DISSERTAÇÃO DE MESTRADO

Employment of Parameter Adaptive Techniques to Bio-Inspired Meta-Heuristics for Mapping Real-Time Applications onto NoC based MPSoCs

Jessé Barreto de Barros

Brasília, Agosto de 2019

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

Employment of Parameter Adaptive Techniques to Bio-Inspired Meta-Heuristics for Mapping Real-Time Applications onto NoC based MPSoCs

Jessé Barreto de Barros

Dissertação submetida ao Departamento de Engenharia Mecânica da Faculdade de Tecnologia da Universidade de Brasília como requisito parcial para obtenção do grau de Mestre Engenheiro em Sistemas Mecatrônicos.

Banca Examinadora

Prof. Dr.Carlos H. Llanos Q., ENM/FT/UnB _____
Orientador

Prof. Dr. Mauricio Ayala Rincón, CIC/UnB _____
Examinador externo

Prof. Dr. Jones Yudi Mori Alves da Silva, ENM/UnB _____
Examinador interno

FICHA CATALOGRÁFICA

BARROS, JESSÉ BARRETO DE
Employment of Parameter Adaptive Techniques to Bio-Inspired Meta-Heuristics for Mapping Real-Time Applications onto NoC based MPSoCs

[Distrito Federal] 2019.

xii, 203 p., 210 x 297 mm (ENM/FT/UnB, Mestre, Sistemas Mecatrônicos, 2019).
Dissertação de Mestrado - Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Mecânica.

1. Meta-heuristics
2. Optimization
3. Real-time Systems
4. Network-on-a-chip

I. ENM/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

BARROS, J.B. (2019). Employment of Parameter Adaptive Techniques to Bio-Inspired Meta-Heuristics for Mapping Real-Time Applications onto NoC based MPSoCs, Dissertação de Mestrado em Sistemas Mecatrônicos, Publicação ENM.DM-71A/14, Departamento de Engenharia Mecânica, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 203 p.

CESSÃO DE DIREITOS

AUTOR: Jessé Barreto de Barros
TÍTULO: Employment of Parameter Adaptive Techniques to Bio-Inspired Meta-Heuristics for Mapping Real-Time Applications onto NoC based MPSoCs.
GRAU: Mestre ANO: 2019

Jessé Barreto de Barros
Depto. de Engenharia Mecânica (ENM) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

Acknowledgements

I would like to express my thanks to my advisor Prof. Carlos H. Llanos, who was always very supportive and always provided me with excellent advice and great insights. I also would like to express my gratitude towards my lab mate Carlos Eduardo for our constructive conversations about the topics shared by our research. Agradezco a Oscar por no haberme ayudado en nada.

Eu gostaria de agradecer à minha familia especialmente à minha mãe, irmã, e avó pela força. Eu também gostaria de agradecer aos meus amigos que adquiri ao longo dos anos na UnB em especial Marquemilho, Carioca (Francisco), Pedro, Andrezinho, e o Teixeira. Moreover, I would like to thank my best friend and girlfriend Chanhui, who helped me a lot during this work giving me much needed support. At last, I would like to make Ozzy Osbourne quote as my words: "Out of everything I've lost, I miss my mind the most!"

Jessé Barreto de Barros

RESUMO

O problema de mapear tarefas de uma aplicação em tempo-real (RTA) em uma plataforma com múltiplos processadores no mesmo chip do tipo (MPSoC) que utiliza uma rede intra-chip (NoC) como arquitetura de comunicação pode ser abordado como um problema de otimização que possui o intuito de melhorar características do design do sistema utilizando uma análise estática. Exemplos de característica que se deseja que sejam melhoradas são a conformidade do sistema com as suas restrições temporais e o use dos seus recursos. Uma meta-heurística bio-inspirada, como o Algoritmo Genético (GA) pode obter mapeamento de tarefas no qual todas as tarefas da aplicação são agendadas. Além disso, no contexto de meta-heurísticas, algoritmos que incorporam técnicas adaptativas para os seus parâmetros e operadores são capazes de mudar os seus mecanismos de busca baseando-se no problema que está sendo otimizado com o intuito de simultaneamente buscar por soluções ótimas para o problema utilizado e por parâmetros melhores. Essas características permitem que meta-heurísticas que empregam esses mecanismos de mitigar a necessidade de um estágio para configurar os seus parâmetros e melhorar os seus desempenhos.

Em luz dessas informações, esse trabalho visa efetuar um estudo para explorar múltiplos técnicas de adaptação para meta-heurísticas bio-inspiradas e empregá-las no desenvolvimento de novas versões adaptativas de meta-heurísticas como o GA, o algoritmo de otimização por enxame de partículas (PSO), e o algoritmo de evolução diferencial (DE). Essas novas meta-heurísticas são subsequentemente aplicadas em múltiplas instâncias do problema de mapeamento de tarefas de uma RTA em uma plataforma do tipo MPSoC que usa NoC utilizando funções com um único ou múltiplos objetivos.

Em adição à aplicação dessas meta-heurísticas, esse trabalho também apresenta uma comparação estatística para verificar os seus desempenhos. O experimento onde esse estudo estatístico é aplicado foi conduzido para utilizar as meta-heurísticas desenvolvidas nesse trabalho e também as suas versões originais em múltiplas configurações do problema de mapeamento utilizando plataformas MPSoC com diferentes tamanhos e uma RTA utilizada como benchmark com RTA geradas de forma sintética. Os resultados obtidos indicam que, para o conjunto de problemas avaliados, meta-heurísticas com técnicas adaptativas são capazes de alcançar melhores resultados, na média, do que as suas versões sem esquemas de adaptação.

Outra contribuição desse trabalho é o desenvolvimento de um plataforma de software orientado a objetos em C++ cuja aplicação não se restringe ao problema de mapeamento de tarefas e pode ser utilizado em outros problemas de otimização como a lista de funções de otimização de benchmark.

Palavras Chaves: Meta-heurísticas, Otimização, Sistemas em Tempo-Real, Redes Intra-chip

ABSTRACT

The task mapping of a Real-Time Application (RTA) onto a Many-/Multi-Processor System-on-a-Chip (MPSoC) with a Network-on-a-Chip (NoC) as on-chip communication architecture can be tackled as an optimization problem to improve desired design features in a static analysis setting. Examples of such features are the system compliance to its time requirements and its resource usage. A search-based bio-inspired meta-heuristic such as the Genetic Algorithm (GA) is capable of achieving task placement solutions in which all tasks are schedulable. Also, in the context of meta-heuristics, algorithms that incorporate adaptive techniques for their parameter values and operators are capable of changing their internal search mechanisms based on the problem at hand to simultaneously search for optimal solutions for the problem at hand and more suitable parameters for the meta-heuristic. These characteristics allow meta-heuristics that employ these mechanisms to mitigate their necessity for a parameter tuning stage and an increase in performance.

In light of this information, this work intends to perform a study to explore adaptive techniques for bio-inspired meta-heuristics and employ them in the development of new adaptive versions of bio-inspired meta-heuristics such as GA, Particle Swarm Optimization (PSO), and Differential Evolution (DE). These novel meta-heuristics are then applied on different instances of task mapping of RTA onto MPSoC platforms with NoC using single-/multi-objective functions.

Additional to the application of these new meta-heuristics, this work also presents a statistical comparison study to assess their performance. The experiments in which this statistical study was conducted uses meta-heuristics developed in this work together with their original counterparts in multiple settings of the task mapping problem using platforms with different sizes and a benchmark RTA jointly with synthetic generated ones. The results obtained indicate that, for the set of problems used, meta-heuristics with adaptive techniques are capable of achieving better performance, on average, than their original counterparts.

Another contribution of this work is the development of an object-oriented software framework in C++ with use not limited to the task mapping problem and can be used to other optimization problems such as the list of benchmark optimization functions.

Keywords: Meta-heuristics, Optimization, Real-Time Systems, Network-on-a-Chip.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

Basic Symbols

$a, b, c, ...$	Scalar number are represented by lower case letters
$\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, ...$	Vectors are represented by bold lower case letters
$\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, ...$	Sets are represented by bold upper case letters
$\lvert\boldsymbol{A}\rvert$	Set $\boldsymbol{A}$ cardinality
$\mathbb{R}$	Real numbers set
$\mathbb{Z}$	Integer numbers set
$\mathbb{N}$	Natural numbers set (including 0)
$\mathbb{B}$	Binary numbers set
$\mathbb{C}$	Complex numbers set
$\mathbb{R}^n$	Real vector with n components
$\mathbb{R}^{n \times m}$	Real matrix with n rows and m columns
$\mathbf{1}$	Vector of ones
$\mathbf{0}$	Vector of zeros

Special Greek Symbols

$\boldsymbol{\Gamma}$	Set of tasks of a real-time application
$\boldsymbol{\Phi}$	Set of messages transmitted by tasks in $\boldsymbol{\Gamma}$
$\boldsymbol{\Psi}$	Model of a MPSoC that uses a RTNoC communication architecture
$\boldsymbol{\Omega}$	Model of a RTA
$\boldsymbol{\Pi}$	Set of processing elements connected to NIs
$\boldsymbol{\Xi}$	Set of Routing Elements
$\boldsymbol{\Lambda}$	Set of Unidirectional Links
π_m	Processing Element connect to a Network Interface
ξ_m	Routing Element in a NoC
λ_{π_m,ξ_m}	Unidirectional Link that connects elements π_m and ξ_m
$\boldsymbol{\tau}_n$	Task of a RTA
$\boldsymbol{\phi}_n$	Message transmitted by a task $\boldsymbol{\tau}_n$

Special Latin Symbols

C_i	WCET of the task $\boldsymbol{\tau}_i$
D_i	Relative Deadline time of the task $\boldsymbol{\tau}_i$
J_i	Release jitter time of a task $\boldsymbol{\tau}_i$
K_i	Release jitter for a message $\boldsymbol{\phi}_i$ transmission
l_i	Lateness for an unschedulable task $\boldsymbol{\tau}_i$
L_i	Latency of transmission of a the task $\boldsymbol{\phi}_i$ without network contention
M_i	Memory code in bytes allocated in memory for a task $\boldsymbol{\tau}_i$
P_i	Priority-level of the task $\boldsymbol{\tau}_i$
R_i	WCRT of the task $\boldsymbol{\tau}_i$
s_i	Slack time for a schedulable task $\boldsymbol{\tau}_i$
S_i	WCLT of a message $\boldsymbol{\phi}_i$
T_i	Inter-arrival period of the task $\boldsymbol{\tau}_i$
$U_{\boldsymbol{\tau}_i}$	Utilization factor of a task $\boldsymbol{\tau}_i$
U_π	Utilization factor of a processor π
U_λ	Utilization factor of a link λ
$U_{\boldsymbol{\Gamma}}$	Utilization factor of all tasks in a RTA
Z_i	Size in bytes of the message $\boldsymbol{\phi}_i$ sent by task $\boldsymbol{\tau}_i$

Acronyms

AGA	Adaptive Genetic Algorithm
AOS	Adaptive Operator Selection
APCS	Adaptive Parameter Control Strategy
APSO	Adaptive Particle Swarm Optimization
APV	Adjusted p-Value
AVA	Autonomous Vehicle Application
BA	Bat Algorithm
BDF	BreakDown Frequency
COP	Combinatorial Optimization Problem
DA	Dragonfly Algorithm
DE	Differential Evolution
DPCS	Deterministic Parameter Control Strategy
DPSO	Discrete Particle Swarm Optimization
EERT	End-to-End Response Time
EHO	Elephant Herd Optimization
FPGA	Field-Programmable Gate Array
FWER	Family-Wise Error Rate
GA	Genetic Algorithm
GWO	Gray Wolf Optimization
IC	Integrated Circuit
I/O	Input/Output
MFO	Moth-Flame Optimization
MOOP	Multi-Objective Optimization Problem
MPSoC	Many-/Multi- Processors System-on-a-Chip
NSGA	Non-dominant Sorting Genetic Algorithm
NFL	No Free Lunch
NI	Network Interface
NoC	Network-on-a-Chip
PE	Processing Element
PF	Pareto Front
PO	Pareto Optimal
PSO	Particle Swarm Optimization
RM	Rate Monotonic
RTA	Real-Time Application
RTNoC	Real-Time Network-on-a-Chip
RTS	Real-Time System
SLAM	Simultaneous Localization And Mapping
SLHD	Symmetric Latin Hypercube Design
SoC	System-on-a-Chip
SOOP	Single-Objective Optimization Problem
SSA	Salp Swarm Algorithm
VC	Virtual Channel
WCET	Worst Case Execution Time
WCLT	Worst Case Latency Time
WCRT	Worst Case Response Time
WOA	Whale Optimization Algorithm

1 INTRODUCTION

1.1 Contextualization

System-on-a-Chip (SoC) [8] is the paradigm to integrate a whole computing system with multiple specialized modules, processor cores, FPGAs, memory caches, memory controllers, inside of the same chip die. It results in a complex Integrated Circuit (IC) with hundreds of millions of transistors that, thanks to the development in IC manufacturing techniques, is heavily relied upon on modern commercial products. For example, a current modern smartphone such as the Samsung's flagship phone Galaxy S10 uses a Qualcomm SoC chip [9] that has components such as processors, graphical processing units, and telephony related modules.

In the information age, SoC products, and their development and production represent the economic source of hundreds of billions of dollars annually. For example, Apple the third most valuable company (as of 2019), has a large part of their market value due to products using SoCs such as smartphones and tablets.

It is common for current commercial SoC to contain multiple processors core inside of the chip. Systems with this attribute are characterized as a Multi-/Many-Processors System-on-a-Chip (MPSoC). In SoCs, and specifically in MPSoCs, as the number of components inside of the chip increases, so does the on-chip communication architecture, the subsystem responsible for data transmission inside of the chip, importance for the system correct functionality and complexity to properly handle the large data transmitted. This growth in complexity is so accentuated in modern SoCs that the on-chip communication architecture has become one of the primary sources of bottlenecks in terms of timing performance due to communication delays, and power and area [10].

In light of these problems, there has been a push from the research community towards a paradigm of an on-communication architecture inspired by computer networks. This new paradigm should be capable of coping with the scalability necessary for the crescent number of elements inside of current and future SoCs design. This on-chip communication paradigm is the Network-on-a-Chip (NoC) [8] [11].

An NoC is an on-chip communication architecture that can be defined by the following characteristics, as stated by [8]:

- The on-chip communication architecture can be separated in multiple abstraction layers similar to computer networks. For example, characteristics of the implementation, such as wiring can be attached to the physical layer design, while abstracted during the data layer design to deal with data transmission protocols. Figure 1.1 represents the mapping on the standard Open System Interconnection (OSI) network protocol stack model of an implementation of an NoC.

- The communication control is distributed inside of the network designated to specialized components called *routers*.
- High customization capability with the possibility to use, for example, different topological connection for its routers, and multiple types of specialized components connected.
- NoC is scalable since the number of communication channels, and routers increases as the number of components grow.
- Better power efficiency due to reduced wiring length (depending on the topology chosen) between routers and communication control for large systems with less complexity.
- Independency between modules since each router connects its system module. It offers a high re-usability between modules since their implementation is independent of the on-chip communication architecture.

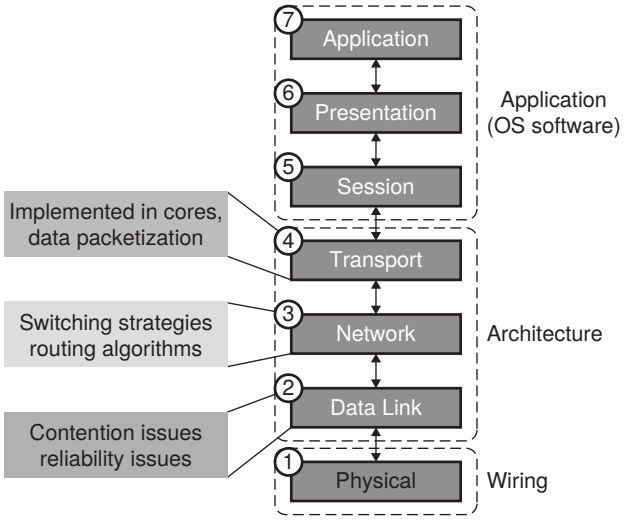


Figura 1.1 – Illustration of the arbitration of a NoC in the OSI computer network protocol extracted from [1].

In the context of embedded devices, the field advances with an ever-growing number of possible critical applications from the expected growth in the development of Internet-of-Things (IoT) devices, the maturation of 5G telecommunication infrastructure, and the deployment of autonomous vehicles. Current and future embedded devices have a large number of processing elements, power usage, and time constraints that are likely to be met by using NoC-based MPSoC designs. Since some of the systems are employed in time-sensitive applications, they are categorized as Real-Time Systems (RTS), and their performance is heavily dependent on the system complying with the Real-Time Application (RTA) time constraints.

Due to the broad range of design possibilities in an NoC-based system allied with the time constraints inherent from RTS, it is imperative for the design of an MPSoC deployed for such systems to consider the time characteristics of its RTA since the early stages of the project. RTAs can be subdivided into tasks that are a chunk of the application software that represents one of

the system tasks. Mapping RTAs onto MPSoCs is the process to define tasks placement onto specific processor cores in the system [12]. This process for an NoC-based system is illustrated in Figure 1.2.

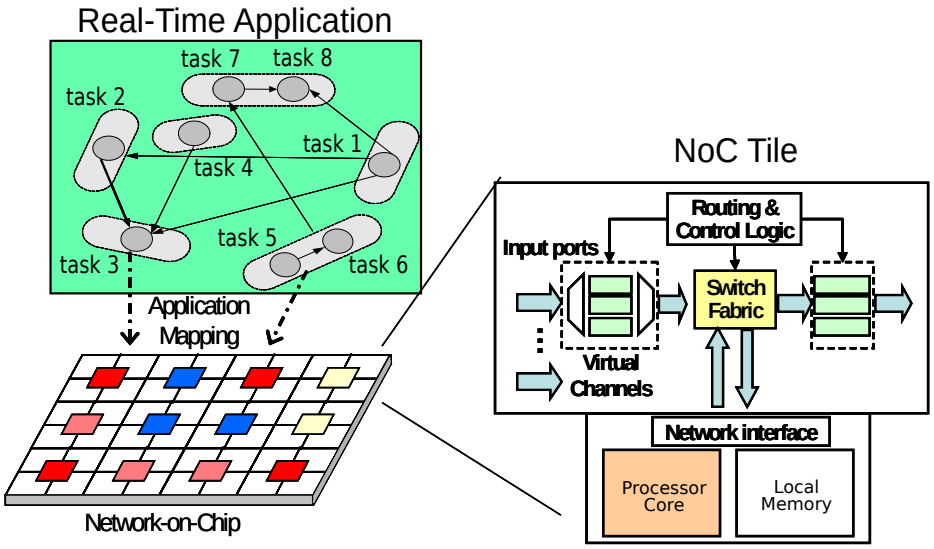


Figura 1.2 – Illustration of a RTA mapping onto a NoC-based MPSoC (adapted from [2]).

When performed during the design time, the RTA mapping onto MPSoCs can be faced by defining an analytic method to evaluate each task mapping solution as an optimization problem quickly. The search for an optimal task placement using a specific quality metric for the system is an NP-hard problem akin to the bin packaging problem [13]. Bio-inspired meta-heuristics such as the Genetic Algorithm (GA) has been successfully used in previous related works to search for task mappings that result in RTAs to comply with their time constraints ([14]).

This work proposes alternative bio-inspired meta-heuristics that employ adaptive techniques that could be used in the task mapping of RTAs onto NoC-based MPSoCs both for single-objective optimization as well as multi-objective ones. These meta-heuristics are composed of recent algorithms from literature as well as brand new ones developed during this work.

Additionally, this work also presents a comparative study to analyze which meta-heuristic is more suitable, on average, for the task mapping optimization problem using different objectives on multiple scenarios with synthetic RTA and MPSoCs. Including one objective developed during this work that evaluates at the same time the number of tasks that do not comply with their timing requirements, and, if all of them respect their time constraints, the minimum slack ratio starts to be optimized.

1.2 Commercial Applications of MPSoC and NoCs

The leading manufacturers of desktop processors, Intel and AMD, use MPSoCs in their new high-end line of products. It highlights the importance of the themes studied in this work and their application in current and future commercial products.

In AMD case, their most recent processors that use their microarchitecture Zen [15] that is aimed to be used in a broad range of products from low-end notebooks with Ryzen processors up to high-performance server computers branded as EPYC. Their high-end EPYC Zen-based products have completely integrated MPSoCs containing memory-controllers, caches and a proprietary on-chip communication architecture called Infinity Fabric that connect both components inside of the same die as well as external modules such as graphical processors.

Intel high-end recent products aimed to servers is part of the Xeon family of processors [16]. These processors are MPSoC since they have up to 56 processor cores in the same die as well as components such as caches, memory controllers, and I/O modules. To connect these internal components, the new Xeon processors use a 2-dimensional mesh-grid NoC to connect them to cope with the increased data traffic in the die.

1.3 Objectives

These work objectives are subdivided into two types: (a) primary: the main objectives for this work; (b) secondary: objectives met during the process to achieve the main objectives;

1.3.1 Primary Objective

The general objective of this work is to apply bio-inspired meta-heuristics with adaptive mechanisms to search for optimal task mapping solutions of real-time applications onto NoC-based MPSoC using current state-of-the-art analytical methods to evaluate the timing characteristics for the systems.

1.3.2 Secondary Objectives

The specific objectives of this work are the following:
(a) Development of a high-performance framework containing the implementation of multiple bio-inspired meta-heuristics and modules to analyze NoC models as optimization problems;
(b) Use of bio-inspired meta-heuristics from literature;
(c) Development and implementation of bio-inspired meta-heuristics that contain adaptive techniques;
(d) Implementation of single-/multi-objective benchmark optimization problems to quickly evaluate the developed meta-heuristics performances;
(e) Development and implementation of objective functions related to task placement of RTAs onto MPSoCs;
(e) Perform a comparative study to assess which meta-heuristics are more suitable for the task mapping problem of RTAs onto MPSoCs.

1.4 Methodology

This work objective uses the following methodological proceedings to meet its objectives:

- Research and study about the fundamentals of the themes: (a) MPSoC, (b) NoCs, and (c) Bio-inspired meta-heuristics.
- Implementation of a framework to analyze multiple characteristics of an MPSoC with NoC based on the mapping of a modeled application onto a modeled system. These characteristics are: (a) real-time schedulability analysis, (b) resource use, and (c) dissipated energy on communication architecture.
- Implementation and development of objective functions that use modeled NoC based system characteristics as metrics.
- Development and implementation of bio-inspired meta-heuristics that uses adaptation techniques.
- Validation of explored methods though the use of experiments using the meta-heuristics in a range of optimization problems, including benchmarks and problems related to NoCs.
- Analysis of experimental results using non-parametric statistical tests that includes hypothesis tests to assess the significance of the improvement obtained by the different meta-heuristics used.

1.5 Contributions

This work has the following contributions:
(a) Development of a C++ framework for bio-inspired optimization meta-heuristics with modules to analytically analyze NoC-based MPSoCs timing characteristics;
(b) Development of a pair of new bio-inspired meta-heuristics based on Differential Evolution (DE) that contain adaptation of parameter values as well as adaptation of operations named Single-Objective Adaptive with Modified Selection Differential Evolution (SOAMSDE) and Multi-Objective Non-dominated Sorting Adaptive Differential Evolution (MONSADE);
(c) Development of four adaptive versions of the GA named AGAv1, AGAv2, AGAv3, and AGAv4;
(d) Development of two adaptive versions of the Particle Swarm Optimization (PSO) name APSO, and APSOv2;
(e) Development of a modified version of a hybrid discrete version of PSO that contains a local-search method specialize for the task mapping problem of RTAs onto MPSoCs named Hybrid Discrete PSO-Utilization-based (HDPSO-U);
(f) Development of a version of HDPSO-U that contains adaptive mechanisms for parameter control named Adaptive HDPSO-U (AHDPO-U);
This meta-heuristic was presented in the article "An Adaptive Discrete Particle Swarm Optimization for Mapping Real-Time Applications onto Network-On-Chip based MPSoCs" [17];

(g) Development of a single-objective metric to evaluate task mapping that contains both information of the number of tasks that do not comply with their timing constraints and slack awareness.

1.6 Manuscript Organization

This work has seven other chapters divided as follows:

- Chapter 2 introduces the mathematical framework for problem optimization, including the reasoning for the modeling of a problem as a single- or multi-objective optimization problem.
- Chapter 3 introduces concepts related to SoCs, NoCs, and RTS, including the modeling for NoC-based MPSoCs and the definition of task mapping as an optimization problem.
- Chapter 4 introduces bio-inspired meta-heuristics, including concepts and taxonomy of bio-inspired meta-heuristics that contain adaptive techniques to modify their characteristics mid-execution.
- Chapter 5 presents a brief survey of related works in the area of search-based bio-inspired meta-heuristics and task mapping of RTAs onto MPSoCs.
- Chapter 6 presents the C++ software developed in this work, including the design choices for classes hierarchy, and tutorials for users.
- Chapter 7 presents the experimental setup and the results for the multiple optimization problems present in this work with emphasis for the task mapping optimization of RTAs onto MPSoC with NoCs using bio-inspired meta-heuristics.
- Chapter 8 concludes this work by presenting possible future works that one can continue from this work.

2 OPTIMIZATION PROBLEMS

This chapter briefly introduces the concepts used in problem optimization. It also presents a mathematical framework to assist the intelligibility of other parts of this work. The chapter is divided as follows: First, in Section 2.1, it briefly introduces optimization problems, then, in Section 2.2, it describes, in a high-level abstraction, the process to model and encodes a real-world problem as an optimization problem with single or multi-objectives. In Section 2.3, it describes examples of optimization problems used in this work, including combinatorial/discrete and continuous problems. Lastly, in Section2.4, concludes the chapter and contextualizes it with other parts of this work.

2.1 Introduction

Optimization problems is a multi-disciplinary field that is a standard part of all areas of sciences and engineering and, for this reason, it is a complex and vast field with a myriad of approaches to tackle it. This work intends to only briefly describe some mathematical framework of the field to assist intelligibility of the work as a whole. In general, optimization problem is a process to minimize or maximize an objective function by searching for an input solution for said function inside a domain of possible solutions that results in its optimal result. In this work, the focus of the optimization problems will be minimization ones. Notice that even maximization problem can be treated as a minimization one, it can be done simply by multiplying its numeric metric values by -1.

2.2 Problem Modeling and Codification

As present in [18], a real-world problem $\mathcal{R}_w$ that needs to have some of its characteristics optimized with optimal or near-optimal parameters, given one or multiple specific goals, can be described by a computational model $\mathcal{M}$. The model $\mathcal{M}$ can have its numerical parameters encoded as a problem $\mathcal{P}(\mathcal{M})$ that defines the parameters being manipulated as an n-dimensional vector $\boldsymbol{x}$, namely its decision variable, that ranges inside of a set of possible parameters S called the search space.

Vector $\boldsymbol{x} \in$ S is not limited to a single number type and its components can be real, integer, binary, complex numbers, or even a combination of these sets $\{\mathbb{R} \times \mathbb{Z} \times \mathbb{B} \times \mathbb{C}\}$. The selection of an approach to encode the model $\mathcal{P}(\cdot)$ defines S and its size, and, therefore, this selection impacts

the complexity of the problem at hand and the capacity or performance of a possible optimization method to solve the problem $\mathcal{R}_w$. Figure 2.1 illustrates the abstraction of modeling and encoding of a real-world problem $\mathcal{R}_w$ into an optimization problem.

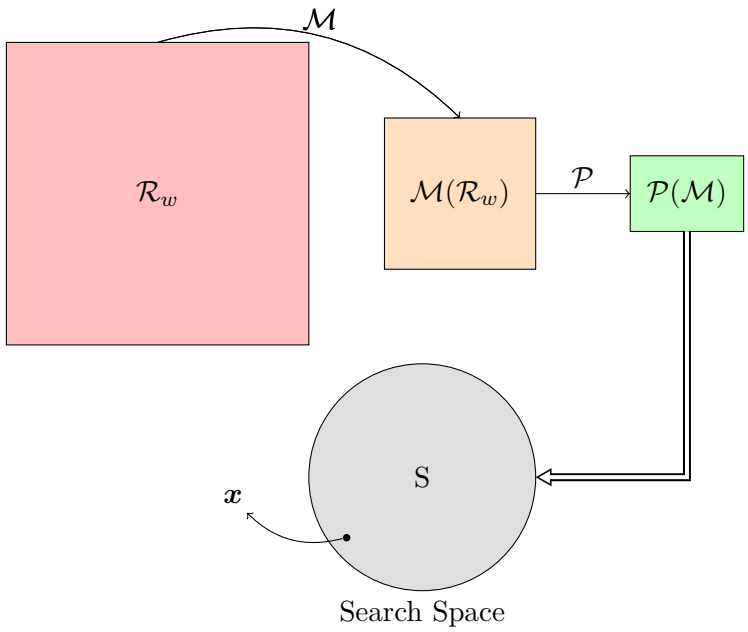


Figura 2.1 – Representation of the process to define a model $\mathcal{M}$ for a real-world problem $\mathcal{R}_w$ and then encoding this model as an optimization problem $\mathcal{P}(\mathcal{M})$ with a search space S composed of possible solutions $\boldsymbol{x} \in \mathrm{S}$.

The optimization of $\mathcal{P}(\mathcal{M})$ is done with respect to one or multiple objectives by defining an objective function $f(\cdot)$ or a vector of objective functions $\boldsymbol{F}(\cdot)$. This objective function is then used as a numeric metric to define the quality of a possible solution and be used for a comparison between these solutions. In the same fashion that a real-world problem can be modeled in different ways, it possible to select multiple objective functions for a single problem depending on the desired metric. Similar to the decision variables, an objective function is not restricted to a single number type. However, in this work, for simplicity, the used objective functions have their images always defined as real or integer numbers.

2.2.1 Single-Objective Optimization Problem

A Single-Objective Optimization Problem (SOOP) uses in its encoded model $\mathcal{P}(\mathcal{M})$ a single evaluation metric for its solutions defined as a single-objective function f defined as following:

$$f(\boldsymbol{x}) : \mathrm{S} \rightarrow \mathrm{O}, \tag{2.1}$$

where O is the objective space that for this work purposes is a subset of $\mathbb{R}$, i.e., $\mathrm{O} \in \mathbb{R}$.

Some problem models also have regions in its search space S that contains unfeasible solutions. It means that this type of problems have constraints equations that represent whether a possible

solution is feasible. These constraint equations can be divided into m equality constraints h_i ($i \in [1, m]$) and p inequality constraints g_j ($j \in [1, p]$).

With the previous concepts in mind, it is possible to provide a general definition of an optimization problem, a minimization, for a given model $\mathcal{M}$ as in Eq. 2.2. In this equation, $\boldsymbol{x}^*$ is a solution that satisfy all restrictions and holds the optimal result for $f(\cdot)$.

$$\begin{aligned} \boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, & f(\boldsymbol{x}) \\ & \boldsymbol{x} \in \mathrm{S} \\ \text{Subject to} \quad & \\ & g_i(\boldsymbol{x}) \leq 0,\ i \in \{1, ..., m\} \\ & h_i(\boldsymbol{x}) = 0,\ i \in \{1, ..., n\} \end{aligned} \quad . \tag{2.2}$$

Figure 2.2 illustrates an example SOOP with a single-objective function f that maps the search space $\mathrm{S} \in \mathbb{R}^2$ to the objective space $\mathrm{O} \in \mathbb{R}$.

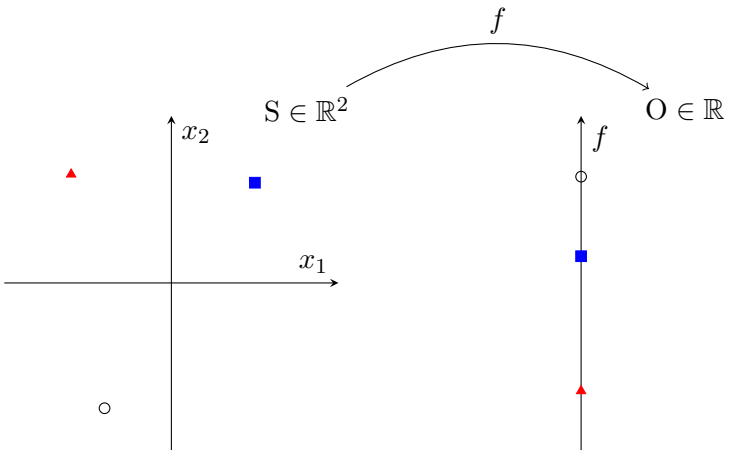


Figura 2.2 – Illustration of a SOOP with an objective function mapping the search space to the objective space.

2.2.2 Multi-Objective Optimization

A Multi-Objective Optimization Problem (MOOP) [19] uses in its encoded model $\mathcal{P}(\mathcal{M})$ multiple numeric metrics as objectives. Generally, these metrics can not be combined as terms of a single-objective function f due to: (a) conflicting objectives, when one objective improves the others deteriorate; (b) different order of magnitude, when an objective scale is small while others are large.

Multi-objective Optimization Problems is defined as following:

$$\begin{aligned} \boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, \boldsymbol{F}(\boldsymbol{x}) \\ \boldsymbol{x} \in \mathrm{S} \\ \text{Subject to} \\ g_i(\boldsymbol{x}) \leq 0,\ i \in \{1, ..., m\} \\ h_i(\boldsymbol{x}) = 0,\ i \in \{1, ..., n\} \end{aligned} \quad , \tag{2.3}$$

where $\boldsymbol{F}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), ..., f_o(\boldsymbol{x}))$ is a vector composed of o objective functions and $\boldsymbol{x}^* = (x_1, ..., x_n)$ satisfy all restriction while resulting the optimal solutions for all component f_i of $\boldsymbol{F}$. $\boldsymbol{F}$ maps the decision variable from their search space S to the objective space O defined as $\boldsymbol{F} : \mathrm{S} \rightarrow \mathrm{O}$. Figure 2.3 illustrates an example of MOOP with a multi-objective function $\boldsymbol{F}$ that maps the search space $\mathrm{S} \in \mathbb{R}^2$ to the objective space $\mathrm{O} \in \mathbb{R}^2$.

