



Adaptação automática de algoritmos de otimização metaheurística

JOÃO MARCELO FERNANDES DE CARVALHO

julho de 2022



Automatic adaptation of metaheuristic optimization algorithms

João Marcelo Fernandes de Carvalho

Student nº: 1150892

**Dissertation for obtaining the Degree of
Master in Artificial Intelligence Engineering**

Advisor: Dr. Tiago Manuel Campelos Ferreira Pinto, Invited Assistant Professor from Instituto Superior de Engenharia do Instituto Politécnico do Porto and Assistant Professor from Universidade de Trás-os-Montes e Alto Douro

Co-advisor: Dr. Rubén Augusto Romero, Full Professor from Universidade Estadual Paulista “Júlio Mesquita Filho”

Evaluation Committee:

President:

Dr. Luiz Felipe Rocha de Faria, Associate Professor, Instituto Superior de Engenharia do Instituto Politécnico do Porto

Members:

Dr. Ana Maria Marques Moura Gomes Viana, Associate Professor, Instituto Superior de Engenharia do Instituto Politécnico do Porto

Dr. Tiago Manuel Campelos Ferreira Pinto, Invited Assistant Professor, Instituto Superior de Engenharia do Instituto Politécnico do Porto, and Assistant Professor, Universidade de Trás-os-Montes e Alto Douro

Porto, June 2022

**Dedico este trabalho ao meu pai que não
teve oportunidade de me ver concluí-lo.**

Resumo

A maioria dos problemas do mundo real tem uma multiplicidade de possíveis soluções. Além disso, usualmente, são encontradas limitações de recursos e tempo na resolução de problemas reais complexos e, por isso, frequentemente, não é possível aplicar um método determinístico na resolução desses problemas. Por este motivo, as meta-heurísticas têm ganho uma relevância significativa sobre os métodos determinísticos na resolução de problemas de otimização com múltiplas combinações. Ainda que as abordagens meta-heurísticas sejam agnósticas ao problema, os resultados da otimização são fortemente influenciados pelos parâmetros que estas meta-heurísticas necessitam para a sua configuração. Por sua vez, as melhores parametrizações são fortemente influenciadas pela meta-heurística e pela função objetivo. Por este motivo, a cada novo desenvolvimento é necessária uma otimização dos parâmetros das metas heurísticas praticamente partindo do zero. Assim, e, atendendo ao aumento da complexidade das meta-heurísticas e dos problemas aos quais estas são normalmente aplicadas, tem-se vindo a observar um crescente interesse no problema da configuração ótima destes algoritmos.

Neste projeto é apresentada uma nova abordagem de otimização automática dos parâmetros de algoritmos meta-heurísticos. Esta abordagem não consiste numa pré-seleção estática de um único conjunto de parâmetros que será utilizado ao longo da pesquisa, como é a abordagem comum, mas sim na criação de um processo dinâmico, em que a parametrização é alterada ao longo da otimização. Esta solução consiste na divisão do processo de otimização em três etapas, forçando, numa primeira etapa um nível alto de exploração do espaço de procura, seguida de uma exploração intermédia e, na última etapa, privilegiando a pesquisa local focada nos pontos de maior potencial. De forma a permitir uma solução eficiente e eficaz, foram desenvolvidos dois módulos um Módulo de Treino e um Módulo de Otimização. No Módulo de Treino, o processo *de fine-tuning* é automatizado e, conseqüentemente, o processo de integração de uma nova meta-heurística ou uma nova função objetivo é facilitado. No Módulo de Otimização é usado um sistema multiagente para a otimização de uma dada função seguindo a abordagem de pesquisa proposta.

Com base nos resultados obtidos através da aplicação de otimização por enxame de partículas e algoritmos genéticos a várias funções *benchmark* e a um problema real na área dos sistemas de energia, o Módulo de Treino permitiu automatizar o processo de *fine-tuning* e, conseqüentemente, facilitar o processo de introdução no sistema de uma nova meta-heurística ou de uma nova função relativa a um novo problema a resolver. Utilizando a abordagem de otimização proposta através do Módulo de Otimização, obtém-se uma maior generalização e os resultados são melhorados sem comprometer o tempo máximo para a otimização.

Palavras-chave: algoritmos genéticos, configuração automática de algoritmos, otimização meta-heurística, otimização por enxame de partículas, parametrização dinâmica, sistemas multiagente

Abstract

Most real-world problems have a large solution space. Due to resource and time constraints, it is often not possible to apply a deterministic method to solve such problems. For this reason, metaheuristic optimization algorithm has earned increased popularity over the deterministic methods in solving complex combination optimization problems.

However, despite being problem-agnostic techniques, metaheuristic's optimization results are highly impacted by the defined parameters. The best parameterizations are highly impacted by the metaheuristic version and by the addressed objective function. For this reason, with each new development it is necessary to optimize the metaheuristic parameters practically from scratch. Thus, and given the increasing complexity of metaheuristics and the problems to which they are normally applied, there has been a growing interest in the problem of optimal configuration of these algorithms.

In this work, a new approach for automatic optimization of metaheuristic algorithms parameters is presented. This approach does not consist in a static pre-selection of a single set of parameters that will be used throughout the search process, as is the common approach, but in the creation of a dynamic process, in which the parameterization is changed during the optimization. This solution consists of dividing the optimization process into three stages, forcing, in a first stage, a high level of exploration of the search space, followed by an intermediate exploration and, in the last stage, fostering local search focused on the points of greatest potential. In order to allow an efficient and effective solution, two modules are developed, a Training Module and an Optimization Module. In the Training Module, the fine-tuning process is automated and, consequently, the process of integrating a new metaheuristic or a new objective function is facilitated. In the Optimization Module, a multi-agent system is used to optimize a given function following the proposed research approach.

Based on the results obtained using particle swarm optimization and genetic algorithms to solve several benchmark functions and a real problem in the area of power and energy systems, the Training Module made it possible to automate the fine-tuning process and, consequently, facilitate the process of introducing in the system a new metaheuristic or a new function related to a new problem to be solved. Using the proposed optimization approach through the Optimization Module, a greater generalization is obtained, and the results are improved without compromising the maximum time for the optimization.

Keywords: automatic algorithm configuration, dynamic parameterization, genetic algorithms, metaheuristic optimization, multi-agent systems, particle swarm optimization

Acknowledgements

I would like to start by thanking the professors at Instituto Superior de Engenharia do Instituto Politécnico do Porto (ISEP) who passed me the necessary knowledge, throughout the master's degree, which allowed me to develop this work. I would also like to give a special thanks to my advisor Dr. Tiago Pinto for his availability and contribution throughout these two years of master's degree. And another to Dr. Carlos Ramos for the concept and the structuring of the master's program. Additionally, I also thank Dr. Rubén Romero who accepted to co-supervisor and have contributed towards making this work, even from another institution.

Secondly, I would like to thank ISEP for giving me the proper tools for the development of this work and Grupo de Investigação em Engenharia de Computação Inteligente para a Inovação e o Desenvolvimento (GECAD) for the resources that they made available, and which were used as the basis for the development of the project.

I also thank my colleagues at ISEP, especially those who were part of my work team.

I would like to extend my thanks as well to my mother, my sisters, and my friends for all their support and patience over these years.

Finally, my sincere thank goes to my girlfriend Diana. Not only for letting me know about this master's degree, also for all the encouragement and motivation she gave me over these two years.

Contents

1	Introduction	1
1.1	Problem definition	1
1.2	Contextualization	2
1.3	Main contributions	3
1.4	Document structure.....	5
2	Theoretical background	7
2.1	Planning and scheduling problems	7
2.2	Metaheuristics on planning and scheduling problems	8
2.3	Metaheuristics with dynamic parameterization.....	9
2.4	Metaheuristics validation	11
2.5	Gap on the literature	11
3	Used Libraries	13
3.1	Pyticle Swarm.....	13
3.1.1	Pyticle Swarm basis library	13
3.1.2	Pyticle Swarm contributions	15
3.2	Genetic Algorithm.....	16
3.2.1	Genetic Algorithm basis library	16
3.2.2	Genetic Algorithm contributions	17
3.3	Spade	17
3.4	Benchmark	17
4	Proposed solution	21
4.1	Proposed solution overview.....	21
4.2	Training module	24
4.2.1	Generate combinations based on input file	26
4.2.2	Evaluate generated combinations	28
4.2.3	Calculate best parameterizations by training mode	29
4.2.4	Store parameterizations on the file system	31
4.3	Optimization module	32
4.3.1	Agents.....	34
4.3.2	Multi-agent system	35
4.3.3	Multi-agent system communication.....	37
4.3.4	Metaheuristics response processing.....	37
5	Results and discussion	41
5.1	Impact of the configured thread number in the training execution time	41

5.2	Impact of the configured batch size in the training execution time.....	43
5.3	Impact of the training combinations in the training execution time	43
5.4	Impact of the training combinations number in the optimization results.....	44
5.5	Optimization module - execution time analysis	46
5.6	Impact of the number of trials on the optimization result.....	48
5.7	Impact of the training function on the optimizations results.....	50
5.8	Proposed solution applied to a real-world problem.....	51
5.9	Proposed solution using a genetic metaheuristic	53
5.9.1	Genetic Algorithm training configuration.....	53
5.9.2	Genetic Algorithm customization	53
5.9.3	Genetic Algorithm results and comparison with Pyticle Swarm	57
5.10	Summary.....	59
6	Conclusion	61
6.1	Achieved objectives	61
6.2	Limitations and future work	62
	Bibliography.....	65

List of figures

Figure 1 - Proposed solution overview	4
Figure 2 - Flow chart of the PSO algorithm [40].....	14
Figure 3 - Proposed solution component diagram	22
Figure 4 - Training module sequence diagram.....	26
Figure 5 -Training mode input file example	27
Figure 6 - Example of generated combinations for the PSO (Pyticle) metaheuristic	28
Figure 7 - Evaluate combinations sequence diagram	29
Figure 8 - Calculate parameterizations traditional mode sequence diagram.....	30
Figure 9 - Calculate parameterizations dynamic mode sequence diagram	30
Figure 10 - Structure of the parameters' repository.....	31
Figure 11 - Example of the parameterization files content	32
Figure 12 - Optimization module input examples.....	34
Figure 13 - Multi-agent system component diagram	35
Figure 14 – Multi-agent system sequence diagram.....	36
Figure 15 - Example of the process of selecting the best solutions after a minimization	38
Figure 16 - Example of the process of select the initial solution in a minimization	39
Figure 17 - Metaheuristic execution time by number of configured threads AMD Ryzen 7 4700	42
Figure 18 - Metaheuristic execution time by number of configured threads AMD Ryzen 7 5700	42
Figure 19 - Metaheuristic execution type by the batch size	43
Figure 20 - Impact of the number of combinations in the training execution time	44
Figure 21 - Impact of the number of training combinations on fitness value. Trained and tested with Schwefel(50)	45
Figure 22 - Impact of the number of training combinations on the fitness value. Trained with Schwefel(50) and tested with Rastrigin(50).....	46
Figure 23 - Impact of number of trials in the optimization results. Trained and tested with Schwefel(50).....	48
Figure 24 - Impact of number of trials in the optimization results. Trained with schwefel(50) and tested with Rastrigin(50).....	49
Figure 25 - Impact of training function on the optimization results of the Schwefel(50).....	50
Figure 26 - Impact of training function on the optimization results of the Rastrigin(50)	51
Figure 27 - Impact of the trials number on the optimization results using a real-word problem	52
Figure 28 - Genetic Algorithm training dynamic configuration	53
Figure 29 - Genetic algorithm - Impact of number of trials in the optimization results. Trained and tested with Schwefel(50)	58
Figure 30 - Comparison between the GA and PSO results obtained optimizing the Schewefel(50) function	59

List of tables

Table 1 - Configurable parameters on the Pyticle Swarm library	15
Table 2 - Configurable parameters on the Genetic Algorithm library [46]	16
Table 3 - Benchmark functions definition [19].....	18
Table 4 - Training module system properties	25
Table 5 - Optimization module system properties description	33
Table 6 - Optimization module execution time (milleseconds) analysis.	47
Table 7 - Optimization module time analysis summary.....	47

List of acronyms and symbols

List of acronyms

ABC	Artificial Bee Colony
ACL	Agent Communication Language
ACO	Ant Colony Optimization
APSO	Adaptative Particle Swarm Optimization
BA	Bat Algorithm
BF	Bacterial Foraging
BFO	Biogeography-based Optimization
COP	Combinatorial Optimization Problems
DSM	Demand Side Management
FIPA	Foundation for Intelligent Physical Agents
GA	Genetic Algorithm
GARF	Genetic Algorithm based on Random Forest
GECAD	Grupo de Investigação em Engenharia de Computação Inteligente para a Inovação e o Desenvolvimento
GP	Genetic Programming
GPALS	Gradient-based Parameter Adaption with Line Search
GPOL	General Purpose Optimization Library
GWO	Grey Wolf Optimization
PSO	Particle Swarm Optimization
SMAC	Sequential Model-based Algorithm Configuration
SSO	Salp Swarm Optimization
XMPP	Extensible Messaging and Presence Protocol

List of Symbols

b	Baseline solution value
f_s	Filter size
m	Static number
m_{ax}	Maximum valid value
m_{in}	Minimum valid value
n	New objective function parameter value
n_c	Number of combinations
n_{me}	Number of metaheuristic executions
p_n	Preprocessing repetition number
r_{float}	Random real number
r_{int}	Random integer number
r_n	Repetitions number

1 Introduction

1.1 Problem definition

The main objective of this work is to study and develop an online parameter adaptation model for metaheuristic optimization algorithms [1]. The proposed model should be problem-agnostic and should be able to finetune the metaheuristic parameters. On the other hand, the configuration of algorithm specific (parameters) and problem specific (function variables) must be easy. In this way, the adaptation of the developed metaheuristic to a real-world problem will be faster and the effectiveness of the problem resolution will increase significantly, taking into consideration that the parameterization can be updated more than once in each execution.

The proposed dynamic parameter adaptation model is applied and experimented using two of the most widely used metaheuristic algorithms, namely Particle Swarm Optimization (PSO) and Genetic Algorithm (GA) [2], which have been selected after a thorough analysis of the characteristics of multiple metaheuristic algorithms. Furthermore, the validation process considers the experimentation using traditional benchmark functions and also using a real problem related to electricity market transactions [3].

To accomplish the main objective, the following specific objectives are proposed:

- Review the state of the art. The main metaheuristic optimization algorithms will be analyzed in two perspectives. Firstly, to determine the most used metaheuristics in the literature in order to solve real-world problems. Secondly, to analyze the metaheuristics' characteristics as means to incorporate the online parameter adaptation.
- Create the proposal solution design.
- Develop a metaheuristic to be used as basis for development of the proposed solution.
- Develop the generic online parameter adaptation model, which must be problem and metaheuristic agnostic.
- Test and validate the metaheuristic using the developed model and compare both results – with and without online parameter adaptation - using several widely adopted benchmark functions and, in addition, against a real scenario on the energy sector.

- Analyze the performance impact of the developed algorithm in the optimization execution time.
- Incorporate the developed model in a different metaheuristic in order to experiment and validate the metaheuristic-agnosticism of the proposed model.

1.2 Contextualization

Most of the real-world optimization problems, especially Combinatorial Optimization Problems (COP) have a large solution space [4]. This is caused by a high complexity, nonlinear constraints, and interdependencies among variables. Due to resources and time constraints, it is often not possible to achieve the optimal solution for these problems, since it is not possible to evaluate all the possible solutions as required to implement an exact (deterministic) method. Consequently, in practice, metaheuristic optimization algorithms are being used in order to get an acceptable solution in a reasonable amount of time [4, 5, 6].

Metaheuristics are generally problem-agnostic techniques that can be applied to several optimization problems. These algorithms usually execute an iterative search process in order to find a near-optimal solution [7]. For these reasons, metaheuristics have earned more popularity over the deterministic methods in solving Combinatorial Optimization Problems (COP) [2], and they are being widely used by researchers and developers in several contexts, such as financial [8], medical [9], mechanical [10], chemical [11], electrical/energy [12] and social [13].

However, despite being problem-agnostic techniques, without some customization and fine-tuning on the metaheuristic's parameters, they generally present a low performance. The customization allows the introduction of problem-specific knowledge to adapt the metaheuristic to the particularities of the COP [7]. The parameterization strongly influences the optimization results. This influence can be caused by the value of the parameters and by the relation among them [14]. After these processes the algorithm can become problem tailored, performing well for some optimization problems only [15]. Consequently, the researchers must start "almost" from scratch, building their own metaheuristic version [7]. Taking into account that most of the new generation metaheuristics have a huge complexity and a large number of parameters [16], the time required to configure a complex optimization algorithm is likely to increase in the future.

One possible solution to overcome this problem is the online parameter adaptation. Using this approach, the metaheuristic parameters are updated dynamically based on the algorithm's performance, instead of being defined exclusively in a pre-processing phase [1]. Besides simplifying the customization work, this approach, according with literature, allows to increase the effectiveness and the performance of metaheuristics, by adapting the search process throughout the optimization process [1, 17].

In order to contribute towards the dynamic adaptation of metaheuristics' search process, this work conceives, develops and validates a generic dynamic parameter definition model. This model is able to fine-tune the metaheuristic parameters and automatize the optimization

process in order to facilitate the integration of new objective functions and new metaheuristic algorithms in the system.

1.3 Main contributions

In this work, an online parameter adaptation model for metaheuristic optimization algorithms is developed. The main contributions of this project are the following, as depicted by the Figure 1:

- Training module that allows to automatically train metaheuristics in two modes:
 - Traditional approach - a single static parameterization is chosen
 - Dynamic approach - three parameterizations are chosen and applied consecutively throughout the search process, aiming at balancing the focus on exploration and exploitation
- Optimization module, modeled as a multi-agent system that allows to perform the optimization of a given problem using a selected metaheuristic.
- Validation of the proposed model, considering multiple problems and metaheuristics, including a real-world problem in the power and energy system's domain.
- Publication of an abstract, entitled "Online adaptation of the search process of metaheuristic algorithms" and respective presentation at the "Artificial Intelligence Technique for the Optimization of Electric Power Distribution Systems" workshop [18].
- Development of two articles in progress. One entitled "Multi-agent based model for the dynamic adaptation of metaheuristic optimization", to be submitted in the IEEE Transactions on Industrial Informatics journal; the other entitled "Dynamic parameterization of metaheuristics using a multi-agent system for the optimization of electricity market participation" to be submitted at the 21st International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS) 2023.
- The development of this thesis contributed to development of the international project "Development of Artificial Intelligence Techniques for the Optimization of Electric Power Distribution Systems (FCT/CAPES 2019.00141.CBM)"

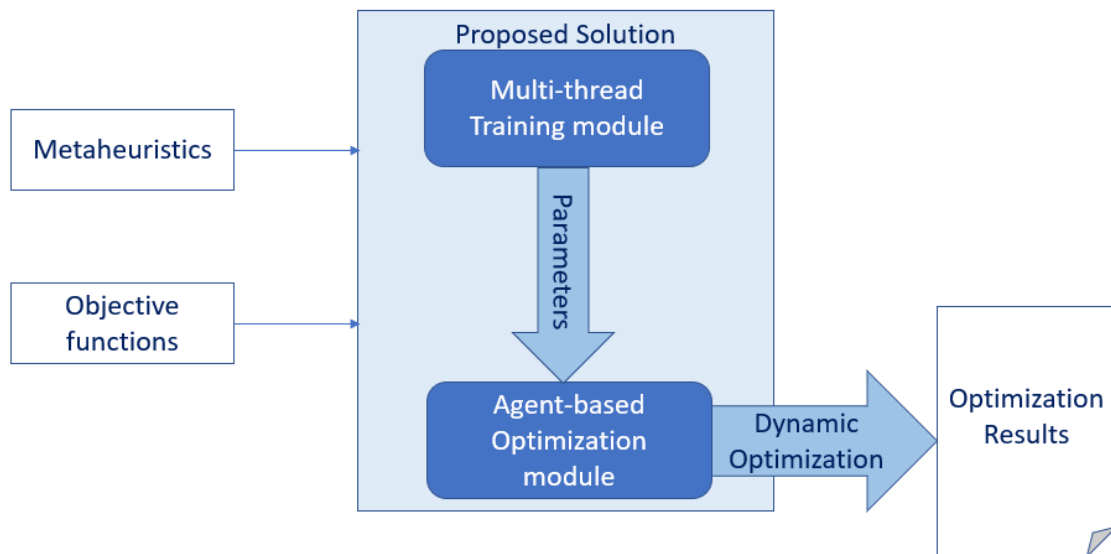


Figure 1 - Proposed solution overview

As shown by Figure 1, the proposed solution is composed by two different modules, a multi-thread Training module and an agent-based Optimization module. These two modules use as input metaheuristics and objective functions and together they have the capacity of generate good parameterizations, execute the optimization, and retrieve the results.

The Training module is implemented comprising two different modes: (i) a traditional model, in which several valid parameterizations are tested and the best one is chosen, and (ii) a dynamic mode, in which three different parameterizations are chosen. These parameterizations are chosen taking into consideration two characteristics, namely the exploration degree and the results obtained. Thereby, the first parameterization should enable a deep exploration of the solution space, in the second the exploration should be reduced, and in the last parameterization, the focus is the local search.

The Optimization module, using the parameterizations obtained on the dynamic training mode, executes the optimization sequentially using each of the parameterizations obtained in the training phase. For the first optimization, the initial solution must be empty, while for the others an initial solution is calculated by the algorithm according to the results achieved in the previous executions.

Under this project, several of the most widely used metaheuristics in the literature have been analyzed, including, PSO, Artificial Bee Colony (ABC), Ant Colony Optimization (ACO), and GA [2]. Considering their characteristics, PSO and GA have been chosen as the algorithms to test and validate the proposed model, enabling assessing the algorithm-agnostic nature of the proposed model. Complementarily, as means to test the capability of the proposed model in dealing with different types of problems, several optimization problems have been considered, namely several benchmark functions widely used in the literature [19] and a real-world problem related with electricity market participation portfolio optimization [3].

1.4 Document structure

The document structure is composed by six different chapters. In chapter 1 the problem is defined and contextualized, and the main objectives of the work are proposed.

In the second chapter, the theoretical background is reviewed, including the definition of planning and scheduling problems, the use of metaheuristic algorithms on planning and scheduling problems, a review on dynamic parameterization and on metaheuristic validation. This chapter concludes by identifying the gaps in the literature.

In the third chapter, the libraries used throughout the project are presented, namely Pyticle Swarm, Spade, Benchmark and Genetic Algorithm.

In the fourth chapter, the solution design is proposed for the online algorithm optimization problem. This chapter presents the details on the implementation of the two main components of the solution: the Training module, and the Optimization module.

In the fifth chapter, the results obtained in the several conducted experiments are discussed. These comprise the analysis of objective function results when using the proposed model, as well as the execution time analysis and the applicability and impact of applying the proposed model to different optimization problems and using different metaheuristic algorithms.

Finally, in chapter 6, the main conclusions, the results achieved, as well as the limitations, and suggestions for future work are presented.

2 Theoretical background

This chapter presents the theoretical background, including the definition of planning and scheduling problems, the use of metaheuristic algorithms on planning and scheduling problems, and then is described dynamic parameterization and metaheuristic validation.

2.1 Planning and scheduling problems

In order to solve various real-world engineering and management problems, driven by the rapid advances in the computing technology, large optimization theories and algorithms have been proposed [20]. These algorithms and theories can be considered deterministic and non-deterministic. On the deterministic approach all the possible solutions are validated, and the best solution is always found [21]. On a non-deterministic approach, a heuristic is implemented in order to find a good solution without validating all possible solutions [7].

So, the deterministic approaches always converge to the global best solution. On the other hand, heuristic approaches are more flexible and efficient, namely for solving nonconvex or large complex optimization problems. For these problems, the deterministic methods may not be able to derive to the solution in a reasonable amount of time. The heuristics are developed in order to reduce the computational resources; however, a feasible or a globally optimal solution is not always guaranteed [20]. Despite the disadvantage, in large solution space problems, the non-deterministic methods are the only feasible option [4, 5, 6].

Metaheuristics are a subset of the non-deterministic techniques characterized by being problem-agnostic techniques that can be applied to several optimization problems. Using these algorithms, usually, an iterative search process is executed in order to find a near-optimal solution [7]. Due to the performance and effectiveness obtained with these techniques, the metaheuristics have earned more popularity over the deterministic methods in solving Combinatorial Optimization Problems [2].

2.2 Metaheuristics on planning and scheduling problems

Influenced by the informatization and the amount of data available to process, during the past years a lot of metaheuristic related works have been proposed. For the mentioned reasons, several different metaheuristics has been created. Generally, they are grouped and classified by some characteristics, nature inspired versus non-nature inspired, population-based versus single-point search, iterative versus greedy, dynamic versus static objective function, memory usage versus memory less, one versus multiple neighborhood structures [4].

The number of algorithm parameters has direct effect on the complexity and, consequently, on the time/iterations required to optimize the metaheuristic parameters. Among these characteristics, from the fine-tuning optimization perspective, the population-based ones can be more complex because it implies the parameterization of a whole family of solutions [4, 22].

Dokeroglua et al. classified the metaheuristics created by the first time before the year 2000 as “classical” metaheuristics and the others as “new generation”. Based on this assumption a state of art review was done and the most implemented classical metaheuristics are the Genetic Algorithm, Particle Swarm Optimization, Ant Colony Optimization and Genetic Programming (GP). In the other hand, the most implemented new generation metaheuristics are Artificial Bee Colony, Bacterial Foraging (BF), Bat Algorithm (BA) and Biogeography-based Optimization (BFO) [16].

Another reason for the success of the metaheuristics is their versatility and the application to all the real-word combinatorial optimization problems. So, these algorithms are being implemented for a lot of different contexts, financial, medical, chemical, electrical/energy and social.

Corazza et al. [23] implemented a hybrid PSO-based algorithm for costly portfolio selection problems. On this paper an exact method, a classic PSO metaheuristic with static parameters and a PSO metaheuristics with dynamic parameterization was compared. On the last approach, a huge reduction of computational time was obtained.

Paul et al. [24] implemented a new feature selection strategy, GA based on Random Forest (GARF) in order to predict the esophageal cancer. In this paper, when compared with the existing algorithms on the literature, the outcome was improved in more than 8% for the predictive study and more than 11% for the prognostic study. These improvements were obtained by the optimization of the selected features and by the tuning of the metaheuristic parameters.

A structural design optimization method of fiber reinforced plastic was proposed by *Kai et al.* [25]. The metaheuristic implemented was the ABC, through 20 executions with different optimization control parameters, an improvement of 8.31% was obtained. However, the proposed implementation requires a lot of time to get an optimum. The researchers, after some experiments, concluded that the execution time is dependent of the defined parameters and the model size. To optimize this time, model simplifying, and the improvement of the optimization algorithm has been proposed.

El-Gendy et al. [26] implemented a hybrid of GA and PSO technique for tuning a Proposal-Integral-Derivative controller parameters used in a chemical process. The proposed solution gets

better results when compared with the existing solutions in the literature. All the parameters, of GA and PSO, are configured without turning the metaheuristic parameters, i.e. the used parameterization is defined based on the existing papers on the literature.

In order to satisfy the user demands and make the best use of the available power on a community parker charging station, Álvarez *et al.* [27] proposed an ABC algorithm with local search. The implemented solution has been compared with the state of art and a similar result was obtained. However, the computation time was reduced significantly allowing the utilization in the real environments for online scheduling. The time was optimized by some manual updates on the metaheuristic parameters.

In Ozbay and Alatas [28] two novel optimization approaches for fake news detection were implemented. These approaches are implemented following the Grey Wolf Optimization (GWO) and the Salp Swarm Optimization principals (SSO). Taking into consideration the obtained results, these two metaheuristics are considered very promising. However, to improve the algorithm's performance, the researchers suggest a dynamic parameterization implementation and a refactor on the metrics used on the model construction.

In summary, metaheuristics are very dynamic approaches, and for this reason they have been used in several different contexts in order to solve combinatorial optimization problems. However, a problem identified on the existing algorithms in the literature is the metaheuristics parameterization. The parameterization influences significantly the performance of the algorithm and consequently the computation time required to archive a good result. On the papers referred on this section, the problems were already partially addressed with some manual testing for each implementation or by problem specific dynamic online parameterization. Using these solutions, the optimization code implemented cannot be reused in other contexts and the research must start optimizing from scratch. A possible way to overcome the limitation is to implement metaheuristic models with dynamic online parameterization.

2.3 Metaheuristics with dynamic parameterization

The metaheuristics effectiveness depends on the interaction among several components. Generally, without an online parameter configuration strategy, the components are configured based on the similar problems on the literature or based on a preliminary experimentation with different parameter combinations [29]. However, in the metaheuristics configuration there are many possible choices, and it is almost impossible to guarantee that the best configuration has been chosen. Besides this, the best configuration may be conditioned by the specific data of each execution [17].

To overcome this, to get better results and to improve the algorithm performance, online parameter adaption algorithms can be implemented. Using this approach, the metaheuristic is able to modify, in runtime, the configuration and the strategic parameters. The parameters are set based on the feedback obtained in the current execution and the knowledge acquired previously [17, 30].

Motivated by the potential of this approach, several algorithms with dynamic parameterization have been developed in the last years. Generally, the outputs are not a new metaheuristic, but an improved version of the most used metaheuristics, such as GA and PSO.

Melin *et al.* [31] developed three fuzzy systems for online parameter adaptation. These algorithms are incorporated in a base PSO metaheuristic. The four versions are tested against the same benchmark functions and a significant improvement was obtained in two of the three implemented PSO versions. The results allowed to conclude that dynamically adjusting parameters can improve the quality of the metaheuristic results. Zhan *et al.* [32] extended the PSO metaheuristic to an Adaptive Particle Swarm Optimization (APSO). The APSO is formulated by an elitist learning strategy and an evolutionary state estimation technique. The developed APSO algorithm was tested against 12 benchmark functions, and it was concluded that the adaptive algorithm enhances substantially the convergence speed, the global optimality, the solution accuracy and the algorithm reliability.

Motivated by the time and results inconsistency of the heuristic algorithms on solving vehicle routing problems, Zakharov and Mugayskikh [33] implemented a dynamic adaptation procedure for the GA metaheuristic. The procedure was evaluated using the Traveling Salesman library and a significant improvement was achieved. The generated solutions are significantly better, and the time needed to achieve the solution was more consistent. Moctezuma *et al.* [34] implemented a self-parameter adaptation mechanism that can be used with the GA metaheuristic. The algorithm was optimized to dynamic problems. When a change on the problem occurs, the system is able to apply a diversification on the mutation and the crossover probabilities. The solution was tested against benchmark functions, and it demonstrates an improvement on the metaheuristic performance.

Tatsis and Parsopoulos [1], in order to reduce the effort needed to the metaheuristic's parameter tuning and control, proposed a gradient-based parameter adaptation method. The implemented solution is metaheuristic agnostic and can be by any population-based metaheuristic and it use a reinforcement learning approach to adapt dynamically the algorithm parameters. A high-dimensional test suite was used. Based on the obtained results, it is possible to conclude that the algorithm is able to get automatically a good parameters configuration without increasing the execution time.

Tatsis and Parsopoulos [29] proposed a Gradient-based Parameter Adaptation with Line Search (GPALS). The algorithm works using a primary population and several secondary populations with a set of different parameters configured. Periodically, the results obtained by the secondary populations are evaluated and, if it is appropriate, the settings of the main population are updated. The developed model was able to correctly identify the adequate parameters, taking that responsibility away from the user. The results obtained when compared with related literature demonstrate a very competitive performance. The comparison was made using benchmark functions.

Shadkam [35] developed a hybrid algorithm called Demand Side Management (DSM). The algorithm, optimizing the metaheuristic parameters, is able to maximize its efficiency. On this paper, the algorithm was applied to the cuckoo optimization metaheuristic; however, in a future work it can be applied to other metaheuristics. In order to evaluate the performance, the

metaheuristic was tested against a covid-19 management problem. When compared with the baseline metaheuristic, the metaheuristic with DSM shows a better performance in terms of solution time, number of iterations, efficiency, and accuracy.

Han and Xiao [36] proposed an improved adaptative genetic algorithm. The proposed solution updated online the crossover probability and mutation probability in order to enhance the global optimization ability of the algorithm. The solution was validated using the traveling salesman problem and the experimental data revealed improvements in the convergence speed and in the operation efficiency.

2.4 Metaheuristics validation

As mentioned on the previous sub chapters, the metaheuristics are problem agnostic optimization algorithms that can be applied to a lot of real-world optimization problems. However, even with the automated parameter tuning, to utilize a developed metaheuristic in a real-word problem it is required some problem specific knowledge in order to enable suitable feature selection and objective function definition. For this reason, to evaluate the metaheuristics effectiveness and efficiency several benchmark functions have been used on the literature [29, 31, 32, 34].

Following the metaheuristic development, in the past decades, a huge variety of benchmark problems and collections have been developed. However, according with Sala and Müller [37], there is a gap between the benchmark functions and the real-word optimization problems. The possible explanation for this gap is the growing complexity of the real-world problems, the focus of the research community on the benchmark functions that are secondary for the real-word problems and the small size/complexity of the benchmark functions.

To overcome the mentioned gap, some authors suggest the development of more realistic and complex benchmark functions. Nevertheless, overall, these suggestions were not followed by the researchers and, consequently, there are no benchmark functions in the literature that are sufficiently representative, complex, and realistic [37].

2.5 Gap on the literature

In the past years, several online parameter adaption algorithms have been developed and published on the literature. Generally, with these algorithms, the metaheuristics efficiency and effectiveness have been increased significantly. However, the existing online parameter adaption methods were optimized to the problem in study (overspecialized), and it implies the inclusion and configuration of new critical parameters [29].

Besides the referred weaknesses, these algorithms that already exist in the literature were not tested against a representative set of problems. Generally, the algorithm's validation was made following one of two approaches. The first one is testing against only one real-world problem, causing overspecialization. The second one is testing only against benchmark functions. Despite

the improvements obtained with the algorithms in the literature, taking into consideration the overspecification of the first testing approach and that the benchmark functions are not representative, complex, and realistic, it is not possible to guarantee that the algorithms will have a good performance when applied to a significant set of real-world problems.

3 Used Libraries

This chapter presents the libraries used throughout the project, namely Pyticle Swarm, Genetic Algorithm, Spade and Benchmark. In addition, the contributions developed to the Pyticle Swarm and the Genetic Algorithm are also presented.

3.1 Pyticle Swarm

3.1.1 Pyticle Swarm basis library

This work uses the PSO metaheuristic as basis since it is one of the most widely implemented metaheuristics and it is a population-based approach that has several parameters to configure. Besides this, the existing online parameter adaption methods applied to the PSO algorithm demonstrate a good improvement on the metaheuristic efficiency.

The PSO algorithm is a population-based stochastic optimization technique proposed by Kennedy and Eberhart in 1995 [38]. It is inspired on the social behavior of the swarming animals, like insect, herds, birds, and fishes. The swarms working together in order to establish a cooperation between them changing their behavior based on the previous experiences [39].

In order to uniformize and optimize the swarm artificial life systems, Millonas proposed five basic principles that should be followed in the PSO, and similar metaheuristics. The principals are proximity, quality, diverse response, stability, and adaptability. To fulfill the proximity principle the swarm should be able to carry out simple space and time computations. To satisfy the quality principle, the environment quality factors should be responded. To carry out the diverse response principle, the algorithm should not limit the response to a subset of the domain solutions. The last two principles can be considered contradictory, but it means that the swarm should not change its behavior after each environment change. The system should be able to realize when it is beneficial to adapt to the environment [39].

A flowchart of the basic PSO metaheuristic is depicted in Figure 2.

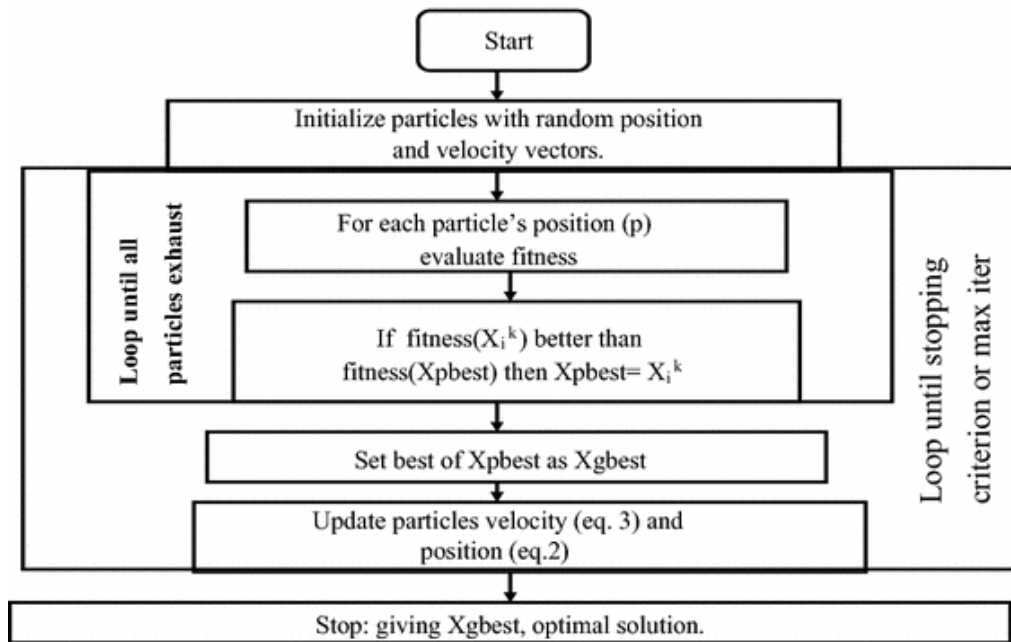


Figure 2 - Flow chart of the PSO algorithm [40]

As shown by Figure 2, the algorithm starts by creating a population of particles in a random location, for each particle a random velocity is associated. Iteratively, an objective function evaluates each particle location and determines the better located particle. Based on the current velocity, the velocities of the best located particles and their neighbors are updated. After each iteration the particles are moved to the next location. The next location is calculated based on the previous one and the velocity defined by the algorithm. This process is executed until a stop condition is reached or the maximum number of iterations is exhausted [40].

In order to get valid results in a reasonable time, before each utilization, the PSO parameters must be optimized. The parameter list, generally, may depend on the PSO version, but generally, it includes the number of particles, acceleration constant, inertia weight and maximum limited velocity [39, 40].

In the past years several PSO libraries, like PySwarms [41], General Purpose Optimization Library (GPOL) [42] and Pyticle Swarm [43], have been published. On this project, the Pyticle Swarm is used as baseline since it is a work in progress open-source library developed by Grupo de Investigação em Engenharia de Computação Inteligente para a Inovação e o Desenvolvimento (GECAD), aiming at a fully flexible version of PSO, offering the possibility to configure all parameters and apply the PSO to any optimization problem. This means that, in the end of the project, the developments made in the scope of this work can be included on the library.

Using Pyticle Swarm, it is possible to optimize any problem. To do so, the library allows the configuration of several parameters, such as the initial and final weigh inertia, the learning factors, the total amount of particles and total number of iterations. The full list of parameters and the associated description can be consulted in Table 1.

Table 1 - Configurable parameters on the Pyticle Swarm library

Parameter Name	Default Value	Description
initial_solution	[]	Matrix or array containing the initial solutions or solution
brm_function	4	Function to handle boundary constraint violation
n_jobs	-2	Number of concurrently running jobs
direct_repair	None	The direct repair function
perc_repair	0	Value between 0 and 1 that determines the percentage of iterations starting from the end where a repair function is applied
wmax	0.5	The maximum value of the inertia weight
wmin	0.1	The minimum value of the inertia weight
c1min	0	Minimum value of the acceleration coefficient c1
c1max	0.4	Maximum value of the acceleration coefficient c1
c2min	0.1	Minimum value of the acceleration coefficient c2
c2max	2	Maximum value of the acceleration coefficient c2
n_iterations	100	The total number of iterations
n_particles	10	The total number of particles
n_trials	30	The total number of trials
show_fitness_graphic	False	Boolean that indicates if the fitness graphic is to be shown or not
show_particle_graphics	False	Boolean that indicates if the particles graphics are to be shown or not (Only works with solutions of 2 dimensions)
verbose	True	Boolean that indicates if the results are to be logged on the console

From the architectural perspective, the library is composed by three different modules. The function module contains all the mathematic functions required to the algorithm, such as initial solution creation function, inertia updating function, local and global acceleration coefficients updating function, velocity updating function, velocity limits control function and position updating function. The main module that should be customized by the user and the results class module that can be used to return the metaheuristic results compacted and standardized.

3.1.2 Pyticle Swarm contributions

Pyticle Swarm was used as basis to develop the proposed algorithm. In this process, some enhancements have been made:

- Update the metaheuristic to support maximization problems. The basis version of Pyticle Swarm was only prepared to deal with minimization problems.
- Allow, by configuration, that all evaluated solutions are stored and returned.
- Improve the metaheuristic API to be dynamic on the received parameters. In this way it is not necessary to always send all the parameters.