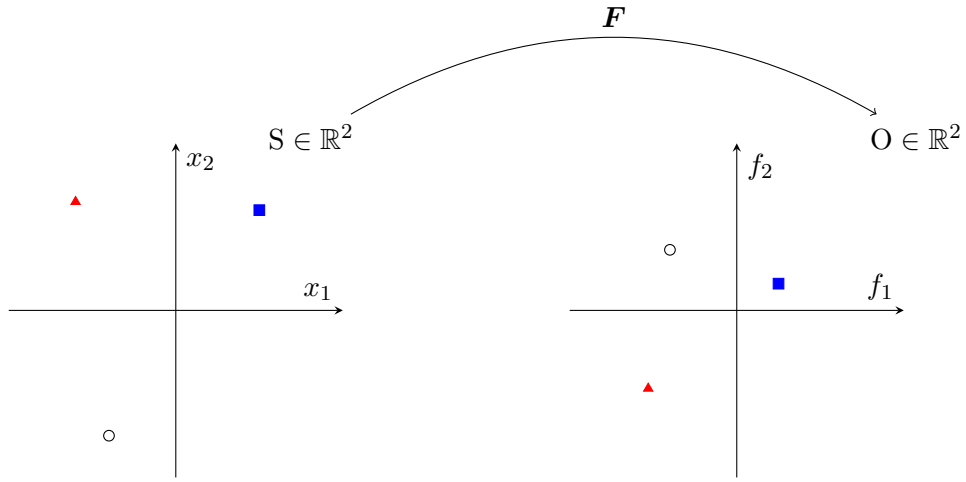


Figura 2.3 – Illustration of a MOOP with a vector of objective functions mapping the search space to the objective space.

The definition of an optimal solution for MOOP is different from that of SOOP that consist of picking the solution with the minimum(maximum) objective value on an ordered objective space, see Fig. 2.2. Meanwhile, MOOP solutions have to meet different objectives that, as previously mentioned, are generally contradictory. With that in mind, MOOP needs a different definition of comparison between solutions that treat all objective functions as equally important. The concept of Pareto optimality and the Pareto dominance criterion covers this necessity, and for this reason, it is used in this work as a ranking method between solutions in MOOP.

2.2.2.1 Pareto Dominance

A vector $\boldsymbol{u}$ dominates a second vector $\boldsymbol{v}$, denoted as $\boldsymbol{u} \preceq \boldsymbol{v}$, if and only if $\boldsymbol{u}$ is at least partially smaller than $\boldsymbol{v}$. In other words, $\forall j \in \{1, .., n\},\ u_j \leq v_j \wedge \exists j \in \{1, .., n\} : u_j < v_j$.

If a vector $\boldsymbol{u}$ does not dominate $\boldsymbol{v}$, $\boldsymbol{u}$ is considered as non-dominant, meaning that both vectors are equal in the Pareto dominance criterion denoted as $\boldsymbol{u} \npreceq \boldsymbol{v}$.

In light of the concept of Pareto dominance, given a pair of solutions $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ for a multi-objective function $\boldsymbol{F}$. $\boldsymbol{x}_1$ dominates $\boldsymbol{x}_2$ if $\boldsymbol{F}(\boldsymbol{x}_1) \preceq \boldsymbol{F}(\boldsymbol{x}_2)$. It means that $\boldsymbol{F}(\boldsymbol{x}_1)$ obtain less or equal results in all objectives when compared with $\boldsymbol{F}(\boldsymbol{x}_2)$ and with at least one of its objectives smaller than $\boldsymbol{F}(\boldsymbol{x}_2)$. If none of these cases are true then $\boldsymbol{x}_1 \npreceq \boldsymbol{x}_2$.

A concept that originates from the Pareto dominance criterion is the idea of non-dominant fronts in the objective space. A set of solutions $\boldsymbol{X}$ forms a front in the search space when its components are non-dominant solutions between themselves. In other words, $\boldsymbol{X}$ forms a front when $\boldsymbol{X} = \{\{\boldsymbol{x}_i, \boldsymbol{x}_j\} \in \boldsymbol{X} : \boldsymbol{x}_i \neq \boldsymbol{x}_j \wedge \boldsymbol{F}(\boldsymbol{x}_i) \npreceq \boldsymbol{F}(\boldsymbol{x}_j) \wedge \boldsymbol{F}(\boldsymbol{x}_j) \npreceq \boldsymbol{F}(\boldsymbol{x}_i)\}$.

A set of solutions $\boldsymbol{X}^*$ is considered as Pareto Optimal (PO) when it is a set of non-dominant solutions.

2.2.2.2 Pareto Optimality

A set of solutions $\boldsymbol{X}^* \in \mathrm{S}$ for a multi-objective function $\boldsymbol{F}$ is defined as PO by following the condition:

$$\boldsymbol{X}^* = \{\boldsymbol{x} \in \mathrm{S} : \nexists \boldsymbol{x}', \boldsymbol{F}(\boldsymbol{x}') \preceq \boldsymbol{F}(\boldsymbol{x})\}. \tag{2.4}$$

The set of non-dominant solutions in the objective space is called the Pareto Front (PF) $\boldsymbol{PF}^*$ and follows:

$$\boldsymbol{PF}^* = \{\boldsymbol{u} = \boldsymbol{F}(\boldsymbol{x}) : \boldsymbol{x} \in \boldsymbol{X}^*\}. \tag{2.5}$$

PF dominate all other fronts in the objective space.

Figure 2.4 illustrates a representation of the objective space with two objective functions $\boldsymbol{F} = (f_1, f_2)$, i.e., $\mathrm{O} \in \mathbb{R}^2$. The worst solutions have their images in the objective space represented in red. All other solutions colored in blue and magenta dominate the worst solutions.

The intermediate solutions have images represented in blue, and they are dominated by the magenta solutions and dominates the red ones. The best solutions are PO, and they have their images colored in magenta, forming the PF. PO solutions dominated all other solutions represent in blue and red and by drawing a curve intercepting the images for the PO non-dominant solutions a representation of the PF. Note that not only the PF can be drawn in this manner. The sets of worst and intermediate solutions can also have curves connecting the images of solutions that are not-dominant between themselves. It creates two other fronts, colored in red and blue, respectively, for the worst and intermediate solutions.

These fronts can be ranked denoting a metric of quality between solutions in a multi-objective context. For this example, PF has the smallest possible rank equal to one. The front in blue has a rank greater than one, in this case, two. Lastly, the front formed by the worst solutions has a rank greater than the rank of the blue front, in this case, three. By applying this ranking convention, a MOOP can be tackled by searching for the solutions that form fronts with smaller ranks that are in turn is a part or is as close as possible of PF.

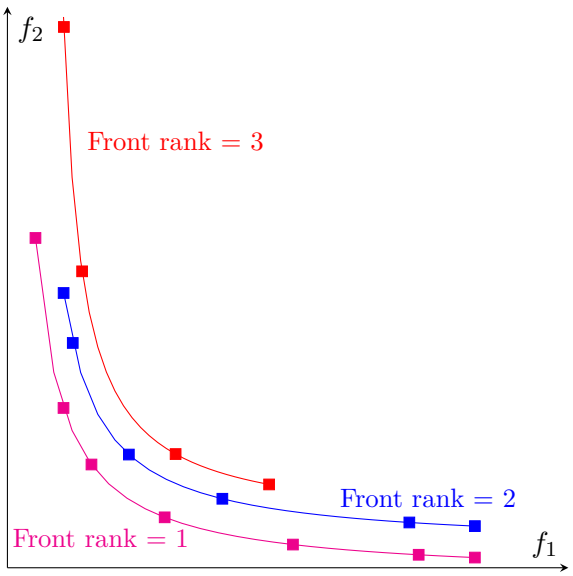


Figura 2.4 – Illustration of a Pareto Front in the objective space of a MOOP as well as representation other non-dominated and dominated solutions that composes other fronts.

2.3 Examples of Optimization Problems

This section presents examples of optimization problems that are later in this work used as benchmark functions.

2.3.1 Combinatorial/Discrete Problems

Combinatorial Optimization Problems (COPs) is a class of optimization problems in which its search space is composed of a finite number of possible solutions, and each of these solutions is associated with single or multiple objectives. In this work, COPs are also called discrete optimization problems.

There are classical algorithms to solve this class of problem optimally, for example, Branch & Bound [20] and Dynamic Programming [21]. Both techniques are greedy divide-and-conquer methods that divide the problem into smaller sub-problems and optimize them and then combine these sub-problems solutions back to form the solution for the problem in hand. Even though these algorithms are exact, in other words, always resulting in the optimal solution for the optimization problem in hand. They can not be applied for all COP problems since there are problems in which their solution components can not be subdivided and evaluated separately. These types of exact algorithm can not tackle problems with objective functions that can not evaluate individual components of the solution.

Another reason in which these algorithms can not be applied is that some problems can not be approached in a reasonable computer time due to their complexity. For example, a problem with so many components that the processing time for an optimal algorithm is prohibitive. In these cases, heuristic techniques present themselves as a possibility to tackle these problems while

being capable of obtaining near-optimal results.

2.3.1.1 N-Queens Placement Problem

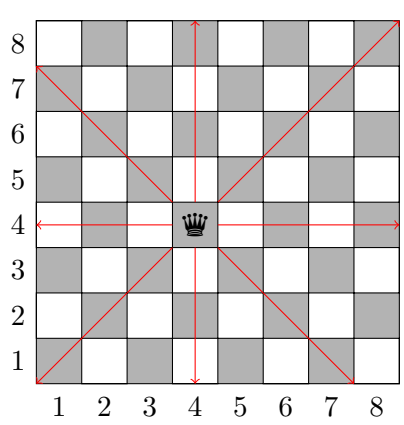
The n-queens placement problem [22] is an expansion of the 8-queens placement problem that aims to place n queen chess pieces in a $n \times n$ chessboard in a configuration in which no two queens can attack each other. In other words, the search of a configuration where no placed n points in an $n \times n$ grid has the same row, column, and diagonals. It is a well-known problem that is commonly used as a benchmark problem in different machine learning-based methods, for example, constructive backtracking algorithms [23] and evolutionary algorithm based meta-heuristics such as GA [24] and PSO [25].

The search process for a solution for the n-queen placement can be represented as an optimization problem

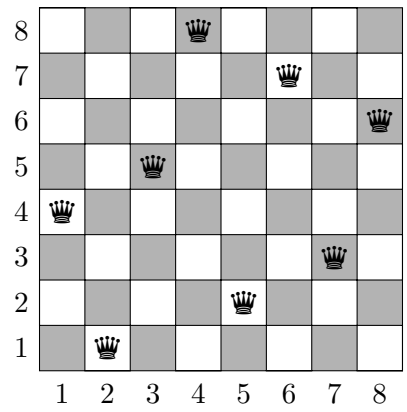
$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\ f_{attack}(\boldsymbol{x}), \tag{2.6}$$

where each candidate placement solution is encoded as a n-dimensional integer vector $\boldsymbol{x}$ and the goal is to minimize a function $f_{attack}(\boldsymbol{x}) : \mathrm{S} \rightarrow \mathbb{Z}^+$ that maps the subset S of n^n possibilities, i.e. a subset of the total $\frac{(n^2)!}{n!(n(n-1))!}$ possible placements, and returns a positive integer with the number of attacking movements for all n queen pieces. If there is no attacking movement, $f_{attack}(\boldsymbol{x})$ returns 0 and $\boldsymbol{x}$ is a placement solutions that satisfies the n-queens placement problem.

Each of the jth component of a solution $\boldsymbol{x}$ represents the placement of a queen piece onto the jth row of the chessboard, and the positive integer held by this component $x_j \in [1, n]$ depicts the column of this placement. In this manner, each queen is placed at the position (j, x_j) of the chessboard. Figure 2.5a illustrates the placement of a queen at (4, 4) position on a 8×8 board and Fig. 2.5b shows a queen placement with no queen attacking each other.



(a) Example of movement of a queen chess piece.



(b) Example of a placement $\boldsymbol{x}$ where $f_{attack}(x) = 0$

Figura 2.5 – Illustration of queen chess pieces interaction on a 8×8 board.

The function f_{attack} calculates the number of attack movements between queen pieces using an algorithm expressed in pseudo-code form in Algorithm 1. It uses two sets $\boldsymbol{dgp}$ and $\boldsymbol{dgs}$ to identify

attacks on the principal and secondary diagonals, respectively. By using a sorting algorithm, such as the *QuickSort* [26] on both sets, and then comparing these sets adjacent elements, it is possible to identify queen pieces in the same diagonal. The same principle can be used directly onto a placement vector $\boldsymbol{x}$ to count the number of pieces sharing the same rows.

Algorithm 1 N-Queens Attack Counter (f_{attack})

INPUT: Pieces Placement ($\boldsymbol{x}$)
OUTPUT: Number of Attacks ($attacks$)

1: **procedure** $NQueensAttacks(\boldsymbol{x})$
2: $attacks = 0$
3: $\boldsymbol{dgp} \leftarrow \emptyset$
4: $\boldsymbol{dgs} \leftarrow \emptyset$
5: **for** $j = 1$ to $|\boldsymbol{x}|$ **do**
6: $\boldsymbol{dgp} \leftarrow x_j - j$
7: $\boldsymbol{dgs} \leftarrow (|\boldsymbol{x}|+1) - x_j - j$
8: **end for**
9: $QuickSort(\boldsymbol{dgp})$
10: $QuickSort(\boldsymbol{dgs})$
11: $QuickSort(\boldsymbol{x})$
12: **for** $j = 2$ to $|\boldsymbol{x}|$ **do**
13: **if** $dgp_j = dgp_{j-1}$ **then**
14: $attacks = attacks + 1$
15: **end if**
16: **if** $dgs_j = dgs_{j-1}$ **then**
17: $attacks = attacks + 1$
18: **end if**
19: **if** $x_j = x_{j-1}$ **then**
20: $attacks = attacks + 1$
21: **end if**
22: **end for**
23: **return** $attacks$
24: **end procedure**

2.3.2 Continuous Problems

Continuous Problems are part of a subset of optimization problems with search-space formed by infinitely many solutions composed of only real numbers. For example, an optimization problem with solutions composed of d-dimensional real vectors $\boldsymbol{x} \in \mathbb{R}^d$.

In science and engineering, the majority of optimization problems is part of this class. Similarly to COP, there are classical methods that can be applied for this type of problems. For example, if the objective function is known analytically, differentiable, and unimodal, the best approach is to use a gradient descent based method. However, if that is not the case, the gradient descent may be rendered unusable or with poor performance. Therefore, search-based heuristic methods can be useful to solve this class of problems as well.

The majority of benchmark functions for heuristics used in numerical optimization competitions (SOOP CEC2017 [27] and MOOP CEC2017 [28]) fall into this category. The benchmark functions used in this work will be present in the next subsections for both single- and multi-objective benchmark optimization functions.

2.3.2.1 Mobile Robot Probabilistic Scan Matching in SLAM Applications

In the context of autonomous mobile robots, Simultaneous Localization and Mapping (SLAM) is a well-known problem defined as a process to concurrently construct an environmental map

(**mapping**) and estimate the robot position inside this map (**localization**), as described by [29]. SLAM problems have a high dependency on methods to correctly associate new observations with previous known sensor readings. This process of matching old and new data from scan sensors is known as *scan matching* and it is a front-end algorithm used by both the localization and mapping processes in SLAM.

The main focus of scan matching algorithms is to estimate the pose displacement of a robot between two consecutive sensor readings.

Probabilistic scan match algorithms can be viewed as an optimization problem to maximize the posterior distribution $p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{u}, \boldsymbol{m}, \boldsymbol{z})$ given a robot pose in space $\boldsymbol{x}_t$ at time t. Each one of these poses are expressed as a 3-dimensional vector (x_t, y_t, θ_t) where (x_t, y_t) represents the robot's position in a 2D map and θ_t represents its heading angle. $\boldsymbol{u}$ represents a prior knowledge that can be an odometer reading or a control input, $\boldsymbol{m}$ is the environmental knowledge as a map, and $\boldsymbol{z}$ is a set of the latest sensor measurements. This optimization problem is represented in Eq. (2.7). By applying Bayes' rule, this distribution can be divided into two terms: (a) the observation model $p(\boldsymbol{z}|\boldsymbol{x}_t, \boldsymbol{m})$ representing the likelihood of the robot to make a measurement $\boldsymbol{z}$, if its pose and map is known; and (b) the motion model $p(\boldsymbol{x}_t, \boldsymbol{x}_{t-1}, \boldsymbol{u})$ that is the likelihood for the robot to be in a specific pose given its previous pose $\boldsymbol{x}_{t-1}$ and prior knowledge $\boldsymbol{u}$;

$$\boldsymbol{x}_t^* = \underset{\boldsymbol{x}_t}{\operatorname{argmax}}\ p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{u}, \boldsymbol{m}, \boldsymbol{z}) \propto \underset{\boldsymbol{x}_t}{\operatorname{argmax}}\ p(\boldsymbol{z}|\boldsymbol{x}_t, \boldsymbol{m}) \cdot p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{u}) \tag{2.7}$$

Assuming that a set of point readings z obtained from a laser scan sensor forms $\boldsymbol{z}$, and each of these points is uncorrelated to each other, the observation model can be reduced into the form present in Eq. (2.8).

$$p(\boldsymbol{z}|\boldsymbol{x}_t, \boldsymbol{M}) = \prod_{z \in \boldsymbol{z}} p(z|\boldsymbol{x}_t, \boldsymbol{m}) \tag{2.8}$$

Given that the prior probability $p(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{u}, \boldsymbol{m}, \boldsymbol{z})$ is known and a constant value, from now on represented as $p(\boldsymbol{x})$, and, since maximizing the posterior is the same as minimizing the negative of the log-posterior, the optimization can be rewritten as shown in Eq. (2.9).

$$\boldsymbol{x}_t^* = \underset{\boldsymbol{x}_t}{\operatorname{argmin}} - log\Big(p(\boldsymbol{x}) \prod_{z \in \boldsymbol{z}} p(z|\boldsymbol{x}_t, \boldsymbol{m})\Big) \tag{2.9}$$

Figure 2.6 shows a representation of the search-space and likelihood values for the robot to be in a range of possible poses in the map relative to its current position. It is possible to visualize the presence of multiple local optima when varying θ. This characteristic that makes this problem difficult to tackle with simple gradient descent method. Not mentioning that, as previously stated, since the SLAM process is executed continuously, the optimization method is processed every new sensor reading. In cases with a large number of possible solutions, a brute-force method may take too much computation time or resources.

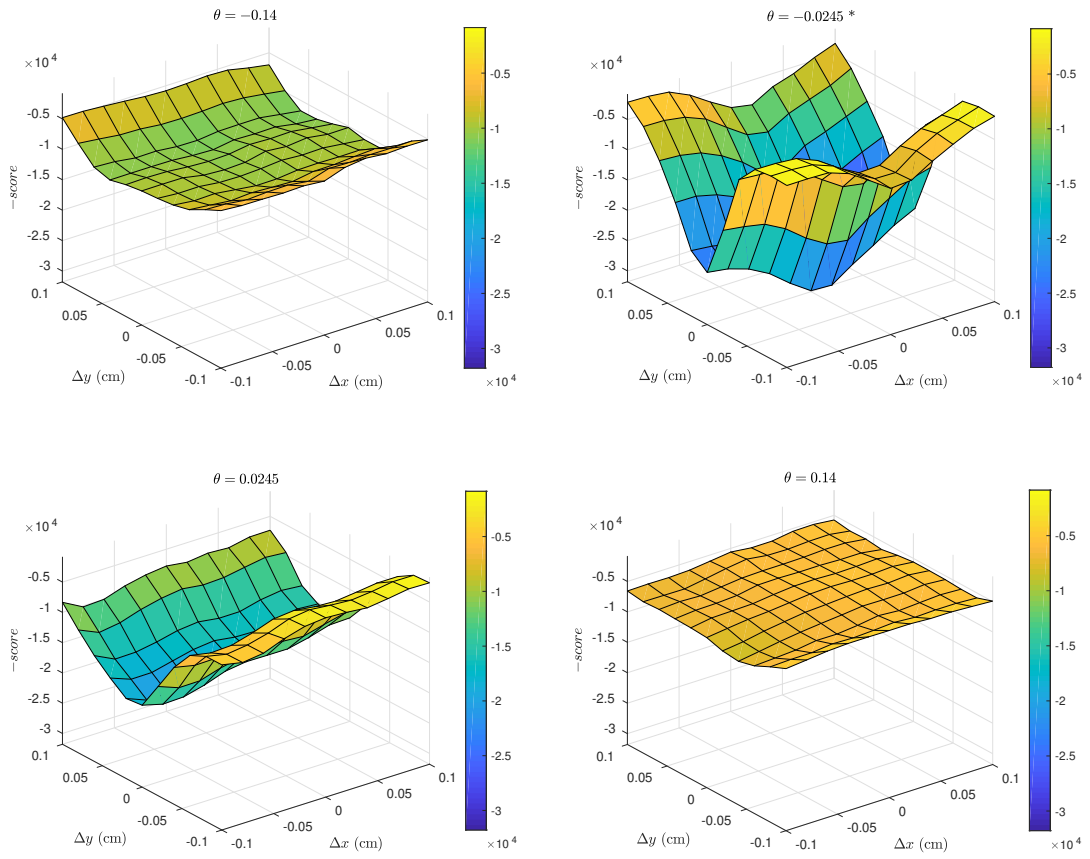


Figura 2.6 – 2D plots of likelihood values given robot positions in the map with fixed heading angles. In this example setting, the optimal point is found at heading angle $\theta = -0.0245$.

2.3.2.2 Single-Objective Benchmark Functions

The single-objective benchmark fitness functions used are listed in Table 2.1 and they include n-dimensional problems with different characteristics such as unimodal, flat, and multimodal, separable, and non-separable. A common characteristic of SOOP benchmark functions used in this work is that their variable components are homogeneous, i.e., each component boundaries are the same.

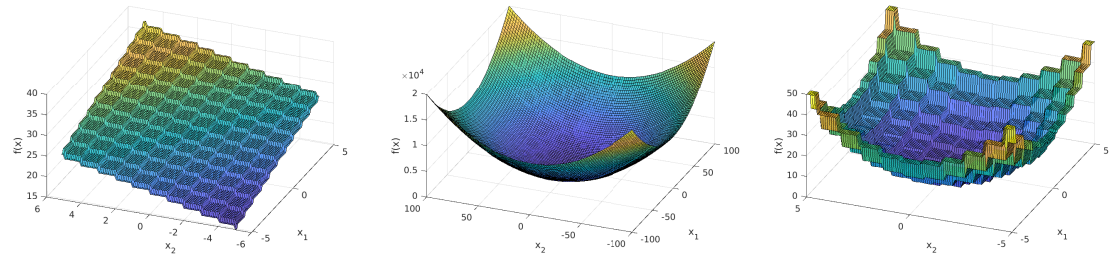
A function is considered to be unimodal when it has a single optimum point with no single local optimum point in its search-space while multimodal functions have countable or uncountable many local optima points with only a single optimum point. Figure 2.7 illustrates these characteristics by plotting each SOOP benchmark functions in a 2-dimensional search space. An example of a unimodal function is the *Sphere*, and an example of a multimodal function is the *Rastrigin* or *StepInt*. The use of multimodal function tests the search-based optimization algorithms capabilities to avoid convergence in local optima points while exploring the search space. Flat functions, such as *Stepint*, has another characteristic that makes them difficult to solve: their flat regions does not present any gradient information for the algorithms to aim their search towards better regions.

A separable function $f : \mathbb{R}^d \to \mathbb{R}$ can be expressed as the sum of d one variable functions. Meanwhile, non-separable can not be expressed similarly, since its input vector components have an interdependence relationship between them. Optimization of non-separable problems is generally more complicated than separable ones.

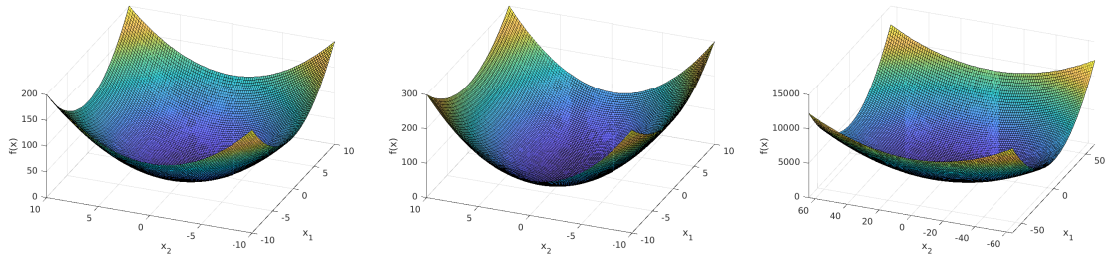
Another important note is the multidimensional characteristics of these benchmarks. As the number of dimensions of a problem increases, so does the volume of solutions inside of the search space turning the search process for the optimal point more complex. This problem is called the "**curse of dimensionality**". In this context, an optimization algorithm that obtains excellent results in a few dimensions setting may deteriorate its optimization performance as the number of dimensions increase.

Tabela 2.1 – Benchmark functions used. Features: **U** - Unimodal, **M** - Multimodal, **S** - Separable, **N** - Non-Separable, and **F** - Flat

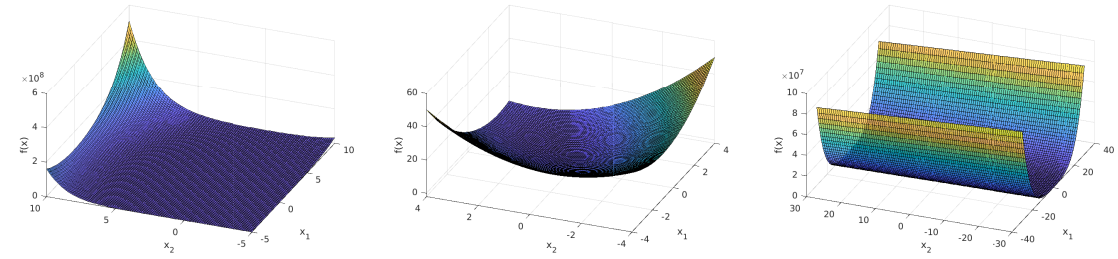
Name	**Range**	**Features**	**Equation**
StepInt	$[-5.12, 5.12]$	**U-F-S**	$f(\boldsymbol{x}) = 25 + \sum_{j=1}^{d} \lceil x_j \rceil$
Step	$[-100, 100]$	**U-F-S**	$f(\boldsymbol{x}) = \sum_{j=1}^{d} (\lfloor x_j + 0.5 \rfloor)^2$
Sphere	$[-100, 100]$	**U-S**	$f(\boldsymbol{x}) = \sum_{j=1}^{d} x_j^2$
SumSquares	$[-10, 10]$	**U-S**	$f(\boldsymbol{x}) = \sum_{j=1}^{d} j x_j^2$
RotatedEllipsoid	$[-64, 64]$	**U-S**	$f(\boldsymbol{x}) = \sum_{j_1=1}^{d} \sum_{j=1}^{j_1} x_j^2$
Zakharov	$[-5, 10]$	**U-N**	$f(\boldsymbol{x}) = \sum_{j=1}^{d} x_j^2 + (\sum_{j=1}^{d} 0.5 j x_j^2)^2 + (\sum_{j=1}^{d} 0.5 j x_j^2)^4$
Trid	$[-n^2, n^2]$	**U-N**	$f(\boldsymbol{x}) = \sum_{j=1}^{d} (x_j - 1)^2 - \sum_{j=2}^{d} x_j x_{j-1}$
Rosenbrock	$[-30, 30]$	**U-N**	$f(\boldsymbol{x}) = \sum_{j=1}^{d-1} \left(100(x_{j+1} - x_j^2)^2 + (x_j - 1)^2\right)$
Rastrigin	$[-5.12, 5.12]$	**M-S**	$f(\boldsymbol{x}) = \sum_{j=1}^{d} \left(x_j^2 - 10\cos(2\pi x_j) + 10\right)$
Schwefel	$[-500, 500]$	**M-S**	$f(\boldsymbol{x}) = 418.9829d - \sum_{j=1}^{d} -x_j \sin(\sqrt{\lvert x_j \rvert}))$
Michalewicz	$[0, \pi]$	**M-F-S**	$f(\boldsymbol{x}) = -\sum_{j=1}^{d} \sin(x_j)(\sin(j\frac{x_j^2}{\pi}))^{20})$
Griewank	$[-600, 600]$	**M-N**	$f(\boldsymbol{x}) = \frac{1}{4000} \sum_{j=1}^{d} x_j^2 - \prod_{j=1}^{d} \cos(\frac{x_j}{\sqrt{j}}) + 1$
Ackley	$[-32, 32]$	**M-N**	$f(\boldsymbol{x}) = -20e^{(-0.2\sqrt{\frac{1}{d}\sum_{j=1}^{d} x_j^2})} - e^{(\frac{1}{d}\sum_{j=1}^{d} \cos(2\pi x_j))} + 20 + e$



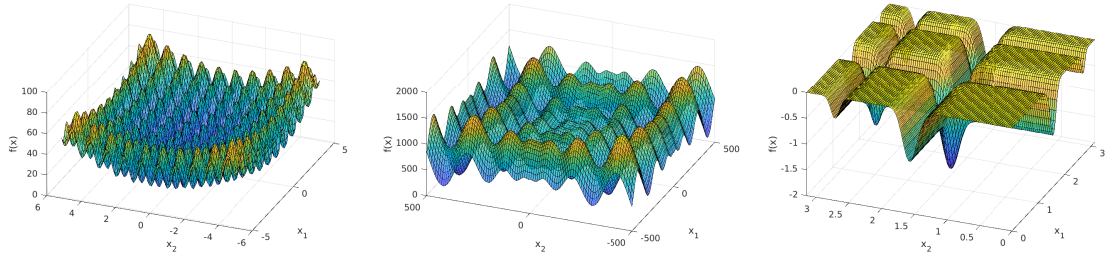
(a) StepInt (b) Step (c) Step zoomed-in $[-5, 5]$



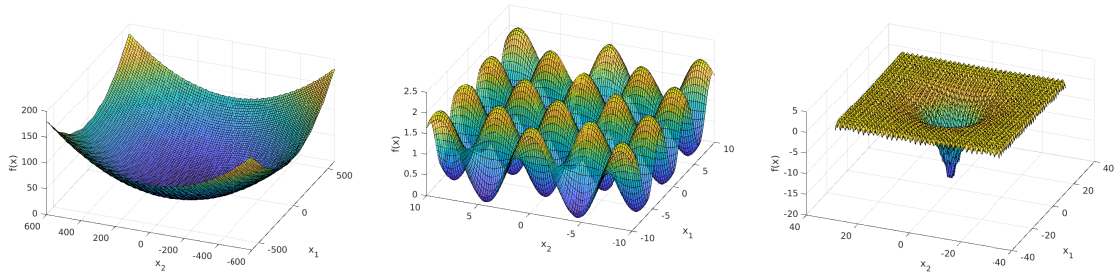
(d) Sphere (e) SumSquares (f) Rotated Ellipsoid



(g) Zakharov (h) Trid (i) Rosenbrok



(j) Rastrigin (k) Schwefel (l) Michalewicz



(m) Griewank (n) Griewank zoomed-in $[-10, 10]$ (o) Ackley

Figura 2.7 – Plot of the used single-objective benchmark functions in a 2-dimensional search space.

2.3.2.3 Multi-Objective Benchmark Functions

The multi-objective functions used are listed in Table 2.2 and they include problems with d-dimensional search spaces with homogeneous similarly to SOOP benchmark problems. Other shared characteristics between these two types of benchmark problems concern their search space. The multi-objective benchmark functions used are also unimodal, multimodal, separable, and non-separable.

Apart from their search space characteristics, these MOOP also have characteristics related to their PF formed by their PO solutions. In this work, these characteristics will be presented based on the geometry formed by PF. All MOOP benchmark functions will be presented with two or three objectives to facilitate the visualization of the geometry. However, some of these problems can be used with an objective space with m-dimensions.

The used MOOP benchmark problems have the following geometric characteristics in their PF: (a) linear: the PF forms a hyper-plane in the objective space; (b) concave: the PF forms a concave curve in the objective space; (c) convex: the PF forms a convex curve in the objective space; (d) mixed concave and convex: the PF forms a mixture of concave and convex regions; (e) disconnected: the PF is composed of disconnected regions.

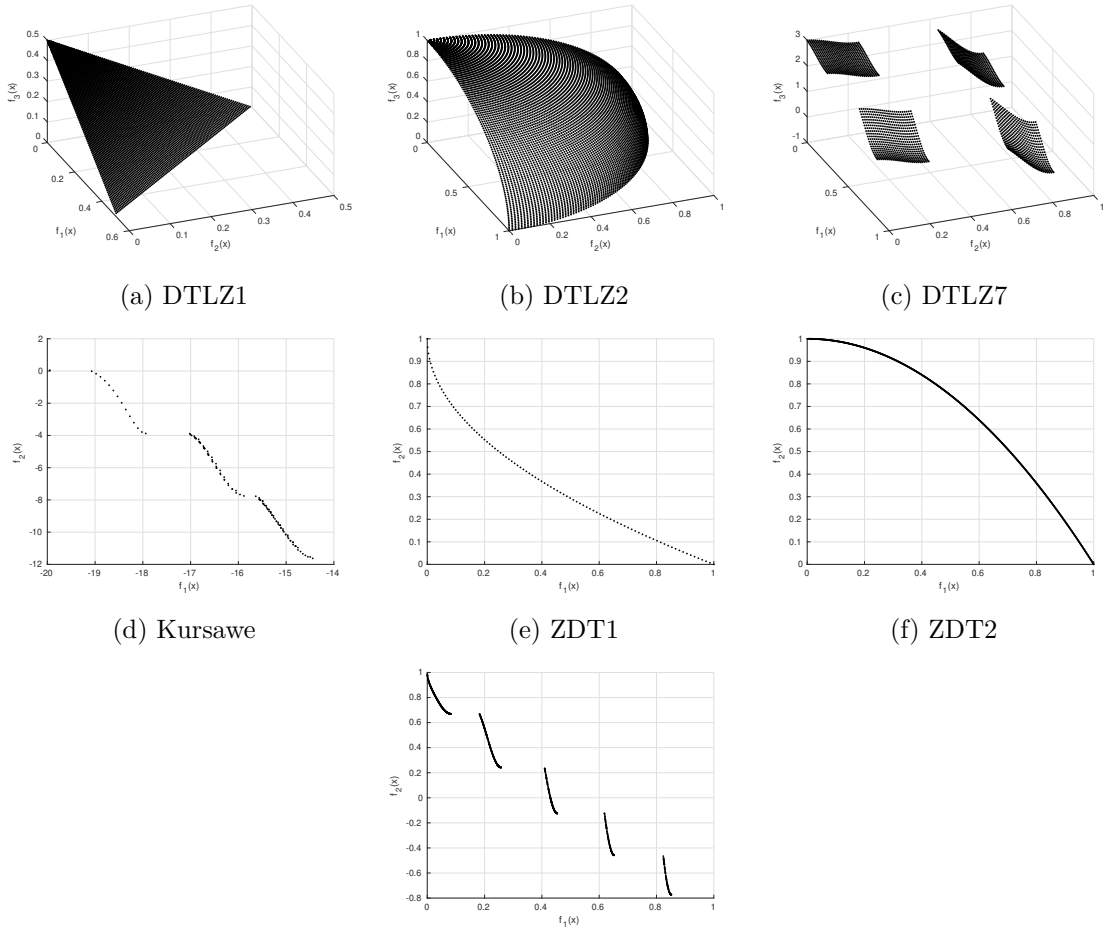
At last, MOOP benchmark functions also suffer from the **curse of dimensionality** in many cases, even more than SOOP benchmark functions. Since solutions in large dimensions are mapped onto multidemensional images.