3.2 Genetic Algorithm

3.2.1 Genetic Algorithm basis library

This works uses the GA metaheuristic since it is one of the most widely implemented metaheuristics. It is a population-based approach that has several parameters to configure.

GA is a search metaheuristic inspired by Charles Darwin's evolution theory. Through an iterative process, the natural selection process is simulated. It means that during the optimization, the species who can adapt better to the environment changes will survive and consequently will be a part of the creation of the next generation. To create the next generation, the previous one is used as basis and three main types of rules are applied. The selection rules impose that only some individuals are filtered out of the population. The crossover rules consider that two selected individuals are combined in order to generate a valid and different solution. The mutation rules introduce some random values that are randomly applied in the generated population [44].

In the past years several GA libraries, like Pygad [45], Genetic Algorithm [46] and Genetic-Algorithms [47], have been published. On this project, the Genetic Algorithm is used since it is an open-source library that offers the possibility of configuring all the parameters and applying the GA to any optimization problem.

Using Genetic Algorithm, it is possible to optimize any problem. To do that, the library receives the configuration of several parameters. The full list of parameters, the default value and the associated description can be consulted in Table 2.

Table 2 - Configurable parameters on the Genetic Algorithm library [46]

Parameter Name	Default Value	Description
max_num_iteration	None	Determines the max number of iterations that can be executed in the optimization. If the value is None, this param is not considered.
population_size	100	Determines the number of trial solutions in each iteration.
mutation_probability	0.1	Determines the chance of each gene in each individual solution to be replaced by a random value
elit_ratio	0.01	Determines the number of elites in the population.
crossover_probability	0.5	Determines the chance of an existed solution to pass its genome to new trial solutions.
parents_portion	0.4	The portion of population filled by the members of the previous generation.
crossover_type	uniform	There are three options including one_point, two_point, and uniform cross.
max_iteration_without_improv	None	Max number of iterations without improving until stop the optimization.

3.2.2 Genetic Algorithm contributions

On this project, the Genetic Algorithm library was used to validate the adaptability of the proposed solution to different metaheuristics. In the process of integrating this metaheuristic in the proposed system, some enhancements have been made, as follows:

- Allow, by configuration, that all evaluated solutions are stored and returned.
- Addition of a new parameter named *initial_population*. By default, it is an empty list and if filled, it is used as basis in the optimization.

3.3 Spade

Spade is an async-based multi-agent systems platform developed in Python. It is based on an Extensible Messaging and Presence Protocol (XMPP) instant messaging, it supports the Foundation for Intelligent Physical Agents (FIPA) [48] metadata using XMPP Data Forms, and the development of agents based on behaviors [49].

The communication can be done using templates. On these templates it is possible to define the sender, the receiver, the message body, and the metadata. The metadata is a strings dictionary containing useful information, such as the FIPA attributes [49].

The agents are implemented based on behaviors. The spade supports four different behaviors: (i) one shot behaviors that are only executed once, (ii) cyclic behaviors that are used by agents that are waiting for messages, (iii) periodic behaviors that are executed periodically and (iv) fine state machine behaviors that are more complex and are composed by a set of registered states and transitions [49].

On this project, the agents are implemented using the one shot, the cyclic and the periodic behaviors. The communication between agents is performed using templates through a local instance of the ejabberd software [50]. The body of the templates is a Python dictionary converted in json.

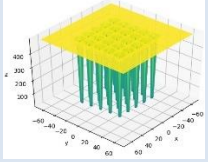
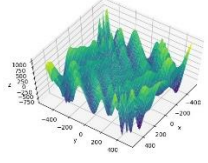
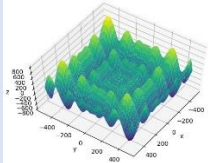
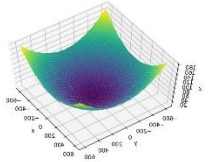
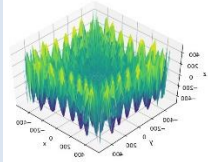
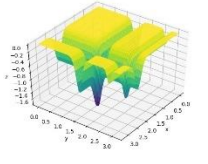
3.4 Benchmark

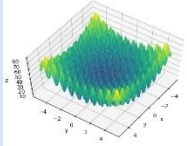
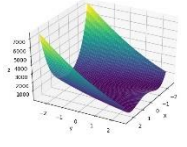
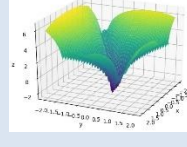
The metaheuristics are problem agnostic techniques that can be used in several real-world problems. For this reason, to evaluate the accuracy and the effectiveness of a metaheuristic and to avoid the metaheuristic overspecification, they should be tested under several different contexts. However, the process of customizing a metaheuristic to a real-world problem is too long and requires knowledge about the application sector. To overcome this, and enable a solid validation of metaheuristic performance, several benchmark functions have been created.

The Python library Benchmark Function is an open-source and contains a collection of benchmark functions written in Python 3.X version. This is a very useful collection because it is possible to increase the functions dimension in an arbitrary way, consequently it increases the complexity of the function. The dimension default value is 2, it is the smallest value, and it means

low complexity. Besides this, the library allows to configure the function to maximize instead of minimizing, consequently it increases the list of covered scenarios. As a result, the function allows the visualization in a graphic way [19]. In Table 3 a subset of the benchmark functions available on the mentioned library is explained. All the data on the table was extracted from the library official documentation.

Table 3 - Benchmark functions definition [19]

Name	Image	Description
De Jong 5		Continuous, multimodal, multiple symmetric local optima with narrow basins on a plateau. It is defined only for 2 dimensions. $f(x) = (0.002 + \sum_{i=1}^{25} (i + (x_1 - A_{1i})^6 + (x_2 - A_{2i})^6)^{-1})^{-1}$ <p>(1)</p>
Egg Holder		Non-convex, contains multiple asymmetrical local optima. $f(x) = - \sum_{i=0}^{N-2} (x_{i+1} + 47) \sin \sqrt{ x_{i+1} + 47 + 0.5x_i } + x_i \sin \sqrt{ x_i - (x_{i+1} + 47) }$ <p>(2)</p>
Schwefel		Non-convex and (highly) multimodal. Location of the minima are geometrical distant. $f(x) = 418.9829N \sum_{i=0}^{N-1} x_i \sin(\sqrt{ x_i })$ <p>(3)</p>
Griewank		Non-convex and (highly) multimodal, it shows a different behavior depending on the scale (zoom) that is used. $f(x) = \sum_{i=0}^{N-1} \frac{x_i^2}{4000} - \prod_{i=0}^{N-1} \cos \frac{x_i}{\sqrt{i+1}} + 1$ <p>(4)</p>
Rana		Highly multimodal symmetric function. $f(x) = \sum_{i=0}^{N-2} x_i \cos \sqrt{ x_{i+1} + x_i + 1 } \sin \sqrt{ x_{i+1} + x_i + 1 } + (+x_{i+1}) \sin \sqrt{ x_{i+1} + x_i + 1 } \cos \sin \sqrt{ x_{i+1} + x_i + 1 }$ <p>(5)</p>
Michaewicz		Non-convex and (highly) multimodal. Contains n! local minimum. Use a parameter m that defines the stepness of the curves. Global minimum around f([2.2,1.57])=-1.8013 for n=2, f(x)=-4.687 for n=5 and f(x)=-9.66 for n=10 (no optimal solution given).

Name	Image	Description
		$f(x) = - \sum_{i=0}^{N-1} \sin(x_i) \sin^{2m}\left(\frac{x_i^2(i+1)}{\pi}\right)$
		(6)
Rastrigin		Non-convex and (highly) multimodal. Location of the minima are regularly distributed. $f(x) = 10N + \sum_{i=0}^{N-1} (x_i^2 - 10 \cos(2\pi x_i))$
		(7)
Rosenbrock		Non-convex and unimodal. Global minimum difficult to approximate. $f(x) = \sum_{i=0}^{N-2} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$
		(8)
Pycheny, Goldstein and Price		Non-convex, multimodal with multiple asymmetrical slopes and global minimum near local optima. It is defined only for 2 dimensions. $f(x) = 2.427^{-1} \left(\log \left[\left(1 + (x_0 + x_1 + 1)^2 (19 - 14x_0 + 3x_0^2 - 14x_1 + 6x_0x_1 + 3x_1^2) \right) \cdot (30 + (2x_0 - 3x_1)^2 (18 - 32x_0 + 12x_0^2 + 48x_1 - 36x_0x_1 + 27x_1^2)) \right] - 8.693 \right)$
		(9)

This library is used to validate the proposed solution. However, as concluded on the previous chapters, the benchmark functions, when compared with some real-world problems are not complex enough. On this project, to test the developed model, a real-world problem in the power and energy systems' domain is used. The problem consists of the portfolio optimization for electricity market participation. The main goal is to replicate the solution proposed by Faia *et al.* [51] and compare the results obtained with the results obtained on the published paper. The data required is provided and anonymized by GECAD.

4 Proposed solution

This section presents the proposed solution for the online algorithm optimization problem. The solution is composed by two main modules, the Training module and the Optimization module. Initially an overview of these modules and the relation between them is described. After that, these modules are separately described in detail.

4.1 Proposed solution overview

The objective of this project is to implement an online parameterization algorithm capable of optimizing the fine-tuning process of metaheuristic algorithms and at the same time improve the optimization results. In addition, the algorithm must be metaheuristic- and problem-agnostic.

Taking into consideration that the metaheuristic parameters are not standardized, two modules are built, the Training and the Optimization modules. These modules can be used for maximization and minimization problems, and are totally generic, i.e. they can be used with any metaheuristic and objective function.

The Training module is responsible for generating and evaluating a list of valid parameterizations in order to create a set of parameters that can be used on the optimization. The Training module is supported and configured through an input file. On the input file the metaheuristic, the objective function, the metaheuristic parameters, the parameters range, and the relation between these parameters must be configured. This module supports two modes (i) a traditional mode where several parameterizations are evaluated, and the best one is chosen; and (ii) a dynamic mode where the parameterizations are chosen assuming that optimization will occur in three steps. On the first step, the optimization should occur trying to explore several locations

on the search space. On the second, the exploration should be reduced and on the third, the local search should be privileged.

The Optimization module is composed by a multi-agent system with three different agent types that uses a set of parameters already defined and can be used to optimize an objective function. For each supported metaheuristic, the Optimization module requires, at least, three different configurations. This is mandatory because the system optimizes the metaheuristic following the dynamic approach already described, where the optimization is executed in three steps. So, on the first execution, the parameters should force a high exploration on the solutions; on the second, parameters should ensure that exploration is reduced and excludes the non-worth exploring places of the search space; on the last execution, the parameterization should represent a high exploitation from the metaheuristic algorithm. In this way, before converging to a good solution, the optimization explores a wide range of solutions avoiding local minimum or maximum solutions. These parameters can be obtained using the Training Module or can be defined manually by the user.

Figure 3 presents a component diagram of the proposed solution.

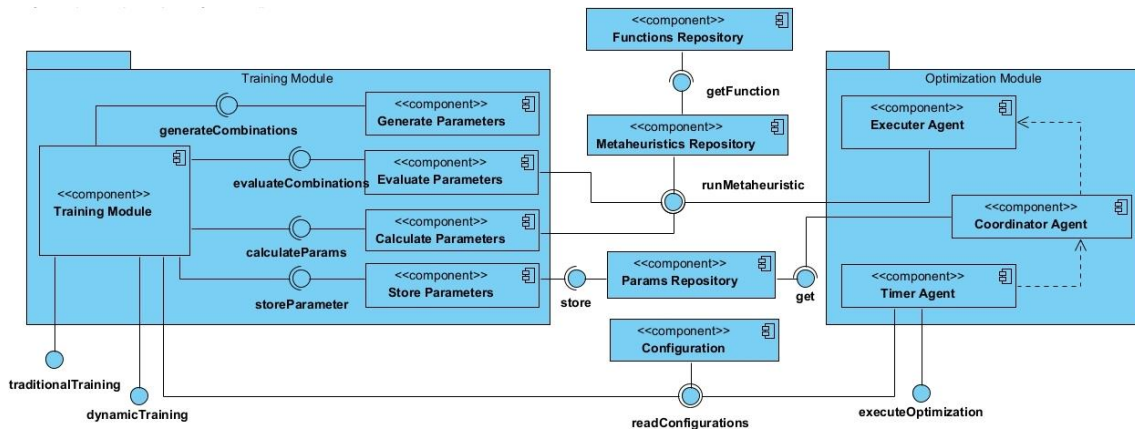


Figure 3 - Proposed solution component diagram

As can be observed by the diagram of Figure 3, other components have been implemented in addition to the two main modules, as means to guarantee the adequate interaction between them and with the user. The Configuration component is used by both modules and allows the user to customize the executions. The Params Repository is used to store and retrieve optimized parameters. The repository is a configured directory, and the parameterizations can be added manually or automatically by the Training Module. The Functions Repository and the Metaheuristics Repository are externalized and must be updated for each supported metaheuristic or objective function.

The Functions Repository is a Python file that must be always implemented on the directory `<base_dir>/customization/`. In the code snippet 1 is an example of the mentioned Python function.

```

import benchmark_functions as bf

def get_function(function_id):
    if function_id == " Schwefel150":
        return bf.Schwefel(50)
    if function_id.startswith("Rana50"):
        return bf.Rana(50)
    raise ValueError("The function received is not supported!")

```

Code snippet 1 – FunctionsRepository.py implementation example

So, the function *get_function* that receive an identifier and based on it retrieve the right implementation. If the identifier is not known the function must retrieve a *ValueError* exception.

The Metaheuristic repository is also a Python class that must also be implemented inside the directory *<base_dir>/customization/*. In the code snippet 2 is an example of the mentioned Python function.

```

from metaheuristic.example import run
from ftoptimizer.model.optimization_result import OptimizationResult
from customization.functions_repository import get_function

def execute_optimization(algorithm_id, function_id, mode, parameters):
    if algorithm_id == "example":
        return run_example(function_id, parameters)
    raise ValueError("The metaheuristic received is not supported!")

def run_example(function_id, parameters):
    parameters["fitness_function"] = get_function(function_id)
    # It is possible to add custom configurations
    parameters["example"] = True
    res = run(parameters)
    return OptimizationResult(res.fitness_value, res.solution,
                              res.all_fitness_values, res.all_solutions)

```

Code snippet 2 – MetaheuristicsRepository.py implementation example

This class must implement the function *execute_optimization* that receives the following parameters:

- *algorithm_id* – It is unique, and it identifies a metaheuristic that must be used.
- *function_id* – It is unique, and it identifies the objective function that must be optimized.
- *mode* – It can assume the values “maximization” or “minimization”, and it indicates if the generic algorithm is trying to maximize or minimize the objective function.
- *parameters* – It is a Python dictionary with all the parameters that should be considered on the current execution.

The function retrieves an *OptimizationResult* object. This is an internal object that has the following attributes:

- *best_fitness_value* – It should contain the best fitness function value.
- *best_solution* – It should contain the solution used to obtain the best fitness value.

- `fitness_values` – It should contain the list of all fitness values obtained on the optimization.
- `solutions` – It should contain the list of solutions analyzed on the algorithm. The values on this list must match the values on the `fitness_values` attribute.

If the memory used on the optimization is a critical factor, instead of all the solutions, the metaheuristic can only return the best one hundred evaluated solutions.

On the development of the proposed solutions the Python libraries Benchmark, Pyticle Swarm and Genetic Algorithm were used. In the integration of the Pyticle Swarm library some improvements described on the section 3.1.1 are implemented.

4.2 Training module

The Training module is a problem- and metaheuristic- agnostic module developed in Python that allows the automatization of the metaheuristics fine-tuning process. The module supports two different modes. The traditional mode where several combinations are evaluated and the best are chosen, and the dynamic mode where several combinations are evaluated and a high exploration, an intermediate and a high exploitation parameterization are chosen.

In order to facilitate the integration with different metaheuristics, the Training module is configurable by system properties. These properties must be defined in a *training.properties* file located in the base directory. The Table 4 describes all the system properties and shows an example value for each property.

Table 4 - Training module system properties

Property	Description	Example value
training.queue.directory	Directory where the input files are located.	C:\tmp\training_queue
training.archive.directory	Directory where the input files are moved after being processed.	C:\tmp\archive_dir
output.params.base.dir	Base directory where the information with the output parameters will be stored.	C:\tmp\params
parallel.threads.number	Number of parallel threads used on the training.	6
thread.exec.batch.size	Size of the batch's processed for each thread execution. Condition the interval of a progress log is printed on the console.	20
pre.processing.rept.number	Number of repetitions executed on the preprocessing. It means that for each combination created, n trails will be executed.	3
calculate.params.filter.size	Number of parameters filtered for evaluation after the preprocessing phase.	100
calculate.params.rept.number	Number of repetitions executed for each combination selected after the preprocessing phase.	20

The system properties `pre.processing.rept.number`(p_n), `calculate.params.filter.size` (f_s) and `calculate.params.rept.number`(r_n) impact directly the number of times the objective function is evaluated and consequently it directly impacts the training execution time. The number of metaheuristic executions can be calculated using the equation 10.

$$n_{me} = p_n \times n_c + m \times (4 \times f_s \times r_n) \quad (10)$$

where n_{me} is the number of metaheuristic executions, p_n is the preprocessing repetition number, n_c is the number of combinations, m is a static number, 1 for the traditional mode and 3 for the dynamic mode, f_s is the filter size and r_n is the calculate parameters repetitions number.

Figure 4 presents a sequence diagram describing the training workflow.

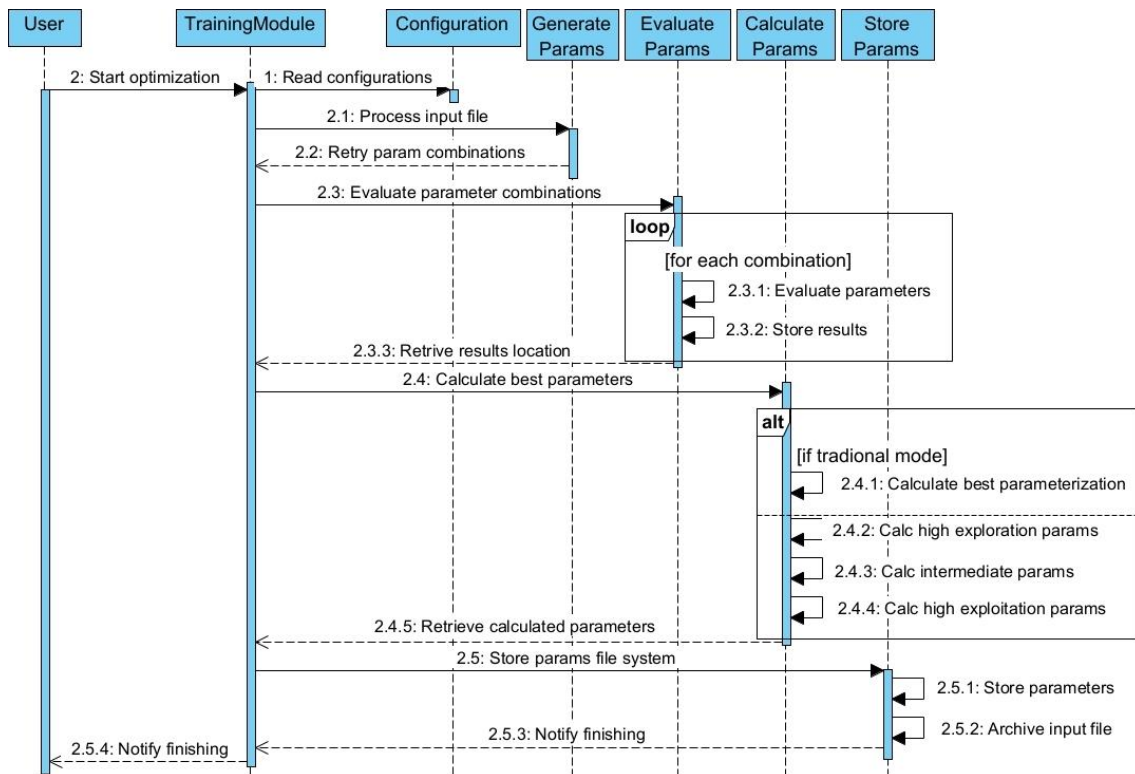


Figure 4 - Training module sequence diagram

As can be observed by Figure 4, the system properties are read on the start-up and four components invoked sequentially. The *GenerateParams* is responsible by input file processing and by generating all the parameterizations that must be evaluated. After this, the *EvaluateParams* evaluates n configured times each combination and stores the results. The *CalculateParams* processes the results and chooses the best parameterizations taking into consideration the training mode. In the end, the *StoreParams* stores the calculated parameters and archives the processed input file.