Figure 2.8 illustrates the geometry for the PF of the benchmark MOOP.

2.4 Conclusions of the Chapter

This chapter briefly introduces the mathematical framework used in optimization problems, as well as the process to model and encode a real-world problem as an optimization problem. The terminology presented in this chapter is essential to understand other aspects of this work, for example, the modeling of an RTNoC task mapping process as an optimization problem in Chapter 3 and the use of search-based meta-heuristics to solve optimization problems in Chapter 4.

This chapter also contextualizes the optimization problems in science and engineering application, including presenting examples and benchmark problems to evaluate meta-heuristics algorithms that will be revisited and used in the experiments in Chapter 7.



(a) DTLZ1 (b) DTLZ2 (c) DTLZ7

(d) Kursawe (e) ZDT1 (f) ZDT2

(g) ZDT3

Figura 2.8 – Plot of the used multi-objective benchmark functions pareto frontiers.

Tabela 2.2 – MOOP Benchmark functions used. Search Space Features (**SS**): **U** - Unimodal, **M** - Multimodal, **S** - Separable, **N** - Non-Separable, and **F** - Flat. Objective Space Features (**OS**): **L** - Linear, **Ce** - Concave, **Cx** - Convex, **Mx** - Mixed Concave/Convex, and **D** - Disconnected.

Name	**Range**	**SS**	**OS**	m	**Equation**
DTLZ1	$[0, 1]$	**M-N**	**L**	3	$g(\boldsymbol{x}) = 100((d-2) + \sum_{i=m}^{d}(x_i - 0.5)^2 - \cos(20\pi * (x_i - 0.5)))$ $f_1(\boldsymbol{x}) = 0.5x_1x_2(1 + g(\boldsymbol{x}))$ $f_2(\boldsymbol{x}) = 0.5x_1(1 - x_2)(1 + g(\boldsymbol{x}))$ $f_3(\boldsymbol{x}) = 0.5(1 - x_1)(1 + g(\boldsymbol{x}))$ $\boldsymbol{F}(\boldsymbol{x}) = \{f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), f_3(\boldsymbol{x})\}$
DTLZ2	$[0, 1]$	**M-N**	**Ce**	3	$g(\boldsymbol{x}) = \sum_{i=m}^{d}(x_i - 0.5)^2$ $f_1(\boldsymbol{x}) = \cos(\frac{\pi}{2}x_1)\cos(\frac{\pi}{2}x_2)(1 + g(\boldsymbol{x}))$ $f_2(\boldsymbol{x}) = \cos(\frac{\pi}{2}x_1)\sin(\frac{\pi}{2}x_2)(1 + g(\boldsymbol{x}))$ $f_3(\boldsymbol{x}) = \sin(\frac{\pi}{2}x_1)(1 + g(\boldsymbol{x}))$ $\boldsymbol{F}(x) = \{f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), f_3(\boldsymbol{x})\}$
DTLZ7	$[0, 1]$	**M-N**	**Cx**	3	$g(\boldsymbol{x}) = 1 + \frac{9}{22}\sum_{i=m}^{d} x_i$ $h(\boldsymbol{x}) = 3 - \sum_{i=1}^{2}\left(\frac{x_i}{(1+g(\boldsymbol{x}))}(1 + \sin(3\pi x_i))\right)$ $f_1(\boldsymbol{x}) = x_1$ $f_2(\boldsymbol{x}) = x_2$ $f_3(\boldsymbol{x}) = (1 + g(\boldsymbol{x}))h(\boldsymbol{x})$ $\boldsymbol{F}(x) = \{f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), f_3(\boldsymbol{x})\}$
Kursawe	$[-5, 5]$	**U-S**	**Mx-D**	2	$f_1(\boldsymbol{x}) = -10\exp(0.2)\sum_{i=1}^{d-1}\sqrt{(x_i^2 + x_{(i+1)}^2)}$ $f_2(\boldsymbol{x}) = \sum_{i=1}^{n}\lvert x_i\rvert^{0.8} + 5\sin(x_i^3)$ $\boldsymbol{F}(x) = \{f_1(\boldsymbol{x}), f_2(\boldsymbol{x})\}$
ZDT1	$[-5, 5]$	**U-N**	**Cx**	2	$g(\boldsymbol{x}) = 1 + \frac{9}{d-1}\sum_{i=m}^{d} x_i$ $f_1(\boldsymbol{x}) = x_1$ $f_2(\boldsymbol{x}) = g(\boldsymbol{x})(1 - \sqrt{\frac{x_1}{g(\boldsymbol{x})}})$ $\boldsymbol{F}(x) = \{f_1(\boldsymbol{x}), f_2(\boldsymbol{x})\}$
ZDT2	$[-5, 5]$	**U-N**	**Ce**	2	$g(\boldsymbol{x}) = 1 + \frac{9}{d-1}\sum_{i=m}^{d} x_i$ $f_1(\boldsymbol{x}) = x_1$ $f_2(\boldsymbol{x}) = g(\boldsymbol{x})\left(1 - \left(\frac{x_1}{g(\boldsymbol{x})}\right)^2\right)$ $\boldsymbol{F}(x) = \{f_1(\boldsymbol{x}), f_2(\boldsymbol{x})\}$
ZDT3	$[0, 1]$	**M-N**	**Cx-D**	2	$g(\boldsymbol{x}) = 1 + \frac{9}{d-1}\sum_{i=m}^{d} x_i$ $f_1(\boldsymbol{x}) = x_1$ $f_2(\boldsymbol{x}) = g(\boldsymbol{x})$ $\boldsymbol{F}(x) = \{f_1(\boldsymbol{x}), f_2(\boldsymbol{x})\}$

3 REAL-TIME NETWORK-ON-A-CHIP BASED MPSOC

This chapter introduces SoCs, NoCs, and RTSs. Information present here contextualizes the task mapping of an RTA onto an MPSoC as an optimization problem. Section 3.1 presents SoCs. Section 3.2 presents communication architectures used in SoCs. Section 3.3 introduces NoCs and their components and possible design choices to then, Section 3.4 to define RTNoC as a special set of NoCs used in this work. Section 3.7 presents RTSs. Section 3.6 presents the framework used to model multiprocessor RTS, including concepts about scheduling algorithms. Section 3.7 models an RTNoC as a particular case of multiprocessor RTS and presents analytical methods to evaluate such a system. Section 3.8 presents the task mapping as an optimization problem. Finally, Section 3.9 concludes the chapter and contextualize it with other parts of the work.

3.1 System-on-a-Chip (SoC)

The following classes can categorize SoCs, depending on their application:

- *General-purpose on-chip multiprocessors*: they are high-performance chips designed to support general-purpose software applications. Example, modern CPUs with multiple processors inside of the same silicon die generally fall into this category.
- *Application-specific SoCs*: they are dedicated chips for a single very specific application with specialized architectures and components. They generally are programmable, but all details of their software applications are known *a priori*.
- ***SoC platforms***: they are application-specific chips for a family of applications in a particular domain. Their design is versatile to attend multiple applications and could be used in different families of embedded devices and end up in a large volume of products.

This work is intended to be used in the latter class of SoCs, more specifically, in Multi-/Many-Processors SoC platforms.

3.1.1 Multi-/Many- Processors System-on-a-Chip (MPSoC)

Multi-/Many-Processors System-on-a-Chip (MPSoC) [30] [31] is a type of SoC that is composed of tens or even hundreds of processor cores inside of a single chip. Systems of this type are ideal for high-processor computing applications, being capable of tackling considerable complex scientific and engineering problems using massive parallelism approaches. However, in the past decade, MPSoC has been introduced into embedded applications by companies due to the increase of transistor count in ICs and the rise of processing demanding applications for embedded platforms such as, for example, image and video processing in security devices, addressed to automotive and aviation critical applications, robotics, etc. This trending of high demanding applications in embedded devices seems to be in the market to stay and keep increasing the importance of MPSoCs. An example of application in which MPSoCs are essential for the near future is in the field of autonomous vehicles and driver assistance systems [32].

3.2 Communication Architectures

The on-chip communication architecture is the subsystem responsible for connecting all components inside of the chip [10] [33]. As the number of on-chip Processing Element (PE) increases, the importance of communication architecture grows along. Because in an SoC platform with dozens of components, such as MPSoCs, the complexity of the on-chip communication architecture causes the increase of delays and power consumption.

Modern SoCs has in its disposal two main paradigms to deal with the design of their communication architectures: (a) buses and (b) network-on-a-chip.

3.2.1 Buses

A bus is an on-chip communication that consists of a shared channel in which components inside of the SoC exchange their data. There are two main types for Buses:

- *Single Shared Bus*: it is the most straightforward approach, where a single bus channel connects all components. Only one component (a master) may transmit data in a given time, while all the other receives its data (slaves).
- *Hierarchical Bus:* it has multiple shared buses connected hierarchically through bridges. This approach permits the increase of bandwidth of the connection between high-performance components that constantly transmit data between them and use smaller buses to connect peripheral components.

The advantages of bus-based on SoCs are: (a) their simplicity, (b) substantial standardization inside of the industry, and (c) the intrinsic broadcast nature for the data messages transmission.

While the main disadvantage is that since all components share the same "communication channel", the global arbitration system grows more complex and requiring more power with longer

on-chip wires as the number of components increases inside of the SoC. It makes its communication bandwidth not scale together with the number of components of the system because there is a single channel where only one component may transmit data in a given time. Figure 3.1 illustrates a system that uses the ARM Advanced Microcontroller Bus Architecture (AMBA). AMBA is an example of hierarchical bus communication architecture widely used in the industry.

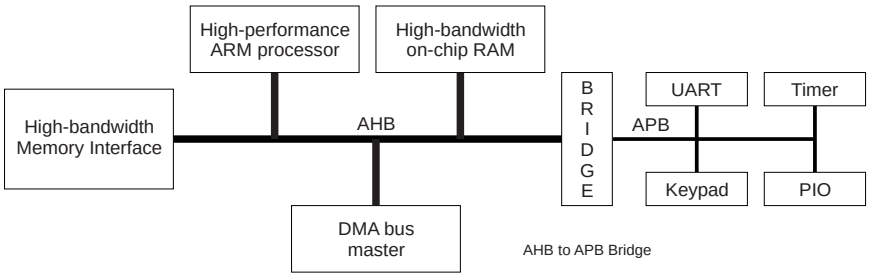


Figura 3.1 – Illustration of an AMBA communication architecture extracted from [3], where AHB stands for Advanced High-performance Bus for fast components, and APB stands for Advanced Peripheral Bus for peripheral and low-power components.

3.2.2 Network-on-a-Chip

Network-on-a-Chip (NoC) [8] [11] is an on-chip communication paradigm inspired by computing networks where the internal SoC components connect each other using routers, which are in turn connected in a myriad of topological approaches. Since routers connect all components, the arbitration decisions are distributed inside of the NoC, allowing for parallelism between the communication components and increasing the scalability of the system as the number of components increase [34]. Section 3.3 will further provide concepts and information related to NoCs.

3.3 Network-on-a-Chip (NoC)

3.3.1 Components

An NoC communication architecture is structured as a series of interconnected tiles with each tile containing a group of the following components: (a) links, (b) PEs, (c) Network Interfaces (NIs), and (d) routers.

Figure 3.2 illustrates an example of a simple NoC with tiles in green containing unidirectional links as black arrows, routers as blue rectangles, PEs as yellow squares, and NIs attached to PEs connecting routers as red rectangles.

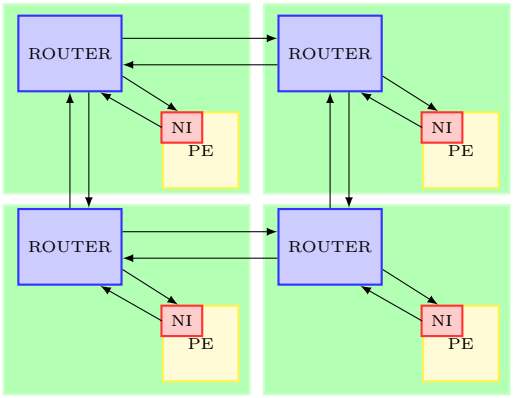


Figura 3.2 – Representation of a mesh-grid 2×2 NoC.

3.3.1.1 Links

Links are the physical wire connection between routers. It is possible to classify links properties under two lenses: (a) low-level abstraction parameters such as the type of repeaters used to amplify the signal, as well as (b) high-level abstraction such as the width of bits for the links when transmitting data.

3.3.1.2 Processing Elements (PE)

Processing Elements (PEs) are the intellectual property components that are modules of the system. Depending on the system, these elements can be processor cores, digital signal processors, graphical processor units, and application-specific modules. For the NoC, PEs are components that produce and consume messages regardless of the NoC design. Since an SoC based on an NoC is modularized, a PE implementation is independent of the communication architecture increasing the re-usability of PEs designs inside of different NoCs.

In MPSoC, PEs connected to the network is in its majority processor cores. In this work, a PE is a processor core connected to a local memory that sends and receives data from other processor cores that has the same properties. In other words, different PEs hold the same processing capabilities and local memory sizes.

3.3.1.3 Network-Interface (NI)

Network Interfaces (NIs) are the components responsible for wrapping the data transmitted independently of the type of PE used in a format suitable for NoC design choices, in terms of communication protocols. One of the main attributions of a NI is to provide a layer of hardware abstraction that separates the communication and processing aspects of the system. The goal for NI use is that for a PE inside of an SoC, the NoC as a communication architecture is invisible. NIs are responsible, for example, to packetize and un-packetize data, depending on the switching policy used and implement end-to-end error correction of the messages.

3.3.1.4 Routers

Routers are the components responsible for controlling the flux of data transiting inside of the NoC. Routers implement communication protocols, such as the arbitration policy when messages compete for shared resources (*links*), routing strategies to decide the path in which a message should transverse through the network, and switching strategy.

Figure 3.3 shows a representation of a router architecture for an NoC that have: (a) a packet switching strategy such as the wormhole switching, (b) virtual channels in the buffers of input links, and (c) credit-based flow control.

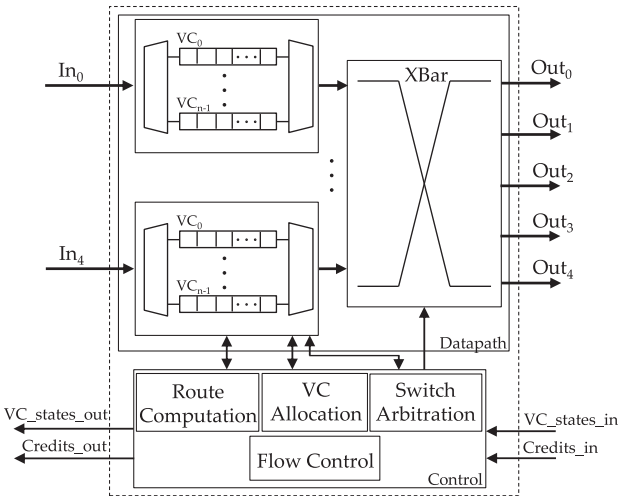
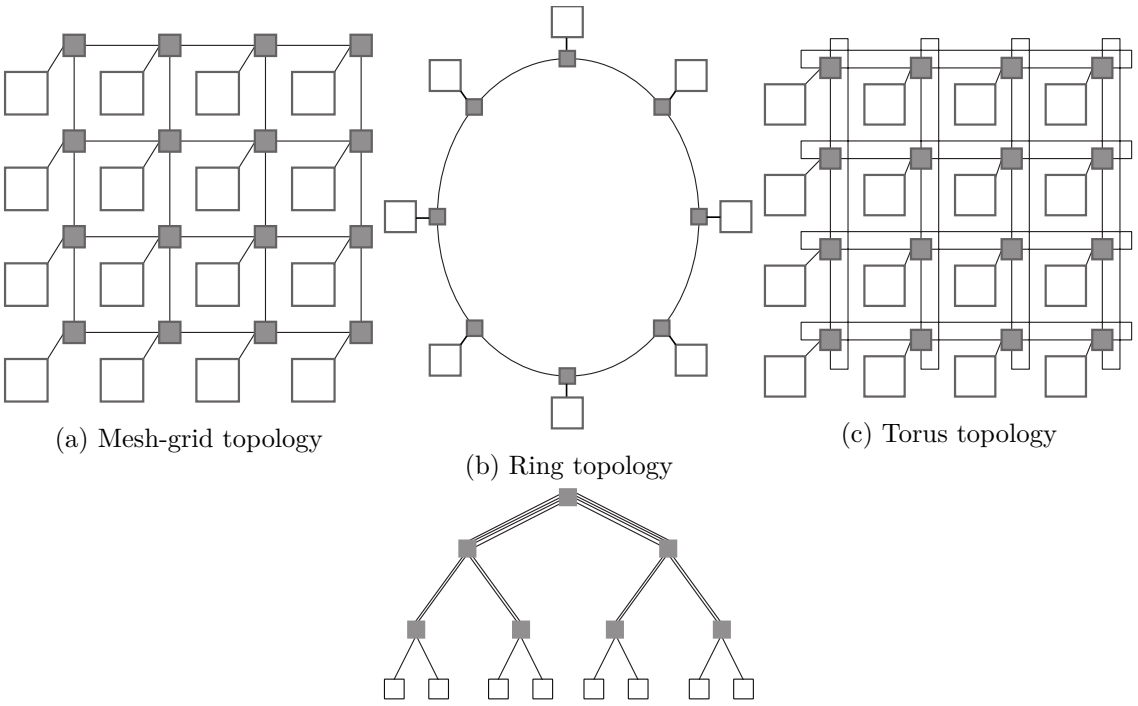


Figura 3.3 – Illustration of an example of a router architecture for a packet switching NoC with virtual channels. (Image extracted from [4]).

3.3.2 Connection Topology

Its the topological organization in which the tiles are connected and organized inside of the NoC. There are two classes of NoC topologies: (a) direct and (b) indirect networks. In *direct networks*, each router is connected to a PE and then connected directly to other neighboring routers. The advantage of this type topology is that, when implemented in a regular fashion, such as a mesh-grid or a torus, it is capable of scalability of the system bandwidth because as the number of PE grows the number of communication architecture components grows as well. *Indirect networks* have routers that do not have a single PE connected to it. Instead, it may connect a set of PEs. Compared to direct networks, this type of topology sacrifices their scalability and performance so as reducing the area used by on-chip communication components.

Figure 3.4 displays examples of NoC with components organized in different topologies where routers are gray squares, and white squares represent PEs. Figure 3.4a shows a direct network mesh-grid topology, Fig. 3.4c displays a direct network using a torus topology, Fig. 3.4b shows a direct network with ring topology, and, last, Fig. 3.4d shows an indirect network with fat-tree topology.



(a) Mesh-grid topology

(b) Ring topology

(c) Torus topology

(d) Fat-tree topology

Figura 3.4 – Illustration of multiple NoC topological organizations, where gray squares represent routers and white squares represent PEs. These images where adapted from [1].

This work focus on NoCs with 2D mesh-grid topology [35], as shown in Fig. 3.4a, due to the following characteristics present in this type of topology: (a) it is a direct and regular network that scales with the number of PEs connected in the NoC; (b) It has short links with the same length that provide similar communication latencies for any link in the network resulting in predictability for the system.

3.3.3 Routing Algorithm

The routing algorithm decides the route composed by links and routers in which a message travels inside of the network from a source PE to a destination one. Routers are the NoC components responsible for the implementation of the routing algorithm. There are two types of routing algorithms: (a) deterministic and (b) adaptive.

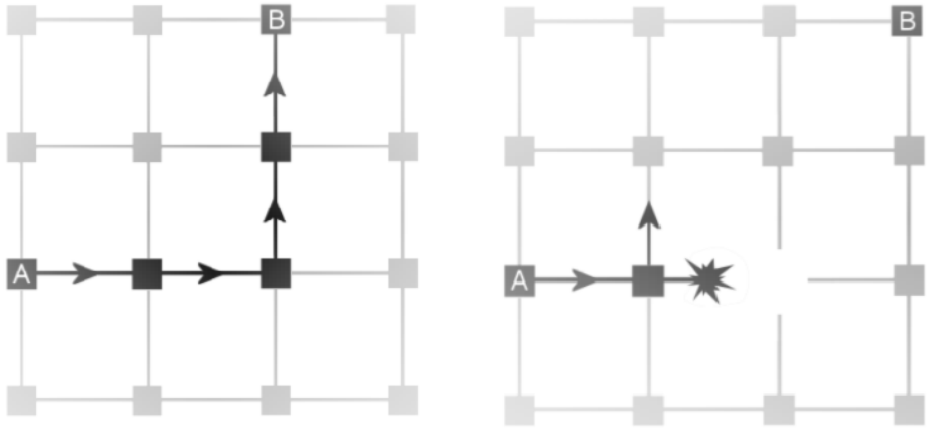
3.3.3.1 Deterministic Routing

In deterministic routing algorithms, any message with the same source and destination PEs always uses the same route for their transmission. The advantages of deterministic algorithms are their predictability, that makes it ideal for real-time systems, and simplicity, that reduces the logic necessary and the router size, and, finally, the guarantee of ordering of delivery for data in a message.

3.3.3.2 Adaptive Routing

In adaptive routing algorithms, the router uses information about the network conditions to adapt the path in which messages use to traffic inside of the NoC in a way that any two messages sharing the same source and destination PE may have different routes. Adaptive algorithms are capable of distributing the use of NoC components better. However, the problems with this approach for routing is their greater complexity causing large routers and lesser predictability than its deterministic counterpart.

Figure 3.5 presents the transmission of a message being transmitted between PEs A and B in a mesh-grid topology under two routing algorithms. Fig. 3.5a shows a deterministic XY-routing in which a message is first transmitted horizontally and then vertically. Meanwhile, Fig. 3.5b shows an example of an adaptive routing algorithm in which a message can be re-routed due to possible problems, for example, faulty components in a route or overuse of some links in the network.



(a) Deterministic XY-algorithm (b) Adaptive routing algorithm

Figura 3.5 – Illustration of a message transmission using two types of routing algorithm.

3.3.4 Switching Strategies

The switching strategy [1] defines how data is transmitted inside of the network by determining the granularity of data transmission and resource sharing. Switching strategies can be sub-divided into two types: (a) **circuit** and (b) **packet** switching.

In *circuit switching*, a route composed of links and routers have all of these components allocated to transmit a single message per time. Once that message has finished being transmitted, that route, and their links and routers, is de-allocated, and then another message can allocate that path or a link or router that before formed the de-allocated path. The advantage of this type of switching strategy is that it guarantees low latency communication by preventing other messages to use the same resources. Nonetheless, circuit switching strategies do not scale with the number of PE because there is a significant addition of time necessary to allocate a route as the number

of messages contention inside of the system grows.

In *packet switching* strategies, messages are packetized by a NI and divided into smaller transmission elements called **flits**, named as flow control units. These flits are then transmitted through the NoC, and, once the message has all its data arrived into its destination, it is de-packetized by the destination's NI. Another characteristic of this type of switching strategy is that a message is transmitted through the reservation of individual links to transmit a single flit as the message advances.

For this reason, multiple messages can be transmitted at the same time, even when they have routes that share links. Contention resolution is only necessary when two or more messages try to reserve the same link. In this case, a message waits for links contention to be resolved when other messages that are blocking it finish their transmissions. During this waiting period, this message is held in buffers present in the routers in the path defined by the routing algorithm. The size of a flit is typically the width in bits for the links, i.e., the number of bits transmitted per clock cycle. Figure 3.6 illustrates the division of a data message into packets then flits. **For simplicity, in this work, a message is considered to be formed by a single packet.**

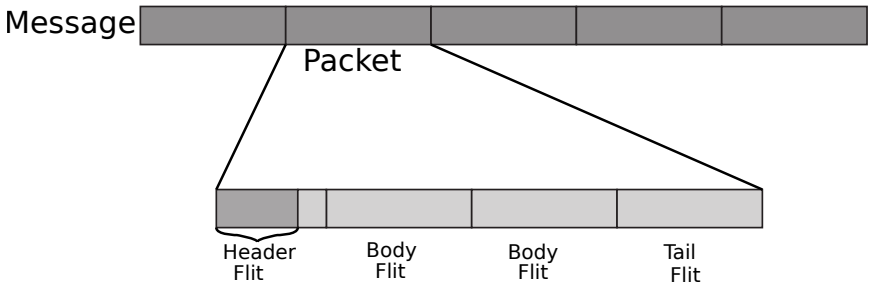


Figura 3.6 – Illustration of the typical packetization of a data message inside of a NoC (adapted from [1]).

There are three main types of packet switching strategies: (a) store-and-forward, (b) virtual cut through, and (c) wormhole.

3.3.4.1 Store-and-Forward Switching

In Store-and-Forward switching strategy, a message has its *flits* transmitted from a router to another only if the receiving router has enough buffer space to store the entire message. Beyond that, a router only forwards a message when it has received all of its flits. For this reason, buffer sizes inside of routers in NoCs that implement store-and-forward switching has to be at least of the size of a message packet. Therefore, this packet switching strategy needs routers with large buffer sizes.

3.3.4.2 Virtual-Cut-Through Switching

Routers that implement virtual-cut-through switching strategies have less latency when transferring data than routers with store-and-forward. It is possible since a router can transmit flits

from a message as soon as it has received it. However, virtual-cut-through switching strategy also requires that receiver routers have to have enough space to store the whole message. Otherwise, no flit is transmitted. This characteristic causes large buffer requirements.

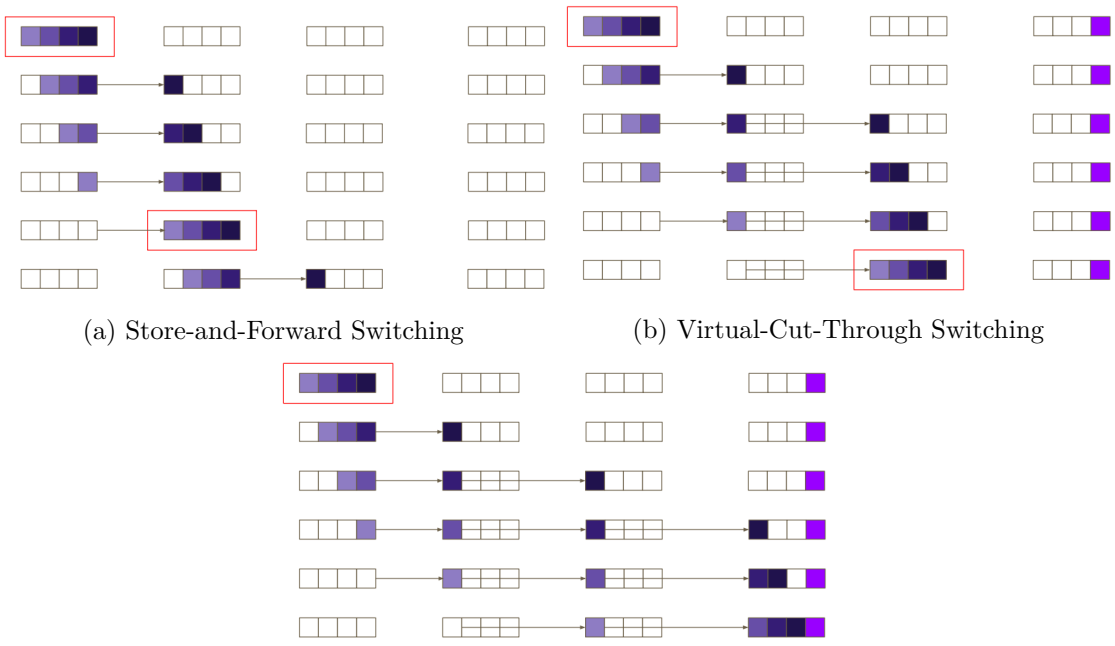
3.3.4.3 Wormhole Switching

Wormhole switching strategy reduces the buffer requirement since a router can transmit a flit as soon as it has received the flit even though the receiver router can only store a single flit for that message. For this reason, a message can have its flits buffered by multiple routers since they are transmitted in a pipeline fashion.

Wormhole switching causes higher congestion than store-and-forward and virtual-cut-through strategies that results in blocking of links such as **deadlocks** and **livelocks**. A **deadlock** happens when a message is interrupted and never reaches its destination due to a cyclic dependency, which causes the message to be blocked for an indefinite amount of time waiting for an event that may never happen. Similarly, **livelock** happens when a message may never reach its destination but not for being blocked but instead for being transmitted around the same group of routers. This type of blockage scenarios can be avoided using routing algorithms that do not allow cycles such as, for example, the XY-algorithm.

Figure 3.7 shows examples representing buffers being used across different routers when transmitting a message formed by a set of flits. In these representations, each buffer router is composed of four flit slots, and each message is formed of four flits represented as purple squares. Each line represents a stage of the message being transmitted. In the first line, the message is stored in the first router highlighted in red. Black arrows represent a flit transmission between two router buffers.

Figure 3.7a shows the transmission under store-and-forward switching, and in this case, a message is only transmitted when the next buffer has enough space for the message and the current router has the whole message stored. Figure 3.7b shows the transmission of a message using virtual-cut-through switching, and in this case, the message is only transmitted when the next buffer has enough space for the whole message. It is possible to see that the message cannot be sent to the last router because its buffer does not have space for the whole message. Lastly, Fig 3.7c shows the transmission of a message using a wormhole switching strategy where the transmission of *flits* is done regardless if the next buffer has enough space for the whole message.



(a) Store-and-Forward Switching

(b) Virtual-Cut-Through Switching

(c) Wormhole Switching

Figura 3.7 – Illustration of a message transmission between routers buffers with different packet switching strategies.

3.3.5 Arbitration Policies

A router implements an arbitration policy to solve contention when multiple messages try to use the same link at the same time, by deciding which message is sent and which has to wait to be transmitted. In a router architecture such as the one present in Fig.3.3, an arbitration logic decides which input link is connected to an output link by the central crossover. Two examples of arbitration policies are the round-robin and priority-based arbitration policy.

3.3.5.1 Round-Robin Arbitration

The round-robin arbitration, in a contentious scenario, selects in a rotation fashion which message buffered in different input links should be transmitted. This arbitration gives equal opportunities for different messages mitigating starvation problems. However, it has no predictability of which message should be sent.

3.3.5.2 Priority-based Arbitration

In a router that uses a priority-based arbitration, messages have information about their level of importance (priority) and, in a case of contention, the input link storing a message with higher priority is selected over a link that is storing a lower priority message. Priority preemptive arbitration is suitable for NoC-based SoCs used in real-time applications due to its predictability when solving contentious scenarios.

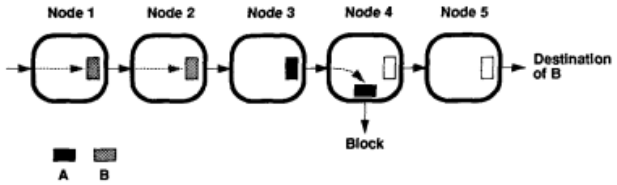
3.3.6 Virtual Channels

In this work, the chosen flow control is the use of Virtual Channels (VCs) [5] [8]. Each input link buffer is sub-divided in different communication channels, and for this reason, it is capable of storing flits from multiple messages. By using VCs, when there is a contention, blocked messages are capable of keeping being transmitted.

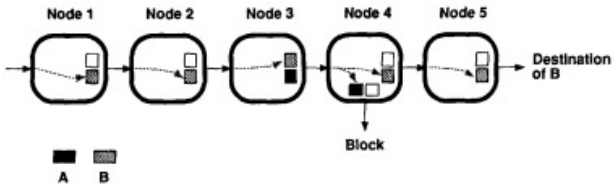
The analogy used by Dally [5] is to compare each link buffer as a road and VCs are the equivalents of using lanes. A link without a VC is the equivalent of a single lane road, and, in this case, if a message is blocked, all following messages are blocked. The same link with VCs is analogous to multiple lanes in the road. If a message is blocked, other messages that are using other "lanes"are capable of keeping moving.

The advantages of using VCs are: (a) it increases the throughput of data in the network by mitigating link idle time; (b) it reduces deadlock scenarios; (c) it allows message preemption by blocking its transmission and holding it in its virtual channel;

Figure 3.8 illustrates example cases of blockages due to contention between two messages using input links without and with VCs, respectively. In these examples, NoCs have priority-based arbitration policy, with message **A** having a higher priority-level than message **B**. In Fig. 3.8a, message **A** is blocked down its route in router 4. Since message **B** uses links busy with message **A**, message **B** will only continue its transmission when message **A** finishes its transmission. In Fig. 3.8b, the same message **A** is blocked. However, since the input links have VCs, message **B** is capable of keep being transmitted during the interval in which message **A** is blocked.



(a) Transmission of messages A and B in scenario without VCs.



(b) Transmission of messages A and B in scenario with VCs.

Figura 3.8 – Illustration of messages with scenarios with and without VCs (adapted from [5]).

3.4 Real-Time Network-on-a-Chip

Real-Time Networks-on-a-Chip (RTNoC) [4] are NoC-based SoC designs that offer architectural support to guarantee Quality-of-Service (QoS) for message transmissions. QoS is essential for messages to comply with time constraints imposed by real-time embedded devices application. In other words, an RTNoC is an SoC-based real-time system that has an NoC as its on-chip communication architecture.

This work focus is over the use of an RTNoC-based MPSoC platform in a real-time setting. The goal is for this platform to be capable of meeting all of the time constraints imposed by possible real-time application considering both their tasks processing as well as the messages transmitted by these tasks.

An RTNoC considered in this work has the following characteristics, unless stated otherwise: (a) it uses wormhole switching strategy; (b) it uses a 2D mesh-grid topology; (c) it uses an XY-routing algorithm that is deterministic and predictable; (c) it uses priority-based arbitration; (d) it uses virtual channels [5] that together with priority-based arbitration allows preemption of a message transmission. An example of an NoC architecture design that fits these characteristics in its design is present in [36].

3.5 Real-Time Systems

A Real-Time System (RTS) [6] [37] [38] is a computing system in which its correctness depends upon both its operations returning logically correct results as well as them fulfilling their time restrictions. These time restrictions, namely deadlines, are generally inherited from the dynamic of the desired application of an RTS. For example, in automotive applications, possible PEs have short time restrictions to process periodic routines related to engine management and the time constraints of these routines are directly dependent on the mechanical system. In this example, if a routine result is logically wrong, it may have catastrophic results permanently damaging the vehicle mechanically or even threatening the life of its occupants. The same outcome may happen if a routine result does not comply with its timing restrictions.