4.2.1 Generate combinations based on input file

The metaheuristic is not standardized and for this reason to build a training the available parameters and their range must be configured. These must be configured in an input file. For the two supported modes, the input structure is the same, however the file must start with “*dynamic_*” for the dynamic mode and with “*traditional_*” for the traditional mode.

In Figure 5 is presented an example of a valid configuration for the PSO (Pyticle) metaheuristic and for the Schwefel objective function.

```
dynamic_Schwefel_pytticle.txt X
1 metaheuristic_id:PytticleSwarm
2 function_name:Schwefel50
3 mode:minimization
4 fixed_param:n_particles:100
5 fixed_param:n_iterations:50
6 relations:c1min<c1max,c2min<c2max,wmin<wmax
7 dynamic_param:c1min:0.1,1.1:5
8 dynamic_param:c1max:0.6,1.6:5
9 dynamic_param:c2min:1,2:5
10 dynamic_param:c2max:1.5,2.5:5
11 dynamic_param:wmin:0.4,1.4:5
12 dynamic_param:wmax:1,2:5
```

Figure 5 -Training mode input file example

On the input file is configured the metaheuristic identifier, the name of the function, the optimization mode, the fixed parameters, the dynamic parameters, and the relations between the dynamic parameters that must be followed on the process of generating the combinations.

The metaheuristic identifier and the function name must be on the respective repositories and the mode can be *maximization* or *minimization*. The *fixed_param* attribute is used to configure the parameters that are valid for all the combinations and for all the parameters that impact the number of objective function evaluations. It means that, for every generated combination the number of objective function evaluations will be the same.

The *dynamic_param* attributes must follow the structure:

- *<parameter name>:<minimum value>:<maximum value>:<split factor>*

On this configuration, the split is used to define the list of parameters for each dynamic parameter. So, for the minimum value 0 and for maximum value 1, if the split factor is 5 the list of parameters used build the combinations will be (0, 0.2, 0.4, 0.6, 0.8, 1), if it is 4 the list will be (0, 0.25, 0.5, 0.75, 1). This factor conditionate the number of combinations generated and, consequently, it impacts the training execution time.

The relations attribute contains the restrictions that must be followed for the parameterization to be considered valid. For example, if the *wmin* domain is (0, 0.5, 1) and the *wmax* domain is (0.5, 1, 1.5), taking into consideration the restriction *wmin<wmax*, for *wmin = 1*, the domain for the *wmax* value will only be 1.5.

In Figure 6 are presented examples of the combinations generated using the input file depicted in the Figure 5.

```

{
  "c1min":0.1,
  "c1max":0.6,
  "c2min":1.0,
  "c2max":1.5,
  "wmin":0.4,
  "wmax":1.0,
  "n_particles":100,
  "n_iterations":50,
  "function_name":"Schwefel50",
  "alg_name":"PyticleSwarm",
  "opt_mode":"minimization"
}
{
  "c1min":0.1,
  "c1max":0.6,
  "c2min":1.0,
  "c2max":1.5,
  "wmin":0.4,
  "wmax":1.75,
  "n_particles":100,
  "n_iterations":50,
  "function_name":"Schwefel50",
  "alg_name":"PyticleSwarm",
  "opt_mode":"minimization"
}
{
  "c1min":0.1,
  "c1max":0.6,
  "c2min":1.0,
  "c2max":1.5,
  "wmin":0.4,
  "wmax":1.25,
  "n_particles":100,
  "n_iterations":50,
  "function_name":"Schwefel50",
  "alg_name":"PyticleSwarm",
  "opt_mode":"minimization"
}
{
  "c1min":0.1,
  "c1max":0.6,
  "c2min":1.0,
  "c2max":1.5,
  "wmin":0.4,
  "wmax":1.5,
  "n_particles":100,
  "n_iterations":50,
  "function_name":"Schwefel50",
  "alg_name":"PyticleSwarm",
  "opt_mode":"minimization"
}

```

Figure 6 - Example of generated combinations for the PSO (Pyticle) metaheuristic

As one can see by Figure 6, each parameter combination contains the identifiers, the optimization mode, the fixed and dynamic parameters. These parameterizations are stored in memory and are sent to back to the Training component in order to be used by the component Evaluate combinations described on the following subchapter.

4.2.2 Evaluate generated combinations

The evaluate generated combinations component receives a list of parameterizations and evaluates them, stores the parameterization and results in a file on the file system and retrieves the file location to the Training main component. To optimize this process, the user has the possibility of configuring, by system property, the number of parallel threads, the batch size processed for each thread, the output file directory, and the number of times each combination is evaluated.

Figure 7 shows a sequence diagram describing the process of evaluate the parameterizations.

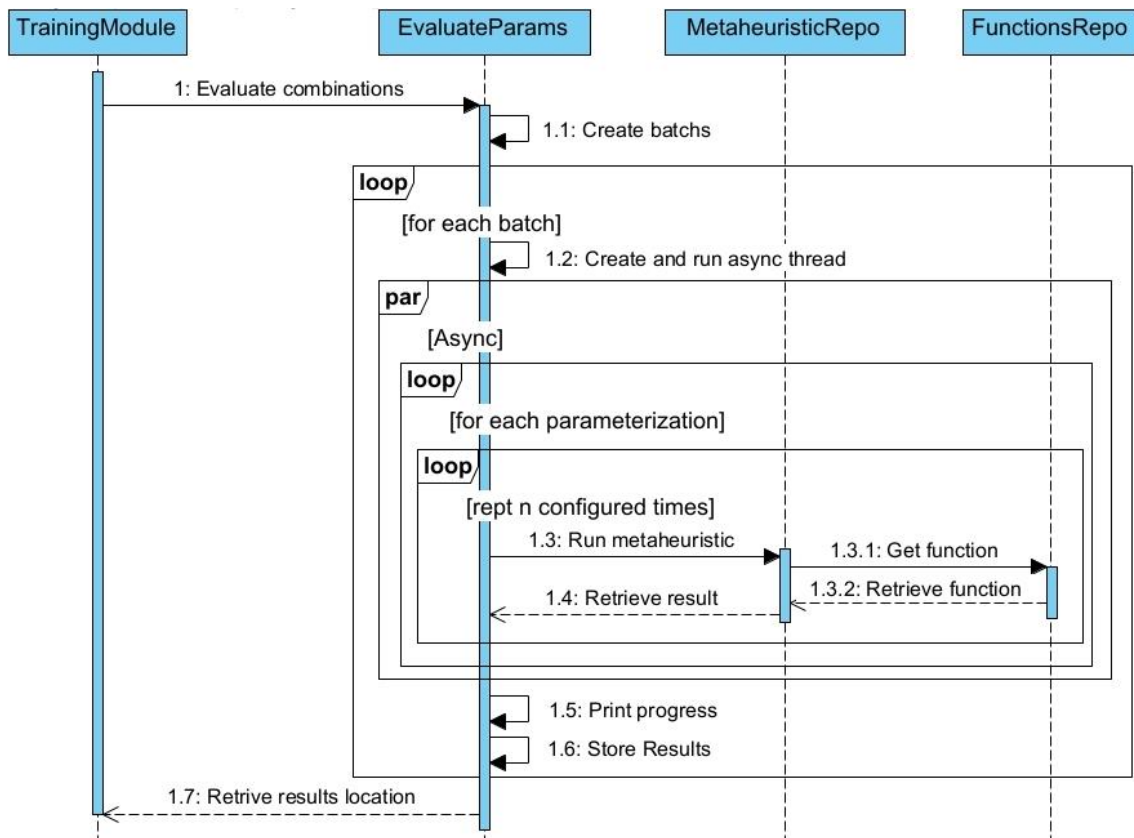


Figure 7 - Evaluate combinations sequence diagram

The list of parameter combinations, generated on the previous step, are grouped in batches, and each batch is executed where there is a thread available. The thread, using the metaheuristic repository, evaluates the configured number of repetitions of all the received combinations. After each asynchronous operation, the results are stored, and the progress is printed in the console. This training step, from the execution time perspective, is highly impacted by the number of combinations and by the number of repetitions.

On the next section, it is described how these results are processed.

4.2.3 Calculate best parameterizations by training mode

This component calculates and selects the best parameterization to the selected training mode, traditional or dynamic. These supported modes have different behaviors but are conditioned by the same system properties. On this component, it is possible to configure the number of parallel threads to be used, the number of parameterizations that should be reevaluated and the number of repetitions for each reevaluation. On this step the number of repetitions is a determinant factor to minimize the impact of the randomness associated to the optimization. For this reason, to get a good parameterization, it should not assume a small value.

Figure 8 presents the sequence diagram for the traditional static mode.

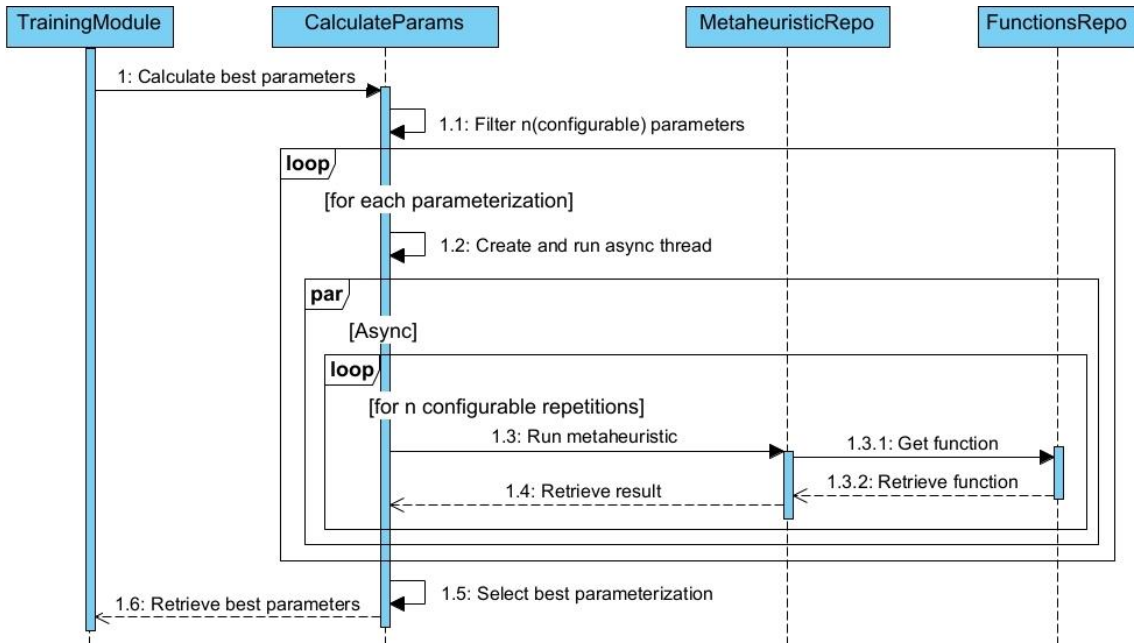


Figure 8 - Calculate parameterizations traditional mode sequence diagram

As shown by Figure 8, initially the parameterizations with best results on the preprocessing evaluation are filtered and a thread is created for each parameterization. On the parallel thread the parameterization is evaluated several times. After this, the best parameterization is chosen and retrieved. Figure 9 presents the sequence diagram for the dynamic mode training.

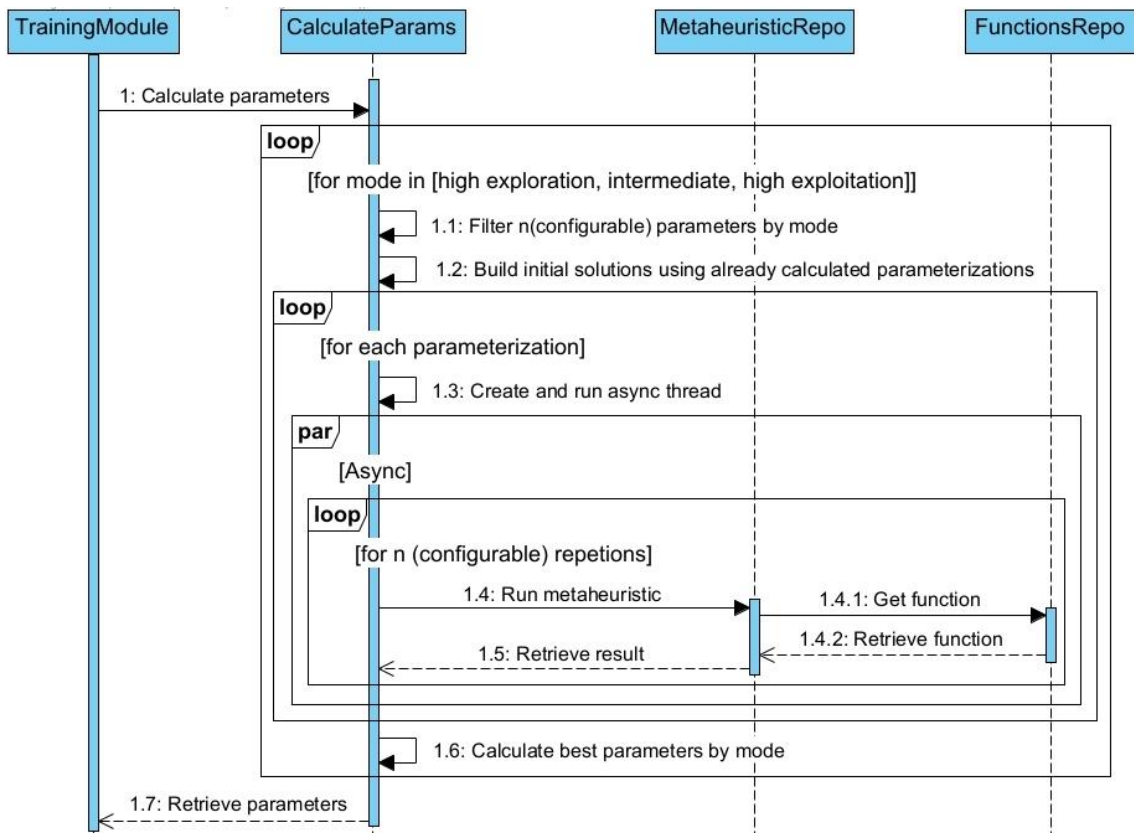


Figure 9 - Calculate parameterizations dynamic mode sequence diagram

The dynamic training process is similar to the traditional training, however three parameterizations are chosen. To do that, the parameterizations are grouped by the standard variation of all the solutions in three different groups. For each group a configurable number of options are filtered by best fitness value. These options are re-evaluated. For the intermediate and high exploitation execution, the initial solutions used on the training are calculated using the already calculated parameterizations. So, for the intermediate, using the high exploration configuration already calculated, the system generates a set of initial solutions to be used on the intermediate training. For the high exploitation, the same happens, but the initial solutions are generated calling the optimization service twice, one using the high exploration and other using the intermediate parameterization. This strategy allows to simulate the optimizing strategy in the Training module.

In the end the best parameterizations are retrieved to the Training main component in order to be stored by the *StoreParams* module described on the next section.

4.2.4 Store parameterizations on the file system

This component is responsible by storing the parameterizations obtained on the configured directory and to archive the processed files. Figure 10 presents an example of the directories created to store the information on the configured repository.

<ul style="list-style-type: none"> ▼ params 	Base repository directory
<ul style="list-style-type: none"> ▼ PyticleSwarm 	Metaheuristic identifier
<ul style="list-style-type: none"> ▼ dynamic 	Dynamic mode base directory
<ul style="list-style-type: none"> <ul style="list-style-type: none"> ▼ Schwefel50_V1_2197 	Training results folder. The folder name concatenates the function id, the version and the number of combinations
<ul style="list-style-type: none"> <ul style="list-style-type: none"> ▼ Schwefel50_V2_10830 	
<ul style="list-style-type: none"> <ul style="list-style-type: none"> <ul style="list-style-type: none"> high_exploitation.txt high_exploration.txt intermediate.txt 	Files containing the best parameterizations, taking in consideration the dynamic approach.
<ul style="list-style-type: none"> ▼ traditional 	Traditional mode base directory
<ul style="list-style-type: none"> <ul style="list-style-type: none"> ▼ Schwefel50_V1_2197 	Training result folder
<ul style="list-style-type: none"> <ul style="list-style-type: none"> <ul style="list-style-type: none"> best_configuration.txt 	File containing the best parameterization

Figure 10 - Structure of the parameters' repository

The parameterizations are stored separately by metaheuristic. It can be seen by Figure 10 that for each metaheuristic, two sub folders are created, identifying the mode used to generate the parameterizations. Inside these folders, a folder is created following the pattern:

- <function Id>_<number of executed trains by function>_<number of combinations used to train>

The first two attributes are important and mandatory because they are used to identify the parameterization. The last one is purely indicative. Inside each folder there are three text files for the dynamic training and one for the traditional. The file names are important and must match the example. Figure 11 shows an example of the content of the mentioned parameterization files.

```
{
  "c1min":0.77,
  "c1max":1.27,
  "c2min":1.0,
  "c2max":1.5,
  "wmin":0.4,
  "wmax":1.0,
  "n_particles":100,
  "n_iterations":50,
  "function_name":"Rastrigin65",
  "alg_name":"PyticleSwarm",
  "opt_mode":"minimization"
}
```

Figure 11 - Example of the parameterization files content

These files, as shown by Figure 11, contain a Python dictionary converted in json, concatenating the objective function, the algorithm id, the optimization mode used on train and the calculated parameters. This can be added manually, as long as the described standards are met.

4.3 Optimization module

The Optimization module is a problem- and metaheuristic- agnostic multi-agent system developed in Python that allows to maximize or minimize different objective functions. The algorithm reads the configurations from an externalized configurations file and executes the optimization using an external service. The system is developed using the Spade library and it is composed by three different agents.

- A Timer agent responsible for starting the optimization and notifying when the timeout is close
- A Coordinator agent responsible for coordinating the optimization
- An Executer agent responsible for calling the metaheuristic and process the results

The agents communicate using a XMPP server. On this project the used server is an instance of the ejabberd installed locally [50]. The chosen XMPP server is not relevant and can be replaced by any other similar application.

This module is configurable by system properties. These properties must be defined in an *optimization.properties* file located on the base directory. Table 5 contains all the system properties described and an example value for each property.

Table 5 - Optimization module system properties description

Property	Description	Example value
foptimizer.logs_base_dir	Directory where the application logs will be stored.	C:\tmp\ftoptimizer\logs
foptimizer.timer.agent	The XMPP identifier of the timer agent.	timerAgent@localhost
foptimizer.coordinator.agent	The XMPP identifier of the coordinator agent.	coordAgent@localhost
foptimizer.executer.agents	The XMPP of the executer agent.	execAgent@localhost
foptimizer.agents.password	The password to connect to the XMPP server.	password1
foptimizer.number.trials	Number of trials executed on the optimization.	4
foptimizer.archive.directory	Directory where the input files are moved after being processed.	C:\tmp\opt_archive
foptimizer.params.directory	Directory that contains the parameters to be used on the optimization.	C:\tmp\params
foptimizer.results.directory	Directory where the results of the optimization should be stored.	C:\tmp\opt_results
foptimizer.timer.agent.period	Time interval in seconds when the timer agent will check the optimization queue.	1

The critical system properties of this module are the *foptimizer.timer.agent.period* and the *foptimizer.number.trials*. The first has impacts on the execution time because it conditions the interval when the queue is verified. In real time environments this value should be small in order to avoid optimizations in the queue without being processed. The second impacts the number of times the objective function is evaluated.

In Figure 12 are presented two examples of the input files structure for the Optimization module. The input files must be text files that start with *exec_* and must be located on the configured optimization queue directory.

Example 1	Example 2
<code>metaheuristic_id:PyticleSwarm</code>	<code>metaheuristic_id:PyticleSwarm</code>
<code>function_id:Schwefel150</code>	<code>function_id:Schwefel150</code>
<code>function_parameters:Schwefel150</code>	<code>function_parameters:Schwefel150_V5</code>
<code>time_to_run:60</code>	<code>time_to_run:60</code>
<code>mode:minimization</code>	<code>mode:maximization</code>

Figure 12 - Optimization module input examples

As can be observed by Figure 12, the following attributes must be defined in the input file:

- *metaheuristic_id* – it is an optimization algorithm identifier that must be defined on the metaheuristic repository.
- *function_id* – it is the objective function identifier that must be known by the system.
- *function_parameters* – identifies the parameters that should be used on the optimization. On this attribute, the function used to define them does not need to match the function to be optimized. Apart from that, it can be configured with and without the version associated. If the version is not defined, the most recent version will be chosen, if the version is defined, the parameters of the specific version are used.
- *time_to_run* – it is a number in seconds, and it is used to condition the maximum time available to run the algorithm.
- *mode* – it can be minimization or maximization and it conditions the optimization execution.

In the following section, the agents and their behaviors are described.

4.3.1 Agents

The developed multi-agent system has a complex architecture with three different agents, the Timer agent, a Coordinator agent, and an Executer agent. The communication between these agents is direct, without using a facilitator, and the implemented communication strategy is the transmission of messages between agents. From the coordination point of view, the model follows the master-slave architecture. The timer agent request services from the coordinator and the coordinator requests services from the Executer.

Figure 13 presents the component diagram of the developed model. It contains the agents, the behaviors, and the initialization process.

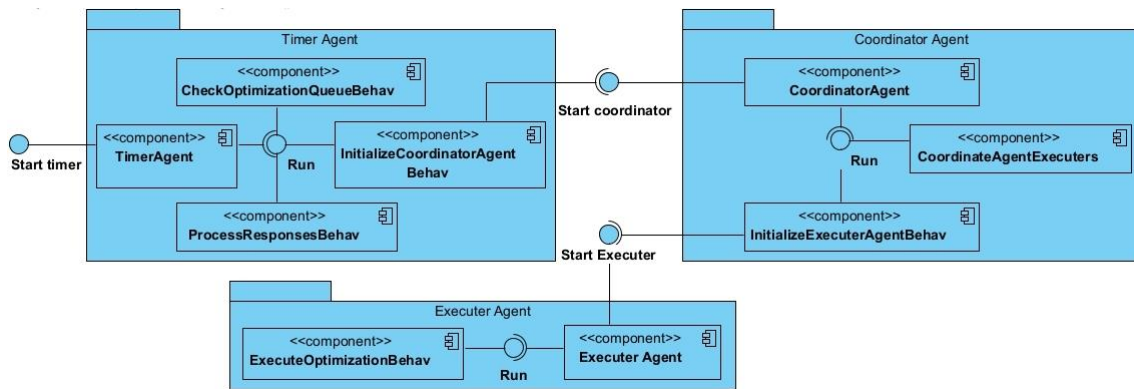


Figure 13 - Multi-agent system component diagram

As one can see by Figure 13, the Executer is initialized by the Coordinator and the Coordinator is initialized by the Timer.

The Timer agent is a cognitive agent capable of initializing the Coordinator, reading the queue, starting the optimization, managing the execution time, and requesting a faster solution if the time to run is being exceeded. To do that, the agent is composed by 3 behaviors. A one-shot behavior, responsible by starting the coordinator (*InitializeCoordinatorAgentBehav*), a period behavior, responsible by checking periodically the training queue (*CheckOptiomizationQueueBehav*) and a cyclic behavior, responsible by processing the received messages (*ProcessResponsesBehav*).

The Coordinator is a cognitive agent capable of initializing the Executer, calculating the best parameterizations for each execution, requesting an optimization, storing and managing the best responses and sending the optimization result to the Timer agent. To do that, 2 behaviors were implemented. A one-shot behavior, responsible by starting the Executer (*InitializeExecuterAgentBehav*) and a cyclic behavior responsible by processing the received messages (*CoordinatorAgentExecuters*).

The Executer agent is a reactive agent capable of receiving a request to execute the necessary optimization, processing the optimization response, and sending back the answer. This agent is composed by only one cyclic behavior, which is executed every time a request is received by the agent (*ExeciteOptimiziationBehav*).

The following section contains the flow of the developed multi-agent model.

4.3.2 Multi-agent system

Besides the agents, the system requires a custom implementation of the *MetaheuristicRepo* and *FunctionRepo* components. These two modules are described in detail on the section 4.1. In Figure 14 is presented the sequence diagram of the optimization process using the developed model.

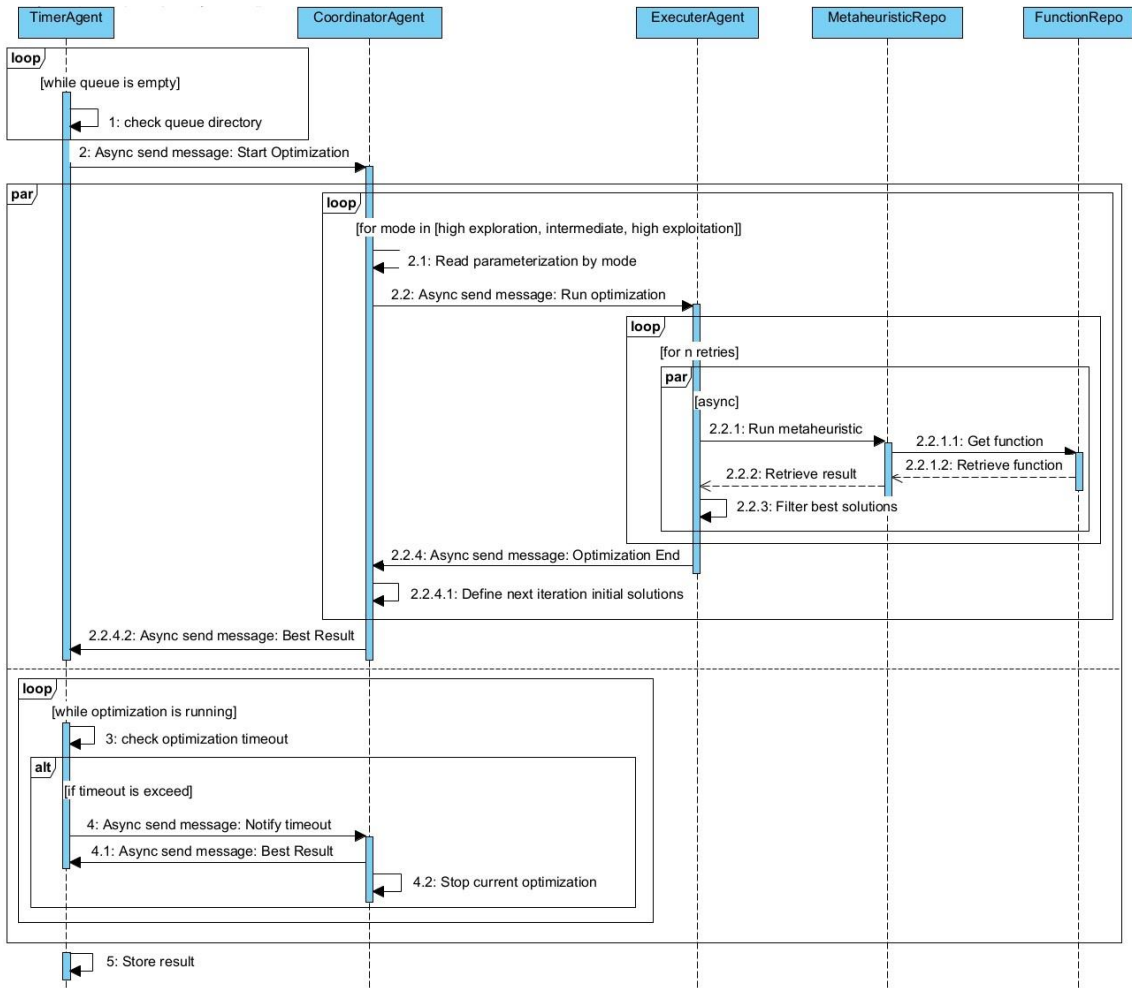


Figure 14 – Multi-agent system sequence diagram

From Figure 14 it can be seen that, after being initialized, the Timer agent periodically checks the optimization queue until a new optimization request is registered. Once a request arrives, the input file is processed, and an asynchrony message is sent to the Coordinator. In the meantime, the Timer starts an iterative process of checking the elapsed time since the optimization started. If the time to run is close to being exceeded, a new message is sent requesting a valid result. In the end, after receiving the result, this agent stores it on the configured output directory.

Regarding the Coordinator agent, after receiving a message to start, it reads the metaheuristic parameterization and sends a message to the Executer indicating the required data to run the algorithm, parameterization, the number of trials, the function identifier, the metaheuristic identifier, the execution mode, and the initial solutions. Afterwards, the agent waits for the executer response. When the response is received, the agent calculates the next iteration initial solutions, reads the parameterization of the next iteration, and sends another message to the Executer agent so that it may start the new execution using the new parameter set. Generally, the loop ends after the end of the high exploitation execution. However, if the agent receives a timeout notifying message, it retrieves the current best response and stops the execution at the end of the next iteration.

The Executer agent reacts based on the received messages. When it receives a message, it performs the optimization using the metaheuristic and functions repositories, and it sends the response back. During the optimization a configured number of trials received on the request is executed. These are executed using asynchronous threads. Each thread executes only one trial, and the parallel processing includes the optimization and the processing of the optimization results, described in section 4.3.4. In the end, the results are aggregated, and the response is sent back to the Coordinator agent.

The following section describes the messages used by the system to support the communications.

4.3.3 Multi-agent system communication

As mentioned on the previous sections, the developed multi-agent system uses FIPA ACL (Agent Communication Language) messages to communicate. The body of all the messages is a Python dictionary converted in a valid json structure. The supported messages are:

- **Start optimization** from **Timer** to **Coordinator**. This has the **inform** performative code and it indicates to the Coordinator that an optimization must be supported.
- **Notify timeout** from **Timer** to **Coordinator**. This has the **request** performative code, and it means the timeout is being exceeded and a faster response is needed.
- **Run optimization** from **Coordinator** to **Executer**. This has the **request** performative code, requesting for an optimization execution.
- **Optimization result** from **Coordinator** to **Timer**. This has the **inform** performative code and it informs of the optimization result.
- **Execution result** from **Executer** to **Coordinator**. This has the **inform** performative code, and it contains the optimization execution result.

4.3.4 Metaheuristics response processing

In order to improve the optimization results, as mentioned before, after each optimization the three best solutions are selected. This is important since for each optimization the system executes several trials. So, if a specific trial does not retrieve good results, it can be completely excluded, and it is not processed on the following iterations. After being selected, the best solutions of all the trials are analyzed and the most appropriate are used as the initial solutions of the next iteration.

Figure 15 shows a diagram explaining the first step of the mentioned process. The example is purely exemplificative, and it is not a result of an optimization.

Process Best solutions - Minimization

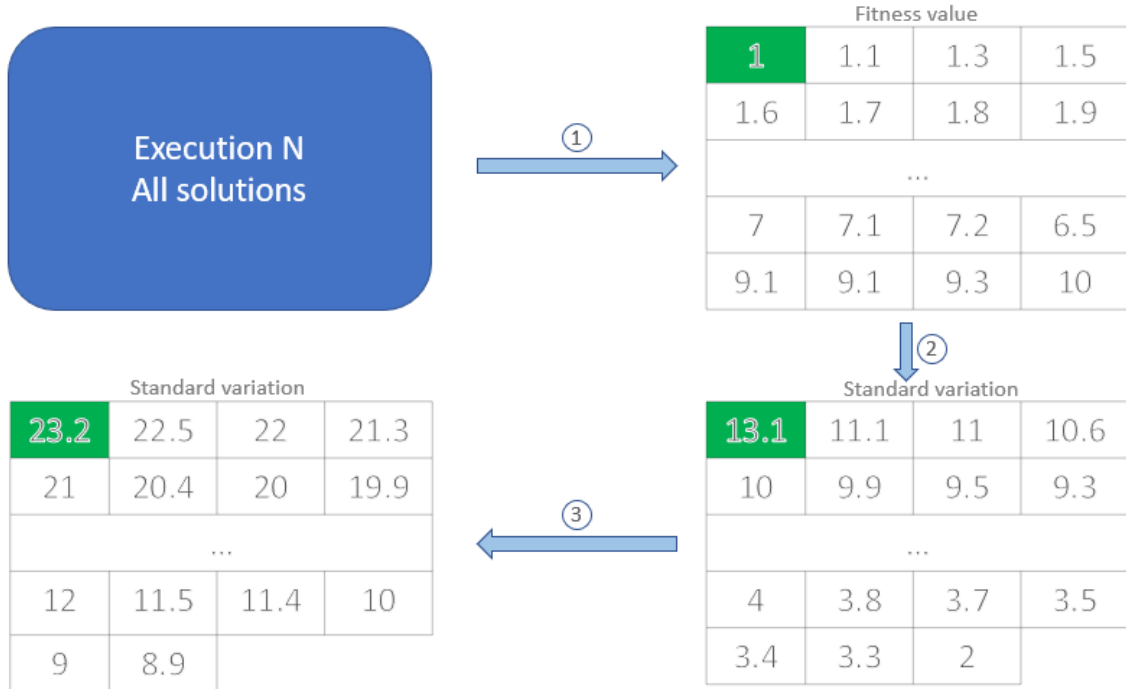


Figure 15 - Example of the process of selecting the best solutions after a minimization

The best responses are processed in three steps, as shown by Figure 15. Initially the best 50 solutions obtained are selected, sorted by fitness values and the best one is selected. The sort can be descending or ascending, depending on the optimization mode: maximization or minimization. On the second step, it is calculated the variation of each objective function result, calculated between the selected best solution and the others. The solutions are sorted by the calculated value and the solution with highest variation is chosen. This is performed so that the three selected solutions, to be used as initial solutions for the next execution, are sufficiently spread along the solution space and are not focused on the same search point. On the third step, the processing is similar; however, the variation is calculated using the two already selected solutions. In this way, a third, distinct, solution is identified.

These selected solutions are combined with the solutions of the other trials and enable starting the process that filters the most appropriate solutions. This process is conditioned by the current step of the optimization. In Figure 16 is explained how the initial solutions are selected for the intermediate configuration when the number of trials configured is 4.

Select initial solutions - Minimization

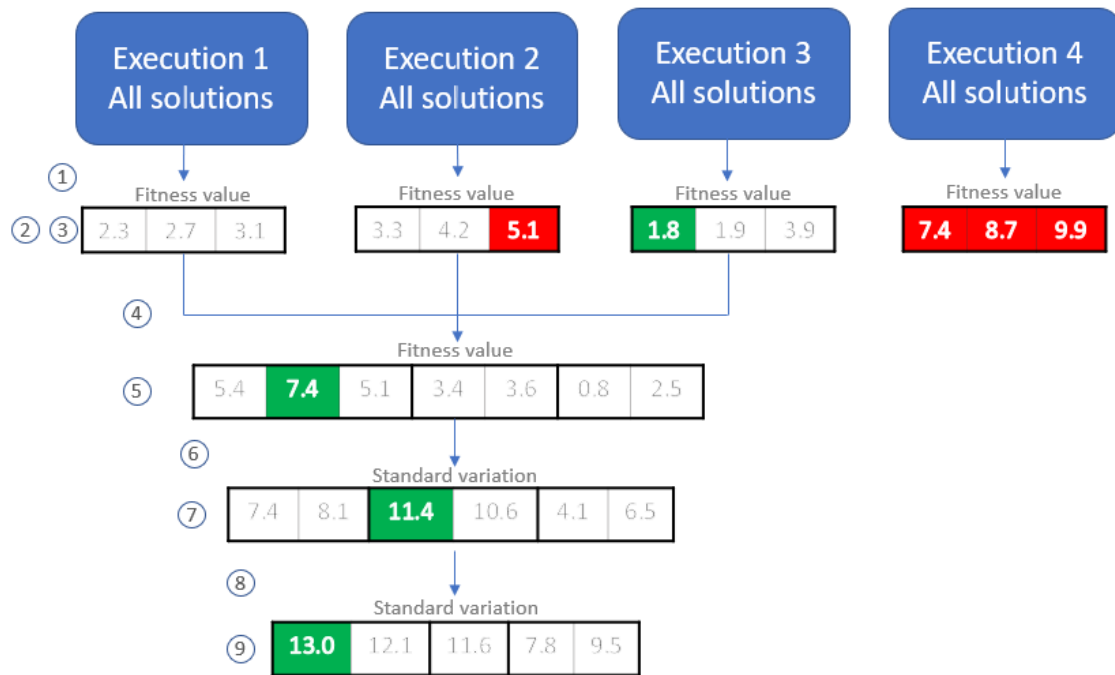


Figure 16 - Example of the process of select the initial solution in a minimization

The process, as depicted by Figure 16, is executed by the following steps:

1. Process the three best solution for each trial. Process described in Figure 15.
2. Exclude the worst 1/3 solutions. The highest values for minimizations and the smallest values for maximizations.
3. Sort by fitness value and select the smaller value in minimizations and the highest value in maximizations.
4. Calculate the variation between the selected solution and the others.
5. Select the solution with the highest variation. It means that, theoretically, it is, among the best solutions, the solution farthest from the already selected.
6. Calculate the variation between the two selected solutions and the others.
7. Select the solution with highest variation.
8. Calculate the variation between the three selected solutions and the others.
9. Select the solution with highest variation.

Following this approach, it is possible to ensure that the best solution is always considered on the next iteration. Besides that, if an optimization retrieves bad results, it can be excluded, and it is not taken into consideration in the second step. At the same time, selecting by variation, instead of fitness value, the chances of selecting two similar initial solutions are reduced significantly.

For the high exploration parameterization, the initial solution is not defined and for the high exploitation, the process is similar to the intermediate parameterization. However, taking into consideration that the focus of the high exploitation optimization is the local search, instead of

selecting the initial solutions by the variation, the list is sorted by fitness value, and the best solutions are selected.

5 Results and discussion

In this section, the proposed solution is tested and validated by being compared with the traditional approach for executing metaheuristic models. The analysis includes the training execution time, the optimizations results based on the number of combinations used to train, the analysis of the performance of the Optimization module, the impact of the number of trials, and a comparison between the results obtained using different training functions. The test and validation process includes the optimization of multiple benchmark functions, using the Benchmark Python library and also the experimentation using a real-world problem in the power and energy domain. PSO is used as the basis metaheuristic for the performed experiments, using the Pyticle Swarm Python library, but a GA, using the Genetic Algorithm Python library is also experimented as means to validate the applicability of the proposed solution to different metaheuristic algorithms.

5.1 Impact of the configured thread number in the training execution time

The Training module developed on this project supports parallel processing. In order to evaluate the impact of the configured thread number on the execution time, the module was tested using two different processors, the AMD Ryzen 7 4700 (8 threads) and the AMD Ryzen 7 5700 (16 threads). Figure 17 shows the analysis for the processor AMD Ryzen 7 4700 and Figure 18 shows the analysis for the processor AMD Ryzen 7 5700, both considering the training process for optimizing two benchmark functions, namely Rana(50) and Schewefel(50).

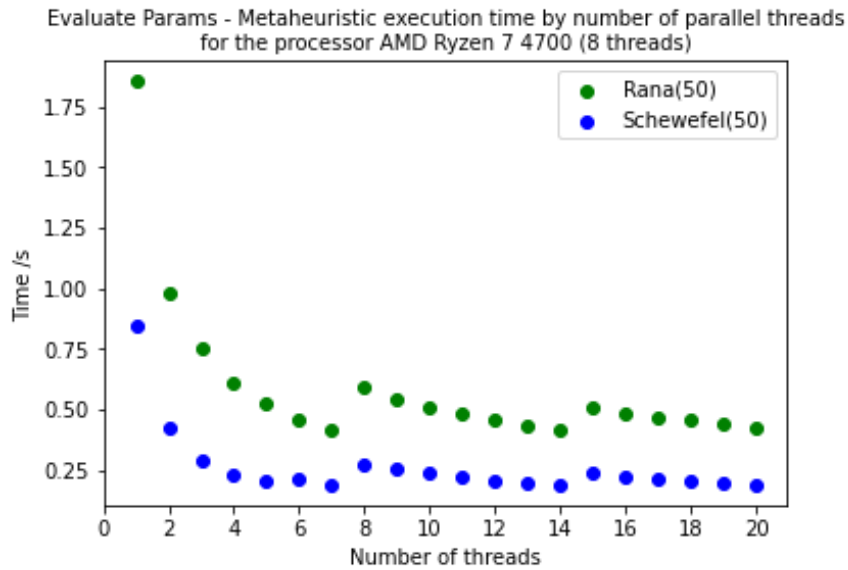


Figure 17 - Metaheuristic execution time by number of configured threads AMD Ryzen 7 4700

As shown by Figure 17, for processor AMD Ryzen 7 4700, with parallel processing, the optimization execution time was reduced significantly when compared with the execution without multi-processing. The reduction can be considered exponential until it reaches the minimum value, when the number of threads configured is 7. Besides the expected differences on the execution time, the same pattern was observed on both functions used on this test.