RTS can be classified based on the consequences when tasks missing deadlines and their results usability after the deadline miss. These three groups are the following:

- **Hard RTS**: It is composed of RTSs where a task missing its deadline produces catastrophic failure. For this reason, no task finishing to execute after its deadline is allowed. An example is the critical system detection in industrial applications. For instance, if a pressure sensor malfunction in a boiler in a factory, the lives of people working in this factory may be in danger.
- **Firm RTS**: It is composed of RTSs where a task missing its deadline renders its result useless, but it is not damaging for the system as a whole. For this reason, few tasks rarely missing their deadlines are allowed. An example is the industrial automatic quality control

of production line on a conveyor belt. If there is a deadline miss, a possible failed product has passed to the next production stage, and the processing results are useless. The consequence of a deadline miss is a failure of the quality control system. However, the system can keep running.

- **Soft RTS**: It is composed of RTSs where a task missing its deadline deteriorates the system performance, but its result is still useful for the system. For this reason, frequent tasks missing their deadlines are allowed. An example is a system-user interaction applications such as graphical user interfaces. If there is a deadline miss, the user perceives the quality degradation but waits for the process to finish.

Figure 3.9 has an illustration of the degradation of the usefulness of the results of an operation in different types of systems as the lateness of the results increases. In this figure, the time unit used is the task deadline (D). The concepts of tasks and deadlines are further explored in Section 3.6.1

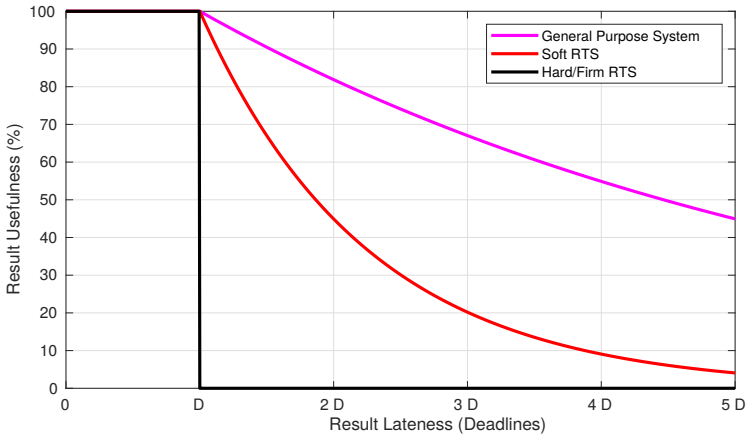


Figura 3.9 – Illustration of the the degradation of a task result usefulness with time after its deadline has passed.

Due to their critical applications, an RTS needs to support the following features:

- **Timeliness:** An RTS results have to be correct in value as well as in delivery time.
- **Efficiency:** An RTS is normally part of an embedded system, and, for this reason, there are other constraints in terms of power consumption, memory, and computational resources. In these systems, the efficient usage of the available resources is essential for its correct application.
- **Robustness:** An RTS needs to be functional even under overloaded conditions. In this manner, its design predicts and cover even the worst usage scenario to ensure maximum robustness.
- **Fault Tolerance:** An RTS should be tolerant of possible software or hardware failures and should handle these possibilities in a way to prevent the system from being in a complete

halt.

- **Maintainability:** An RTS must be quick to repair and maintain to ensure that it is always online and working. Generally, RTSs achieves this feature by having a modular architecture in which a possible repair or substitution of a component of the system can be done independently of other components.
- **Predictability:** An RTS needs to be predictable in order to verify whether it is capable of meeting its timing constraints under any scenario. If not, the RTS has to be capable to notify its failure to be then handled by other modules/parts of the system.

3.6 Multiprocessor Real-Time System Modeling and Scheduling Algorithms

In this section, the author presents a brief introduction of the process to model a real-time application as a set of tasks for a general multiprocessors real-time system. It also includes concepts, definitions, and examples of scheduling algorithms. These topics are essential to aid the reader to understand the modeling process of an RTNoC-based MPSoC as an RTS as well as the theory involved for the analytic scheduling algorithm present in Section 3.7.

3.6.1 Real-Time Application Modeling

A Real-Time Application (RTA) can be defined as a set of n concurrent system tasks represented as $\boldsymbol{\Gamma} = \{\boldsymbol{\tau}_1, ..., \boldsymbol{\tau}_n\}$. In [6], a task is defined as follows: "Task is a computation in which the operations are executed by the processor one at a time. A task may consist of a sequence of identical jobs, also called instances. The word *process* is often used as a synonym".

In this work, a given task $\boldsymbol{\tau}$ from an RTA follows the same definition by being considered as an abstraction of a set of programming instructions that can be dispatched to be executed by a processor core. Another important definition is that of a *job*. A *job* is a single instance of a task, i.e., a task emits a job to be executed by a processor.

In RTS, a task model can be divided into two types: (a) *aperiodic* tasks that emit jobs only once or in irregular intervals; and (b) *periodic* tasks that emit jobs on regular well-defined intervals. In this work, **all tasks are periodic**, and the following timing characteristics can define each ith task $\boldsymbol{\tau}_i$ of an RTA $\boldsymbol{\Gamma}$:

- **Cost**: Represents the Worst-Case Execution Time (WCET) of a task, The maximum time for a processor core to execute a job emitted from task $\boldsymbol{\tau}_i$. It is represented as C_i.
- **Period**: Represents the inter-arrival period between emitted jobs of a task $\boldsymbol{\tau}_i$. It is represented as T_i.
- **Deadline**: Represents the relative deadline or the bound of the time interval in which a job has to finish to process given its arrival time. It is represented as D_i.

- **Jitter**: Represents the possible difference between the expected arrival time and the real starting time of a task.
- **Priority**: Represents the priority level of the task that corresponds to its criticality. It is represented as P_i. **In this work, the convention used is the smaller the value of P_i, the greater the priority level of a task.**
- **Arrival Time**: Represents the activation time in which a task arrives into the processing queue to be executed. It is represented as a_i.
- **Finishing Time**: Represents the time in which a job emitted by the task $\boldsymbol{\tau}_i$ is completed to be executed by a processor core. It is represented as e_i.
- **Response Time**: Represents the total time interval between the arrival of a task into the processor queue until its completion, including the WCET and possible interruptions suffered by tasks with larger priorities. It is represented as $R_i = e_i - a_i$. During the worst-case scenario, R_i represents the Worst-Case Response Time (WCRT).
- **Lateness**: Represents the time interval of the difference between the deadline and the finishing time of the task. It describes how late a task has finished executing. It is represented as $l_i = R_i - D_i$.
- **Slack Time**: Also known as laxation, it represents the time interval of the difference between the finishing time of the task and its deadline. It describes the remaining time that a task has to finish to process and still respect its deadline. It is represented as $s_i = D_i - R_i$.

Notice that in this work, the worst-case scenario represents the **critical instant** in time in which all tasks emit jobs that concurrently compete for the RTS resources. From a designer point-of-view, this moment represents the case in which the system suffers the most degradation of their quality, and if it is capable of performing well under the worst-case scenario, it is capable of achieving even better results under other scenarios.

Depending on the platform on use, processing cores may allow interruption of an executing task returning them to the queue of processing tasks. This process is called *preemption*. Preemption is an essential feature for RTS since it allows critical tasks to be processed before less important ones.

Figure 3.10 illustrates the execution process of multiple tasks in a processor core with available mechanisms for preemption including the steps for (a) activation of a task, (b) scheduling representing decision of the order of tasks to be processed, (c) dispatching a task to be processed, (d) preemption of a task from execution back to the queue, and (e) termination of a task execution.

Figure 3.11 displays an example of the processing timeline for three tasks processed by a single processor. In this example, an *upward arrow* represents the arrival time of a task, and a *downward arrow* represents a task deadline. The time interval that a task is executed is colored in gray. The time interval that a task suffers preemption due to higher priority tasks interruption is represented

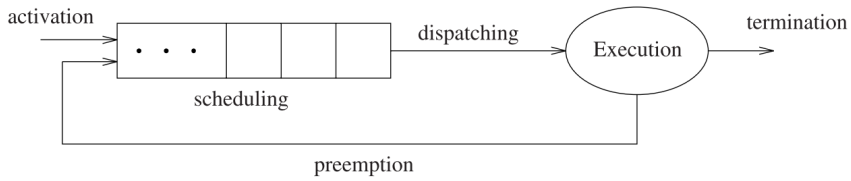


Figura 3.10 – Representation of a processing queue in a processing core that enable preemption. Source [6].

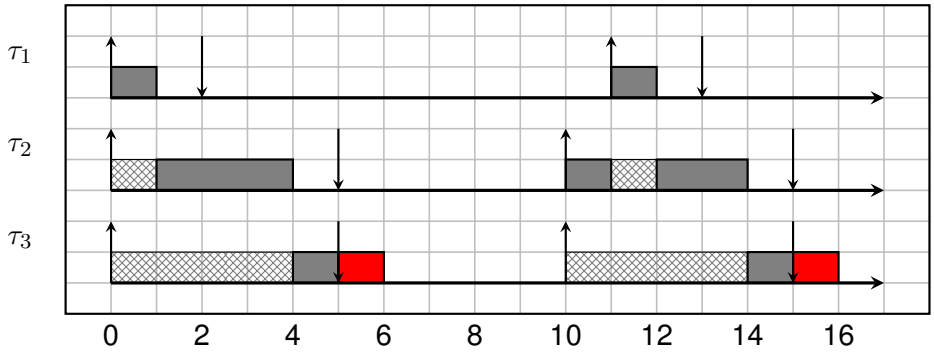


Figura 3.11 – Illustration of a timeline for instances of three tasks in a processing core.

as a cross-hatching pattern. Lastly, if a task does not comply with its deadline, the lateness is colored in red.

In this example, three tasks τ_1, τ_2, and τ_3 are activated an arrives into the processor scheduling queue. These tasks indices dictate their priority levels, in this way, $P_1 < P_2 < P_3$, so τ_1 is the most critical task, and τ_3 is the least critical one. Since τ_1 has greater priority, τ_1 preempts task τ_2 and τ_3 interrupting them during its execution. As soon as τ_1 finishes to be executed, τ_2 starts its execution. Finally, when τ_2 ends its execution, task τ_3 starts to be processed, and, due to its interruption by τ_1 and τ_2, its execution finishes after its deadline (represented as an downward arrow). The period in which τ_3 is processing while late is represented in red.

3.6.2 Scheduling Problems and Algorithms

As present in the surveys [37] and [39], real-time scheduling problems can be divided into two main types: (a) **Allocation**: The definition for each processor in the system of a subset of tasks in the RTA that it should execute; (b) **Priority Assignment**: The process to assign priority levels for the tasks of an RTA.

A scheduling algorithm is a method to solve scheduling problems, and they can be subdivided into the following classes:

- **Preemptive**: For systems that allow tasks to be interrupted during their execution.
- **Non-Preemptive**: For systems that once a task starts, it has to execute until the end.

- **Static**: For systems where the scheduling decisions are made with static parameters that do not change during its use.
- **Dynamic**: For systems where the scheduling decisions are made with information that changes during its use.
- **Offline**: For systems where the scheduling decisions are made and saved before the system go online.
- **Online**: For systems where the scheduling decisions are made during the system run-time.
- **No-Migration**: For systems where tasks are only executed by the processor assigned to them. Also known as **partitioned** scheduling algorithms.
- **Migration**: For a system that allows a task to have their jobs to be executed by different processors than the one assigned for it. Another possibility is job migration, for systems that allow a job to start its execution in a processor and then migrate to another one. Both Migration classes are also known as **global**.
- **Fixed-Priority**: For systems that do not allow for jobs to have a different priority than their tasks.
- **Dynamic-Priority**: For systems that allow for jobs to have a different priority than their source tasks.

Other important definitions related to scheduling algorithms are whether they or their results are: (a) **feasible**, (b) **schedulable**, (c) **sufficient**, (d) **necessary**, and (e) **optimal**.

An RTA is deemed **feasible** for an RTS if there is a scheduling algorithm that can schedule all possible sequences of jobs emitted by its tasks are processed without violating their deadlines. Namely, all tasks response times are bounded by their deadlines. Other possible restrictions that increase the difficulty of the existence of feasible solutions are: (a) tasks with a specific order of precedence; and (b) resource constraints such as the use of memory and I/O components;

An RTA is deemed *schedulable* given a scheduling algorithm result if all of its composing tasks respect their deadlines under their WCRTs. A schedulability analysis test is used to check if an RTA is schedulable. This test is defined as the process to check whether a solution for a scheduling has all its tasks attending their respective deadlines with a simple true/false answer. A schedulability test is deemed **sufficient** for a schedulability algorithm being applied on an RTS, if all RTAs are deemed schedulable using a sufficient test, they are de-facto schedulable. In the same fashion, a test is deemed **necessary**, if all RTA that is deemed unschedulable are indeed unschedulable. A schedulability test that is both sufficient and necessary is **exact**.

Lastly, a schedulability algorithm is deemed **optimal**, if it can schedule all RTA for a particular model of tasks that are feasible on RTS.

In this work, the focus is over **preemptive, static, offline, partitioned with fixed-priority scheduling algorithms**. The advantages and disadvantages of such methods are:

1. Static and Offline

    - **Advantage**: It allows the use of complex and computing intensive approaches to solve the scheduling problem.
    - **Disadvantage**: It lacks flexibility for possible changes in the system after the scheduling decisions are made.

2. Preemptive

    - **Advantage**: As previously mentioned, It allows the use of an essential feature for RTS.
    - **Disadvantage**: It increases system complexity.

3. Partitioned

    - **Advantage**: It allows the use of real-time scheduling algorithms for single-processor systems after an allocation process has been done. It also permits the use of local processing queue per processor instead of a large global one for the whole system that needs a more complex logic to control and manage all RTA tasks.
    - **Disadvantage**: Task allocation problem for partitioned approaches is similar to the bin packing problem that is a well-known NP-hard problem [13].

3.6.3 Priority Assignment

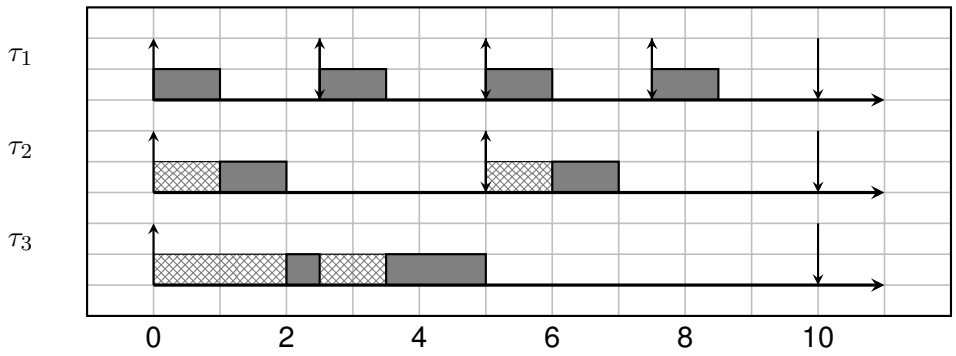
As previously mentioned, the priority assignment problem is the process to define priority levels for the tasks of an RTA. Since in this work, the tasks used are periodic, and the scheduling algorithms used have fixed-priority, it is crucial to present the well-established priority assignment algorithm called *Rate Monotonic Scheduling.*

3.6.3.1 Rate Monotonic Priority Assignment Policy

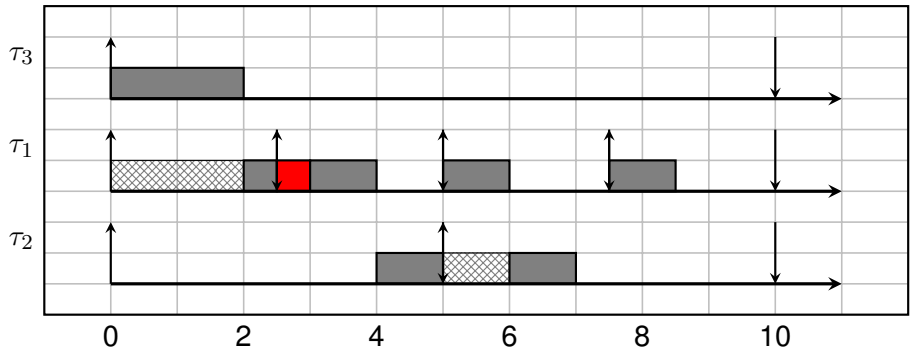
Rate Monotonic (RM) priority policy algorithm assign priority levels for the tasks in an RTA given their inter-arrival periods. Tasks with higher activation rates, i.e., shorter periods, are treated as more critical than those with lower activation rates. For example, three tasks $\boldsymbol{\tau}_1$, $\boldsymbol{\tau}_2$, and $\boldsymbol{\tau}_3$ with their respective periods following $T_1 < T_2 < T_3$. Under RM scheduling their assigned priorities would be $P_1 < P_2 < P_3$. It is worth to highlight that, in this work, the lower the priority-level numeric value, the more critical is the task. So a task j with $P_j = 1$ is the most critical task in the application.

The seminal work of Liu and Layland [40] proofs that RM scheduling is the optimal priority assignment policy in the context of a single processor running an RTA, with periodic tasks with periods smaller or equal than their deadlines. This proof is revisited in [41]. Note that since this work focus on partitioned scheduling algorithm, it is possible to use RM priority assignment to define the priority levels for the tasks, because after the allocation of tasks into processor cores for individual tasks the system can be reduced as a single processor.

Figure 3.12 presents a simple example illustrating a processor timeline for three tasks $\boldsymbol{\tau}_1$, $\boldsymbol{\tau}_2$, and $\boldsymbol{\tau}_3$ during the worst-case scenario. In this example all three tasks have their deadlines and periods with equal value, i.e., $D_i = T_i$, with $i = 1, ..., 3$. For this reason, the arrows that represent the arrival and deadline times merge, forming upward and downward arrows. In the first case (Fig. 3.12a), an RM policy is used to define the tasks priorities, and all three tasks are schedulable. In the second case (Fig. 3.12b), an arbitrary policy was used to set tasks priorities and, as a consequence, it causes $\boldsymbol{\tau}_1$ first instance to miss its deadline and then compete with its second instance, therefore, becoming an unschedulable task under the worst-case scenario.



(a) Illustration of timelines for instances of tasks with their priorities assigned with RM where $P_1 < P_2 < P_3$.



(b) Illustration of timelines for instances of tasks with their priorities assigned where $P_3 < P_1 < P_2$.

Figura 3.12 – Illustration of timelines for instances of three tasks ($\boldsymbol{\tau}_1$, $\boldsymbol{\tau}_2$ and $\boldsymbol{\tau}_3$) in a processing core. Upward arrows represents the arrival time and the interval between them presents the period T_j for $\boldsymbol{\tau}_j$, and downward arrows represents the deadline time since the last arrival

3.6.4 Task Assignment

As previously mentioned, the task assignment problem is the allocation of subsets of an RTA to be executed exclusively in different processor cores of a multiprocessor system. In this work, the schedulability tests used are defined based on the task assignment problem in a way to assess whether a scheduling algorithm result is schedulable or not.

The author believes that it is vital for the reader to understand two essential schedulability analysis for multiprocessor systems that are extended for the problem of schedulability analysis for RTNoC-based MPSoC. These two analyses are (a) processor utilization factor test, and (b)

response-time analysis.

3.6.4.1 Schedulability Analysis - Processor Utilization Factor

The schedulability analysis, based on processor utilization factor, as established by [40], uses the utilization factor of each task to define whether the processor cores are being used under its maximum capacity. The utilization factor of a processor core is the fraction of total processing time spent executing tasks. So for example, if a processor core has a utilization factor $U_\pi = 0.8$, it means that this processor spends 80% of its time executing tasks and 20% idle. The same goes if a processor core has a utilization factor $U_\pi = 1.5$, it means that this processor is over its maximum capacity of 100% of time executing tasks.

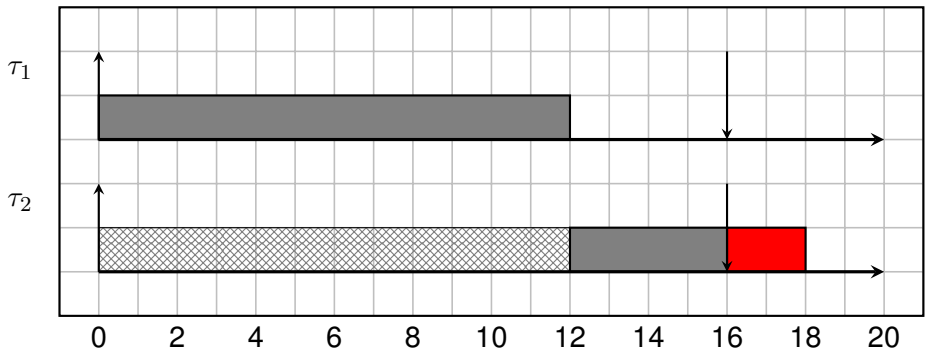
The utilization factor for a processor core is the sum of the utilization factors for the tasks mapped into that processor. For example, given a set $map(\pi)^{-1}$ of tasks mapped into a processor π, the utilization factor for this processor U_π to be under its maximum capability has to follow:

$$U_\pi = \sum_{\boldsymbol{\tau}_j \in map(\pi)^{-1}} \frac{C_j}{T_j} \leq 1, \tag{3.1}$$

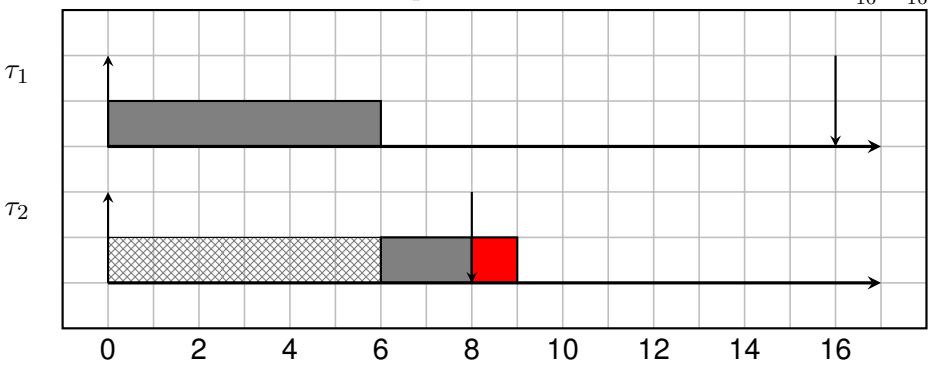
where $map^{-1}(\pi)$ represents the set of tasks mapped to a processor π, C_j and T_j are, respectively, the WCET and period for task $\boldsymbol{\tau}_j$.

This schedulability analysis is simple to perform. However, this test is necessary, but not sufficient. In other words, if a set of tasks into a processor has the sum of their utilization factor greater than 1, it means that the processor core is over its maximum capacity and some tasks do not respect their deadlines. However, if this sum of utilization factors is less than 1, it does not guarantee that all tasks respect their deadlines.

Figure 3.13 illustrate examples of two tasks mapped into the same processor core. In the first case (Fig. 3.13a), task $\boldsymbol{\tau}_1$ has WCET $C_1 = 12$, and equal periods and deadlines $T_i = D_i = 16$, and task $\boldsymbol{\tau}_2$ has $C_2 = 6$ and period and deadline $T_i = D_i = 16$. In this case, the sum of utilization factors $U_\pi = \frac{12}{16} + \frac{6}{16} = 1.125$ is over 1 resulting in one of them being unschedulable. In the second case (Fig. 3.13b), task $\boldsymbol{\tau}_1$ has WCET $C_1 = 6$, and equal periods and deadlines $T_i = D_i = 16$, and task $\boldsymbol{\tau}_2$ has $C_2 = 3$ and period and deadline $T_i = D_i = 8$. In this case, the sum of utilization factors $U_\pi = \frac{6}{16} + \frac{3}{8} = 0.75$, and even though the sum of utilization factor is below 1, due to the priority policy, task $\boldsymbol{\tau}_2$ is unschedulable.



(a) Illustration of a timeline for a set of tasks in a processor with utilization factor $U_\pi = \frac{12}{16} + \frac{6}{16} = 1.125 > 1$.



(b) Illustration of timeline for a set of tasks in a processor with utilization factor $U_\pi = \frac{6}{16} + \frac{3}{8} = 0.75 < 1$.

Figura 3.13 – Illustration of timelines for instances of two tasks ($\boldsymbol{\tau}_1$ and $\boldsymbol{\tau}_2$) in a processing core. Upward arrows represents the arrival time and the interval between them presents the period T_j for $\boldsymbol{\tau}_j$, and downward arrows represents the deadline time since the last arrival

This schedulability test can be extended for all tasks in the RTA [42], to asses at least how many m processor cores are necessary. Given that the sum of utilization factor of tasks in an RTA $\boldsymbol{\Gamma}$ is $U_{\boldsymbol{\Gamma}}$, the number of processor cores in the used RTS has to be at least greater than $U_{\boldsymbol{\Gamma}}$ as expressed in:

$$U_{\boldsymbol{\Gamma}} = \sum_{\boldsymbol{\tau}_i \in \boldsymbol{\Gamma}} \frac{C_i}{T_i} \leq m, \tag{3.2}$$

where $\boldsymbol{\Gamma}$ is the set of tasks in an RTA, $U_{\boldsymbol{\Gamma}}$ are the utilization factor for the application representing the number of required processors, m is the number of processor cores, C_i is the WCET of the ith task in RTA, and T_i is the period of task $\boldsymbol{\tau}_i$.

The utilization factor ($\boldsymbol{\Gamma}$) schedulability test for an RTA executing in a multiple processor system is the test whether all processors are running under their maximum capacity. The complexity of this method is polynomial with big-O notation $\boldsymbol{O}(nm)$, where n is the number of tasks that composes the RTA used, and m is the number of processor cores in the multiple processors RTS.

3.6.4.2 Schedulability Analysis - Response Time Analysis

The schedulability analysis, based on response time, under the worst-case scenario, as established by [43], uses the WCRT of each task R_i to check whether it respects these *tasks time*

constraints D_i.

$$R_i = C_i + I_i, \tag{3.3}$$

where I_i is the interference term due to higher priority tasks.

If a task $\boldsymbol{\tau}_j$ is mapped into the same processor π that a task $\boldsymbol{\tau}_i$ and has greater priority level, i.e. $P_j < P_i$, this task $\boldsymbol{\tau}_j$ interferes $\boldsymbol{\tau}_i$ inside of its response time interval $[0, R_i]$ and this interference time $I_{i,j}$ is expressed as:

$$I_{i,j} = \left\lceil \frac{R_i}{T_j} \right\rceil C_j, \tag{3.4}$$

where R_i is the WCRT for task i, T_j is the period for task j, and C_j is the WCET for task $\boldsymbol{\tau}_j$.

The interference time I_i suffered by a task $\boldsymbol{\tau}_i$ due to all higher priority tasks running in the same processor preempting this task is calculated using equation:

$$I_i = \sum_{\boldsymbol{\tau}_j \in cont(\boldsymbol{\tau}_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j, \tag{3.5}$$

where $cont(\boldsymbol{\tau}_i)$ represents the set of contentious tasks with higher priority than task $\boldsymbol{\tau}_i$ that shares the same processor core, in other words $cont(\boldsymbol{\tau}_i) = \{\boldsymbol{\tau}_j \in \Gamma : P_j < P_i\}$.

Combining Equations 3.3 and 3.5, it is possible to express the Equation 3.6 to the response time under worst-case as follows:

$$R_i^{(t+1)} = C_i + \sum_{\boldsymbol{\tau}_j \in cont(\boldsymbol{\tau}_i)} \left\lceil \frac{R_i^{(t)}}{T_j} \right\rceil C_j. \tag{3.6}$$

No simple solution exists for Equation 3.6 since R_i appears on both sides. The solution for the WCRT of a task $\boldsymbol{\tau}_i$ can be obtained through an iterative process illustrated by Algorithm 2. Once the WCRT of a task has been calculated, if its value is greater than its deadline, it means that that task is unschedulable.

Algorithm 2 Response Time for a Task ($WCRTf$)

INPUT: task ($\boldsymbol{\tau}_i$)
OUTPUT: WCRT R_i for the task $\boldsymbol{\tau}_i$
1: **procedure** $WCRTf(\boldsymbol{\tau}_i)$
2: $R_i^{(0)} = C_i$
3: $R_i^{(1)} = 0$
4: **do**
5: $R_i^{(t+1)} = C_i + \sum_{\boldsymbol{\tau}_j \in const(\boldsymbol{\tau}_i)} \lceil \frac{R_i^{(t)}}{T_j} \rceil C_j$
6: **while** $R_i^{(t+1)} \neq R_i^{(t)}$
7: **return** $R_i^{(t+1)}$
8: **end procedure**

An example case is present here to aid the reader and illustrate the process to calculate the WCRT values for three tasks ($\boldsymbol{\tau}_1$, $\boldsymbol{\tau}_2$, and $\boldsymbol{\tau}_3$) allocated to a processor. Table 3.1 presents information for these three tasks while Tab. 3.2 presents the calculation through the iterative method to obtain the WCRT for these three tasks. Finally, Fig. 3.14 displays the timeline for these three tasks during the worst-case scenario.

Tabela 3.1 – Characteristics of three tasks in an example case. Where C is the task WCET, D is the deadline, T is the period, and P is the priority-level.

Task	C	D	T	P
τ_1	1	3	3	1
τ_2	2	8	8	2
τ_3	2	10	10	3

Tabela 3.2 – Example of iterations to calculate the WCRT (R_i) mapped to a processor.

	Task 1	**Task** 2	**Task** 3
Iteration 0:	$R_1^{(0)} = 1$	$R_2^{(0)} = 2$	$R_3^{(0)} = 2$
Iteration 1:	$R_1^{(1)} = 1$	$R_2^{(1)} = 2 + \lceil \frac{2}{3} \rceil 1 = 3$	$R_3^{(1)} = 2 + \lceil \frac{2}{3} \rceil 1 + \lceil \frac{2}{8} \rceil 2 = 5$
Iteration 2:	**Return** $R_1^{(1)} = R_1^{(0)} = 1$	$R_2^{(2)} = 2 + \lceil \frac{3}{3} \rceil 1 = 3$	$R_3^{(2)} = 2 + \lceil \frac{5}{3} \rceil 1 + \lceil \frac{5}{8} \rceil 2 = 6$
Iteration 3:	-	**Return** $R_2^{(2)} = R_2^{(1)} = 3$	$R_3^{(3)} = 2 + \lceil \frac{6}{3} \rceil 1 + \lceil \frac{6}{8} \rceil 2 = 6$
Iteration 4:	-	-	**Return** $R_3^{(3)} = R_3^{(2)} = 6$

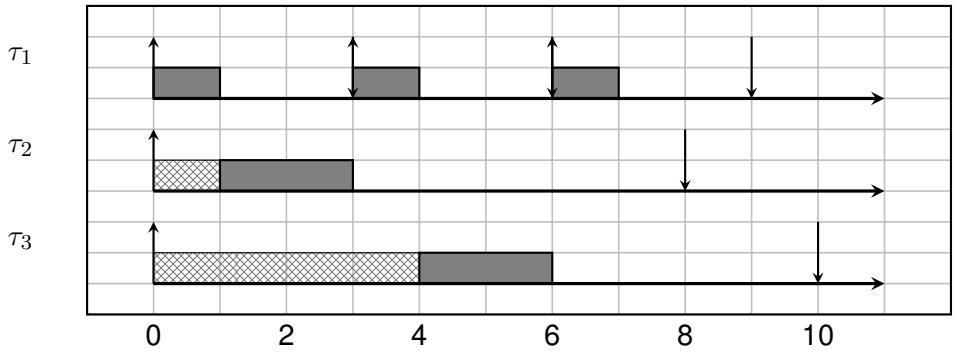


Figura 3.14 – Illustration of timeline for the example of WCRT (R_i) calculation.

The response time schedulability test for an RTA is the use of Alg. 2 to identify if all tasks in an application comply with their deadlines, i.e., $R_i \leq D_i$. The complexity of this method is pseudo-polynomial with big-O notation $\boldsymbol{O}(nN)$ [6], where n is the number of tasks in the application, and N represents an unknown factor of relation between the periods of the tasks.

3.7 Modeling and Scheduling Algorithms for RTNoC-based MPSoCs

An MPSoC applied in a real-time environment that uses an RTNoC as its communication needs a more sophisticated approach for possible schedulability tests. Such approaches need to take into consideration both its multiple processor cores, as well as its communication architecture, since, the communication latencies strongly influences the system and may cause its tasks to miss deadlines. For this reason, End-to-End Schedulability Analysis (EESA) take into consideration

the required time necessary for a task processing from its arrival until the message (flow of flits) arrive at its destination, hence, the end-to-end in the name.

The schedulability analysis used in this work including the system and task models is based on those present in the works of Indrusiak [44] [45] that are based on the works of Shi and Burns [46] and Xiong [47] [48]. The system modeling also contains the memory model C presented by Still [49]. In this work, the schedulability algorithm is preemptive, static, offline, partitioned with fixed-priority that are intended to be used in a setting where the task mapping is performed offline, i.e., before the system deployment.

The following assumptions are made for the analysis and task and platform models:

- RTNoC used has a wormhole switching strategy with 2D mesh-grid topology and virtual channels as previously stated on Section 3.4.
- Each NoC tile has a processor core with its local memory connected to a NI.
- All processing cores are equal, i.e., the system is composed of homogeneous processor cores.
- All processor cores have a processing queue ordered by tasks priority with mechanisms that permit preemption, such as Fig. 3.10.
- Tasks deadlines are equal to their periods $D_i = T_i$ (*Implicity Deadlines*).
- Tasks transmit a single flow of flits (namely, messages) at the end of their executions. Since the schedulability of such systems also depends on communication characteristics of the platform using an NoC.
- Each message (flow of flits) transmitted by a task has the same priority level as its source task, i.e., messages inherit priority from their source tasks.
- The analysis is done on the worst-case scenario. Therefore all tasks arrive at the processing queue of their processor cores at the same time, $a_i = 0$ with $J_i = 0$.

These assumptions limit the scope of MPSoC based on NoC platforms in which these analytical model can be applied. However, this analytical framework can be easily expanded to treat cases where the previous assumptions are not valid. For example, to cover cases where processor cores are not homogeneous, one may alter the WCRT (Section 3.6.4.2) calculation using a scaling value for processor cores with higher frequencies.

3.7.1 Platform Model

The platform model $\mathbf{\Psi}$ is formed by a **set of homogeneous processor cores** connected to their respective NIs and private local memory $\mathbf{\Pi} = \{\pi_1, \pi_2, ..., \pi_n\}$. Each processor core is connected to a router in a set $\mathbf{\Xi} = \{\xi_1, \xi_2, ..., \xi_n\}$. Both routers and processor cores are then connected by a set of unidirectional links $\mathbf{\Lambda} = \{\lambda_{\pi_1,\xi_1}, \lambda_{\xi_1,\pi_1}, \lambda_{\xi_1,\xi_2}, ..., \lambda_{\xi_{n-1},\xi_n}\}$. Each link connects

a pair of components with a specific direction, for example, λ_{π_1,ξ_1} only sends data from π_1 to ξ_1 and λ_{ξ_1,π_1} is another link that connects the same pair but with the opposite direction.