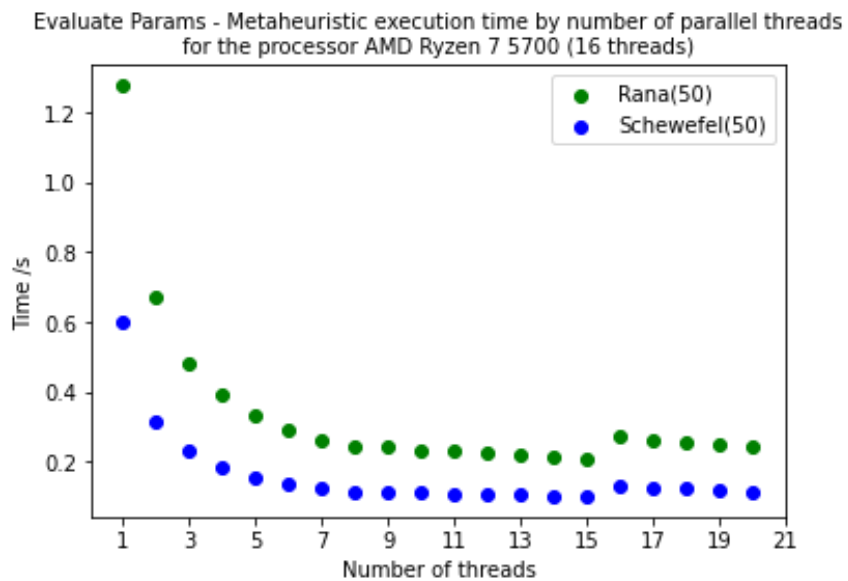


Figure 18 - Metaheuristic execution time by number of configured threads AMD Ryzen 7 5700

For the AMD Ryzen 7 5700 the behavior is similar, as shown by Figure 18. However, the minimum time was obtained when the number of threads configured was 15. This can be explained by the architecture differences between these processors. The first supports 8 threads running at the same time and the second one 16.

Therefore, the number of threads significantly impacts the training execution time. The number of threads should be smaller than the number of multi-threading supported by the used processor.

In the following tests the AMD Ryzen 7 4700 processor was used with 4 parallel threads. This value was chosen since the time difference is not significant and using only 4 threads, the execution does not impact the other processes running on the machine.

5.2 Impact of the configured batch size in the training execution time

The batch size is a system property that can be configurable. It conditions the number of combinations evaluated for each thread. Figure 19 presents the analysis of the impact of this system property on the metaheuristic execution time for the Rana (50) and for the Schewefel(50) objective functions.

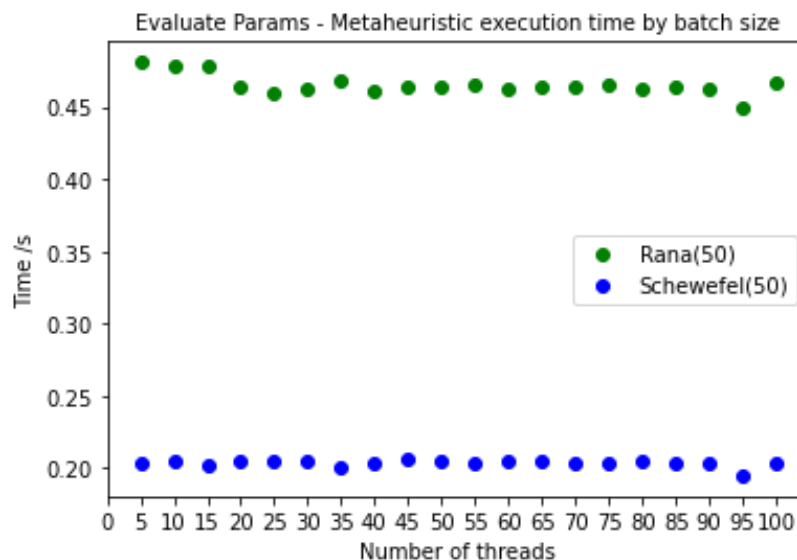


Figure 19 - Metaheuristic execution type by the batch size

As can be observed by Figure 19, the size of the batch does not impact the execution time. Nevertheless, it is still useful as it conditions how often the progress is printed in the console.

5.3 Impact of the training combinations in the training execution time

The Training module is composed by 4 components. The *GenerateParams* that generates the combinations to process, the *EvaluateParams* that evaluates all the generated combinations, the *CalculateParams* that processes the evaluations executed on the previous module and calculates the best parametrization for the traditional mode and three parameterizations for the dynamic mode, and the *StoreParams* that stores the calculated parameters. From these

components, only the *EvaluateParams* and the *CalculatedParams* have real impacts on the training execution time.

The training execution time is highly influenced by the number of combinations that are generated to be processed. Figure 20 presents the comparison analysis between the training execution times for the *Schewefel(50)* function for the two execution modes.

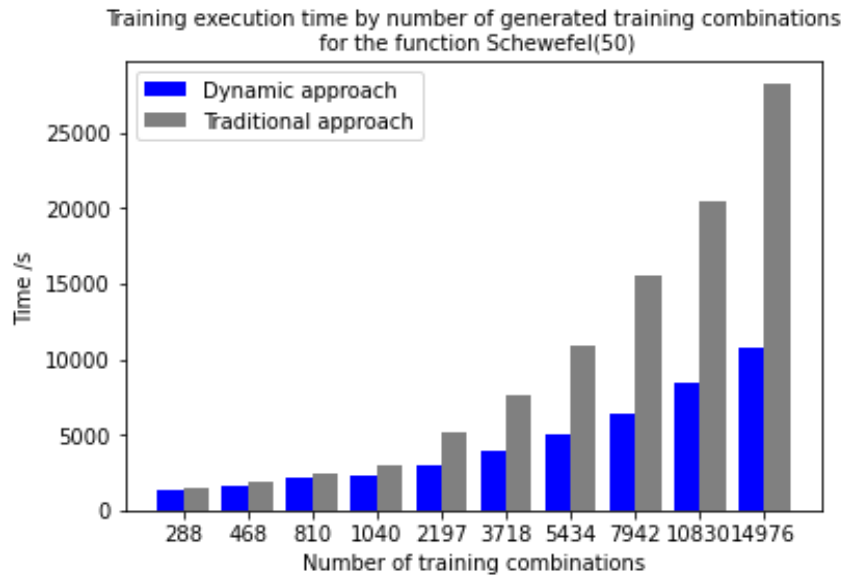


Figure 20 - Impact of the number of combinations in the training execution time

Based on the obtained results, shown by Figure 20, by increasing the combinations number, the traditional mode execution time increases greatly comparing to the dynamic mode. It occurs since running the *EvaluateParams* in the traditional mode, the objective function is evaluated three times more. It occurs because, in this step, all the combinations are evaluated the same number of repetitions and two training execution are only considered equivalents when the number of executions configured for the traditional is three times bigger.

When the number of combinations is lower, the difference is not significant, as it is related with the *CalculateParams* component. For the dynamic approach, in this component, three different parameterizations must be calculated instead of one. So, the *EvaluateParams* is faster on the dynamic mode and the *CalculateParams* is faster on the traditional. However, the execution time of the second component is not impacted by the number of combinations.

5.4 Impact of the training combinations number in the optimization results

Another important aspect in the optimization of metaheuristic parameters is the optimization results. Theoretically, the increase of evaluated combinations of parameters improves the results obtained. In Figure 21 is shown the comparison between the number of combinations used to train and the results obtained on the minimization of the *Schwefel(50)* function. The parameterizations used on this analysis were generated using the same function.

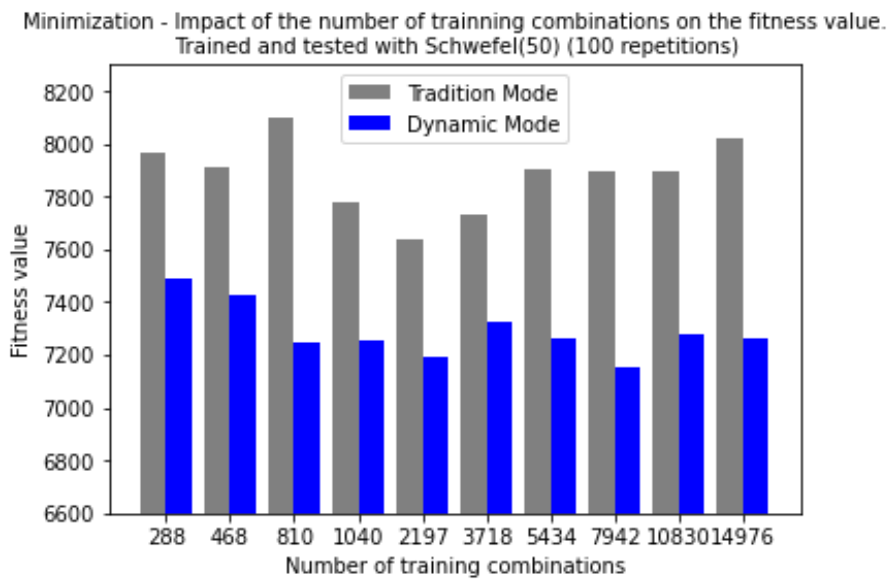


Figure 21 - Impact of the number of training combinations on fitness value. Trained and tested with Schwefel(50)

Contrarily to the formulated hypothesis, as can be seen by Figure 21, the obtained results do not improve consistently with the increase of the number of tested parameter combinations. For the dynamic mode, the objective function has a decreasing trend with the increase of combinations. However, for the number of 3718 combinations, the value is considerably worst in comparison to the value obtained with 2197 combinations. For the traditional mode, the best result is obtained with 2197 combinations and with the increasing of the number of parameterizations considered the results become substantially worse.

This unexpected behavior can be explained by the combination's generation process and the small number of parameterizations considered on the training, taking into consideration the metaheuristic complexity. As explained on section 4.2.1 the combinations are generated based on an input file. This input file must contain, for each parameter, the maximum and minimum values, and the number of different values that must be considered for the parameter. For example, if the parameter x must be between 0 and 2, for the factor 5, the list of values considered is (0, 0.5, 1, 1.5, 2). On the other hand, for the factor 6, the list of values considered is (0, 0.4, 0.8, 1.2, 1.6, 2). As can be observed, for this scenario, only the extremes are in both lists, and there is no guarantee that the values in the longer list are better and will produce better results.

Figure 22 presents a similar analysis; however, in this case the training is executed with the *Schwefel(50)* function, but the tests are run using the *Rastrigin(50)* function.

Minimization - Impact of the number of training combinations on the fitness value.
Trained with Schwefel(50) and tested with Rastrigin(50) (100 repetitions)

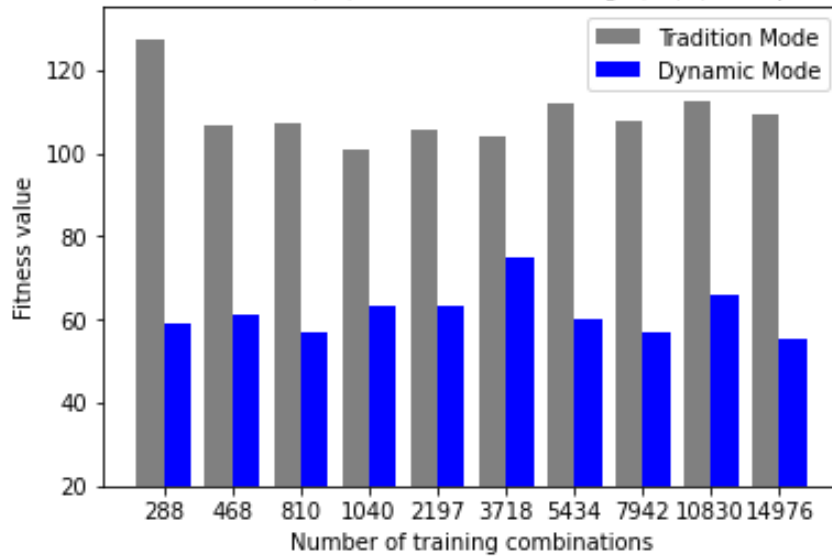


Figure 22 - Impact of the number of training combinations on the fitness value. Trained with Schwefel(50) and tested with Rastrigin(50)

For this second test, displayed in Figure 22, the same inconsistency is observed, for both modes. There is no pattern that represents the evolution of the results improvement by number of combinations used to train.

Another conclusion achieved from these experiments is that the proposed solution gets better results than the traditional solution. The difference is greater and more noticeable for the second scenario, where the function used to train is different than the function used to test, which means that the proposed dynamic Training model enables reaching a parameterization set that is more suitable to deal with multiple problems, and not as problem specific as the one achieved with the traditional mode.

5.5 Optimization module – execution time analysis

As described in section 4.3, the Optimization module is a multi-agent system composed by three different modules that optimize in parallel a configured number of trials. This developed model incorporates in the optimization additional processing. Namely communication between agents, thread management and handling optimization responses. The additional processing execution time, in real time environments, cannot be significant. In Table 6 is presented a performance analysis, in milliseconds, of the proposed Optimization module by number of optimization trials. In Table 7 is presented a summary containing the most relevant times of the mentioned analysis, in percentage. The processor used in these tests is the AMD Ryzen 7 4700 (8 threads).

Table 6 - Optimization module execution time (milleseconds) analysis.

Trial Number	1	2	3	4	5	6	7	8	9	10
Optimization execution time	1377	1367	1402	1434	1467	1565	1622	1643	1581	1630
Process response time	227	234	222	230	236	242	256	262	256	261
Parallel processing time	1606	1636	1681	1725	1806	1915	1978	3898	3894	4095
Threading management time	2	35	57	61	103	108	100	1993	2057	2204
Total Time	1672	1688	1728	1788	1869	1978	2057	3945	3958	4156
Algorithm additional time	66	52	47	63	63	63	79	47	64	61
Difference to 1 trial time	0	16	56	116	197	308	385	2373	2386	2484

Table 7 - Optimization module time analysis summary

Trial Number	1	2	3	4	5	6	7	8	9	10
Algorithm additional time (%)	3.9	3.1	2.7	3.5	3.4	3.2	3.8	1.2	1.6	1.5
Threading management time (%)	0.1	2.0	3.3	3.4	5.5	5.5	4.9	50.5	52.0	53.1
Process response (%)	13.6	13.9	12.9	12.8	12.6	12.2	12.4	6.6	6.5	6.2
Optimization time (%)	82.4	81.0	81.1	80.3	78.5	79.1	78.9	41.6	39.9	39.2

As once can see by Table 6 and Table 7, the additional algorithm time (agents communication, read and store files, etc.) is not impacted by the number of trials and it is not significant. In percentage, the highest value is less than 4% and in absolute numbers, it is stable, always below 80 milliseconds.

Thread management time is the time it takes to initialize the optimizing threads. This time is highly impacted by the number of trials configured. For 1 trial the number is close to 0, the time goes up slightly until reaching 8 trials, in which it has a sharp rise representing 50% of algorithm's execution time. It can be explained by the characteristics of the used processor; it supports only 8 threads running at the same time. So, for this processor the maximum number of trials that should be used is 7. However, this analysis is only valid for this processor, being required a similar analysis is using the proposed model in different processors.

The process response time is not impacted by the number of trials, and it is always below than 250 milliseconds. The optimization time, time spent running the optimization algorithm, excluding when number of trials is higher than 7, represents about 80% of the total time.

So, using the algorithm it is possible to automatize the process of running a metaheuristic. This algorithm includes additional processing that for a reasonable number of trials (below the number of threads supported by the processor) does not significantly impact the optimization time.

5.6 Impact of the number of trials on the optimization result

As concluded in the previous section, the number of configured trials, depending on the processor used to run the algorithm, can have impacts on the optimization execution time. However, it is not the only aspect that must be taken into consideration. The execution time analysis must be associated with the obtained results. In theory, by increasing the number of trials, the optimization results will be better.

Figure 23 shows the comparison between the fitness values obtained for the traditional mode, for the dynamic mode considering only the best solution, and the proposed model. These values are obtained using parameterizations generated with the Training module.

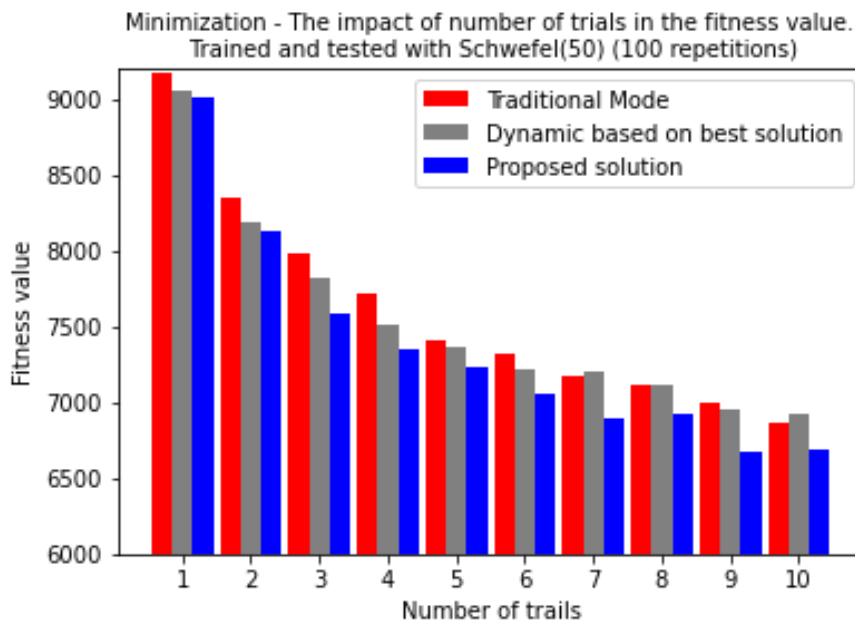


Figure 23 - Impact of number of trials in the optimization results. Trained and tested with Schwefel(50)

As can be observed by Figure 23, the obtained results support the initial theory, with the increase of the number of trials, the optimizations results are improved. These improvements are more significant when the number of trials is low. It occurs because in complex optimizations, there is a considerable randomness in the results.

Comparing the three optimization modes, the proposed solution always achieves better results. Considering the solution that uses only the best solution after each iteration, for a low number of trials, the results are close to the proposed solution; when the number of trials increases, the difference increases and ends up getting closer to the traditional solution.

Figure 24 shows a comparison using the same parameterizations, reached after training with Schwefel(50) function; however, the function used to test is the *Rastrigin(50)*.

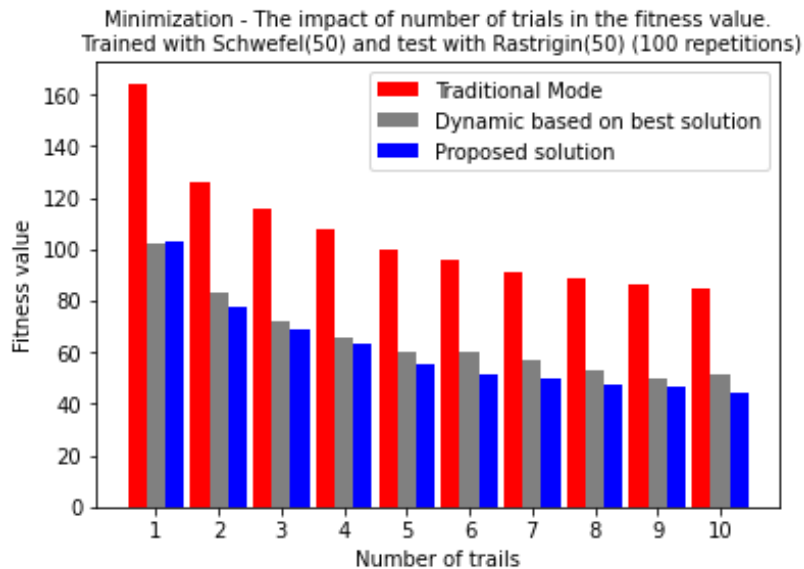


Figure 24 - Impact of number of trials in the optimization results. Trained with schwefel(50) and tested with Rastrigin(50)

For this test, as shown by Figure 24 the same can be observed: the best results are obtained using the proposed solution. However, the difference between the traditional solution and the others is much higher if compared with the previous example.

It occurs because on the traditional mode, several combinations are tested, and the best is chosen. This choice is made only and exclusively considering the fitness value obtained. So, the probability of overspecialization is high. Using the proposed solution, the parameterizations are not chosen taking into consideration only the objective function fitness value. First, the choice is made focused on the proposed approach, which consists of reducing the exploration as the number of iterations increase. So, regardless of the objective function to be optimized, in the first iteration the variability of the solution is higher, in the second it will decrease, and in final iteration the focus will be the local search.