Figure 3.15 presents an example of a platform model of an MPSSoC with an NoC that has 3×3 processors connected in a mesh-grid topology.

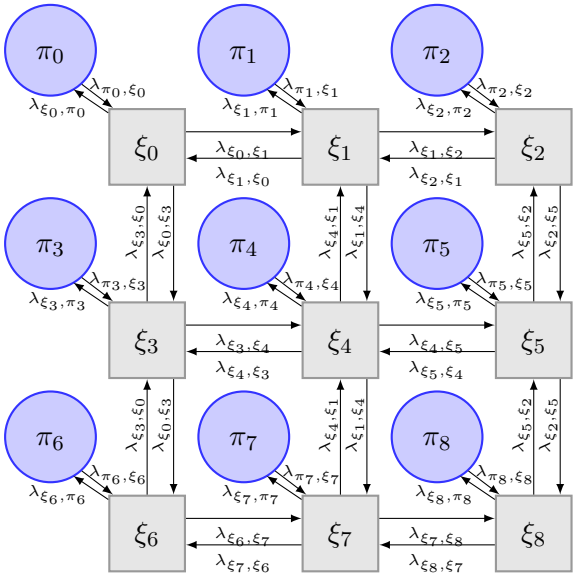


Figura 3.15 – 3x3 Mesh-grid RTNoC platform. Blue circles represent the processors and their network interfaces (NI), and gray squares represent routers.

3.7.2 Real-Time Application Model

The real-time application $\mathbf{\Omega}$ is represented as a set of tasks $\mathbf{\Gamma} = \{\boldsymbol{\tau}_1, ..., \boldsymbol{\tau}_n\}$ and a set of messages $\mathbf{\Phi} = \{\boldsymbol{\phi}_1, ..., \boldsymbol{\phi}_m\}$. These tasks are modeled as tuples $\boldsymbol{\tau} = \langle C, T, D, P, R, M, \boldsymbol{\phi}\rangle$ composed by the following characteristics, respectively: (a) C is the WCET, (b) T is the inter-arrival period, (c) D is the relative deadline time, (d) P is the task priority-level, (e) R is the task WCRT, (f) M is the task required code memory in bytes, and (g) $\boldsymbol{\phi}$ is a message sent to another task.

The messages are also represented as tuples $\boldsymbol{\phi} = \langle \boldsymbol{\tau}_d, L, Z, K, S\rangle$ with $\boldsymbol{\tau}_d$ is the destination task, L is the basic latency time (latency without network contention), Z is the message size in bytes, K is the maximum release jitter time, and S is the Worst-Case Latency Time (WCLT).

The tuple of a message is smaller because it inherits characteristics from its source task, namely, its priority, inter-arrival period, and deadline. The jitter suffered by a flow $\boldsymbol{\phi_i}$ is due to how long it takes for its source to emit it, and, since a task only emits its message when it has finished processing. For this reason, the jitter time suffered by a message is equal to its source task WCRT $K_i = R_i$.

The following functions are defined to aid notation for real-time analysis and system models:

- $index : \mathbf{\Pi} \rightarrow \mathbb{N}$ returns the index of a processor core π. For example, $index(\pi_1) = 1$.

- $map : \mathbf{\Gamma} \rightarrow \mathbf{\Pi}$ returns the processor core π that a task $\boldsymbol{\tau}$ is mapped onto. For example, if task $\boldsymbol{\tau}_1$ was mapped onto the processor core π_0, then $map(\boldsymbol{\tau}_i) = \pi_0$.

- $map^{-1} : \mathbf{\Pi} \rightarrow \mathbf{\Gamma}$ returns a set of tasks mapped to a processor π, i.e. $map^{-1}(\pi) = \{\boldsymbol{\tau} \in \Gamma : map(\boldsymbol{\tau}) = \pi\}$. For example, if only $\boldsymbol{\tau}_1$, $\boldsymbol{\tau}_2$, and $\boldsymbol{\tau}_3$ were mapped onto π_0, then $map^{-1}(\pi_0) = \{\boldsymbol{\tau}_1, \boldsymbol{\tau}_2, \boldsymbol{\tau}_3\}$.

- $route : \mathbf{\Phi} \rightarrow \mathbf{\Lambda}$ returns a set of ordered links that forms a path used by a flow $\boldsymbol{\phi}$ moving from its source task to its destination ordered. For example, if a message $\boldsymbol{\phi}$ transverse links λ_{π_a,ξ_a}, λ_{ξ_a,ξ_b}, and λ_{ξ_b,π_b} then $route(\boldsymbol{\phi}) = \{\lambda_{\pi_a,\xi_a}, \lambda_{\xi_a,\xi_b}, \lambda_{\xi_b,\pi_b}\}$.

- $route : \{\mathbf{\Pi}, \mathbf{\Pi}\} \rightarrow \mathbf{\Lambda}$ returns a set of ordered links that forms a path between two processing cores. Given a message $\boldsymbol{\phi}_i$ transferred between tasks $\boldsymbol{\tau}_i$ and $\boldsymbol{\tau}_j$, then it is equivalent $route(\boldsymbol{\phi}_i) = route(map(\boldsymbol{\tau}_i), map(\boldsymbol{\tau}_j))$. For example, $route(\pi_a, \pi_b) = \{\lambda_{\pi_a,\xi_a}, ..., \lambda_{\xi_b,\pi_b}\}$ and $route(\pi_b, \pi_a) = \{\lambda_{\pi_b,\xi_b}, ..., \lambda_{\xi_a,\pi_a}\}$, this order is defined by the routing algorithm.

- $map^{-1} : \mathbf{\Lambda} \rightarrow \mathbf{\Phi}$ returns a set of messages that passes through a link λ in other words $map^{-1}(\boldsymbol{\phi}) = \{\boldsymbol{\phi} \in \mathbf{\Phi} : \lambda \in route(\phi)\}$. For example, if only $\boldsymbol{\phi}_1$ and $\boldsymbol{\phi}_2$ uses a link λ, then $map^{-1}(\lambda) = \{\boldsymbol{\phi}_1, \boldsymbol{\phi}_2\}$.

- $cd : \{\mathbf{\Phi}, \mathbf{\Phi}\} \rightarrow \mathbf{\Lambda}$ returns a set of ordered contented links shared by two messages $\boldsymbol{\phi}_i$ and $\boldsymbol{\phi}_j$ in other words $cd(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j) = route(\boldsymbol{\phi}_i) \cap route(\boldsymbol{\phi}_i)$. For example, if a message $\boldsymbol{\phi}_1$ transverse the links $route(\boldsymbol{\phi}_1) = \{\lambda_{\pi_a,\xi_a}, \lambda_{\xi_a,\xi_b}, \lambda_{\xi_b,\pi_b}\}$ and a message $\boldsymbol{\phi}_2$ transverse the links $route(\boldsymbol{\phi}_2) = \{\lambda_{\pi_c,\xi_c}, \lambda_{\xi_c,\xi_a}, \lambda_{\xi_a,\xi_b}, \lambda_{\xi_b,\pi_b}\}$, then $cd(\boldsymbol{\phi}_1, \boldsymbol{\phi}_2) = \{\lambda_{\xi_a,\xi_b}, \lambda_{\xi_b,\pi_b}\}$.

- $vc : \mathbf{\Xi} \rightarrow \mathbb{N}$ returns the number of VCs supported by a router ξ. For example, if a system has routers with three VCs in each input buffer of a router ξ_1, then $vc(\xi_1)$.

- $buffer : \mathbf{\Xi} \rightarrow \mathbb{N}$ returns the buffer size of each virtual channel in number of flits. For example, if a system has routers with buffers with 3 VCs with each supporting 3 flits, then $buffer(\xi) = 3$.

- $mreq : \mathbf{\Pi} \rightarrow \mathbb{N}$ returns the private local memory required by a processor core π to store code and data in bytes. For example, if a processor π should have $\approx 32KB$ of memory to execute an application correctly, then $mreq(\pi) = 32768$ bytes.

- $mcap : \mathbf{\Psi} \rightarrow \mathbb{N}$ returns the amount of private local memory available in each processor core of a platform $\mathbf{\Psi}$ in bytes.

- $order : \{\mathbf{\Lambda}, \mathbf{\Lambda}\} \rightarrow \mathbb{N}$ returns the position of a link λ given a path between two processor cores π_a and π_b. For example, if a message $\boldsymbol{\phi}_1$ is transferred between tasks mapped onto processors π_a and π_b that passes through links $route(\boldsymbol{\phi}_1) = \{\lambda_{\pi_a,\xi_a}, \lambda_{\xi_a,\xi_b}, \lambda_{\xi_b,\pi_b}\}$, then $order(\lambda_{\xi_a,\xi_b}, route(\pi_a, \pi_b)) = 2$.

- $firstl : \mathbf{\Lambda} \rightarrow \mathbf{\Lambda}$ returns the link with the smallest order given a route. For example, if in a route $route(\pi_a, \pi_b)$ where $order(\lambda_{\pi_a,\xi_a}, route(\pi_a, \pi_b)) = 1$, then $firstl(route(\pi_a, \pi_b)) = \lambda_{\pi_a,\xi_a}$.

- $lastl : \mathbf{\Lambda} \rightarrow \mathbf{\Lambda}$ returns the link with the greatest order given a route. For example, if in a route $route(\pi_a, \pi_b)$ where $order(\lambda_{\xi_b,\pi_b}, route(\pi_a, \pi_b)) = |route(\pi_a, \pi_b)|$, then $lastl(route(\pi_a, \pi_b)) = \lambda_{\xi_b,\pi_b}$.

- $cont : \mathbf{\Gamma} \rightarrow \mathbf{\Gamma}$ returns a set of contentious tasks with higher priority levels than a given task $\boldsymbol{\tau}$ that are mapped into the same processor $map(\boldsymbol{\tau})$, in other words $cont(\boldsymbol{\tau}_i) = \{\boldsymbol{\tau}_j \in \mathbf{\Gamma} : map(\boldsymbol{\tau}_j) = map(\boldsymbol{\tau}_i) \wedge P_j < P_i\}$. For example, if tasks $\boldsymbol{\tau}_1$, $\boldsymbol{\tau}_2$, and $\boldsymbol{\tau}_3$ are mapped onto processor core π and $P_1 < P_2 < P_3$, then $cont(\boldsymbol{\tau}_3) = \{\boldsymbol{\tau}_1, \boldsymbol{\tau}_2\}$.

- $cont_d : \mathbf{\Phi} \rightarrow \mathbf{\Phi}$ returns a set of contentious flows with higher priority that directly interfere a given flow $\boldsymbol{\phi}$ sharing at least one link in their path $\boldsymbol{\phi}_i \in \mathbf{\Phi} : route(\boldsymbol{\phi}) \cap route(\boldsymbol{\phi}_i) \neq \emptyset$. The definition and examples of $cont_d$ will be presented further on Section 3.7.4.1.

- $cont_i : \mathbf{\Phi} \rightarrow \mathbf{\Phi}$ returns a set of all flows that indirectly interfere a given flow $\boldsymbol{\phi}$ that does not share any link in their routes ($\boldsymbol{\phi}_i \in \mathbf{\Phi} : route(\boldsymbol{\phi}) \cap route(\boldsymbol{\phi}_i) = \emptyset$) by direct interfering with a third message that in turn directly interfere $\boldsymbol{\phi}$. The definition and examples of $cont_i$ will be presented further on Section 3.7.4.1.

- $cont_i^u : \mathbf{\Phi} \rightarrow \mathbf{\Phi}$ returns a set of all flows that indirectly interfere a given flow $\boldsymbol{\phi}$ upstream. The definition and examples of $cont_d$ will be presented further on Section 3.7.4.1.

- $cont_i^d : \mathbf{\Phi} \rightarrow \mathbf{\Phi}$ returns a set of all flows that indirectly interfere a given flow $\boldsymbol{\phi}$ downstream. The definition and examples of $cont_d$ will be presented further on Section 3.7.4.1.

After the RTA has all its tasks allocated onto the platform processor cores, it is possible to calculate each task $\boldsymbol{\tau}_i$ WCRT R_i as well as its message $\boldsymbol{\phi}_i$ basic latency time L_i and WCLT S_i.

L_i value depends on specific system parameters. In this work, L_i is calculated using the following equation:

$$L_i = |route(\boldsymbol{\phi}_i)| \cdot (L_\lambda + (|route(\boldsymbol{\phi}_i)| - 1) \cdot L_\xi + \left\lceil \frac{8Z_i}{\lambda_{width}} \right\rceil L_\lambda, \tag{3.7}$$

where L_λ is the maximum time necessary for a *flit* to be transmitted through a link, L_ξ is the maximum time necessary for a header *flit* to be routed by a router, λ_{width} is the width of the links in bits, $route(\boldsymbol{\phi}_i)$ is the set of links in the route that message $\boldsymbol{\phi}_i$ use to transit, and Z_i is the data size of message $\boldsymbol{\phi}_i$ in bytes.

The values for the tasks WCRTs are calculated as described in Section 3.6.4.2, where, once again, a task response time during the worst-case scenario depends upon the tasks with higher priority mapped into the same processor.

The values for the messages WCLT are calculated as explained in Section 3.7.4.

3.7.3 Processors and Links Utilization Factors Test

Similar to the schedulability analysis that uses processor utilization factors, as present in Section 3.6.4.1, Indrusiak [45] presents a utilization factor test that can be applied to each of the

platforms links to check whether none of them are over their maximum capacity for the messages that passes through them. A link $\lambda \in \mathbf{\Lambda}$ is not over used if the following is valid:

$$U_\lambda = \sum_{\phi_i \in map^{-1}(\lambda)} = \frac{L_i}{T_i} \leq 1, \tag{3.8}$$

where U_λ is the utilization factor for a link λ, $map^{-1}(\lambda)$ is the set of messages that are transmitted over the link λ, L_i is the basic latency for message $\boldsymbol{\phi}_i$, and T_i is the period for task $\boldsymbol{\tau}_i$ that transmits message $\boldsymbol{\phi}_i$.

Similar to the processor cores utilization test in Section (3.6.4.1), the utilization test for links is necessary but not sufficient because a link being capable of sending all messages that use it in their paths without starvation does not mean that these messages will meet their deadlines.

It is possible to combine both processor and link utilization factors tests into the same schedulability test that evaluates the whole system after the task mapping by counting the number of processor cores and links over their maximum capacity, if it is equal zero then the system is schedulable considering both their tasks and messages. This test is expressed as follows:

$$|\{\lambda \in \mathbf{\Lambda} : U_\lambda > 1\}| + |\{\pi \in \mathbf{\Pi} : U_\pi > 1\}| = 0, \tag{3.9}$$

where U_λ is the utilization factor for a link λ in the set of links $\mathbf{\Lambda}$ in the platform, and U_π is the utilization factor for a processor core π in the set of processor cores $\mathbf{\Pi}$ in the platform.

This simple utilization test is useful, given its simplicity and can be combined as a condition for a more complex schedulability test. The complexity of this method used to evaluate whether a task assignment is schedulable in big-O notation is $\boldsymbol{O}(\max(|\mathbf{\Gamma}||\mathbf{\Pi}|, |\mathbf{\Phi}||\mathbf{\Lambda}|))$ depending on the number of tasks and messages in the RTA as well as the number of processor cores and links in the platform.

3.7.4 Multi-point Progressive Blocking Worst-Case Latency Analysis

The Worst-Case Latency Time (WCLT) of a message $\boldsymbol{\phi}$ is the time necessary for the header flit leave the NI until the moment that the last flit of the message arrives into its destination. As present in this work, the analysis process to calculate the Worst-Case Latency Time (WCLT) of a message transiting inside of a wormhole-based RTNoC is based on the work of Indrusiak et al. [44] that in turn improves the works of Shi and Burns [46] and Xiong [47] [48].

3.7.4.1 Message Interference Types in Wormhole-based NoCs

Before the presentation of the method, it is essential to explain the concept of **direct** and **indirect** interferences. In [50], these are the two types of interference that a message suffers through a wormhole-based NoC.

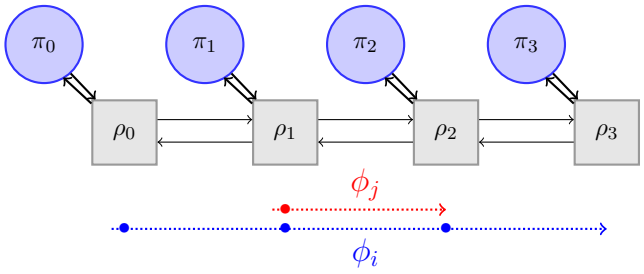
A message suffers **direct** interference when sharing at least one link in its route with messages that have higher priorities, equivalently as the contention suffered by tasks when sharing a processor with higher priority tasks. For example, given a pair of messages $\boldsymbol{\phi}_i$ and $\boldsymbol{\phi}_j$ with $P_j < P_i$

and both of them sharing links in their route, as shown in Fig. 3.16a, it is possible to see that message $\boldsymbol{\phi}_j$ preempts the message $\boldsymbol{\phi}_i$ increasing the WCLT S_i due to interference terms. The set of messages that directly interfere a message $\boldsymbol{\phi}_i$ is expressed as follows:

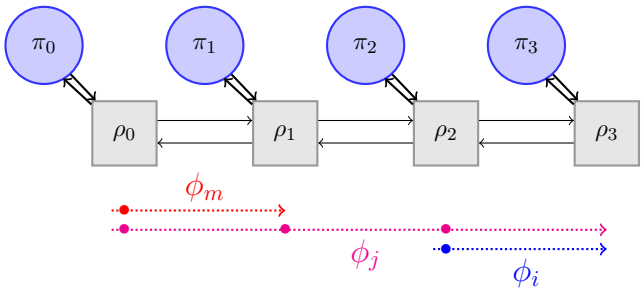
$$cont_d(\boldsymbol{\phi}_i) = \{\boldsymbol{\phi}_j \in \boldsymbol{\Phi} : P_j < P_i \wedge cd(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j) \neq \emptyset\}. \tag{3.10}$$

A message experiences indirect interference when one of the messages that preempts it also suffers interference from a third message that in turn, does not share any link with its route. For example, given three messages $\boldsymbol{\phi}_i$, $\boldsymbol{\phi}_j$, and $\boldsymbol{\phi}_m$ with $P_m < P_j < P_i$ with $\boldsymbol{\phi}_m$ and $\boldsymbol{\phi}_j$ sharing links as well as $\boldsymbol{\phi}_j$ and $\boldsymbol{\phi}_i$. In this situation, a message $\boldsymbol{\phi}_i$ suffers indirect interference from message $\boldsymbol{\phi}_m$. This scenario is expressed in Fig. 3.16b. The set of messages that indirectly interfere with a message $\boldsymbol{\phi}_i$ is expressed as follows:

$$cont_i(\boldsymbol{\phi}_i) = \{\boldsymbol{\phi}_k \in \boldsymbol{\Phi} : \boldsymbol{\phi}_k \in cont_d(\boldsymbol{\phi}_j) \wedge \boldsymbol{\phi}_j \in cont_d(\boldsymbol{\phi}_i) \wedge \boldsymbol{\phi}_k \notin cont_d(\boldsymbol{\phi}_i)\}. \tag{3.11}$$



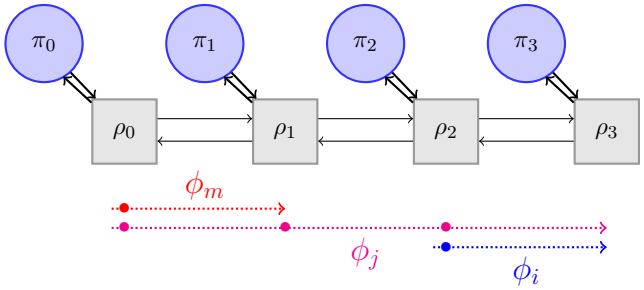
(a) Illustration of an example of a direct interference.



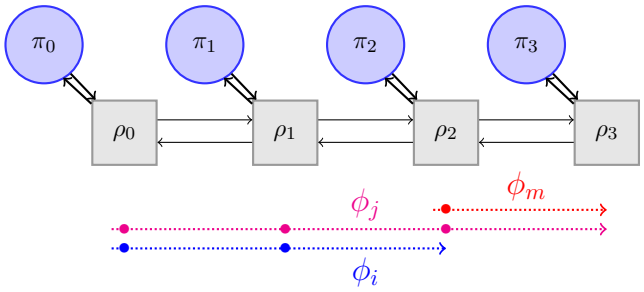
(b) Illustration of an example of an indirect interference.

Figura 3.16 – Examples of two cases where a message $\boldsymbol{\phi}_j$ suffers direct and indirect interference.

As present in [48], the concept of indirect interference is subdivided into two types: (a) upstream, and (b) downstream. These differentiation is due to the position of interference between $\boldsymbol{\phi}_k$ and $\boldsymbol{\phi}_j$ in the contention domain set $cd(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j)$.



(a) Illustration of an example of an upstream indirect interference.



(b) Illustration of an example of a downstream indirect interference.

Figura 3.17 – Examples of two cases where a message $\boldsymbol{\phi}_j$ suffers upstream and downstream indirect interference.

A message $\boldsymbol{\phi}_i$ suffers upstream indirect interference when a higher priority message $\boldsymbol{\phi}_m \in cont_i(\boldsymbol{\phi}_k)$ preempts a message $\boldsymbol{\phi}_j$ before it can direct interfere $\boldsymbol{\phi}_i$. Hence the order of the links of the shared route between these three messages is important. An example case is present in Fig. 3.17a, where a message $\boldsymbol{\phi}_m$ interfere with $\boldsymbol{\phi}_j$ in links before it can interfere $\boldsymbol{\phi}_i$. The set of messages that upstream indirectly interfere with a message $\boldsymbol{\phi}_i$ is expressed as follows:

$$cont_i^u(\boldsymbol{\phi}_j, \boldsymbol{\phi}_i) = \{\boldsymbol{\phi}_k \in cont_i(\boldsymbol{\phi}_i) \cap cont_d(\boldsymbol{\phi}_j) : order(lastl(cd(\boldsymbol{\phi}_j, \boldsymbol{\phi}_k), route(\boldsymbol{\phi}_j))) < order(firstl(cd(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j), route(\boldsymbol{\phi}_j)))\} \tag{3.12}$$

A message $\boldsymbol{\phi}_i$ suffers downstream indirect interference when first it suffers direct interference from a higher priority message $\boldsymbol{\phi}_j$ and then "down" the route $\boldsymbol{\phi}_j$ is interrupted by a third message $\boldsymbol{\phi}_m$ that does not share any link with $route(\boldsymbol{\phi}_i)$. It causes $\boldsymbol{\phi}_i$ to be allowed to keep transmitting during the interval that $\boldsymbol{\phi}_j$ is being blocked. However, as soon $\boldsymbol{\phi}_m$ finishes to interfere with $\boldsymbol{\phi}_j$ and $\boldsymbol{\phi}_j$ restarts to block $\boldsymbol{\phi}_i$, $\boldsymbol{\phi}_i$ in turn restart to be blocked in multiple points of the route, as a "domino" effect, in the buffers of the links of routers in the path of $\boldsymbol{\phi}_i$. This effect shortly presented here is further explained in the works [47] [51]. An example scenario is illustrated in Fig. 3.17a, where a message $\boldsymbol{\phi}_m$ interfere with $\boldsymbol{\phi}_j$ in links after it can interfere $\boldsymbol{\phi}_i$. The set of messages that downstream indirectly interfere with a message $\boldsymbol{\phi}_i$ is expresses as follows:

$$cont_i^d(\boldsymbol{\phi}_j, \boldsymbol{\phi}_i) = \{\boldsymbol{\phi}_k \in cont_i(\boldsymbol{\phi}_i) \cap cont_d(\boldsymbol{\phi}_j) : order(firstl(cd(\boldsymbol{\phi}_j, \boldsymbol{\phi}_k), route(\boldsymbol{\phi}_j))) < order(lastl(cd(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j), route(\boldsymbol{\phi}_j)))\} \tag{3.13}$$

In light of the multiple types of interference that a message $\boldsymbol{\phi}_i$ can suffer, the value of WCLT

S_i for a message $\boldsymbol{\phi}_i$ is formulated as follows:

$$S_i^{(t)} = L_i + \sum_{\boldsymbol{\phi}_j \in cont_d(\boldsymbol{\phi}_i)} \left\lceil \frac{S_i^{(t-1)} + K_j + K_j^I}{T_j} \right\rceil (L_j + I_{j,i}^{down}), \tag{3.14}$$

where $cont_d(\boldsymbol{\phi}_i)$ is the set of messages that interfere $\boldsymbol{\phi}_i$ directly, L_i is the basic latency time for message $\boldsymbol{\phi}_i$ transmission, K_j is the release jitter suffered by a message $\boldsymbol{\phi}_j$ that in turn interferes directly with the message $\boldsymbol{\phi}_i$, and K_j^I is the indirect interference jitter suffered by $\boldsymbol{\phi}_j$ due to messages preempting it and indirect interfering with message $\boldsymbol{\phi}_i$ and its value is defined as $K_j^I = S_j - L_j$.

The term $I_{j,i}^{down}$ is due to downstream indirect interference that the message $\boldsymbol{\phi}_i$ suffers due to other messages interfering with $\boldsymbol{\phi}_i$ and it is calculated using the following equation:

$$I_{j,i}^{down} = \sum_{\boldsymbol{\tau}_k \in cont_i^d(\boldsymbol{\phi}_j, \boldsymbol{\phi}_i)} \left\lceil \frac{S_j + K_k}{T_k} \right\rceil \min(bi(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j), L_k + I_{j,k}^{down}), \tag{3.15}$$

where $cont_i^d(\boldsymbol{\phi}_j, \boldsymbol{\phi}_i)$ is the set of tasks that indirectly interfere task $\boldsymbol{\tau}_i$ and directly interfere task $\boldsymbol{\tau}_j$, S_j is the WCLT of message $\boldsymbol{\tau}_j$, K_k is the jitter time for message $\boldsymbol{\phi}_k$, T_k is the period for task k, L_k is the basic latency time for task $\boldsymbol{\tau}_k$, and $bi(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j)$ is a function that represents the maximum buffered interference time that $\boldsymbol{\phi}_j$ may affect $\boldsymbol{\phi}_j$ in their shared links $cd(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j)$. $bi(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j)$ is formulated as:

$$bi(\boldsymbol{\phi}_j, \boldsymbol{\phi}_i) = buffer(\boldsymbol{\xi})\, L_\xi\, |cd(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j)|, \tag{3.16}$$

where $buffer(\xi)$ is the size of buffers in bytes for routers in the NoC, L_ξ is the maximum time necessary for a router to route and transmit a *flit*, and $cd(\boldsymbol{\phi}_i, \boldsymbol{\phi}_j)$ is the set of links in which both messages $\boldsymbol{\phi}_i$ and $\boldsymbol{\phi}_j$ shares when being transmitted.

Similar to the WCRT schedulability, as present in 3.6.4.2, the WCLT calculation of all WCLT values of messages in an RTA permits the evaluation of whether all these messages $\boldsymbol{\phi} \in \boldsymbol{\Phi}$ arrives in their destinations complying with their respective deadlines, i.e., $S_i \leq D_i$. This schedulability test based on WCLT is both necessary and sufficient.

3.7.5 End-to-End Schedulability Test

End-to-End Schedulability Analysis (EEST) test, as present in [45], take into consideration for the tasks in a RTA being mapped onto a MPSoC based on NoC, whether, each task, in the worst-case scenario, has its WCRT plus the WCLT of its message, if there any, bounded by the task deadline. This summed value is named End-to-End Response Time (EERT), and it covers the time of the task from its arrival time until its message arrives in its destination. Hence, the "end-to-end" in the name.

EERT for a task $\boldsymbol{\tau}_i$ is expressed as $EER_i = R_i + S_i$. This test is necessary and sufficient to show whether a task is schedulable. A task is schedulable when it follows the condition $EER_i \leq D_i$. A RTA $\boldsymbol{\Omega} = \{\boldsymbol{\Gamma}, \boldsymbol{\Phi}\}$ is deemed schedulable, if all tasks with its messages follows the schedulability condition $\boldsymbol{\tau}_i \in \boldsymbol{\Gamma}, EER_i \leq D_i$.

Figure 3.18 displays a simple example timeline where a task execution time is colored gray, and message transmission time is colored yellow similar to the convention used in timelines in Section 3.6.1. In this example, there are two tasks $\boldsymbol{\tau}_a$ and $\boldsymbol{\tau}_b$ with $P_b < P_a$ and with both tasks mapped into the same processor $map(\boldsymbol{\tau}_a) = map(\boldsymbol{\tau}_b) = \pi_0$ emitting, respectively, messages $\boldsymbol{\phi}_a$ and $\boldsymbol{\phi}_b$. Both these messages are send to the same destination and share the path $route(\boldsymbol{\phi}_a) = route(\boldsymbol{\tau}_b) = cd(\boldsymbol{\phi}_a, \boldsymbol{\phi}_b)$.

While a third task $\boldsymbol{\tau}_c$ executes in a different processor $map(\boldsymbol{\tau}_c) \neq \pi_0$ has higher priority than $\boldsymbol{\tau}_a$ and $\boldsymbol{\tau}_b$ with $P_c < P_b < P_a$ and emits a message $\boldsymbol{\phi}_c$ that has contentious links with messages $\boldsymbol{\phi}_a$ and $\boldsymbol{\phi}_b$ with $route(\boldsymbol{\phi}_a) \cap route(\boldsymbol{\tau}_b) \cap route(\boldsymbol{\tau}_c) \neq \emptyset$ and for this reason interrupt directly both messages. As visualized, all three tasks are schedulable, since they do not miss their deadlines with $R_c = 4$, $S_c = 1$, $EER_c = 5$, $R_b = 2$, $S_b = 4$, $EER_b = 6$, $R_a = 3$, $S_a = 5$, and $EER_a = 8$.

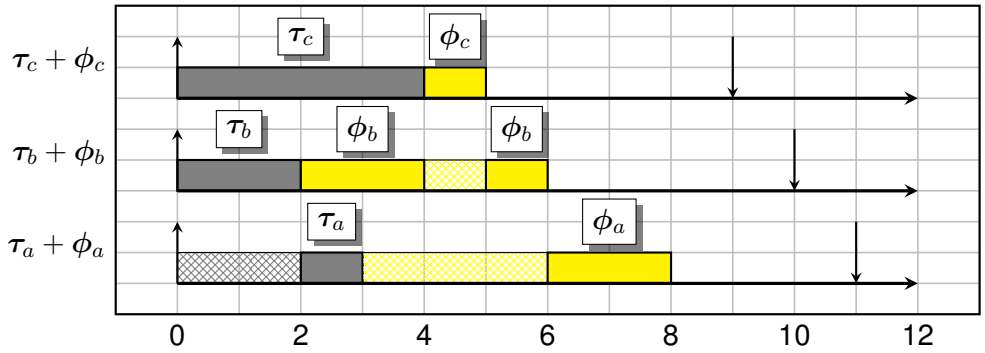


Figura 3.18 – Illustration of timeline for the example of EERT calculation using both WCRT and WCLT.

3.7.6 Required Local Memory for Processor Cores Test

In this work, it is used the required local memory model C present in [49]. The assumption is that each task $\boldsymbol{\tau}_i \in \boldsymbol{\Gamma}$ mapped onto a processor core π_a needs for its execution the following requirements:

- π_a local memory should be capable of storing the task code memory.
- π_a local memory should be capable of storing data from messages in which $\boldsymbol{\tau}_i$ is the destination task. It is considered that $\boldsymbol{\tau}_i$ needs to access this data in local memory for its correct execution.
- π_a local memory should be capable of storing data that $\boldsymbol{\tau}_i$ sends to other tasks.

Given these three requirements, the required local memory for a processor core π_a is the sum of all code memory of tasks mapped to π_a, the sum of data emitted by all messages sent by tasks mapped to π_a, and the sum of data received by tasks mapped onto π_a. The value of required local memory $mreq(\pi_a)$ for processor π_a is calculated as follows:

$$mreq(\pi_a) = \sum_{\boldsymbol{\tau}_k \in map^{-1}(\pi_a)} M_k + \sum_{\boldsymbol{\phi}_s \in send(\pi_a)} Z_s + \sum_{\boldsymbol{\phi}_r \in received(\pi_a)} Z_r, \tag{3.17}$$

where $map^{-1}(\pi_a)$ is the set of tasks mapped to processor core π_a, $send(\pi_a)$ is the set of messages $\boldsymbol{\phi}_s$ sent by tasks mapped onto processor $\pi_a = map(\boldsymbol{\tau}_a)$, $received(\pi_a)$ is the set of messages that has the processor core π_a as their destination, M_k is the memory code in bytes for task $\boldsymbol{\tau}_k$, Z_s is the payload size transmitted by message $\boldsymbol{\phi}_s$, and Z_r is the payload size transmitted by message $\boldsymbol{\phi}_r$.

The set $send(\pi_a)$ is expressed as follows:

$$send(\pi_a) = \{\boldsymbol{\phi}_s \in \boldsymbol{\Phi} : \boldsymbol{\tau}_s \in map^{-1}(\pi_a)\}. \tag{3.18}$$

The set $received(\pi_a)$ has all messages $\boldsymbol{\phi}_r$ received by tasks mapped onto processor $\pi_a = map(\boldsymbol{\tau}_a)$ and it is expressed as follows:

$$received(\pi_a) = \{\boldsymbol{\phi}_r \in \boldsymbol{\Phi} : \boldsymbol{\tau}_{dr} \in map^{-1}(\pi_a)\}, \tag{3.19}$$

where $\boldsymbol{\tau}_d$ is the destination task for a message $\boldsymbol{\phi}_r$ received by the processor π_a.

In order to check whether a processor in MPSoC $\boldsymbol{\Psi}$ is capable to correctly execute an RTA $\boldsymbol{\Omega}$ given a task allocation, all processor cores must have at least the required memory size. This test is expressed as follows:

$$\max\left(\{\pi \in \boldsymbol{\Pi} : mreq(\pi)\right) \leq cap(\boldsymbol{\Psi}). \tag{3.20}$$

3.7.7 Normalized Energy Dissipation Model

In this work, the energy dissipation used is based on the one present in [52]. Power characteristics of RTNoCs depend upon multiple factors, such as, for example, chosen data encoding process for the messages packetization and wire length for the links.

In other words, many of these factors are beyond the scope of this work. The idea is to use a simple yet expressive model that assumes that the energy dissipated by the communication architecture is closely related to the energy dissipated by the sum of energy of the wormhole-based NoC components when routing and transmitting a message.

The resulting model considering the scenario in which the platform is spending the maximum power to transmit a message with one flit of size, namely $\boldsymbol{\phi}_{1flit}$, then a NI connect to a processor $\pi \in \boldsymbol{\Pi}$ dissipates e_π joules when encoding and packetizing $\boldsymbol{\phi}_{1flit}$, a router $\xi \in \boldsymbol{\Xi}$ dissipates e_ξ joules when routing the header flit and buffering and sending the subsequent flit of $\boldsymbol{\phi}_{1flit}$, and a link $\lambda \in \boldsymbol{\Lambda}$ dissipates e_λ joules when transmitting $\boldsymbol{\phi}_{1flit}$.

Assuming that the dissipate energy for each component scale linear with the increase of the size of messages in flits and normalizing for the dissipate energy of a link transmitting $\boldsymbol{\phi}_{1flit}$, then the following formulation is obtained to calculate the normalized energy dissipated during the transmission of a message $\boldsymbol{\phi}_i$:

$$e_{\boldsymbol{\phi}_i} = \left\lceil \frac{8Z_i}{\lambda_{width}} \right\rceil \left(2\frac{e_\pi}{e_\lambda} + (|route(\boldsymbol{\phi}_i)|-1)\frac{e_\xi}{e_\lambda} + |route(\boldsymbol{\phi}_i)|\right), \tag{3.21}$$

where λ_{width} is the flit size of the platform in bits, Z_i is the size of message $\boldsymbol{\phi}_i$ in bytes, and $route(\boldsymbol{\phi}_i)$ is the set of links in which $\boldsymbol{\phi}_i$ is transmitted.

3.8 Task Mapping onto a RTNoC-based MPSoC as an Optimization Problem

In light of the model of the system present in Section 3.7, it is possible to encode the task mapping of a RTA onto the NoC-based MPSoC as an optimization problem and check during the early stages of the design process of a RTNoC whether different characteristics attend restrictions and specifications of a possible application in a real-time embedded system application.

Since multiple characteristics of the system were presented in a fashion that depends upon the task mapping, this section will present each one of them as a different objective function with the same decision variable encoding.

3.8.1 Task Mapping Encoding

The problem of mapping an RTA $\boldsymbol{\Omega}$ with a task set $\boldsymbol{\Gamma}$ into an MPSoC platform with a set of processing cores $\boldsymbol{\Pi}$ is an NP-hard COP similar to the Quadratic Assignment Problem [53]. A mapping solution $\boldsymbol{x}$ pairs each task $\boldsymbol{\tau}_j \in \boldsymbol{\Gamma}$ to a processor core $\pi_n \in \boldsymbol{\Pi}$ and it is encoded as an $|\boldsymbol{\Gamma}|$-dimensional vector of positive integers, where each jth component of $\boldsymbol{x}$ represents the index of the processor responsible for processing the task $\boldsymbol{\tau}_j$, namely $index(map(\boldsymbol{\tau}_j))$, and, for this reason, each x_j ranges inside of $[0, |\boldsymbol{\Pi}|-1]$.

Figure 3.19 illustrates visually the representation of a task mapping encoded as a vector of integers, illustrating the decision variables into the search space.

By using this problem encoding, the search-space of possible task mapping solution is composed of m^n possible mappings, where m is the number of processor cores in the MPSoC platform $|\boldsymbol{Pi}|$ and n is the number of tasks in the task set of the mapped RTA $|\boldsymbol{\Gamma}|$.

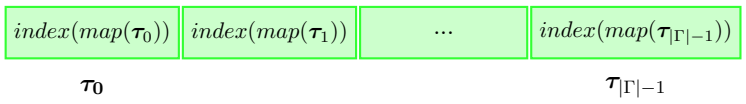


Figura 3.19 – Visual representation of a task mapping where each jth element is an integer value representing the index of a processor where $\boldsymbol{\tau}_j$ is mapped onto.

3.8.2 Utilization Test as an Objective Function

As present in Section 3.7.3, the utilization schedulability test for both processors and links depending on the task assignment of an RTA onto the MPSoC platform. Therefore, it is possible to use this test as an optimization function to find an optimal task placement where the tasks and their messages are schedulable under this test. For this reason, we define the objective function f_{util} as follows:

$$f_{util} : \{\mathbb{N}^{|\boldsymbol{\Gamma}|}, \boldsymbol{\Omega}, \boldsymbol{\Psi}\} \rightarrow \mathbb{N}, \tag{3.22}$$

where f_{util} returns the number of processor cores and links in the platform $\boldsymbol{\Psi}$ are above their capacity given a task mapping $\boldsymbol{x}$ for a RTA $\boldsymbol{\Omega}$. The formulation for f_{util} is equal to equation 3.9 as follows:

$$f_{util}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}) = |\{\lambda \in \boldsymbol{\Lambda} : U_\lambda > 1\}| + |\{\pi \in \boldsymbol{\Pi} : U_\pi > 1\}|. \tag{3.23}$$

The optimization problem formulation for the utilization test is:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f_{util}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), \tag{3.24}$$

where the goal is to find the optimal solution $\boldsymbol{x}^*$ representing a schedulable task mapping solution with none processors and links over their maximum capacity. In other words, $f_{util}(\boldsymbol{x}^*) = 0$. However, depending on both the platform $\boldsymbol{\Psi}$ and the application $\boldsymbol{\Omega}$ a schedulable solution may not exist, in this case, $\boldsymbol{x}^*$ represents the task mapping solution where fewer processors and links are over their maximum capacity. A shortcoming of this method is that it uses a necessary but not sufficient method, so an optimal solution where $f_{util}(\boldsymbol{x}) = 0$ is not guaranteed to be schedulable.

3.8.3 End-to-End Schedulability Test as an Objective Function

In the same fashion as the utilization test, it is possible to express the search for a schedulable mapping solution using the EEST (Section 3.7.5) as an optimization process. To this end, an objective function $f_{unsc} : \{\mathbb{N}^{|\boldsymbol{\Gamma}|}, \boldsymbol{\Omega}, \boldsymbol{\Psi}\} \to \mathbb{N}$ is defined as:

$$f_{unsc}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}) = |\{\boldsymbol{\tau}_i \in \boldsymbol{\Gamma} \,:\, EER_i > D_i\}|. \tag{3.25}$$

This function returns the number of tasks that are unschedulable under the task mapping $\boldsymbol{x}$.

At last, the formulation of the search for a task mapping where all tasks are schedulable using the end-to-end schedulability test as an optimization problem is:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f_{unsc}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}). \tag{3.26}$$

In Eq. 3.26, the goal is to find an optimal mapping solution $\boldsymbol{x}^*$ in which all tasks are schedulable $f_{unsc}(\boldsymbol{x}^*) = 0$. Since such solution may not exist, at least, the optimization of f_{unsch} results in a solution with fewer unschedulable tasks and, depending on the type of RTS at hand, for example, soft RTS, this solution can still be useful. Another advantage is that the method used is necessary and sufficient guarantying that a solution where $f_{unsc}(\boldsymbol{x}^*) = 0$ is schedulable.

3.8.4 Normalized Energy Dissipation as an Objective Function

Since lower power consumed is an essential requirement of many embedded devices including RTS, it is crucial to define as well the search of task mapping that reduces as much as possible the energy dissipated by the system. For this reason, the function $f_{ener} : \{\mathbb{N}^{|\boldsymbol{\Gamma}|}, \boldsymbol{\Omega}, \boldsymbol{\Psi}\} \to \mathbb{R}$ is defined to evaluate the energy dissipated of the platform $\boldsymbol{\Psi}$ given a task mapping $\boldsymbol{x}$ for an application $\boldsymbol{\Omega}$. The formulation for this objective function is as follows: Using Eq. 3.21, it is possible to obtain

the energy dissipated for a NoC transmitting data from a RTA mapped onto the system with the following equation:

$$f_{ener}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}) = \sum_{\phi \in \boldsymbol{\Phi}} e_{\phi}. \tag{3.27}$$

The formulation of the process to reduce the energy dissipated by the system's communication component as an optimization problem is:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f_{ener}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), \tag{3.28}$$

where $\boldsymbol{x}^*$ represents a task mapping solution with the minimum of energy dissipated by the transmission of messages inside of the platform.

3.8.5 Maximum Required Local Memory as an Objective Function

In his work [49], present a model for private local memory as well a secondary objective function in a MOOP that is based on the system maximum required the amount of private local memory for all processors in a platform $\boldsymbol{\Psi}$. This function $f_{mreq} : \{\mathbb{N}^{|\boldsymbol{\Gamma}|}, \boldsymbol{\Omega}, \boldsymbol{\Psi}\} \rightarrow \mathbb{N}$ receives a task mapping for a RTA to be mapped onto the platform and returns the maximum amount of required memory in bytes. The formulation for this function is as follows:

$$f_{mreq}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}) = \max\left(\{mreq(\pi) : \pi \in \boldsymbol{\Pi}\}\right) \tag{3.29}$$

The formulation of f_{bdf} as an objective function to be optimized is:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f_{mreq}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), \tag{3.30}$$

where $\boldsymbol{x}^*$ represents a task mapping solution with the minimum required amount of local memory for the processor cores.

3.8.6 Breakdown Frequency as an Objective Function

In [54], Sayuti presents a possible objective function for an MPSoC-based RTS that is based on the concept of breakdown frequency to scale up or down the processor cores frequency clock up to the point where all tasks are schedulable given a task mapping. BreakDown Frequency (BDF) is the minimal operational frequency of a system where all tasks and messages respect their deadlines.

The BDF scaling method uses the EEST to evaluate whether the use of specific scaling frequencies values, in a range of possible ones, that is capable of turning all tasks running in the system schedulable. The algorithm to search for the minimum closest suitable scaling value uses a binary search, and every iteration checks whether the current scaling value is schedulable or not. If the current scaling value in an iteration results in a schedulable system, the binary search reduces the frequency scaling factor. Otherwise, it increases the scaling value.

Since it uses a binary search, each iteration also reduces by half the number of scaling frequency values to be searched. The search for the minimum BDF value is displayed in Algorithm 3, where $f_{bdf} : \{\mathbb{N}^{|\mathbf{\Gamma}|}, \mathbf{\Omega}, \mathbf{\Psi}\} \rightarrow \mathbb{R}$ is a function that given a task mapping returns the closest minimum BDF scaling in a list of frequency scaling values (attached as appendix I in this document).

Algorithm 3 BreakDown Frequency Search (f_{bdf})

INPUT: Task Placement ($\boldsymbol{x}$)
Platform ($\mathbf{\Psi}$)
Application ($\mathbf{\Omega}$)
OUTPUT: Frequency Scaling Value (f_{scur})

procedure $BDFSearch(\boldsymbol{x})$
  $\boldsymbol{L_f} \leftarrow$ List of Frequency Scalings
  $f_{scur} = f_{snext} = 1$
  while $f_{snext} \in \boldsymbol{L_f}$ **do**
    $f_{scur} = f_{snext}$
    Scale $\mathbf{\Psi}$ *frequency by* f_{scur}
    if $f_{unsc}(\boldsymbol{x}, \mathbf{\Psi}, \mathbf{\Omega}) = 0$ **then**
      Reduce Frequency f_{snext}
    else
      Increase Frequency f_{snext}
    end if
  end while
return f_{scur}
end procedure

The formulation of f_{bdf} as an objective function to be optimized is:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f_{bdf}(\boldsymbol{x}, \mathbf{\Omega}, \mathbf{\Psi}), \tag{3.31}$$

where $\boldsymbol{x}^*$ represents a task mapping solution with the minimum breakdown frequency scaling value for an application $\mathbf{\Omega}$ mapped onto a platform $\mathbf{\Psi}$. Since the energy dissipated by the system is linearly proportional with its frequency [55], a reduction on the frequency also reduces the power consumed by the platform. Another advantage of f_{bdf} is that once its optimization process reaches a task mapping solution $\boldsymbol{x}$ with a scaling values $f_{bdf}(\boldsymbol{x}) \leq 1$, it means that that solution is equivalent to a solution with $f_{unsch}(\boldsymbol{x}) = 0$, therefore, a complete schedulable task mapping solution.

Beyond that, the optimization f_{bdf} is even capable to improve further schedulable solution by finding differences in schedulable task mappings something that the optimization of the function f_{unsch} is not capable to achieve. For example, given two schedulable task mapping solutions $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ with $f_{unsch}(\boldsymbol{x}_1) = f_{unsch}(\boldsymbol{x}_2) = 0$, using the breakdown frequency scaling optimization $\boldsymbol{x}_1$ would be chosen instead of $\boldsymbol{x}_2$, because $f_{bdf}(\boldsymbol{x}_1) < f_{bdf}(\boldsymbol{x}_2)$.

However, there are two main drawbacks with the optimization of f_{bdf}. The first is that the objective function f_{bdf} calculation takes multiple evaluations of f_{unsch}. Since f_{unsch} by itself already takes a considerable execution time due to its complex analysis, f_{bdf} execution time in some cases are prohibitive a characteristic pointed out by the author in [54].

The second point is that for task mapping solutions with resulting scaling values above 1, the f_{bdf} relationship with f_{unsc} is not one-to-one in the sense of a decrement in the number of unschedulable tasks may actually have an increment in the breakdown scaling frequency. For example, given two mappings $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$, for the mapping of a single application in the same platform, where $f_{unsch}(\boldsymbol{x}_a) = 3$ and $f_{unsch}(\boldsymbol{x}_b) = 6$, now imagine that the lateness of the 6

unschedulable tasks in $\boldsymbol{x}_b$ is very small when compared with the lateness of the 3 unschedulable tasks in $\boldsymbol{x}_a$, it would cause the evaluation of these solutions using the BDF method to be $f_{bdf}(\boldsymbol{x}_a) > f_{bdf}(\boldsymbol{x}_b)$ even though $f_{unsch}(\boldsymbol{x}_a) < f_{unsch}(\boldsymbol{x}_b)$.

3.8.7 End-to-End Scheduling Test with Slack Awareness as an Objective Function: a new Proposed Approach

Inspired by breakdown frequency method [54], the author formulated an objective function that could at the same time keep the information of the number of unschedulable tasks in a specific task mapping being capable of differentiating and further improving solutions where all tasks are already schedulable. This two features using evaluating only once f_{unsch} that is a complex objective function with a large computation time when evaluating the mapping of RTAs with large task sets. This objective function called here f_{umsr} where *umsr* stands for *unschedulable tasks* and Minimum Slack Ratio, uses both the information of f_{unsch} together with tasks slack values to achieve just that.

$f_{umsr} : \{\mathbb{N}^{|\boldsymbol{\Gamma}|}, \boldsymbol{\Omega}, \boldsymbol{\Psi}\} \rightarrow \mathbb{R}$ receives a possible task mapping $\boldsymbol{x}$ for a RTA $\boldsymbol{\Omega}$ onto a platform $\boldsymbol{\Psi}$ and assess whether all tasks are schedulable, if they are not, it returns the number of unschedulable tasks exactly as $f_{unsch}(\boldsymbol{x})$.

However, if $f_{unsch}(\boldsymbol{x}) = 0$ (that is all task are schedulable), it then calculates the slack time of all tasks, calculates the ratio between slack time and the deadline for all tasks and then selects the task with the minimum slack time and returns the negative value for the minimum slack time and deadline ratio. Reminding the reader that the slack time used is for the EERT of the tasks, i.e., for a given schedulable task $\boldsymbol{\tau}_i$ its slack is $s_i = D_i - EER_i$.

The formulation for this objective function is as follows:

$$f_{umsr}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}) = \begin{cases} f_{unsch}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), & \text{if } f_{unsch}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}) > 0 \\ -\min(\{\frac{s_i}{D_i} : \boldsymbol{\tau}_i \in \boldsymbol{\Gamma}\}), & \text{otherwise} \end{cases}, \tag{3.32}$$

where f_{unsch} is the EEST as an objective function as presented in Section 3.8.3, s_i and D_i are the slack time and deadline for a schedulable task $\boldsymbol{\tau}_i$. The reason in which the negative of the minimum slack ratio is used ($-min()$) is to focus the evaluation on the task with the minimum slack. In this manner as f_{umsr} is optimized the task with the minimum slack ratio has this metric increasing through the optimization process since the minimization of a function returning a negative value is equivalent to the maximization for that function absolute value.

The optimization process for f_{umsr} as an objective function is:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}}\, f_{umsr}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), \tag{3.33}$$

where $\boldsymbol{x}^*$ represents the task mapping solution with either the minimum number of unschedulable tasks or the minimum slack time and deadline ratio.

The advantage of this approach are three: (a) it only uses a single call of f_{unsch}; (b) it gives information about both number of unschedulable tasks and slack and deadline ratio with a single

real number; (c) advantage is that all tasks slack ratios are increased during the optimization process, and it is possible to use this dimensionless value to calculate a reduction of frequency scale for the system processors. Since even the task with the worst slack ratio will also attend its time requirements;

For example, given a task mapping solution $\boldsymbol{x}_s$ that when used to assign tasks from a RTA $\boldsymbol{\Omega}$ to a platform $\boldsymbol{\Psi}$ results in a schedulable applications $f_{unsch}(\boldsymbol{x}_s) = 0$ and it also results in $f_{umsr}(\boldsymbol{x}_s) = -0.05$. It is the same to say that all tasks have at least 5% of their time as slack time, so, if all tasks have their EER time increased by $\frac{1}{(1-0.05)}$ they would still meet their deadlines. It means that one can use the value $(1 + f_{umsr}(\boldsymbol{x}_s))$ as the scaling value to reduce the processor cores frequency and all tasks would still be schedulable.

3.8.8 Multi-Objective Task Mapping Optimization

In light of the previously defined objective functions, one could unify them in a multi-objectives function to optimize multiple characteristics concurrently of an analyzed RTS.

The $\boldsymbol{F}_{noc} : \{\mathbb{N}^{|\boldsymbol{\Gamma}|}, \boldsymbol{\Omega}, \boldsymbol{\Psi}\} \rightarrow \{\mathbb{R}, \mathbb{N}, \mathbb{R}, \mathbb{N}\}$ has the following formulation for its objectives:

$$\boldsymbol{F}_{noc}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}) = \{f_{umsr}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), f_{mreq}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), f_{ener}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), f_{util}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi})\}. \tag{3.34}$$

This function is formed by the following objectives: (a) number of unschedulable tasks or negative minimum slack and deadline, (b) maximum required memory in bytes, (c) total normalized dissipated energy, and (d) number of processors and links over utilized in the system.

The process of optimize function $\boldsymbol{F}_{noc}$ in a Pareto sense is represented as:

$$\boldsymbol{X}^* = \underset{\boldsymbol{x}}{\operatorname{argmin}} F_{noc}(\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}), \tag{3.35}$$

where $\boldsymbol{X}^*$ is the Pareto Optimal solutions with the set $\boldsymbol{F}_{noc}(\boldsymbol{X}^*)$, that is the PF for this function.

3.9 Conclusions of the Chapter

This chapter introduces the concepts behind system-on-a-chip, network-on-a-chip, and real-time systems. Information that is used to explain the context, modeling, and reasoning used to define the task mapping of a real-time application onto an MPSoC based on an NoC as an optimization problem. Since the search of a task mapping solution where all tasks are schedulable is an NP-hard problem, the optimization problem defined here is tackled using bio-inspired meta-heuristics for single and multi-objective problems as present in Chapter 4.

Between the multiple possible objectives functions defined for the optimization for task mapping of RTAs onto RTNoC-based MPSoC, two of them were developed in this work: (a) f_{umsr} that mix the number of unschedulable tasks under the EERT analysis and the negative minimum slack deadline times ratio; (b) $\boldsymbol{F}_{noc}$ that is multi-objective and contains four objectives, including f_{umsr}, and information about required maximum local memory, dissipated energy by the system, and resource utilization in the form of utilization of processors and links tests.

The task mapping problem is used as the main real-life optimization scenario in this work. It is revisited in Chapter 7 when defining an experimental setup to evaluate different meta-heuristics when optimizing multiple aspects of the design of RTNoC-based MPSoC.

4 SEARCH-BASED OPTIMIZATION BIO-INSPIRED META-HEURISTICS

This chapter introduces bio-inspired meta-heuristics to solve single-/multi-objective optimization problems with emphasis on adaptive techniques to control these algorithms parameters during execution. The chapter is divided as follows: Section 4.1 presents and contextualizes bio-inspired meta-heuristics including a taxonomy for the ones used in this work; Section 4.2 defines adaptive techniques for bio-inspired meta-heuristics including their classification; Section 4.3 presents the single-objective meta-heuristics from other works in the literature; Section 4.4 presents the single-objective meta-heuristics developed in this work; Section 4.5 the multi-objective meta-heuristics from other works in the literature; Section 4.6 presents the multi-objective meta-heuristics developed in this work; Section 4.7 concludes the chapter and contextualizes it with other parts of this work.

4.1 Bio-inspired Meta-Heuristics

The algorithms responsible for solving optimization problems can be categorized into two main groups [56]: (a) exact, and (b) approximated.

Exact methods guarantee that the output solution is the optimal one. However, exact methods have their usability prohibitive in terms of necessary computing time in many problem stances. Meanwhile, approximated methods, also called *soft-computing*, do not hold the guarantee of optimality for their results. Instead, these methods output are near-optimal solutions delivered in a maximum amount of time.

Approximated methods can be further subdivided into two groups: (a) heuristics and (b) meta-heuristics [56] [18] . Heuristic algorithms are search-based optimization methods that try to obtain good solutions using experienced-based techniques to search intelligently, in promising regions of the search space instead of a brute-force approach that enumerates and exhaustively search for solutions. On the other hand, meta-heuristics approaches are composed of multiple heuristics as sub-procedures that are used to search for near-optimal solutions interactively.

In this context, *Bio-inspired meta-heuristic algorithms* are a subset of *soft-computing* methods that use biological phenomena present in natural systems as metaphors to their heuristic strategies to search for solutions of optimization problems [56]. The efficiency of these algorithms is linked

to their capabilities to **explore** a broad range of possible solutions by different search agents and **exploit** the information gathered by them to avoid possible local optima.

Bio-inspired meta-heuristics display the following shared framework: (a) initialize search agents with randomly generated possible solutions inside of the search space; (b) repeat iteratively search operations that are used to explore and exploit solutions in the search space while converging in a (near-)optimal mapping until a stop criterion is met;

The interest of the scientific community in meta-heuristic methods, including the intense generation of new methods in the past decade, uses as theoretical base the *No Free Lunch* (NFL) Theorem. The NFL theorem [57] [58] shows that all meta-heuristics algorithms have the same performance on average when applied to all possible problems. In other words, if a meta-heuristic $Algorithm_1$ achieves better results than another $Algorithm_2$ when applied to a set of problems, $Algorithm_1$ performs worse when applied on another set of problems, where $Algorithm_2$ performs better. For this reason, to decide what is the best bio-inspired meta-heuristic to optimize a complicated family of optimization problems, it is essential to use and compare a significant number of meta-heuristics on a range of combinations of problems.

In this work, bio-inspired meta-heuristics are categorized into two groups based on their metaphors: (a) **evolutionary** and (b) **swarm intelligence** algorithms.

Evolutionary algorithms are inspired by Darwinian evolution theory that describes the method in which a species population changes their individuals genetic material through generations as an adaptation to the environment. Algorithms in this category encode the possible solution as the genetic material of an individual of the population and use the evaluation of the objective function as the environmental pressure, forcing the population to adapt and consequently have better individuals that correspond to possible improved solutions to an optimization problem.

Swarm intelligence algorithms use as inspiration for their metaphors the social behavior displayed by some species of animals. Animals of these species have their members operate in a dispersed fashion but, as a group, they are capable of accomplishing a common goal, for example, search for food, mate or escape predators. Algorithms, in this category, generally encode a possible solution as the position of an individual in the group and use the evaluation of the objective function to force the movement of individuals towards improved solutions.

Figure 4.1 presents the taxonomy for bio-inspired algorithms for SOOP used in this work. They are organized in a tree-like structure that points out the categories and relationships between the different algorithms. The meta-heuristics displayed in Fig. 4.1 are identified by their acronyms present as following: (a) Genetic Algorithm (GA) (Section 4.3.1), (b) Differential Evolution (DE) (Section 4.3.2), (c) Particle Swarm Optimization (PSO) (Section 4.3.3), (d) Salp Swarm Algorithm (SSA) (Section 4.3.4), (e) Gray Wolf Optimization (GWO) (Section 4.3.5), (f) Elephant Herd Optimization (EHO) (Section 4.3.6), (g) Dragonfly Algorithm (DA) (Section 4.3.7), (h) Moth-Flame Optimization (MFO) (Section 4.3.8), (i) Whale Optimization Algorithm (WOA) (Section 4.3.9), (j) Bat Algorithm (BA) (Section 4.3.10), (k) Adaptive DE (JADE) (Section 4.3.11), (l) Crossover Strategy Adaptive - (Self)-Adaptive DE (CSASADE) (Section 4.3.12), (m) Discrete PSO (DPSO) (Section 4.3.13), (n) Self-Adaptive PSO (SAPSO) (Section 4.3.14), (o) Hybrid Discrete PSO

Makespan-based (HDPSO-M) (Section 4.3.15), (p) Single-Objective Adaptive DE (SOAMSDE) (Section 4.4.1), (q) Adaptive GA (AGAv1) (Section 4.4.2), (r) Adaptive GA (AGAv2) (Section 4.4.3), (s) Adaptive GA (AGAv3) (Section 4.4.4), (t) Adaptive GA (AGAv4) (Section 4.4.5), (u) Adaptive PSO (APSO) (Section 4.4.6), (v) Adaptive PSO (APSOv2) (Section 4.4.7), (w) Hybrid Discrete PSO Utilization-based (HDPSO-U) (Section 4.4.8), (x) Hybrid Discrete Adaptive PSO Utilization-based (AHDPSO-U) (Section 4.4.9), and (y) Adaptive Gray Wolf Optimization (AGWO) (Section 4.3.5).

Figure 4.2 presents a similar taxonomy but for bio-inspired meta-heuristics for MOOP used in this work. The list of used meta-heuristic for multi-objective is shorter and are composed of the following algorithms: (a) Non-dominant Sorting GA (NSGA-II) (Section 4.5.1), (b) Adaptive Parameter with Mutation Tournament Multi-Objective DE (APMTMODE) (Section 4.5.2), (c) Non-dominant Sorting Adaptive GA (NSAGA) (Section 4.6.1), and (d) Multi-Objective Non-dominant Sorting Adaptive DE (MONSADE) (Section 4.6.2).

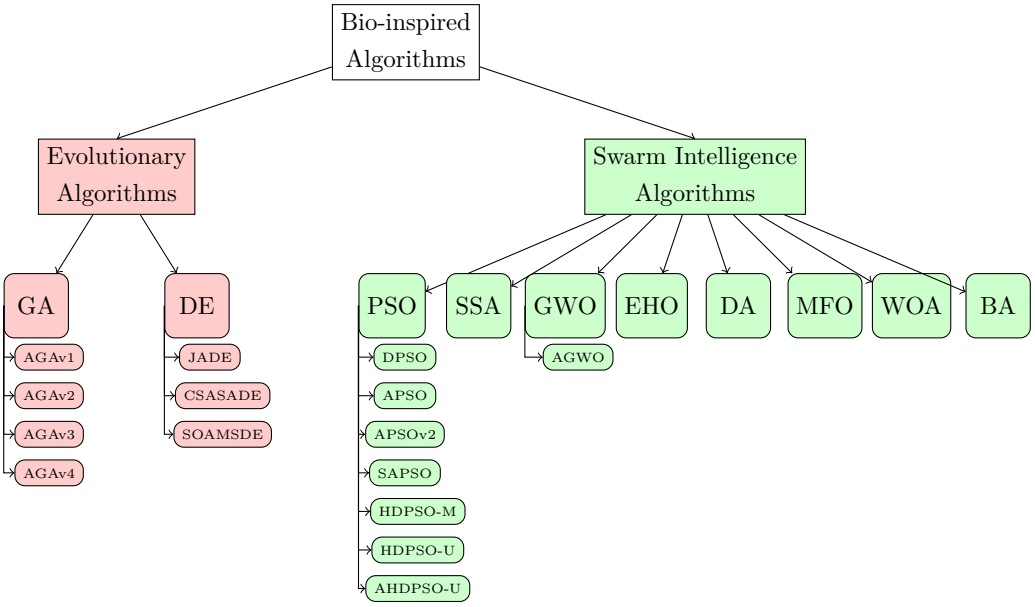


Figura 4.1 – Classification structure of bio-inspired meta-heuristics for SOOP.

4.2 Adaptive Techniques for Bio-Inspired Meta-Heuristics

Adaptive techniques for search-based optimization meta-heuristics can be categorized for their goal: (a) **parameter control strategies** a subset of **parameter setting strategies** (Section 4.2.1) are responsible to adjust parameters during a meta-heuristic execution; and (b) **adaptive operation selection** changes heuristic operators used by a meta-heuristic during their execution (Section 4.2.2).

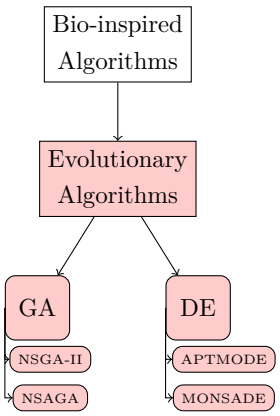


Figura 4.2 – Classification structure of bio-inspired meta-heuristics for MOOP.

4.2.1 Parameter Setting Strategies

Bio-inspired meta-heuristics relies upon parameters values to determine their behavior and generally defines whether their search operators that are responsible for exploration and exploitation act aggressively in a local search or focus on a global search. The selection of these parameters values decides whether the algorithm is capable to correctly converge to an optimal value effectively or stop in a local minimum with premature convergence.

- A taxonomy of the parameters: These parameters are either **internal** or **external** ones. **Internal** parameters are integrated into the meta-heuristic procedures with values decided by the algorithm designer and not being capable of alteration by a user. Meanwhile, **external** parameters are capable of being changed and tuned by a user. Both these types of parameters values have to be configured somehow by the algorithm designer or the algorithm user using a **parameter setting strategy**. The difference is that since the user does not see internal parameters, it is entirely up to the algorithm designer to correctly set them.

- A taxonomy of the parameter setting: Parameters values that are immutable during the execution of a meta-heuristic are deemed as **static**. In many cases, even static parameters that have their values correctly set in a meta-heuristic are only suitable for a specific problem, or family of problems, and does not achieve suitable results when applied to other groups of optimization problems. For some pairs of meta-heuristic static parameters and specific problems, this overspecialization of parameter values goes so far as being only satisfactory for a stage of the problem optimization. This only further illustrates the importance of **parameter setting strategies** in meta-heuristic algorithms.

  As stated in [59] and then revisited in [60], there are two possible modes for parameters values setting strategies in meta-heuristics: (a) **parameter tuning**, and (b) **parameter control**. The search for (near-)optimal parameter values in a meta-heuristic by these parameter setting strategies by themselves are optimization problems that need to be optimized on top of the search-based optimization performed by the bio-inspired meta-heuristic in hand. For some optimization problems, the search space size of possible parameter values is

even larger than the search space of the problem being optimized depending on how many static parameters the meta-heuristic has.

- **Parameter tuning**: it is a strategy to search for (near-)optimal values for **static** parameters for all iterations of the heuristic algorithm. This search is generally an empirical trial-and-error method where the whole optimization process is repeated multiple times to find parameters that bear good results when compared with others. If the objective function evaluation is slow, the parameter tuning then becomes time-consuming or even intractable. An example of parameter tuning in action is presented by [14], where there was an intensive search to identify the best parameter values for a genetic algorithm applied to a family of task mapping problems.
- **Parameter control**: it is an alternative that adds techniques that are capable of dynamically changing parameters values during a meta-heuristic execution while, consequently, mitigating its dependence on a proper time-consuming initial parameter tuning while improving the algorithm performance. **Parameter control** mechanisms are subdivided into three groups:
  - **Deterministic** strategies: The parameter values are changed using rules that do not take into consideration the algorithm performance. An example of this type of strategy is present in iteration varying parameters such as the inertia weight present on PSO-based algorithms [61].
  - **Adaptive** strategies: Adaptation mechanisms that receive feedback from the algorithm performance and alter the values of the parameters based on this information. These modifications can also adjust the direction and speed in which the values of the parameters change.
  - **Self-Adaptive** strategies: Adaptation mechanisms that encode the parameter values into the algorithm decision variables and uses its optimization mechanisms to search for the best set of parameters while optimizing the fitness function.

Figure 4.3 illustrates the taxonomy of **parameter setting strategies** for bio-inspired meta-heuristics.

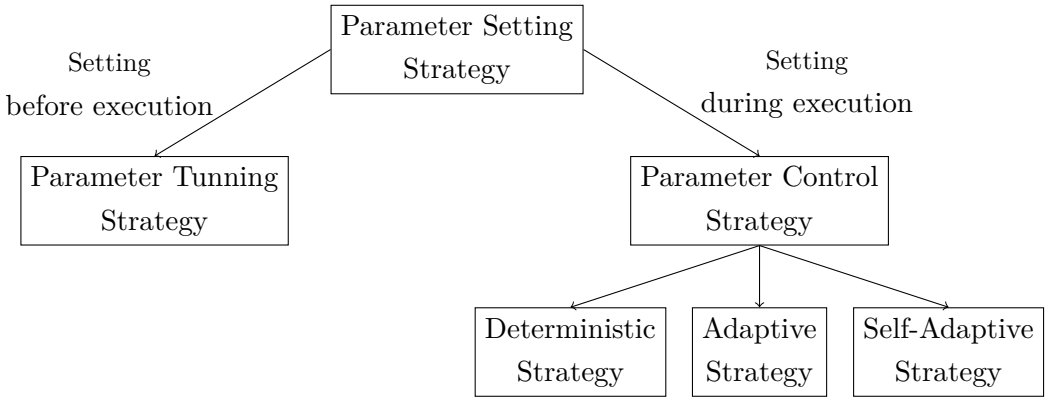


Figura 4.3 – Classification of bio-inspired algorithms parameter setting strategies.

4.2.2 Adaptive Operator Selection

Adaptive Operator Selection mechanisms are part of another category of adaptive techniques for bio-inspired meta-heuristics that, instead of changing parameter values, they change heuristic operators. Generally, **Adaptive Operator Selection** chooses a search operator in a pool of possible operators depending on the performance displayed by a meta-heuristic, a specific operator or a search agent, and then combine or substitute them with other heuristic operators already implemented in the meta-heuristic operator.

In this work, adaptive techniques (**parameter control strategies** and **adaptive operator selection**) are also categorized depending on the following aspects of their functionality as mentioned in [60]:

- **How changes are made**: Defines the type of mechanism used, these mechanisms may be, for example, static (with no change), deterministic, adaptive, and self-adaptive.
- **Parameter changed**: Since a meta-heuristic can adopt multiple different parameter setting strategies for their various parameters, it is important to define which parameter values are changed by which parameter setting strategy.
- **Scope of change**: Defines the range of influence of an adaptive mechanism that could affect the population or just an individual search agent.
- **Evidence for changes**: Defines the metric used to identify the need for change, for example, an adaptation mechanism may rely upon the objective function as a metric, or diversity of solutions in the population of search agents.