To understand better the impacts of the proposed approach on the metaheuristic parameters generalization, on the next section the *Schwefel(50)* and the *Rastrigin(50)* are tested with parameterizations obtained using several benchmark functions.

5.7 Impact of the training function on the optimizations results

In some cases, it is not possible to train the metaheuristic with the objective function that must be optimized, e.g. if the function/problem is new and unknown, and requires a fast response time. For these scenarios a parameterization optimized using different training functions must be used. For this reason, an important factor on the metaheuristic parameters selection is the generalization.

Figure 25 one can see the comparison between the results obtained for the *Schwefel(50)* function using different parameterizations obtained by training with different objective functions. Figure 26 shows the same comparison, but the optimized function is the *Rastrigin(50)*.

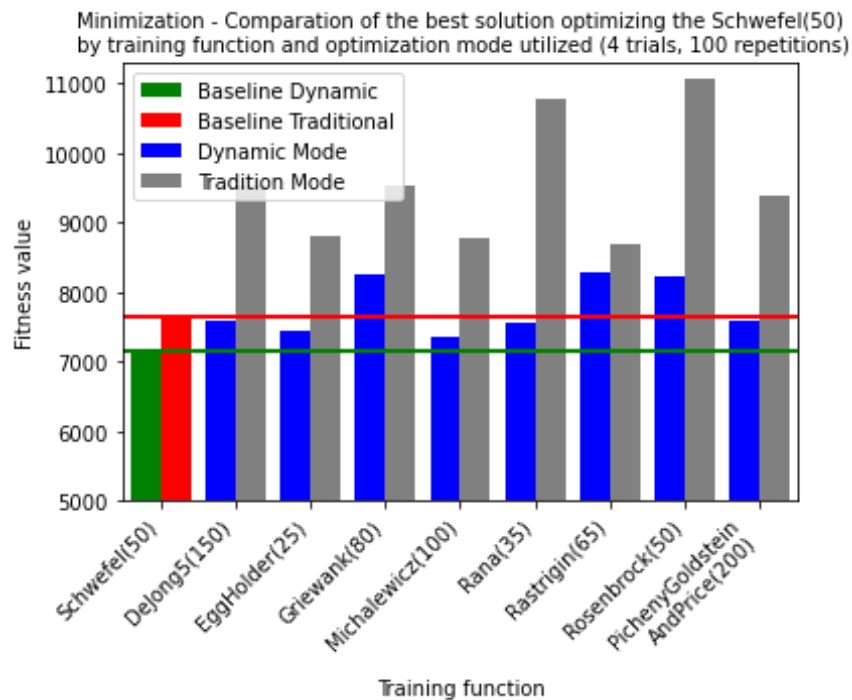


Figure 25 - Impact of training function on the optimization results of the *Schwefel(50)*

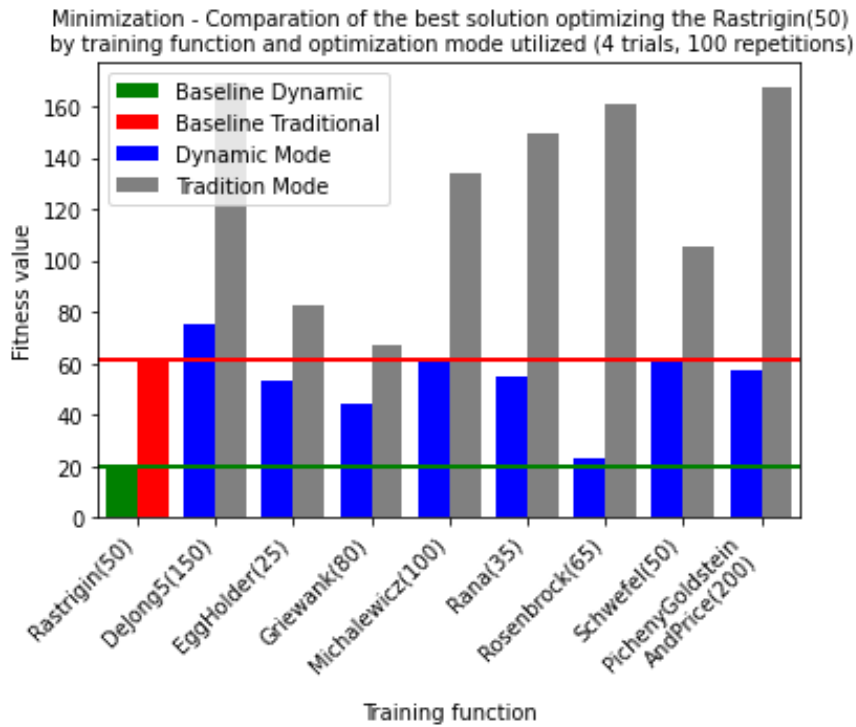


Figure 26 - Impact of training function on the optimization results of the *Rastrigin(50)*

The results from Figure 25 and Figure 26 are very similar. In both situations, as expected, the best result is obtained using the proposed solution with the parameterization obtained using the same function in the training process. When applying the same parameterization reached by the proposed model to optimize other functions that have not been used in the training phase, the results are very close to the dynamic baseline and some of them are better than the traditional baseline. The results obtained with traditional approach are worse than when using the dynamic mode. When comparing with the global results, some high values can be observed.

So, with the proposed approach, the results are better, and the variation between the results with each different parameterization is smaller than the results obtained with the traditional approach. Therefore, besides enabling reaching better results for a specific objective function, this approach allows the increase of the generalizability of the parameterizations.

5.8 Proposed solution applied to a real-world problem

In order to validate the suitability of the proposed model in dealing with real-world problems, a case study considering an existing problem in the power and energy systems' domain is presented. Energy production from renewable sources is characterized by its intermittent nature [52]. When it comes to its distribution this can prove to be a big challenge. Consequently, energy distribution based on renewable sources requires new solutions, capable to deal with these characteristics. The smart grid concept is pointed out as one of the most suitable solutions to facilitate the participation of small players in electric power negotiations while improving energy efficiency [53]. Smart grid considers the management of local generation, loads and storage systems to be independent of the main system

Therefore, since market players as well as regulators have an interest in predicting market behavior, it is crucial that they understand the market and learn how to evaluate their investments in such a competitive environment. And simultaneously to take suitable decisions about how and to participate in each market type [3].

Consequently, Pinto *et al.* [3] developed a part of an ongoing work that proposes a portfolio optimization methodology that analyses different market opportunities and provides the best investment profile for an electricity market player. In summary, this problem considers as input the forecasted market prices for different market opportunities, e.g. day-ahead market, intraday market sessions, bilateral contracts negotiation, local market trading, and given the total amount of generation or consumption of a certain player, optimizes the volume to be transacted in each of these markets in each (defined) transaction period in order to minimize the purchasing cost or maximize the sale profit.

This model was used to validate the proposed solution. To validate it, two training executions with 2197 combinations, one for the traditional and the other for the dynamic modes, were executed. The fine tuned parameters are evaluated and the comparison between the results obtained by number of trials is presented in Figure 27.

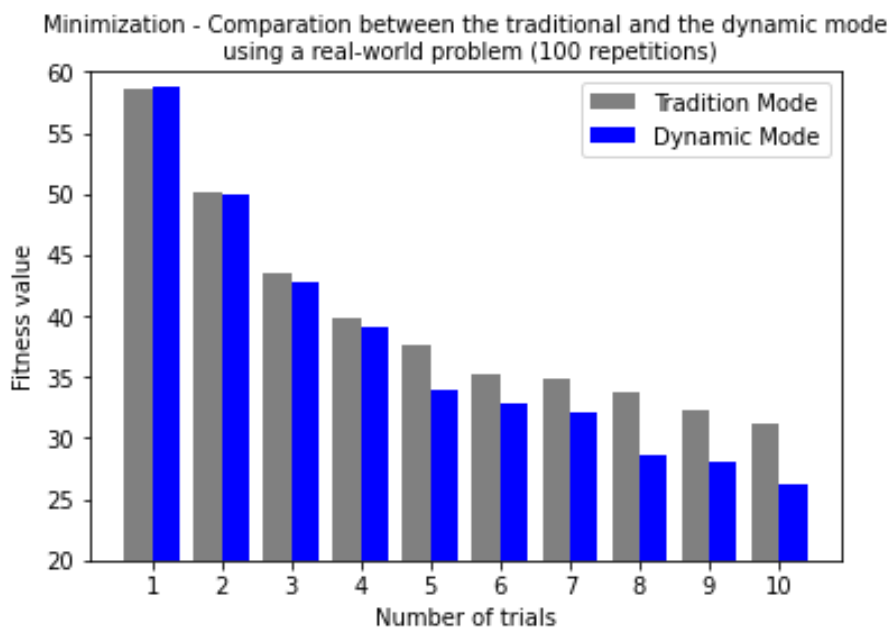


Figure 27 - Impact of the trials number on the optimization results using a real-word problem

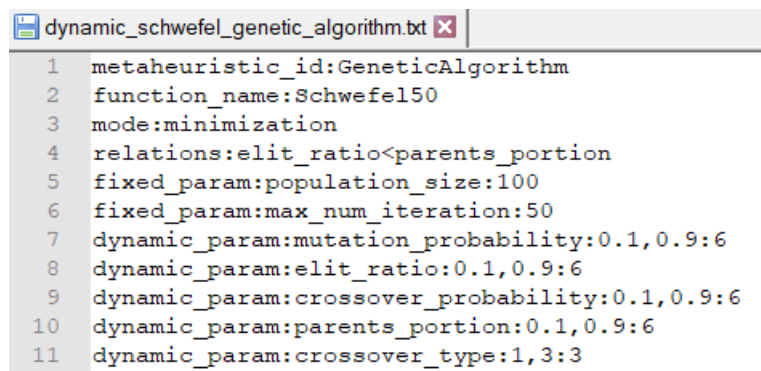
As one can see from Figure 27, the results are similar to the results obtained using the benchmark functions. For a low number of trials, the results obtained are very similar. However, with the increase of number of trials, the results using the dynamic mode increases considerably when compared with the same test using the traditional mode. It occurs because with the proposed solution, using several trials, the bad results obtained on the first iterations are not considered in the following executions.

5.9 Proposed solution using a genetic metaheuristic

In order to validate the capability of the proposed solution in dealing with multiple metaheuristic algorithms, a Python open-source library Genetic Algorithm was integrated with the proposed solution. The integration process includes the configuration for the training process, the customization and the evaluation of the obtained results. The results were also compared with the results obtained using PSO, with the Pyticle Swarm library.

5.9.1 Genetic Algorithm training configuration

The metaheuristics are not standardized. For this reason, to train a metaheuristic, the parameters and their range must be configured. More details regarding the mentioned configuration can be consulted in section 4.2.1. Figure 28 presents an example of a possible configuration for the Genetic Algorithm library.



```
dynamic_schwefel_genetic_algorithm.txt
1 metaheuristic_id:GeneticAlgorithm
2 function_name:Schwefel150
3 mode:minimization
4 relations:elit_ratio<parents_portion
5 fixed_param:population_size:100
6 fixed_param:max_num_iteration:50
7 dynamic_param:mutation_probability:0.1,0.9:6
8 dynamic_param:elit_ratio:0.1,0.9:6
9 dynamic_param:crossover_probability:0.1,0.9:6
10 dynamic_param:parents_portion:0.1,0.9:6
11 dynamic_param:crossover_type:1,3:3
```

Figure 28 - Genetic Algorithm training dynamic configuration

As Figure 28 shows, the metaheuristic, in this configuration, is identified in the system by the id *GeneticAlgorithm*, there are two fixed parameters that condition the number of times the objective function will be executed, five dynamic parameters and one relation, the *elit_ratio* must be smaller than the *parents_portion*. Using this configuration, 1620 different parameterizations are evaluated.

The following section explains the code implemented on the customization layer in order to support the mentioned metaheuristic.

5.9.2 Genetic Algorithm customization

Besides the training configuration file, another important step that must be implemented for each supported metaheuristic is the customization layer. As described in section 4.1, for this layer, it is only standardized the input parameters and the expected output. So, inside it the code that must be introduced is strongly dependent on the algorithm characteristics.

To support the referred metaheuristic, several functions were implemented. In the Code snippet 3 is the *parse_crossover_type* function.


```

# Parse crossover type.It is considered numeric by the training
# Combinations generator
def parse_crossover_type(crossover_int):
    if int(crossover_int) == 1:
        return "uniform"
    if int(crossover_int) == 2:
        return "one_point"
    if int(crossover_int) == 3:
        return "two_point"
    raise ValueError("Crossover type is not supported!")

```

Code snippet 3 – Support genetic algorithm metaheuristic – Parse crossover

This function is required due a limitation on the generate combinations module. In the current version, this module is not able to process string values. So, to overcome the referred limitation, the Generate Combinations component considers the crossover an integer value between 1 and 3. On this function, the integer is converted to a string value that can be processed by the optimization algorithm.

Generally, the objective function variables are conditioned by a maximum and a minimum value. To support this feature, the metaheuristic must receive the number of variables and a vector containing the maximum and minimum values for each variable. The Code snippet 4 contains the function *get_varbound_and_dimensions* implemented.

```

def get_varbound_and_dimensions(func_id, param_index):
    #get function from functions repository
    function = get_function(func_id)
    # get dimension number
    n_vars = function.n_dimensions()
    m = function.suggested_bounds()
    # create vector using suggested bounds
    varbound = np.array([[m[param_index][0], m[param_index][0]]] * n_vars)
    return varbound, n_vars

```

Code snippet 4 – Support genetic algorithm metaheuristic – calculate min and max bound for each parameter

For this project, this metaheuristic is only tested against benchmark functions and for this reason the implemented function uses the function *n_dimensions* and *suggested_bounds* provided by the Benchmark library. In theory, the suggested bounds are conditioned by the objective function and, consequently, the function must be updated according to the functions supported by the system.

Besides that, this library requires an initial population and does not support a single initial solution. Note that the PSO version, using Pyticle Swarm, is able to generate automatically (randomly) all the initial population individuals that are not provided as initial solution. For this reason, taking into consideration that the proposed solution only selects a single solution after each iteration, the code on the Code snippet 5 was developed in order to create a population based on the selected solution.

```

def get_factor_and_bounds_by_iteration(varbound, index):
    max_bound = varbound[0][1]
    min_bound = varbound[0][0]
    factor = (varbound[0][1] - varbound[0][0])/200
    return max_bound, min_bound, factor

def build_initial_population(func_id, parameters, varbound,
population_size):
    if "initial_solution" not in parameters.keys() or
len(parameters["initial_solution"]) == 0:
        return []
    new_population = []
    init_sol = []
    # append initial solution
    for el in parameters["initial_solution"]:
        init_sol.append(el)
    init_sol.append(get_function(func_id)(init_sol))
    new_population.append(init_sol)
    # process other individuals
    for index in range(population_size - 1):
        individual = []
        param_index = 0
        for el in parameters["initial_solution"]:
            calculate_and_append_variation(varbound, param_index)
            param_index += 1
        individual.append(get_function(func_id)(individual))
        new_population.append(individual)
    return new_population

def calculate_and_append_variation(varbound, param_index):
    max_bound, min_bound, factor =
get_factor_and_bounds_by_iteration(varbound, param_index)
    #condition if a negative, a positive or non value is applied
    rd_int = random.randint(-1,1)
    value = random.random() * rd_int * factor
    # add number, if the number exceed the max or
    # min bound, the max or min values are applied
    if el + num > max_bound:
        individual.append(max_bound)
    elif el + num < min_bound:
        individual.append(min_bound)
    else:
        individual.append(el + num)

```

Code snippet 5 – Support genetic algorithm metaheuristic – generate an initial population based on an initial solution.

In summary, using the received initial solution as baseline, and taking into consideration the maximum values for each variable, new valid solutions are created until the population size is met. Equation 11 presents the formula used to calculate the value of each variable

$$n = b + \left(\frac{|m_{ax} - m_{in}|}{200}\right) * r_{int} * r_{float} \quad (11)$$

where n is the new function parameter value, b is the baseline solution value, m_{ax} is the maximum valid value, m_{in} is the minimum valid value, r_{int} is a random integer number between -1 and 1, and the r_{float} is random real number between 0 and 1.

The maximum variation that a parameter calculated by two hundredth part of the module of the difference between the maximum value and the minimum value of objective function parameter. So, if the maximum value is 500 and the minimum value is -500. The maximum variation will be 5. The r_{int} is used to condition the addition type, if the r_{int} is -1 the number to add will be negative (-5), if it is 1 the number to add will be positive (5) and if is 0 the value calculated will be the baseline. The r_{float} is used to randomize the amount that will be added. For example, if the maximum variation is 5 and the r_{int} is 1 and the r_{float} is 0.5, the value added to the baseline is 2.5. This is executed iteratively to all the objective function variables and after each complete iteration, a new algorithm individual is generated.

After this processing, the metaheuristic can be instantiated and executed. The code to execute the algorithm is presented in the Code Snippet 6.

```
def run_genetic_algorithm(function_id, parameters):
    # get dimension and bounds
    varbound, dimension = get_varbound_and_dimensions(function_id)
    # create parameters input dictionary
    algorithm_param = {
        "max_num_iteration": parameters["max_num_iteration"],
        "population_size": parameters["population_size"],
        "mutation_probability": parameters["mutation_probability"],
        "elit_ratio": parameters["elit_ratio"],
        "crossover_probability": parameters["crossover_probability"],
        "parents_portion": parameters["parents_portion"],
        "crossover_type":
    parse_crossover_type(parameters["crossover_type"]),
        "max_iteration_without_improv": None,
        "initial_population": build_initial_population(function_id,
parameters, varbound, parameters["population_size"])
    }
    # Instance genetic algorithm model
    model = ga(
        algorithm_parameters=algorithm_param,
        function=get_function(function_id),
        dimension=dimension,
        variable_type="real",
        variable_boundaries=varbound,
        progress_bar= False,
        convergence_curve=False
    )
    # run optimization
    model.run()
    # process response
    solutions, fitness_values = process_response(model)
    # initialize optimization result
    return process_response(model)
)
```

Code snippet 6 – Support genetic algorithm metaheuristic – initialize and run the Genetic Algorithm library

Initially, a Python dictionary containing the required parameters is initialized, then the instance of the Genetic Algorithm is initialized, and in the end, the optimization is executed.

After the optimization, the optimization response must be processed and an *OptimizationResult* must be created in order to returned. The code that processes the response and creates the mentioned object is in the Code Snippet 7.

```
def process_response(model):
    solutions = []
    fitness_values = []
    for el in model.all_solutions:
        it_solutions = []
        it_fitness_values = []
        for el2 in el:
            it_fitness_values.append(el2[-1])
            it_solutions.append([x for x in el2[0:-1]])
        solutions.append(it_solutions)
        fitness_values.append(it_fitness_values)
    return OptimizationResult(
        model.best_function, model.best_variable, fitness_values,
        solutions)
```

Code snippet 7 – Support genetic algorithm metaheuristic – process response in the *OptimizationResult* format.

As can be observed, the evaluated solutions inside the metaheuristic are iterated and manipulated in order to be in the correct structure.

Summing up, taking into consideration that the huge differences between the Pyticle Swarm and the Genetic Algorithm metaheuristics, it can be concluded that with some customizations, the proposed solution is problem-agnostic and can be integrated with several algorithms without huge developments. However, the metaheuristic must be able to retrieve the solutions evaluated during the optimization process and must be able to receive an initial solution.

On this project, both metaheuristics are improved in order to support the required features and facilitate the integration with the developed modules. More details about the changes developed can be consulted in the section 3.

5.9.3 Genetic Algorithm results and comparison with Pyticle Swarm

To evaluate the effectiveness of the approach with GA, the parameterizations obtained using the configuration file in Figure 28 is compared with a similar configuration obtained using the traditional mode. In Figure 29 is presented the result of this comparison by number of trials.

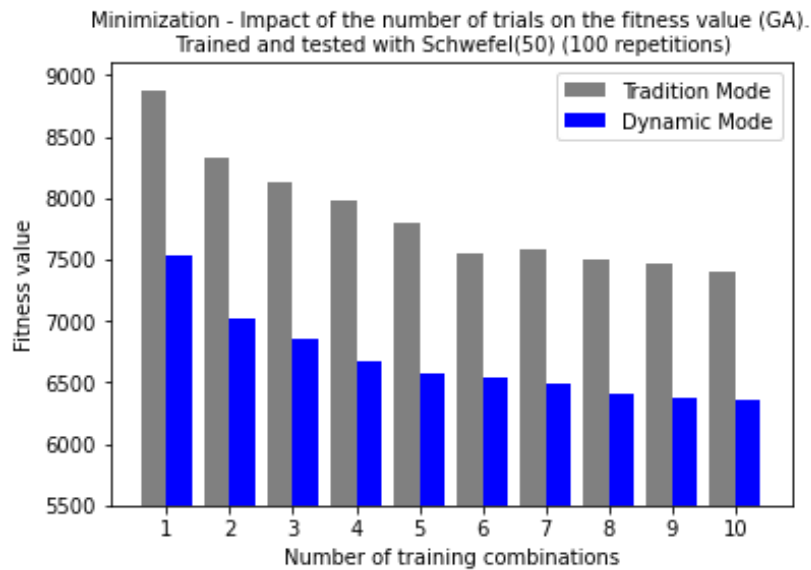


Figure 29 - Genetic algorithm - Impact of number of trials in the optimization results. Trained and tested with Schwefel(50)

As can be observed by Figure 29, using the proposed solution, the fitness values obtained are considerably better when compared with the traditional mode. Additionally, as observed in previous experiments, the difference between the obtained results using the proposed solution and the traditional mode increases with the increase of the trials number.

Therefore, the results reinforce the conclusions obtained in the previous sections, using the proposed approach. Taking into consideration that it is a requirement of the proposed solution to be metaheuristic agnostic, Figure 30 presents the comparison between the results obtained using the PSO and the Genetic Algorithm metaheuristics.

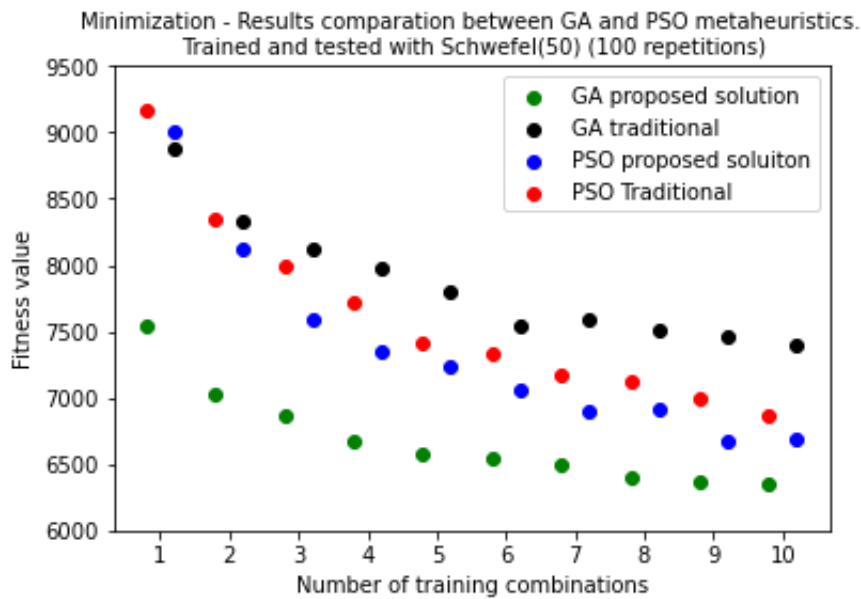


Figure 30 - Comparison between the GA and PSO results obtained optimizing the Schwefel(50) function

From Figure 30 it can be seen that the best results are obtained using the Genetic Algorithms library. This allows to conclude that the developed model is not over specified to the algorithm used on the development, especially considering that the proposed model was developed using the Pyticle Swarm library as the basis throughout the development and implementation phase. The good results achieved with both metaheuristics meet the expectations, taking into consideration that the metaheuristic specifications are externalized in a customization layer, making the proposed mode, in fact, metaheuristic-agnostic.

5.10 Summary

Results show that the developed Training module is highly impacted by the number of threads that is configured and this configuration depends on the processor used to execute the algorithm. In the other hand the batch sizer does not have significant impact.

The training for the proposed solution is faster than the training for the traditional mode. The difference between these two approaches increases exponentially with the increase of the number of parameter combinations that are experimented.

In theory, the increase of the number of combinations used to train, improves the result. Contrary to the expectations, for the number of combinations considered on this project, it is not possible to find a correlation between the number of combinations and the outcomes obtained. It occurs because there is no guarantee that all parameterizations evaluated in one execution are also evaluated in another that evaluates more combinations of parameters.

The Optimization module is highly impacted by the configured number of trials. From the execution time perspective, it increases slightly until it suddenly increases considerably. The

value at which this happens is not static and it is conditioned by the processor used on the optimization. The results, as expected, are improved consistently for each additional trial.

The results obtained using dynamic optimization approach in comparison with the traditional approach obtains better results. This improvement is more evident when the function used to train is different than the function used to test.

Besides the mentioned conclusions, the proposed approach, when applied to a real-world problem or a different metaheuristic obtains similar results. It means that it is problem- and metaheuristic- agnostic.

6 Conclusion

6.1 Achieved objectives

Given the complexity of real-world problems, the impact of the metaheuristic parameterization on the optimization results and the impact of the objective function in the metaheuristic configuration, this thesis developed an automatic parameter configuration solution composed by two modules, the Training and the Optimization module.

The Training module, through a configuration file, is a problem and algorithm agnostic system that can be used to automatize the process of training a metaheuristic, facilitating the fine-tuning process. The module supports two modes, a traditional mode where several combinations are tested and the best one is chosen, and a dynamic mode where fine-tuning is executed assuming that the optimization will occur in three steps and the exploration degree will decrease after each step.

The Optimization module is a multi-agent system that optimizes a function using parameters optimized through the Training module dynamic mode. The module introduces extra processing that increases optimization execution time. However, most of the time spent continues to be in optimizing and this approach improves considerably the obtained results, especially when the function used to optimize is different from the function used to fine tune the metaheuristic parameters. Furthermore, for each execution, the maximum execution time is defined and if the time is being exceeded, the system is prepared to stop the execution and retrieve the current best solution.

Summing up, as proven by experience, using the Training module, the fine-tuning process is automatized, and consequently, the process of supporting a different function or metaheuristic is facilitated. Using the dynamic approach inside the Optimization module, the results are improved without compromising the optimization deadlines. The improvement is bigger when the function used to train is different from the function used to optimize. So, with this approach, the generalization is bigger. It occurs because the fine-tuning is executed taking into consideration the approach (force the reduction of exploration) and not the problem.

The project was developed using the PSO metaheuristic and a set of benchmark functions. In the end, to validate the proposed solution it was tested using a real-world optimization problem that optimizes energy portfolios and using the Genetic Algorithm Python library. In order to guarantee the compatibility of the used metaheuristics with the developed system, the metaheuristics are improved in order to support the required features.

The results achieved during this work have led to the publication of an extended abstract, entitled “Online adaptation of the search process of metaheuristic algorithms”. This publication was integrated in the workshop “Artificial Intelligence Technique for the Optimization of Electric Power Distribution Systems”. Besides that, two articles are being written. One entitled “Multi-agent based model for the dynamic adaptation of metaheuristic optimization” to be submitted in the IEEE Transactions on Industrial Informatics journal; the other entitled “Dynamic parameterization of metaheuristics using a multi-agent system for the optimization of electricity market participation” to be submitted at the 21st International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS) 2023.

The development of this thesis contributed to development of the international project “Development of Artificial Intelligence Techniques for the Optimization of Electric Power Distribution Systems (FCT/CAPES 2019.00141.CBM)”

6.2 Limitations and future work

During the development of this project, some limitations impacted directly the developed work. There are not many real functions that can be used to test metaheuristics and as this project is focused on the optimization process and not on the problem itself, it limited considerably the testing phase.

The performance of the proposed approach is highly influenced by the processor characteristics. It means that in order to configure the ideal number of threads it is necessary to know the processor characteristics every time the algorithm is executed in a different machine, resulting in a specific update of the system configuration.

Both modules of the proposed solution are algorithm agnostic, however, in general the metaheuristics are very complex, and it is essential to understand the metaheuristic in order to define the parameters range correctly. Additionally, if the number of parameters or their range is huge, the number of combinations that must be evaluated in order to get good results should be huge as well. Consequently, the training can become slow making it difficult to perform a specific training process for each supported objective function.

To minimize the mentioned limitations and, at same time, to increase the effectiveness, the generalization, and the value of the proposed solution, several improvements should be implemented in the future.

The Training module needs to be updated in order to evaluate small variations of the calculated metaheuristic parameterizations. On the other hand, other offline training approaches, such as

grid search [54] or Sequential Model-based Algorithm Configuration (SMAC) [55], can be evaluated and compared with the solution proposed in this project.

From the combination's generation perspective, the Generate Combinations must be more dynamic and support other configuration types, such as a static list of values. Besides that, dynamism must be added to the parameters that impact the number of function evaluations. For example, the value for these parameters can be defined based on the variation between each iteration and the maximum time to run.

Regarding the Optimization module, it should be prepared to run using the traditional approach. The number of trials, instead of being static, can be adapted to environment conditions. For instance, the execution time should be predicted and if the time is close to the maximum time to run, the number of trials should be reduced.

In the end, both modules should be tested and validated using other metaheuristics and using further real-world objective functions.

Bibliography

- [1] V. Tatsis and K. Parsopoulos, "Reinforced Online Parameter Adaptation Method for Population-based Metaheuristics," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, Canberra, ACT, Australia, 2020.
- [2] K. Hussain, M. Najib, M. Salleh, S. Cheng and Y. Shi, "Metaheuristic research: a comprehensive survey," *Artificial Intelligence Review*, vol. 52, p. 2191–2233, 2019.
- [3] T. Pinto, Z. Vale and S. Widergren, "Local Electricity Markets," *Academic Press*, pp. 1-3, 2021.
- [4] A. G. & A. Tayal, "Metaheuristics: review and application," *Journal of Experimental & Theoretical Artificial*, vol. 25, no. 4, pp. 503-526, 2013.
- [5] E.-G. Talbi, *Metaheuristics: From Design to Implementation*, Wiley, 2009.
- [6] L. Bianchi, M. Dorigo and L. M. Gambardella, "A survey on metaheuristics for stochastic combinatorial," *Natural Computing*, vol. 8, pp. 239-287, 2009.
- [7] F. Peres and M. Castell, "Combinatorial Optimization Problems and Metaheuristics: Review, Challenges, Design, and Development," *Applied Sciences*, vol. 11, no. 14, 2021.
- [8] A. Soler-Dominguez, A. A. Juan and R. Kizys, "A Survey on Financial Applications of Metaheuristics," vol. 50, no. 15, pp. 1-23, 2017.
- [9] V. Coletto-Alcudia and M. A. Vega-Rodríguez, "A metaheuristic multi-objective optimization method for dynamical network biomarker identification as pre-disease stage signal," *AppliedSoftComputing*, vol. 109, no. 107554, 2021.

- [10] M. A. Elaziz, A. H. Elsheikh, D. Oliva, L. Abualigah, S. Lu and A. A. Ewees, "Advanced Metaheuristic Techniques for Mechanical Design Problems: Review," *Archives of Computational Methods in Engineering*, vol. 29, p. 695–716, 2021.
- [11] A. A. Dadvar, J. Vahidi, Z. Hajizadeh, A. Maleki and M. R. Bayati, "Experimental study on classical and metaheuristics algorithms for optimal nanochitosan concentration selection in surface coating and food packaging," *Food Chemistry*, vol. 335, no. 15, 2021.
- [12] M. Papadimitrakis, N. Giamarellos, M. Stogiannos, E. Zois, N.-I. Livanos and A. Alexandridis, "Metaheuristic search in smart grid: A review with emphasis on planning, scheduling and power flow optimization applications," *Renewable and Sustainable Energy Reviews*, vol. 145, no. 111072, 2021.
- [13] U. Can and B. Alatas, "A novel approach for efficient stance detection in online social networks with metaheuristic optimization," *Technology in Society*, vol. 64, no. 101501, 1021.
- [14] G. Juarez, O. Abarategi, Eguia and P., "Importance of Parameterization to Improve Meta-heuristics Performance for Smart Grid Applications," in *18th International Conference on Renewable Energies and Power Quality*, Granada, Spain, 2020.
- [15] F. Almeida, D. Giménez, J. J. López-Espín and M. Pérez-Pérez, "Parameterized Schemes of Metaheuristics: Basic Ideas and Applications With Genetic Algorithms, Scatter Search, and GRASP," *TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS: SYSTEMS*, vol. 43, no. 3, pp. 570-586, 2013.
- [16] T. Dokeroglua, E. Sevinc, T. Kucukyilmaz and A. Cosar, "A survey on new generation metaheuristic algorithms," *Computers & Industrial Engineering*, vol. 137, 2019.
- [17] A. E. Eiben, R. Hinterding and Z. Michalewicz, "Parameter Control in Evolutionary Algorithms," *RANSACTIONS ON EVOLUTIONARY COMPUTATION*, vol. 3, no. 2, pp. 124-141, 1999.
- [18] J. Carvalho, T. Pinto and R. Romero, "Online adaptation of the search process of meta-heuristic algorithms," in *Abstract for the Workshop on Artificial Intelligence Techniques for the Optimization of Electric Power Distribution Systems*, Online, 2022.
- [19] "Benchmark Functions - A Python Library," 29 September 2021. [Online]. Available: https://gitlab.com/luca.baronti/python_benchmark_functions/-/blob/master/README.md. [Accessed 22 January 2022].
- [20] M.-H. Lin, J.-F. Tsai and C.-S. Yu, "A Review of Deterministic Optimization Methods in Engineering and Management," *Mathematical Problems in Engineering*, vol. 2012, no. 756023, 2012.
- [21] P. Liu, X. Cai and S. Guo, "Deriving multiple near-optimal solutions to deterministic reservoir operation problems," *Water Resources Researchers*, vol. 47, no. 11, 2011.

- [22] I. Boussaïd, J. Lepagnot and P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, pp. 82-117, 2013.
- [23] M. Corazza, G. d. Tollo, G. Fasano and R. Pesenti, "A novel hybrid PSO-based metaheuristic for costly portfolio," *Annals of Operations Research*, vol. 304, pp. 109-137, 2021.
- [24] D. Paul, R. Su, M. Romaina, V. Sébastienb, V. Pierrea and G. Isabellea, "Feature selection for outcome prediction in oesophageal cancer using genetic algorithm and random forest classifier," *Pattern Recognition*, vol. 60, pp. 42-29, 2017.
- [25] L. Kaia, Y. Yanyun, W. Yunlong and H. Zhenwu, "Research on structural optimization method of FRP fishing vessel based on artificial bee colony algorithm," *Advances in Engineering Software*, vol. 121, pp. 250-261, 2018.
- [26] E. El-Gendy, M. M. Saafan, M. S. Elksas, S. Saraya and F. F. G. Areed, "Applying hybrid genetic-PSO technique for tuning an adaptive PID controller used in a chemical process," *Soft Comput*, vol. 24, p. 3455-3474, 2020.
- [27] J. G. Álvarez, M. Á. González, C. R. Vela and R. Varela, "Electric Vehicle Charging Scheduling by an Enhanced Artificial Bee Colony Algorithm," *Energies*, vol. 11, no. 10, 2018.
- [28] F. A. Ozbay and B. Alatas, "A Novel Approach for Detection of Fake News on Social Media Using Metaheuristic Optimization Algorithms," *Elektronika Ir Elektrotechnika*.
- [29] V. Tatsis and K. Parsopoulos, "Dynamic parameter adaptation in metaheuristics using gradient approximation and line search," *Applied Soft Computing*, vol. 74, pp. 368-384, 2019.
- [30] A. Rodríguez-Molina, E. Mezura-Montes, M. G. Villarreal-Cervantes and M. Aldape-Pérez, "Multi-objective meta-heuristic optimization in intelligent control: A survey on the controller tuning problem," *Applied Soft Computing*, vol. 93, no. 106342, 2020.
- [31] P. Melin, F. Olivas, O. Castillo, F. Valdez, J. Soria and M. Valdez, "Optimal design of fuzzy classification systems using PSO with dynamic parameter adaptation through fuzzy logic," *Expert Systems with Applications*, vol. 40, no. 8, pp. 3196-3206, 2013.
- [32] Z.-H. Zhan, J. Zhang, Y. Li and H. S.-H. Chung, "Adaptive Particle Swarm Optimization," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 6, pp. 1362-1381, 2009.
- [33] V. Zakharov and A. Mugaisikh, "Dynamic Adaptation of Genetic Algorithm for Solving Routing Problems on Large Scale Systems," *Advances in Systems Science and Applications*, vol. 20, no. 2, pp. 32-43, 2020.
- [34] C. J. M. Moctezuma, J. Mora and M. G. Mendoza, "A self-adaptive mechanism using weibull probability distribution to improve metaheuristic algorithms to solve

- combinatorial optimization problems in dynamic environments.," *Math Biosci Eng.*, vol. 17, no. 2, pp. 975-997, 2019.
- [35] E. Shadkam, "Parameter setting of meta-heuristic algorithms: a new hybrid method based on DEA and RSM," *Environmental Science and Pollution Research*, 2021.
- [36] S. Han and L. Xiao, "An improved adaptive genetic algorithm," in *2022 International Conference on Information Technology in Education and Management Engineering (ITEME2022)*, SHS Web Conference, 2022.
- [37] R. Sala and R. Müller, "Benchmarking for Metaheuristic Black-Box Optimization: Perspectives and Open Challenges," *Neural and Evolutionary Computing*, 2020.
- [38] J. Kennedy and R. Eberhart, "Particle Swarm Optimization," in *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, Nagoya, Japan, 1996.
- [39] D. Wang , D. Tan and L. Liu, "Particle swarm optimization algorithm: an overview," *Soft Computing*, vol. 22, pp. 387-408, 2018.
- [40] N. K. Jain, U. Nangia and J. Jain , "A Review of Particle Swarm Optimization," *Journal of The Institution of Engineers (India): Series B*, vol. 99, pp. 407-411, 2018.
- [41] L. James and V. Miranda, "PySwarms: a research toolkit for Particle Swarm Optimization in Python," *The Journal of Open Source Software*, vol. 3(21), no. 433, 2018.
- [42] I. Bakurov, M. Buzzelli, M. Castelli, L. Vanneschi and R. Schettini, "General Purpose Optimization Library (GPOL): A Flexible and Efficient Multi-Purpose Optimization Library in Python," *Applied Sciences*, vol. 11, no. 4774, 2021.
- [43] B. Veiga, R. Faia, T. Pinto and Z. Vale, "https://pypi.org/project/Pyticle-Swarm/," 14 January 2022. [Online]. Available: <https://pypi.org/project/Pyticle-Swarm/#description>. [Accessed 22 January 2022].
- [44] S. Katoch, S. S. Chauhan and V. Kumar, "A review on genetic algorithm: past, present, and future," *Multimedia Tools and Applications*, vol. 80, p. 8091–8126, 2021.
- [45] PYGAD, "PyGAD - Python Genetic Algorithm," [Online]. Available: <https://pygad.readthedocs.io/en/latest/>. [Accessed January 2022].
- [46] R. M. Solgi, "geneticalgorithm - Project description," [Online]. Available: <https://pypi.org/project/geneticalgorithm/>. [Accessed January 2022].
- [47] T. Devlin, "Genetic Algorithms for python," [Online]. Available: <https://pypi.org/project/genetic-algorithms/>. [Accessed January 2022].

- [48] F. Bellifemine, A. Poggi and G. Rimassa, "Developing multi-agent systems with a FIPA-compliant agent framework," *SOFTWARE—PRACTICE AND EXPERIENCE*, vol. 31, pp. 103-128, 2001.
- [49] SPADE, "The SPADE agent model," [Online]. Available: <https://spade-mas.readthedocs.io/en/latest/model.html>. [Accessed 5 2022].
- [50] Ejabberd, "Ejabberd," [Online]. Available: <https://docs.ejabberd.im/get-started/>. [Accessed 2022].
- [51] R. Faia, T. Pinto and Z. Vale, "Portfolio Optimization for Electricity Market Participation with Particle Swarm," in *2015 26th International Workshop on Database and Expert Systems Applications (DEXA)*, Valencia, Spain, 2015.
- [52] "Nexus between financial development and renewable energy: Empirical evidence from nonlinear autoregression distributed lag," *Renewable Energy*, vol. 193, pp. 475-483, 2022.
- [53] N. Mostafa, H. S. M. Ramadan and O. Elfarouk, "Renewable energy management in smart grids by using big data analytics and machine learning," *Machine Learning with Applications*, vol. 9, no. 100363, 2022.
- [54] D. M. Belete and M. D. Huchaiah, "Grid search in hyperparameter optimization of machine learning models for prediction of HIV/AIDS test results," *International Journal of Computers and Applications*, 2021.
- [55] M. Lindauer, K. Eggenberger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass and F. Hutter, "SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization," *Journal of Machine Learning Research*, vol. 22, pp. 1-9, 2021.