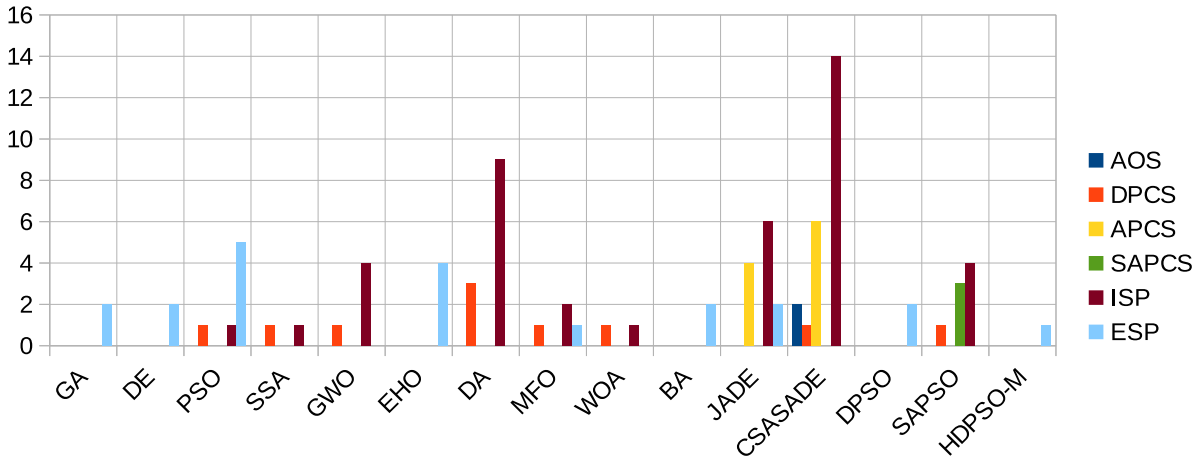
Regarding common parameters in bio-inspired algorithms, the number of search agents in a meta-heuristic is a static external parameter shared by all bio-inspired algorithms. The size of the population of search agents influences whether an algorithm starts with enough information for the problem components to exploit upon and consequently reducing the convergence time [62] and if the algorithm does not have enough search agents to escape a local minima region [63] and [64]. Since it is a **static** parameter, the correct approach is to use parameter tuning strategies to define which population or swarm size should be used by a bio-inspired meta-heuristic when solving a problem. However, in this work, the number of search agents is a parameter that was not focused upon and not treated by parameter control strategies.

There are in the literature meta-heuristics that contains parameter control strategies to deal with the size of the population, such as, for example, LSHADE [65]. Other common types of parameters shared by multiple meta-heuristics in this work are those parameter values that control adaptive techniques, including parameter control and adaptive operation selection strategies. These parameters are commonly treated as intern static ones with values controlled only by the algorithm designer. However, these parameters can also be adjusted by other parameter control strategies affecting meta-heuristic behavior when searching for optimal parameters.

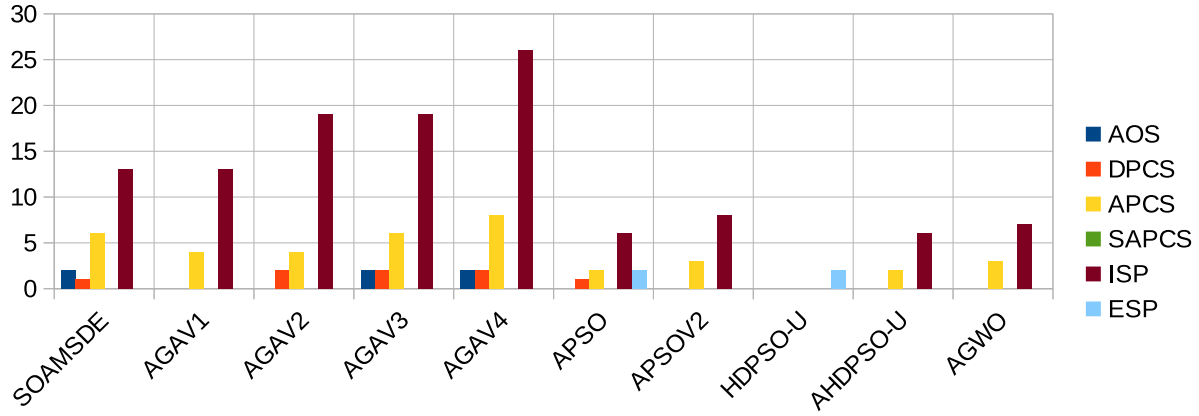
By grouping the meta-heuristics based on the adaptive technique present in its methods, this work has the following groups:

- **No adaptive methods**: The following meta-heuristics does not contain any adaptive methods: (a) GA, (b) DE, (c) EHO, (d) BA, (e) DPSO, (f) HDPO-M, (g) HDPSO-U, and (h) NSGA-II.
- **Deterministic Parameter Control Strategies**: The following meta-heuristics contains this type of adaptation methods: (a) PSO, (b) SSA, (c) GWO, (d) DA, (e) MFO, (f) WOA, (g) CSASADE, (h) SAPSO, (i) SOAMSDE, (j) AGAv2, (k) AGAv3, (l) AGAv4, (m) APSO, (n) APMTMODE, (o) NSAGA, and (p) MONSADE.
- **Adaptive Parameter Control Strategies**: The following meta-heuristics contains adaptation methods of this type: (a) JADE, (b) CSASADE, (c) SOAMSDE, (d) AGAv1, (e) AGAv2, (f) AGAv3, (g) AGAv4, (h) APSO, (i) APSOv2, (j) AHDPSO-U, (k) AGWO, (l) APMTMODE, (m) NSAGA, and (n) MONSADE.
- **Self-Adaptive Parameter Control Strategies**: There are only one meta-heuristic that uses this strategy: SAPSO.
- **Adaptive Operation Selection Techniques**: The following meta-heuristics contains this type of adaptation schemes: (a) CSASADE, (b) SOAMSDE, (c) AGAv3, (d) AGAv4, (e) APMTMODE, (f) NSAGA, and (g) MONSADE.

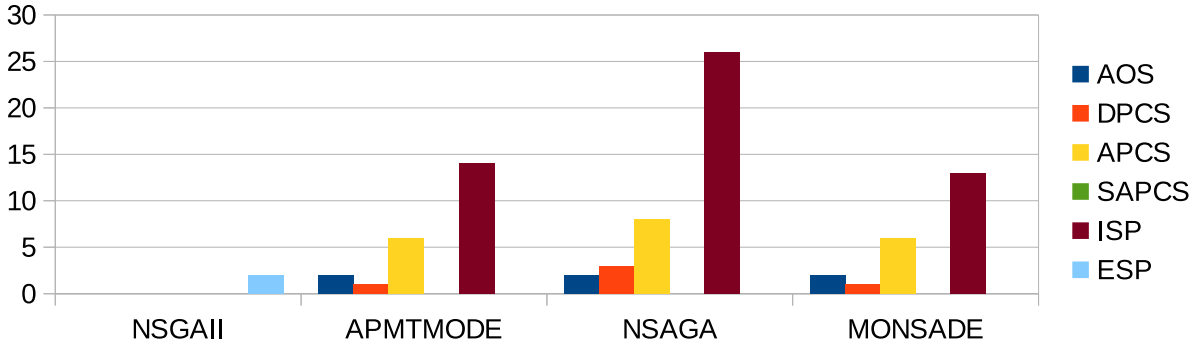
Figure 4.4 holds images that show an enumeration for the parameters, parameters strategies settings and adaptive operation selection mechanisms to control them, for the meta-heuristics used in this work. In this figure, the following acronyms are used (**AOS**) Adaptive Operator Selection, (**DPCS**) Deterministic Parameter Control Strategy, (**APCS**) Adaptive Parameter Control Strategy, (**SAPCS**) Self-Adaptive Parameter Control Strategy, (**ISP**) Internal Static Parameter, and (**ESP**) External Static Parameter.



(a) SOOP from literature, where the y-axis represents the number of parameters or adaptive techniques.



(b) SOOP developed in this work, where the y-axis represents the number of parameters or adaptive techniques.



(c) MOOP.

Figura 4.4 – Number of parameters and parameter setting strategies for each meta-heuristic in this work.

4.3 Single-Objective Optimization Bio-Inspired Meta-Heuristics from Literature

4.3.1 Genetic Algorithm (GA)

4.3.1.1 Meta-Heuristic Description

Genetic Algorithm (GA) is an evolutionary meta-heuristic in which the solutions held by search agents are interpreted to be genetic material in individuals in a population $\mathbf{P}$ of size n. Each ith individual holds a possible solution $\boldsymbol{x}_i$ that is an integer vector formed by d components and each component is called a chromosome. The GA implemented in this work is based on the ones present in [14] and [45]. Algorithm 4 describes the implemented GA pseudo-code.

Algorithm 4 Genetic Algorithm (GA)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of chromosomes (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Mutation rate (p_m)
Crossover rate (p_c)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

1: **procedure** GA($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}, p_m, p_c$)
2: *Initialize* $\boldsymbol{x}_i (i = 1, ..., n)$
3: *Evaluate* $f(\boldsymbol{x}_i)$
4: *Obtain best Individual* $\boldsymbol{x}_{best}$
5: **for** $k = 1$ to $I - 1$ **do**
6: $\mathbf{P}_{new} = \emptyset$
7: **for** $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
8: *Binary Tournament Selection*
9: *One-Point Crossover*
10: *Uniform Mutation*
11: *Evaluate generated offsprings*
12: $\boldsymbol{P}_{new} \leftarrow \boldsymbol{x}_{new1}$
13: $\boldsymbol{P}_{new} \leftarrow \boldsymbol{x}_{new2}$
14: **end for**
15: $\boldsymbol{P} = Sort\ \boldsymbol{P} \cup \boldsymbol{P}_{new}$
16: *Remove individuals in* $\boldsymbol{P}$ *until* $|\boldsymbol{P}| = n$
17: *Update* $\boldsymbol{x}_{best}$
18: **end for**
19: **return** $\boldsymbol{x}_{best}$
20: **end procedure**

Solutions in the population are generated during an initialization step (lines 2 to 4 in Algorithm 4) using a uniform distribution that generates chromosomes randomly inside of the search space given the integer vectors $\boldsymbol{ub}$ and $\boldsymbol{lb}$ that are composed by the upper and lower bounds values for the search space components.

Since the chromosomes are integer values, GA is expected to be used in single-objective COP optimization problems with an objective function f where the search space is a subset of $\mathbb{N}^d$. After the population generation, the solutions in $\boldsymbol{P}$ evolves through I iterations using three genetic operators: (a) **selection**, (b) **crossover**, and (c) **mutation**.

The selection operator chooses the fittest individuals of the population to transmit their chromosomes to the next generation as parents to their offspring [66]. The selection strategy used is a **binary tournament selection** where to every new offspring two pairs of different individuals

are randomly selected from the population $\boldsymbol{P}$ and the fittest individual of each select pairs is in turn selected as one of the parents.

After selection, GA applies a crossover operator on a pair of parents simulating a reproduction scheme by exchanging chromosomes between both parents. In this work, the crossover operator used in GA is the **single-point crossover operator** where, if a pair of parents pass a random check using a **crossover rate** p_c, a random index $r_d \in [1, d]$ called **crossover point** is used to separate and swap chromosomes from both parents to generate two offsprings that are new individuals with new solutions formed with chromosomes from both parents. Figure 4.5 illustrates a graphical representation of a single-point crossover operation functionality being applied on two parents solutions ($\boldsymbol{x}_{p_1}$, $\boldsymbol{x}_{p_2}$) to generate two offsprings ($\boldsymbol{x}_{o1}$, $\boldsymbol{x}_{o2}$) given a crossover point r_d. If the pair of parents does not pass the random check, their solutions are copied on to the next generation as two new offsprings.

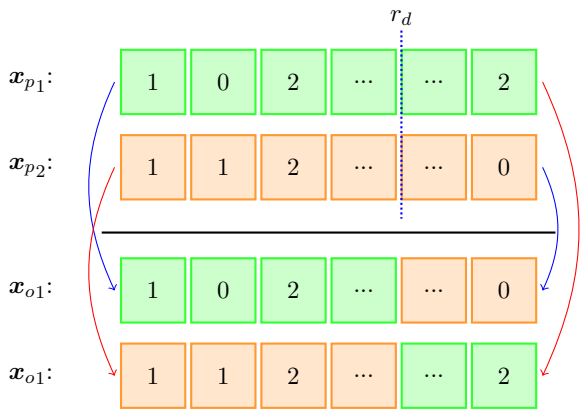


Figura 4.5 – Graphical representation of a one-point crossover operator.

At last, after the crossover operation, GA applies a **random gene flip mutation operator** that modifies the chromosomes of the offspring individuals stimulating the addition of diversity in the solutions present in the population and helping the algorithm to avoid local minima. For every jth chromosome in an offspring solution, the mutation operation performs a random check with the **mutation rate** p_m to substitute this chromosome's original value with a randomly uniform generated value inside of the range $[lb_j, ub_j]$.

After the application of genetic operators, GA unifies both the initial population $\boldsymbol{P}$ and the new population of offsprings $\boldsymbol{P}_{new}$ then sort their evaluation results in ascending order. The new unified population has its size reduced by removing their last individuals since it was ordered these represent the worst solutions until only the best n individuals are left. This new solution is then used as a parent population for the next GA iteration. This elitist process of leaving only the best solution allows the algorithm to converge with the best solution for the single-objective function f at hand.

4.3.1.2 Parameters Description for GA

GA has two **external static** parameters p_m and p_c that controls, respectively, the rate in which the solutions exchange components and the rate in which new solution components are added, or re-added, to the population $\boldsymbol{P}$. For a better GA performance, these parameters values should be configured using a **parameter tuning** strategy depending on the problem being optimized.

4.3.2 Differential Evolution (DE)

4.3.2.1 Meta-Heuristic Description

Differential Evolution (DE) [67] is an evolutionary bio-inspired meta-heuristic algorithm that uses two strategies to explore and exploit the solutions in a search space: (a) **mutation strategy**: Add population diversity by creating new solution components; (b) **crossover strategy**: Transfer information between new and old solutions to exploit known solution components. Even though these strategies are similar to the genetic operators in GA, the approach in which DE generate and keep new solutions is different from GA.

DE has a population $\boldsymbol{P}_c$ of n individuals and each ith individual is composed of a d-dimensional real vector that, during the kth iteration of the algorithm, is represented as $\boldsymbol{x}_i^k$. Vector $\boldsymbol{x}_i^k$ is a possible solution for a continuous SOOP with an objective function f with a search space being a subset of $\mathbb{R}^d$.

Since solutions $\boldsymbol{x}_i^{(k)}$ are continuous, to use DE to optimize a COP problem with a search space in $\mathbb{N}^d$, in this work, continuous solutions are rounded in an element-by-element basis before evaluating the objective function. This same approach is repeated to any bio-inspired algorithm that holds continuous solutions in their search agents.

During DE initialization, $\boldsymbol{P}_c$ has its individuals randomly generated inside of the continuous search-space using uniform distributions for each component of each individual given the real vectors $\boldsymbol{ub}$ and $\boldsymbol{lb}$ that is composed by the upper and lower bounds values for the search space.

After the initialization, the population evolves through multiple iterations using mutation and crossover strategies on each individual until a final iteration I is reached. In this work, DE uses a ***rand/1* mutant strategy** where an ith individual $\boldsymbol{x}_i^{(k)}$ generates a mutant vector $\boldsymbol{v}_i$ according to the following equation:

$$\boldsymbol{v}_i = \boldsymbol{x}_{r_1}^{(k)} + F(\boldsymbol{x}_{r_2}^{(k)} - \boldsymbol{x}_{r_3}^{(k)}), \tag{4.1}$$

where r_1, r_2, $r_3 \in [1, n]$ are different random indices, and F is an **external static** parameter called **scaling factor**.

Following the mutation, DE uses a **binary crossover strategy** that uses the mutant vector $\boldsymbol{v}_i$ to diversify the components of $\boldsymbol{x}_i^{(k)}$ by combining both vector as a trial vector $\boldsymbol{u}_i$ using Eq. 4.2 for every j element of the solution. Note that this process truncates the mutant vector components back to the search space boundaries.

$$u_{i,j} = \begin{cases} \min(ub_j, \max(lb_j, v_{i,j})) & \text{if } r \leq CR \text{ or } j = r_j \\ x_{i,j}^{(k)} & \text{otherwise} \end{cases}, \tag{4.2}$$

where r is a random real number in $[0, 1)$, r_j is a random index in $[1, d]$ kept for all j, and CR is a **external static** parameter in $[0, 1)$ called **crossover rate**. The parameter CR controls the rate in which new solution components are added to the population $\boldsymbol{P}_c$. A DE-based algorithm that uses both the previously described mutation and crossover strategies is called *DE/rand/1/bin*.

The trial vector $\boldsymbol{u}_i$ is evaluated using the objective function f, and its result is then compared with the evaluation previously obtained by the individual $\boldsymbol{x}_i^{(k)}$. If $f(\boldsymbol{u}_i) < f(\boldsymbol{x}_i^{(k)})$, then $\boldsymbol{x}_i^{(k)}$ is replaced by $\boldsymbol{u}_i$. This replacement process in DE-based algorithms is called **selection strategy**, and it is represented by the following equation:

$$\boldsymbol{x}_i^{(k+1)} = \begin{cases} \boldsymbol{u}_i & \text{if } f(\boldsymbol{u}_i) < f(\boldsymbol{x}_i^{(k)}) \\ \boldsymbol{x}_i^{(k)} & \text{otherwise} \end{cases}. \tag{4.3}$$

The pseudo-code for *DE/rand/1/bin* algorithm is present in Algorithm 5.

4.3.2.2 Parameters Description for DE

DE has two **external static** parameters F and CR, and their values control the exploration and exploitative behavior of the meta-heuristic. These parameters control, respectively: (a) F influences the importance of the difference between individuals when generating mutant vectors affecting the algorithm exploration; (b) CR influences the rate in which components of generated mutant vectors are added in possible new solutions; For these reasons, the use of a parameter tuning strategy to adjust F and CR may improve DE performance.

Algorithm 5 Differential Evolution (DE/rand/1/bin)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Scaling factor (F)
Crossover rate (CR)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

1: **procedure** DE($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}, F, CR$)
2: *Initialize* $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
3: *Evaluate* $f(\boldsymbol{x}_i^{(0)})$
4: *Obtain best Individual* $\boldsymbol{x}_{best}$
5: **for** $k = 1$ to $I - 1$ **do**
6: **for** $i = 1$ to n **do**
7: (*rand*/1) *Mutation* ▷ Eq. 4.1
8: *Binary Crossover* ▷ Eq. 4.2
9: *Selection* ▷ Eq. 4.3
10: **end for**
11: Update $\boldsymbol{x}_{best}$
12: **end for**
13: **return** $\boldsymbol{x}_{best}$
14: **end procedure**

4.3.3 Particle Swarm Optimization (PSO)

4.3.3.1 Meta-Heuristic Description

Particle Swarm Optimization (PSO) [61] [68] is a swarm intelligence meta-heuristic inspired by the social behavior of fish schools and flocks of birds. Search agents in PSO are called particles. To simulate the swarm search for food, every particle in PSO has a memory about the best food source it has found so far and also exchanges information with other particles to know where the swarm has found the best food source. This pattern of information exchange exploits the swarm knowledge to increase the algorithm convergence by finding better food sources when exploring. The pseudo-code for PSO is present in Algorithm 6.

Algorithm 6 Particle Swarm Optimization (PSO)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of particles (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Initial weight ($w_{initial}$)
Final weight (w_{final})
Maximum particle speed (v_{max})
Social factor (c_1)
Cognitive factor (c_2)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

1: **procedure** PSO
2: *Initialize* $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
3: *Initialize* $\boldsymbol{v}_i^{(0)}$ $(i = 1, ..., n)$
4: *Evaluate* $f(\boldsymbol{x}_i^{(0)})$
5: $w^{(0)} = w_{initial}$
6: $\boldsymbol{x}_{i,pbest} = \boldsymbol{x}_i^{(0)}$
7: *Obtain best solution* $\boldsymbol{x}_{best}$
8: **for** $k = 1$ to $I - 1$ **do**
9: $w = w_{final} + \frac{k(w_{final} - w_{initial})}{I}$
10: **for** $i = 1$ to n **do**
11: *Calculate* $\boldsymbol{v}_i^{(k)}$ ▷ Eq. 4.4
12: *Calculate* $\boldsymbol{x}_i^{(k)}$ ▷ Eq. 4.5
13: *Evaluate* $f(\boldsymbol{x}_i^{(0)})$
14: *Update* $\boldsymbol{x}_{pbest}$
15: **end for**
16: Update $\boldsymbol{x}_{best}$
17: **end for**
18: **return** $\boldsymbol{x}_{best}$
19: **end procedure**

PSO has a swarm of $\boldsymbol{P}_c$ particles and each ith particle has a d-dimensional real vector $\boldsymbol{x}_i^{(k)}$ that represents its position during iteration k as well as being a solution for a single-objective function f. Besides its position, an ith particle also holds a copy for the position of its personal best evaluation $\boldsymbol{x}_{pbest_i}$, a reference to the global best position found by the swarm $\boldsymbol{x}_{best}$, and, to model the swarm movement, a speed vector $\boldsymbol{v}_i^{(k)}$.

During the initialization, particles positions are initialized using a random uniform distribution in the interval $[lb_j, ub_j]$ for each jth component for each particle with $\boldsymbol{ub}$ and $\boldsymbol{lb}$ being vectors that are composed by the upper and lower bounds values for the search-space. Speed vectors also have their components initialized in a similar fashion as the positions, however, the random uniform distribution used is inside $[-v_{max}, v_{max}]$ where v_{max} is an **external static** parameter called **maximum speed**. In other words, each jth component of the speed vector for the ith

particle is constrained $v_{i,j} \in [-v_{max}, v_{max}]$.

Equation 4.4 calculates the speed components for each ith particle moving inside of the search space.

$$v_{i,j}^{(k+1)} = \min(v_{max}, \max(-v_{max}, w\, v_{i,j}^{(k)} + c_1 r_1 (x_{best\,j} - x_{i,j}^{(k)}) + c_2 r_2 (x_{pbest_j} - x_{i,j}^{(k)}))), \tag{4.4}$$

where r_1 and r_2 are random real values generated using uniform distribution in $[0, 1]$, and w is an **internal** parameter called **inertial weight**. **External static** parameters c_1 and c_2 are called, respectively, **social** and **cognitive** factors.

After calculating the speed for the particles, their position inside of the search space are updated using the following equation:

$$x_{i,j}^{(k+1)} = \min(ub_j, \max(lb_j, x_{i,j}^{(k)} + v_{i,j}^{(k+1)})). \tag{4.5}$$

For every iteration, the **inertial weight** $w^{(k)}$ is adjusted using a **deterministic parameter control strategy** that decreases linearly the values for weights throughout iterations between two **external static parameters** $w_{initial}$ and w_{final} called **initial inertial weight** and **final inertial weight**.

4.3.3.2 Parameters Description for PSO

PSO has five **external static** parameters $w_{initial}$, w_{final}, v_{max}, c_1 and c_2. All of these parameter influence the performance of PSO as following: (a) $w_{initial}$ influences particles movement during the first iterations affecting PSO exploration. It is suggested that $w_{initial} \in (0, 1)$ to improve convergence; (b) w_{final} influences particles movement during last iterations affecting PSO exploitation of information gathered during execution. PSO expects $w_{final} < w_{initial}$ with suggested value $w_{final} \in (0, 1)$ to improve convergence; (c) c_1 and c_2 influences the PSO behaviour towards global or local search, respectively, by affecting the tendency of particles movement in direction of the global optimal or their local optimal. When $c_2 < c_1$, the PSO keeps an aggressive approach prone to fast convergence but susceptible to be a premature one. In the other case, when $c_2 > c_1$, the algorithm acquire a more conservative instance with the particles focusing more on their local than the global optima resulting in converge that takes more iterations. As analyzed in [69], for PSO convergence $c_1 + c_2 \in [0, 4]$; (d) v_{max} influences PSO exploration by controlling the maximum absolute value that a particle is allowed to move in a single dimension in the search space. If particles are allowed to move in long steps, they may escape local minima but are prone to "jump over" possible good solutions. A suggested value for $v_{max} \in [0, 1]||\boldsymbol{ub} - \boldsymbol{lb}||$.

PSO has an **internal** parameter adjusted by a **deterministic parameter control strategy**: the **inertial weight** w. The parameter w influences PSO to alternate between a more exploratory stage with particles moving by large steps in the search space to a more exploitative stage with the particles moving with small steps. This change in behavior is advantageous because, during the last iterations, the algorithm is expected to be closer to the global optima, and only a refinement of the solution is necessary.

4.3.4 Salps Swarm Algorithm (SSA)

4.3.4.1 Meta-Heuristic Description

Salps Swarm Algorithm (SSA) [70] is a swarm intelligence bio-inspired meta-heuristic inspired by the social behavior of small animals called salps. Salps are marine animals similar to jellyfish that live in deep-sea waters and feed on plankton and moves through their environment as a chain of group members following a leader and collaborate foraging for food. The pseudo-code in Algorithm 7 represents SSA.

Algorithm 7 Salp Swarm Algorithm (SSA)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of salps (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure SSA($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize salps $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Sort $\boldsymbol{P}_c$
  Obtain $\boldsymbol{x}_{food}$
  for $k = 1$ to $I - 1$ **do**
    Update c_1 ▷ Eq. 4.7
    for $i = 1$ to n **do**
      if $i = 1$ **then**
        Update leader $\boldsymbol{x}_1^{(k)}$ ▷ Eq. 4.6
      else
        Update follower $\boldsymbol{x}_i^{(k)}$ ▷ Eq. 4.8
      end if
      Evaluate $f(\boldsymbol{x}_i^{(k)})$
    end for
    Update $\boldsymbol{x}_{food}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

SSA model the salps as a swarm $\boldsymbol{P}_c$ of n salps and each ith salp has its position defined as a d-dimensional real vector $\boldsymbol{x}_i^{(k)}$ that is also a possible solution for a objective function f.

The swarm members can assume two types: (a) leader or (b) follower. The definition for a type of a salp takes place during the initialization stage. In this stage, the salps have their positions randomly generated inside of the search space. These positions are then evaluated using a single-objective function f. After the evaluation of every position, SSA sorts the population in ascending order of objective evaluation and using this ordered swarm SSA gives to each salp its rank based on its index in the sorted swarm. The first salp with index $i = 1$ is the leader. Meanwhile, all other salps are followers.

The leader is the salp that moves around the food position, namely, the best solution, and leads the other salps to do the same. The leader salp has its position updated using the following equation:

$$x_{1,j}^{(k+1)} = \begin{cases} x_{food,j} + c_1(r_1(ub_j - lb_j) + lb_j) & \text{, if } r_2 \geq 0.5 \\ x_{food,j} - c_1(r_1(ub_j - lb_j) + lb_j) & \text{, if } r_2 < 0.5 \end{cases}, \tag{4.6}$$

where $x_{1,j}^{(k)}$ is the leader position at dimension j and kth iteration, $\boldsymbol{x}_{food}^{(k)}$ is the food position

representing the best solution found so far. Vectors $\boldsymbol{ub}$ and $\boldsymbol{lb}$ holds the upper and lower bounds values for the search space. Random real numbers r_1 and r_2 are in $[0, 1)$ and defines which direction the leader should move away or towards the food. An **internal** parameterc_1 has its value adjusted every iteration by a **deterministic parameter control strategy**.

Equation 4.7 represents the **deterministic control parameter strategy** calculation of c_1 during iterations.

$$c_1 = 2e^{-\left(\frac{4k}{I}\right)}, \tag{4.7}$$

where k is the current iteration and I is SSA maximum number of iterations.

The followers salps update their positions using the following equation:

$$x_{i,j}^{(k+1)} = 0.5(x_{i,j}^{(k)} + x_{i-1,j}^{(k)}), \tag{4.8}$$

where $x_{i,j}^{(k)}$ with $i > 1$ is jth component for the ith salp position. This update method forces the salps to follow the salp with one rank above it. It has as a consequence that the swarm moves in a chain-like pattern that sweeps across the search space seeking for the food position (best solution).

4.3.4.2 Parameters Description for SSA

SSA has no **external static** parameters and its single **internal** parameter c_1 is adjusted by a **deterministic control parameter strategy**. The parameter c_1 influences SSA by alternating its behavior between exploration and exploitation by defining whether the leader salp moves by large or small steps inside of the search space.

4.3.5 Gray Wolf Optimization (GWO)

4.3.5.1 Meta-Heuristic Description

Gray Wolf Optimization (GWO) [71] is a swarm intelligence meta-heuristic that uses as a metaphor for their heuristic search operators the social behavior observed in wild packs of wolves when hunting. Their behavior presents a hierarchical assignment of different tasks between pack members. Both in the wild as in GWO, wolf packs are divided into four hierarchical groups: a) alpha (α): They are the leading wolves, and the others follow them when hunting; b) beta (β): The second in command and give support to the alpha when hunting. Alpha wolves come from this group when an alpha becomes old or dies; c) delta (δ): The third in command. In the wild, this group has old alphas and betas, sentinels, scouts and caretakers of the pack pups; d) omega (ω): All other wolves in the pack are part of this group, and they submit to wolves in the former groups. In GWO, the omega wolves are responsible for exploring the hunting grounds following the wolves from the upper hierarchical groups α, β and δ. Pseudo-code for GWO is present in Algorithm 8.

The wolf pack $\boldsymbol{P}_c$ is formed by n omega wolves with each ith wolf position at iteration k expressed as $\boldsymbol{x}_i^{(k)}$ in an d-dimensional hunting ground while following the alpha, beta and delta wolves positions expressed as $\boldsymbol{x}_\alpha$, $\boldsymbol{x}_\beta$ and $\boldsymbol{x}_\delta$, respectively, and representing the three best solutions.

Algorithm 8 Gray Wolf Optimization (GWO)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of wolves (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure GWO($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize wolves $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{x}_\alpha$ *Best Solution*
  Update $\boldsymbol{x}_\beta$ *2nd best Solution*
  Update $\boldsymbol{x}_\delta$ *3rd best solution*
  for $k = 1$ to $I - 1$ **do**
    $a = 2 - \frac{2\,k}{I}$
    for $i = 1$ to n **do**
      Update wolf $\boldsymbol{x}_i^{(k)}$ ▷ Eq. 4.13
      Evaluate $f(\boldsymbol{x}_i^{(k)})$
      Update $\boldsymbol{x}_\alpha$ *Best Solution*
      Update $\boldsymbol{x}_\beta$ *2nd best Solution*
      Update $\boldsymbol{x}_\delta$ *3rd best solution*
    end for
  end for
  $\boldsymbol{x}_{best} = \boldsymbol{x}_\alpha$
  return $\boldsymbol{x}_{best}$
end procedure

In GWO initialization, ω wolves positions are generated randomly inside of the search space using a uniform distribution in $[lb_j, ub_j]$ for each jth dimension for each ith omega wolf. Then these positions are evaluated using the single-objective f, and the three best solutions are defined as positions for the wolf leaders, respectively, alpha, beta, and delta.

During every iteration, omega wolves follows each of their superiors using Eq. 4.9, Eq. 4.10 and Eq. 4.13, respectively, for α, β and δ wolves.

$$\begin{cases} \Delta x_{\alpha,j} = |c_1 \cdot x_{\alpha,j} - x_{i,j}^{(k)}| \\ x_{f_\alpha,j} = x_{\alpha,j} - A_1 \cdot \Delta x_{\alpha,j} \end{cases} . \tag{4.9}$$

$$\begin{cases} \Delta x_{\beta,j} = |c_2 \cdot x_{\beta,j} - x_{i,j}^{(k)}| \\ x_{f_\beta,j} = x_{\beta,j} - A_2 \cdot \Delta x_{\beta,j} \end{cases} . \tag{4.10}$$

$$\begin{cases} \Delta x_{\delta,j} = |c_3 \cdot x_{\delta,j} - x_{i,j}^{(k)}| \\ x_{f_\delta,j} = x_{\delta,j} - A_3 \cdot \Delta x_{\delta,j} \end{cases} , \tag{4.11}$$

in these equations, $\Delta x_{\alpha,j}$, $\Delta x_{\beta,j}$, and $\Delta x_{\delta,j}$ are the differences between positions of the leader wolves and the omega wolf $\boldsymbol{x}_i^{(k)}$ in a component j. Vectors $\boldsymbol{x}_(f_\alpha, j)$, $\boldsymbol{x}_{f_\beta,j}$ and $\boldsymbol{x}_{f_\delta,j}$ are the positions for the ith omega wolf in dimension j, if it would only follow a single leader α, β or δ, respectively. Parameters c_1, c_2, and c_3 are randomly generated real values in $[0, 2]$ and A_1, A_2, and A_3 are **internal** parameters calculated using Eq. 4.12.

$$A_m = 2ar_m - a \tag{4.12}$$

where $m \in \{1, 2, 3\}$ and r_m are different randomly generated real values inside $[0, 1]$. An **internal** parameter a adjusted by a **deterministic parameter control strategy** that decreases a linearly throughout the iterations between 2 and 0.

At last, the position of omega wolves for each jth dimension is updated as the arithmetic average between $\boldsymbol{x}_{f_\alpha}$, $\boldsymbol{x}_{f_\beta}$ and $\boldsymbol{x}_{f_\delta}$. It is expressed in the following equation:

$$x_{i,j}^{(k+1)} = \frac{x_{f_\alpha,j} + x_{f_\beta,j} + x_{f_\delta,j}}{3}.. \tag{4.13}$$

4.3.5.2 Parameters Description for GWO

GWO does not have any **external static** parameters. However, its exploratory and exploitative behaviors relies on its **internal** parameters: (a) a, (b) A_1, (c) A_2, and (d) A_3. The value for a is controlled every iteration by a **deterministic parameter control strategy** decreasing linearly between 2 and 0. Parameter a controls the algorithm by affecting the parameters A_1, A_2, A_3. Since a decreases with iterations, the algorithm tends to an exploratory behavior during the first iterations and an exploitative one during the last iterations.

The variables A_m with $m \in \{1, 2, 3\}$ varies every iteration randomly in the interval $[-a, a]$ and they influence whether the wolf should depart from a leader $|A_m| > 1$ or move towards a leader $|A_m| < 1$ for each jth dimension.

4.3.6 Elephant Herd Optimization (EHO)

4.3.6.1 Meta-Heuristic Description

EHO algorithm [72] is a swarm intelligence based meta-heuristic that uses as inspiration the behavior of elephant herds in nature. Elephants are social animals, and a group of them is composed of multiple clans that live together following a matriarch leader. In these clans, males when reaching reproductive age, they separate from these clans to live alone. EHO algorithm is present in pseudo-code form in Algorithm 9.

EHO has a population $\boldsymbol{P}_c$ of n elephants with positions $\boldsymbol{x}_i^{(k)}$ that are d-dimensional real vectors as well as solutions for a single-objective function f. $\boldsymbol{P}_c$ is subdivided into N_{clans} clans $\boldsymbol{P}_n$ with $n = \{1, ..., N_{clans}\}$ with same size $\left\lfloor \frac{n}{N_{clans}} \right\rfloor$. N_{clans} is a **external static** parameter in EHO.

In the initilialization, EHO generates randomly n solutions $\boldsymbol{x}_i^{(0)}$ for f given that each component is bounded by the elements in vectors $\boldsymbol{lb}$ lower-bounds and $\boldsymbol{ub}$ upper-boundaries, and use this solutions as positions for each elephant. Then, EHO subdivides these solutions randomly in its N_{clans} and evaluate each elephant position. For each generated clan, EHO partitions the elephants into two types depending on their initial evaluation: (a) matriarch that represents the best solution, and (b) followers elephants that represent other solutions.

Elephants positions are updated during the iterations by a representation of the social aspect of the elephants in $\boldsymbol{P}_c$ that is expressed by applying two operations over each clan: (a) clan updating and (b) separating operators.

The clan updating operator updates elephants in each clan. Each follower elephant in a clan

Algorithm 9 Elephant Herd Optimization (EHO)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of elephants (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Number of clans (N_{clans})
Number of kept elephants (N_{kept})
Scaling factor (α)
Center scaling factor (β)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure EHO
  Initialize elephants $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  Separate elephants in N_{clans} *clans*
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Define matriarchs for each clan $\boldsymbol{x}_{c,mat}$
  Define worst for each clan $\boldsymbol{x}_{c,worst}$
  for $k = 1$ to $I - 1$ **do**
    Save N_{kept} *best elephants in each clan*
    Update followers in clans ▷ Eq. 4.14
    Update matriarchs in clans ▷ Eq. 4.15
    Separate operators in clans ▷ Eq. 4.17
    Evaluate $f(\boldsymbol{x}_i^{(k)})$
    Define matriarchs for each clan $\boldsymbol{x}_{c,mat}$
    Define worst for each clan $\boldsymbol{x}_{c,worst}$
    Replace N_{kept} *worst in clans*
    Update $\boldsymbol{x}_{best}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

has its position updated using the following equation:

$$\boldsymbol{x}_{c,l}^{(k+1)} = \boldsymbol{x}_{c,l}^{(k)} + r\,\alpha(\boldsymbol{x}_{c,mat} - \boldsymbol{x}_{c,l}), \tag{4.14}$$

where $\boldsymbol{x}_{c,l}^{(k)}$ is a position for the lth elephant that is part of the cth clan at iteration k, and $\boldsymbol{x}_{c,mat}$ is the position for the matriarch in cth clan. The real number r is randomly generated in $[0, 1]$, and α is an **external static** parameter.

The matriarch elephant position of clan c is updated using the following equation:

$$\boldsymbol{x}_{c,best} = \beta\,\boldsymbol{x}_{c,center}, \tag{4.15}$$

where β is an **external static** parameter and $\boldsymbol{x}_{c,center}$ is the center position of the cth clan and it is calculated using the following equation for each jth component of $\boldsymbol{x}_{c,center}$:

$$x_{c,center,j} = \frac{1}{|\boldsymbol{P}_n|} \sum_{j=1}^{|\boldsymbol{P}_n|} x_{c,l,j}, \tag{4.16}$$

where $x_{c,l,j}$ is the jth component of the lth elephant in clan c.

The separating operator updates the worst elephant in each clan by creating a completely new random solution using the following equation:

$$x_{c,worst,j} = r(ub_j - lb_j) + lb_j, \tag{4.17}$$

where $r$is a random real number in $[0, 1]$, $x_{c,worst,j}$ is the jth element for the position of the elephant with the worst evaluation in clan c, ub_j is the search space upper-boundary for jth dimension, and lb_j is lower-boundary for search space jth dimension.

After the application of clan update and separating operators, the elephants are evaluated, and then, an elitist process keeps the N_{kept} best solutions in each clan from previous iterations to replace the elephants with the worst evaluations in the current solution after the evaluation. The integer N_{kept} is an **external static** parameter.

4.3.6.2 Parameters Description for EHO

EHO has three **external static** parameters: (a) α, (b) β, (c) N_{clans}, and (d) N_{kept}. These parameters need to be configured before the algorithm execution through a parameter tuning process for better performance for the algorithm.

The value for $\alpha \in [0, 1)$ influences the exploration of EHO by working as a scaling factor for the difference between a matriarch position and a follower elephant in each clan. If α is large, then the chances for elephants to quickly move towards their matriarch increases and EHO clans suffers premature convergence. Otherwise, if α is too small, the elephants move slower towards their matriarch and EHO has lower convergence speed.

The parameter $\beta \in [0, 1)$ influences the exploration by controlling the movement of the matriarch in each clan towards the center of their clan. For this reason, β controls the speed in which all solutions drift towards the center of each clan. The parameter N_{clans} controls how many clans in the population of elephants, and it influences EHO exploration by defining N_{clans} local optimal in which elephants in each clan will move towards during the initial iterations. The parameter N_{kept} adjust the (re-)entrance of past good solutions in the clan, and it helps the exploitation of the algorithm.

4.3.7 Dragonfly Algorithm (DA)

4.3.7.1 Meta-Heuristic Description

DA [73] is a swarm intelligence meta-heuristic that uses as a metaphor for its procedures the collective behavior of adult dragonflies in nature. Dragonflies are predator insects and have a range of smaller animals in their diet. When flying in a swarm, dragonflies behaves in three types of movements: (a) static movement: when hunting, dragonflies in small groups fly back and forth searching for prey; (b) migration: when hunting, the whole swarm flies in the same direction to migrate from one hunting ground to another; (c) escape: dragonflies are also predated upon by birds and other animals. To survive, the swarm combines an escape movement from predators flying away from identified threats while converging to the prey position. DA is shown in pseudo-code form in Algorithm 10.

DA optimization has a swarm $\boldsymbol{P}_c$ of n dragonflies. Each ith dragonfly holds a position $\boldsymbol{x}_i^{(k)}$ in a d-dimensional continuous search space and a possible solution for the objective function f. The ith dragonfly also contain their speed information represented as $\boldsymbol{v}_i^{(k)}$ at iteration k.

During the initialization, DA generates n random solutions for single-objective function f inside of the search space defined by the upper and lower boundaries present as elements in the

Algorithm 10 Dragonfly Algorithm (DA)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of dragonflies (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure DA($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize wolves $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  Initialize $\boldsymbol{v}_i^{(0)}$ $(i = 1, ..., N)$
  for $k = 1$ to I **do**
    Evaluate $f(\boldsymbol{x}_i^{(k)})$
    Update $\boldsymbol{x}_{best}$
    Update $\boldsymbol{x}_{worst}$
    Update $w, w_{sep}, w_{ali}, w_{coh}, w_{att}, w_{rep}$
    Update n_r ▷ Eq. 4.19
    for $i = 1$ to n **do**
      Obtain $\boldsymbol{X}_{i,nbrs}$
      if $\boldsymbol{X}_{i,nbrs} \neq \emptyset$ **then**
        Calculate $\boldsymbol{x}_{i,sep}$ ▷ Eq. 4.18
        Calculate $\boldsymbol{x}_{i,ali}$ ▷ Eq. 4.20
        Calculate $\boldsymbol{x}_{i,coh}$ ▷ Eq. 4.21
        Calculate $\boldsymbol{x}_{i,att}$ ▷ Eq. 4.22
        Calculate $\boldsymbol{x}_{i,rep}$ ▷ Eq. 4.23
        Update $\boldsymbol{v}_i^{(k+1)}$ ▷ Eq. 4.24
        $\boldsymbol{x}_i^{(k+1)} = \boldsymbol{x}_i^{(k)} + \boldsymbol{v}_i^{(k+1)}$
      else
        Update $\boldsymbol{x}_i^{(k+1)}$ ▷ Eq. 4.28
      end if
    end for
  end for
  return $\boldsymbol{x}_{best}$
end procedure

d-dimensional vectors $\boldsymbol{lb}$ and $\boldsymbol{ub}$.

The position of a dragonfly is updated analogous to particles in PSO (Section 4.3.3) using Eq. 4.5. DA emulates the different movements displayed by a dragonfly in nature by constructing a speed with terms that depend on the following types of movements: (a) Separation (static), (b) Alignment (migration), (c) Cohesion, (d) Food attraction, and (e) Enemy repulsion.

The separation term $\boldsymbol{v}_{i,sep}$ for a dragonfly i is calculated using the following equation:

$$\boldsymbol{v}_{i,sep} = \sum_{\boldsymbol{x}_{l,nbrs} \in \boldsymbol{X}_{i,nbrs}} (\boldsymbol{x}_{l,nbrs} - \boldsymbol{x}_i^{(k)}), \tag{4.18}$$

where $\boldsymbol{x}_i^{(k)}$ is the ith dragonfly position at iteration k, and $\boldsymbol{x}_{l,nbrs}$ is the lth neighbor dragonfly in the set $\boldsymbol{X}_{i,nbrs}$ of neighbors for the ith dragonfly.

The neighborhood for the ith dragonfly $\boldsymbol{X}_{i,nbrs}$ is formed for the dragonflies with positions inside of a hypersphere in the search space with the center in $\boldsymbol{x}_i^{(k)}$ and a radius of n_r. Parameter n_r is the **neighborhood radius**, and it is an **internal** parameter that has its value increased throughout the algorithm execution by a **deterministic parameter control strategy** as described by equations:

$$n_r = ||\boldsymbol{ub} - \boldsymbol{lb}|| \left(0.1 + \frac{k}{I}\right), \tag{4.19}$$

where $\boldsymbol{ub}$ and $\boldsymbol{lb}$ are the vectors holding the upper and lower boundaries values for the search-space.

The alignment term $\boldsymbol{v}_{i,ali}$ for the speed of ith dragonfly is calculated by the following equation:

$$\boldsymbol{v}_{i,ali} = \frac{1}{|\boldsymbol{X}_{i,nbrs}|} \sum_{\boldsymbol{v}_{l,nbrs} \in \boldsymbol{X}_{i,nbrs}} \boldsymbol{v}_{l,nbrs}, \tag{4.20}$$

where $\boldsymbol{v}_{i,nbrs}$ is the speed for the lth dragonfly with position neighboring that of the ith dragonfly by being part of $\boldsymbol{X}_{i,nbrs}$.

The cohesion term $\boldsymbol{v}_{i,coh}$ for the ith dragonfly uses the following equation:

$$\boldsymbol{v}_{i,coh} = \frac{1}{|\boldsymbol{X}_{i,nbrs}|} \sum_{\boldsymbol{x}_{l,nbrs} \in \boldsymbol{X}_{i,nbrs}} \boldsymbol{x}_{l,nbrs} - \boldsymbol{x}_i^{(k)}, \tag{4.21}$$

where $\boldsymbol{x}_{l,nbrs}$ is the lth neighbor dragonfly in the set $\boldsymbol{X}_{i,nbrs}$ in the set of neighbors $\boldsymbol{X}_{i,nbrs}$.

The attraction towards food term $\boldsymbol{v}_{i,att}$ for a dragonfly i is calculated using the following equation:

$$\boldsymbol{v}_{i,att} = \boldsymbol{x}_{best} - \boldsymbol{x}_i^{(k)}, \tag{4.22}$$

where $\boldsymbol{x}_{best}$ is the food source position and it is represented by the best solution found so far.

The repulsion enemy term $\boldsymbol{v}_{i,rep}$ for a dragonfly i is calculated using equation:

$$\boldsymbol{v}_{i,rep} = \boldsymbol{x}_{worst} + \boldsymbol{x}_i^{(k)}, \tag{4.23}$$

where $\boldsymbol{x}_{worst}$ is the predator position and it is represented by the worst solution found so far by DA.

At last, the speed of the ith dragonfly $\boldsymbol{v}_i^{(k)}$ in the swarm is updated using equation:

$$\boldsymbol{v}_i^{(k+1)} = w_{sep}\boldsymbol{v}_{i,sep} + w_{ali}\boldsymbol{v}_{i,ali} + w_{coh}\boldsymbol{v}_{i,coh} + w_{att}\boldsymbol{v}_{i,att} + w_{rep}\boldsymbol{v}_{i,rep} + w\boldsymbol{v}_i^{(k)}, \tag{4.24}$$

where w, w_{sep}, w_{ali}, w_{coh}, w_{att} and w_{rep} are **internal** parameters. The values for w_{sep}, w_{ali}, w_{coh}, and w_{att} are updated using Eq. 4.25. Meanwhile, w_{rep} is adjusted every iteration using a **deterministic parameter control strategy** defined by Eq. 4.26 and w is also controlled by another **deterministic parameter control strategy** represented in Eq. 4.27.

$$w_{type} = r_{type}\, w_{rep}, \tag{4.25}$$

where $type = \{sep, ali, coh, att\}$, and r_{type} is a random real number in $[0, 1]$.

$$w_{rep} = 0.1\left(1.0 - \frac{(k-1)}{I}\right), \tag{4.26}$$

$$w = w_{initial} - (w_{initial} - w_{final})\frac{(k-1)}{I}. \tag{4.27}$$

In Equations 4.26 and 4.27, k is the current DA iteration, and I is the maximum number of iterations. The real numbers $w_{initial}$ and w_{final} are **internal static** parameters.

Whenever a dragonfly i has no neighbors around its position, i.e., $\boldsymbol{X}_{i,nbrs} = \emptyset$, DA enters in an exploitation phase for the ith dragonfly by using a random Lévy fly method to modify its position instead of using its speed vector $\boldsymbol{v}_i^{(k)}$ This update process follows Eq. 4.28, and the Lévy

fly method uses Eq. 4.29 with $\beta = 1.5$, $\sigma = 0.6966$, where r_1 and r_2 being random real numbers in $[0, 1)$.

$$x_{i,j}^{(k+1)} = x_{i,j}^{(k)} + r_{Levy}. \tag{4.28}$$

$$r_{Levy} = 0.01 \frac{r_1 \sigma}{r_2^{\left(\frac{1}{\beta}\right)}}. \tag{4.29}$$

4.3.7.2 Parameters Description for DA

DA is a meta-heuristic that does not hold any **static external** parameter to be adjusted using a **parameter tuning strategy**. However, DA exploration and exploitation depends upon seven **internal** parameters: (a) n_r is the neighborhood radius for dragonflies in the swarm, (b) w_{sep} is the importance factor for the separation term, (c) w_{ali} is the importance factor for the alignment term, (d) w_{coh} is the importance factor for the cohesion term, (e) w_{att} is the importance factor for the attraction term, (f) w_{rep} is the importance factor for the repulsion term, (g) $w_{initial}$ initial inertia weight, (h) w_{final} final inertia weight, and (i) w is the **inertia weight**.

Parameter n_r is adjusted by a **deterministic parameter control** that increases its value over iterations. As n_r increases, DA alternates between a local search stage with multiple small neighborhoods of dragonflies influencing each other to global search stage with a single large neighborhood with a radius as big as the search space.

Parameters w_{sep}, w_{ali}, w_{coh}, and w_{att} depends on w_{rep} and has its values changing randomly. While, w_{rep} is adjusted by a **deterministic parameter control** that decreases its value over iterations. Parameter w_{sep} influences the speed in which dragonflies are repelled from the worst solution found so far, and by doing so, it affects DA convergence. Also, w_{sep} influences other internal parameters.

The values of $w_{initial}$ and w_{final} are **static internal** parameters, and, similar to its counterparts in PSO (Section 4.3.3), they control how large is the steps of dragonflies movement during the initial and final iterations. The values used are $w_{initial} = 0.9$ and $w_{final} = 0.4$. A **deterministic parameter control** adjusts parameter w, and it affects the change between exploratory to exploitative behavior for DA.

4.3.8 Moth-Flame Optimization (MFO)

4.3.8.1 Meta-Heuristic Description

MFO [74] is a swarm intelligence meta-heuristic inspired by the localization method used by moths in nature that brings them towards light sources during night time making spiral movements. Often these erratic pattern movement makes the moths converge to the position of the light source, and MFO optimizes an objective function by simulating this convergence aspect of moths movement. MFO is presented in pseudo-code in algorithm 11.

Algorithm 11 Moth-Flame Optimization (MFO)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of moths (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Parameter (b)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure MFO($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$, b)
  Initialize moths $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  for $k = 1$ to I **do**
    Evaluate $f(\boldsymbol{x}_i^{(k)})$
    Sort $\boldsymbol{P}_c$
    Calculate N_{flames} ▷ Eq. 4.31
    Create flames based on best N_{flames} *moths*
    for $i = 1$ to n **do**
      Map moth i *to closest* j *flame*
      Update $\boldsymbol{x}_i^{(k+1)}$ ▷ Eq. 4.30
    end for
    Update $\boldsymbol{x}_{best}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

MFO has a swarm $\boldsymbol{P}_c$ with n moths with positions represented as d-dimensional real vectors $\boldsymbol{x}_i$ with $i = \{1, ..., n\}$. Apart from the moths positions, MFO also has N_{flame} flames (light sources) with positions $\boldsymbol{x}_{f_l}$ with $l = \{1, ..., N_{flames}\}$ that represent the best found solutions for the objective function f. The integer number N_{flame} is a **internal** parameter controlled by a **deterministic parameter control**.

In MFO initialization, for each moth position is assigned a randomly generated solution $\boldsymbol{x}_i^{(k)}$ inside of the search space with boundaries $\boldsymbol{ub}$ and lb, where $\boldsymbol{ub}$ has in its elements the upper boundaries, and $\boldsymbol{lb}$ is composed of the lower ones. Also, during the initialization, flames positions are assigned as the best solutions for each moth position.

During MFO iterations,each ith moth in the swarm has its position attracted by its visible flame emulated in the following equation:

$$x_{i,j}^{(k+1)} = |\boldsymbol{x}_{f_{l,j}} - x_{i,j}^{(k)}| e^{br} cos(2\pi t) + x_{f_{l,j}}, \tag{4.30}$$

Vector $x_{i,j}$ is the position of the ith moth at jth dimension, $\boldsymbol{x}_{f_j}$ is the position of the lth flame visible by moth i, b is an **external static** parameter, and r is a random value in $[-1, 1]$.

The number of flames N_{flames} linearly over iterations, and as it happens, the mapping of moths to flames stops to be one-to-one and starts to be many-to-one. The process to pair the ith moth to the lth flame is to choose the flame position closest to moth i position. The **adaptive control parameter strategy** that updates N_{flams} is represented as follows:

$$N_{flames} = \left\lfloor n - k\frac{n-1}{I} + 0.5 \right\rfloor, \tag{4.31}$$

where k is the current iteration and I is the maximum number of iterations. During every iteration, the flames are updated with the best N_{flames} found so far by MFO.

4.3.8.2 Parameters Description for MFO

MFO has one **external static** parameter: b that controls the shape and size of the spiral movements of moths in the search space. For an improved MFO performance, it is necessary to tune b before MFO execution.

MFO also contains an **internal** parameters: N_{flames} that controls the number of flames and it is adjusted by a **deterministic parameter control** that reduces it linearly during the iterations. A consequence of the reduction of flames is that MFO alternates between a local search for multiple local minima to global search behavior.

4.3.9 Whale Optimization Algorithm (WOA)

4.3.9.1 Meta-Heuristic Description

WOA [73] is a swarm intelligence meta-heuristic that uses as a metaphor for its search heuristics the hunting behavior of humpback whales. Members of this species of whales are intelligent and social animals that live in groups or alone. This algorithm takes into consideration the bubble net hunting strategy observed in humpback whales in nature preying on krill or small fish. When using this hunting strategy, whales dive down and start to create bubbles with spiral movements around the school of fishes. As a whale encircles the prey, this whale then starts to swim up towards the surface while shrinking the circle of bubbles. Once the prey has been herd towards a favorable position, the same whale, or another whale in the same hunting group, swims up and eat the school of fish in a single gulp. The pseudo-code at Algorithm 12 describes the WOA algorithm.

Algorithm 12 Whale Optimization Algorithm (WOA)

INPUT: Objective Function (f)
Iterations (I)
Number of whales (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure WOA($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize moths $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{x}_{best}$
  for $k = 1$ to $I - 1$ **do**
    Update a
    for $i = 1$ to n **do**
      Generate p
      if $p \leq 0.5$ **then**
        Update $\boldsymbol{x}_i^{(k)}$ ▷ Eq. 4.32
      else
        Update $\boldsymbol{x}_i^{(k)}$ ▷ Eq. 4.33
      end if
      Evaluate $f(\boldsymbol{x}_i^{(k)})$
    end for
    Update $\boldsymbol{x}_{best}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

WOA simulates the bubble net hunting strategy by having a group $\boldsymbol{P}_c$ of n whales with their positions $\boldsymbol{x}_i^{(k)}$ during kth iteration represented as d-dimensional vectors that are also solutions for

an objective function f.

During WOA initialization, positions $\boldsymbol{x}_i^{(0)}$ are generated randomly inside of the search space. During iterations, WOA simulates the movement of whales during hunting by dividing the process into two stages: (a) prey search (**exploring**), and (b) prey attack (**exploiting**).

During the prey search stage, each ith whale position has its jth dimensions $\boldsymbol{x}_{i,j}^{(k)}$ updated as following:

$$x_{i,j}^{(k+1)} = \begin{cases} x_{r_1,j}^{(k)} - a(2r-1)\,|2r\,x_{r_1,j}^{(k)} - x_{i,j}^{(k)}| & \text{if } |a(2r-1)| \geq 1 \\ x_{best,j} - a(2r-1)\,|2r\,x_{best,j} - x_{i,j}^{(k)}| & \text{otherwise} \end{cases}, \tag{4.32}$$

where r_1 is a random index for whales ranging from $[1, n]$ with $r_1 \neq i$, $\boldsymbol{x}_{r_1}^{(k)}$ is the position for a random rth whale in the swarm, and r is a random real value in $[0, 1]$. Parameter a is a **internal** parameter that decreases linearly between 2 and 0 over the iterations using a **deterministic parameter control**

During the prey attack phase, each ith whale moves using a spiral shape around the best position described as follows:.

$$x_{i,j}^{(k+1)} = |x_{best,j} - x_{i,j}^{(k)}| e^{0.5r} cos(2\pi r) + x_{best,j}, \tag{4.33}$$

where $\boldsymbol{x}_{best}$ is the best solution found so far by WOA and r is a random real number in $[0, 1]$.

Since both hunting stages are done simultaneously in nature, a random test decides whether a whale uses prey search or attack movement types. This test has 50/50 chances to use these movement types.

4.3.9.2 Parameters Description for WOA

WOA does not have any **external static** parameter. However, it contains an **internal** parameters a that has its values decreased linearly by a **deterministic control parameter**. Parameter a controls whether WOA updates its values using a difference for another random whale or to the best whale with the chances of using the best whale position increasing with the meta-heuristic iterations. By changing this behavior, a influences whether a focus in an exploratory behavior to an exploitative one during the last iterations.

4.3.10 Bat Algorithm (BA)

4.3.10.1 Meta-Heuristic Description

BA [75] is a swarm intelligence meta-heuristic that uses as inspiration the approach in which microbats use their echolocation ability to search for food in nature while flying. These bats diet is composed of small fruits as well as insects, and they heavily depend on their ability to fly around the environment only using their hearing due to their incredibly poor eyesight. BA simulates this localization process by mimicking the bats' pulse loudness and pulses emission rates variation pattern during flying. As the bats fly towards the food, their pulse loudness decrease

while the rate in which they emit ultrasound sounds increases. Pseudo-code Algorithm 13 displays BA optimization meta-heuristic.

Algorithm 13 Bat Algorithm (BA)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of bats (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Pulse loudness (A)
Pulse emission rate (p_r)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

1: **procedure** BA($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$, p_r, A)
2: *Initialize bats* $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
3: *Initialize speeds* $\boldsymbol{v}_i^{(0)} (i = 1, ..., n)$
4: *Evaluate* $f(\boldsymbol{x}_i^{(0)}, \boldsymbol{\Omega}, \boldsymbol{\Psi})$
5: *Update* $\boldsymbol{x}_{best}$
6: **for** $k = 2$ to I **do**]
7: **for** $i = 1$ to N **do**
8: *Generate* F_i *in* $[F_{min,F_{max}}]$
9: *Update* $\boldsymbol{v}_i^{(k)}$ ▷ Eq. 4.34
10: *Update* $\boldsymbol{x}_i^{(k)} = \boldsymbol{v}_i^{(k)} + \boldsymbol{x}_i^{(k-1)}$
11: **if** $r < p_r$ **then**
12: *Update* $\boldsymbol{x}_i$ ▷ Eq. 4.35
13: **end if**
14: *Evaluate* $f(\boldsymbol{x}_i^{(k)})$
15: **end for**
16: *Update* $\boldsymbol{x}_{best}$
17: **end for**
18: **return** $\boldsymbol{x}_{best}$
19: **end procedure**

BA has a swarm of n bats in a group $\boldsymbol{P}_c$ that is used as search agents and each ith bat holds a d-dimensional real vector $\boldsymbol{x}_i^{(k)}$ at iteration k that represents a solution to the objective function f. Beyond that, bats in BA also contains a d-dimensional real vector that represents the bats speeds $\boldsymbol{v}_i^{(k)}$ at iteration k.

During the initialization, BA generates random solutions for f inside of the search space defined by the vectors $\boldsymbol{ub}$ and $\boldsymbol{lb}$ that represents, respectively, a vector containing the upper and lower boundaries for the search space. In this process, n solutions are assigned as positions $\boldsymbol{x}_i^{(0)}$ for bats in the swarm and another n solutions are assigned to the bats speeds $\boldsymbol{v}_i^{(0)}$.

During the iterations, BA updates the position for each ith bat with component $j \in [1, d]$ analogous as PSO (Section 4.3.10) updating their particles positions. BA updates the bats speed using equation:

$$v_{i,j}^{(k+1)} = v_{i,j}^{(k)} + F(\boldsymbol{x}_{best,j} - \boldsymbol{x}_{i,j}^{(k)}), \tag{4.34}$$

where $v_{i,j}$ is the jth dimension of the ith bat speed, $x_{i,j}$ is the ith bat position at component j, $x_{best,j}$ is the jth dimension of the best solution found so far, and F is a randomly generated real number inside of the interval $[F_{min}, F_{max}]$ with F_{max} and F_{min} being two **internal static** parameters.

BA also performs a random test to check whether it should another method to update a single bat position based on a **external static** parameter p_r called **emitted pulse rate**. If a bat pass a random check where a random generated real number r is $r < p_r$, then this bat position is updated using the following equation:

$$x_{i,j}^{(k+1)} = x_{i,j}^{(k)} + rA, \tag{4.35}$$

where r is a random real number in $[-1, 1]$, and A is an **external static** parameter called **pulses loudness**.

4.3.10.2 Parameters Description for BA

BA contains two **external static** parameters: (a) p_r emitted pulse rate, and (b) A pulses loudness. The parameter p_r influences BA by altering its chances to change the motion of bats from a more exploratory and global search based behavior of moving towards the best solution to a more exploitative and local search behavior of moving around its current position. While A influences BA by defining how possible large can the movement steps during the exploitative type of movements.

4.3.11 Adaptive Differential Evolution (JADE)

4.3.11.1 Meta-Heuristic Description

JADE [76] is a DE-based evolutionary meta-heuristic that adds **adaptive parameter control strategies** for DE **external static** parameters, namely, the **scaling factor** and the **crossover rate**. JADE also uses a different mutation strategy called $current-to-pbest/1$ one, and an archive of previous individuals in the population that can be used to improve the algorithm performance by adding more diversity of solutions in the population. The pseudo-code for JADE is present in Algorithm 14.

JADE has a population $\boldsymbol{P}_c$ containing n individuals holding d-dimensional real vectors $\boldsymbol{x}_i^{(k)}$ at iteration k. Each ith individual also contains information about their individual scaling factor F_i and crossover rate CR_i with these parameters being treated by JADE as **internal** ones.

During initialization stage (see lines 2 to 7 of Algorithm 14), JADE generates n solutions for an objective function f randomly inside of the search space given that each component is bounded by the elements in vectors $\boldsymbol{lb}$ lower-bounds and $\boldsymbol{ub}$ upper-boundaries. The parameters for each ith individual are also initially tuned as $F_i = \mu_F^0 = \mu_{Finitial}$ and $CR_i = \mu_{CR}^0 = \mu_{CRinitial}$ where μ_F^k and μ_{CR}^k are two **internal** parameters adjusted by an **adaptive control strategy** in JADE. The real numbers $\mu_{Finitial}$ and $\mu_{CRinitial}$ are **internal static** parameters.

During every iteration, JADE updates generate the mutant vectors $\boldsymbol{v}_i$ for each ith individual using the mutation strategy $current - to - pbest/1$ defines as follows:

$$\boldsymbol{v}_i = \boldsymbol{x}_i^{(k)} + F_i \left(\boldsymbol{x}_{pbest} - \boldsymbol{x}_i^{(k)} + \boldsymbol{x}_{r_1}^{(k)} - \boldsymbol{x}_{r_2}^{(k)} \right), \tag{4.36}$$

where $\boldsymbol{x}_{pbest}$ is a randomly chosen individual that is part of the best $100p\%$ of individuals in the population where p is an **external static** parameter called **mutation greediness**, and F_i is the adapted **scaling factor** for the ith individual at the kth iteration. The real numbers r_1 and r_2 are different random indices ($r_1 \neq r_2$) with $r_1 \in [1, n]$ and $r_2 \in [1, |\boldsymbol{P}_c \cup \boldsymbol{A}|]$ where $\boldsymbol{A}$ is an archive set that contains previous solutions of individuals in the population. Whenever $r_2 > n$, it becomes an index for an individual present in the archive $\boldsymbol{A}$ instead of an individual in the population.

Algorithm 14 Adaptive Differential Evolution (JADE)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Parameter Adaptation Rate (c)
Mutation Greediness Value (p)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure JADE($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}, c, p$)
  Initialize $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Obtain best Individual $\boldsymbol{x}_{best}$
  $\mu_F^0 = \mu_{F_{initial}}$
  $\mu_{CR}^0 = \mu_{CR_{initial}}$
  $\mathbf{A} = \emptyset$
  for $k = 1$ to $I - 1$ **do**
    $\boldsymbol{S}_F = \emptyset$
    $\boldsymbol{S}_{CR} = \emptyset$
    for $i = 1$ to n **do**
      Generate CR_i ▷ Eq. 4.39
      Generate F_i ▷ Eq. 4.41
      Randomly choose $\boldsymbol{x}_{pbest}$ *from* $100p\%$ *best solutions*
      (current-to-pbest/1) Mutation ▷ Eq. 4.36
      Binary Crossover ▷ Eq. 4.38
      Selection ▷ Eq. 4.3
      Update $\boldsymbol{S}_F$ ▷ Eq. 4.43
      Update $\boldsymbol{S}_{CR}$ ▷ Eq. 4.44
      Update $\boldsymbol{A}$ ▷ Eq. 4.37
    end for
    Remove solutions in $\boldsymbol{A}$ *until* $|\boldsymbol{A}| \leq n$
    Update $\mu_F^{(k+1)}$ ▷ Eq. 4.42
    Update $\mu_{CR}^{(k+1)}$ ▷ Eq. 4.40
    Update $\boldsymbol{x}_{best}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

The archive set $\boldsymbol{A}$ starts empty and is then populated throughout the algorithm iterations during the selection process. Whenever an individual has a worse evaluation than its trial vector, JADE then saves this individual solution $\boldsymbol{x}_i^{(k)}$ in the archive set. Whenever the archive size $|\boldsymbol{A}|$ surpasses the population size $\boldsymbol{P}_c$, a randomly chosen archived solution is removed from $\boldsymbol{A}$ until $|\mathbf{A}| \leq n$. The process to populate the archive set is expressed by the following equation:

$$\boldsymbol{A} \leftarrow \begin{cases} \boldsymbol{x}_i^{(k)} & \text{if } f(\boldsymbol{u}_i) < f(\boldsymbol{x}_i^{(k)}) \\ \emptyset & \text{otherwise} \end{cases} \tag{4.37}$$

JADE uses a **binary crossover strategy** analogous to the one employed by DE Section 4.3.2 in Eq. 4.2 to generate trial vectors. However, similar to the mutation strategy, each ith individual contains its crossover rate represented as CR_i. This updated binary crossover strategy is as following:

$$u_{i,j} = \begin{cases} \min(ub_j, \max(lb_j, v_{i,j})) & \text{if } r \leq CR_i \text{ or } j = r_j \\ x_{i,j}^{(k)} & \text{otherwise} \end{cases}, \tag{4.38}$$

where r is a random real number in $[0, 1)$, r_j is a random index in $[1, d]$ kept for all j for the generation of a trial-vector $\boldsymbol{u}_i$, and CR_i is the crossover rate used by individual i.

The **adaptive parameter control strategy** to adjust the crossover rates for each individual

work in an individual-scope using the following equation:

$$CR_i = \min(1, \max(0, \mathcal{N}(\mu_{CR}^{(k)}, 0.1))), \tag{4.39}$$

where CR_i is new randomly generated **crossover rate** for the ith individual truncated in $[0, 1]$, and $\mathcal{N}(\mu_{CR}^{(k)}, 0.1)$ is a normal distribution centered at $\mu_{CR}^{(k)}$ and standard deviation 0.1. Also, $\mu_{CR}^{(k)}$ is an **internal static** parameter that is controlled by another **adaptive control strategy** that vary as follows at every iteration k:

$$\mu_{CR}^{(k+1)} = \begin{cases} (1-c) \cdot \mu_{CR}^{(k)} + c \cdot \frac{\sum_{CR \in \boldsymbol{S}_{CR}} (\boldsymbol{S}_{CR})}{|\boldsymbol{S}_{CR}|} & \text{, if } |\boldsymbol{S}_{CR}| \neq 0 \\ \mu_{CR}^{(k)} & \text{, otherwise} \end{cases}, \tag{4.40}$$

where c is an **external static** parameter called **adaptation rate**, and $\boldsymbol{S}_{CR}$ is a set of successful crossover rates from the previous iteration.

In a equivalent manner, there is an **adaptive parameter control strategy** that update for every ith individual in the population the **scaling factor** F_i as follows:

$$\begin{aligned} \nu &= \mathcal{C}(\mu_F^{(k)}, 0.1) \\ F_i &= \begin{cases} \mathcal{C}(\mu_F^{(k)}, 0.1) & \text{, if } \nu \in (0, 1) \\ 1 & \text{, otherwise} \end{cases}, \end{aligned} \tag{4.41}$$

where F_i is randomly generated scaling factor for individual i, $\mathcal{C}(\mu_F^{(k)}, 0.1)$ is a Cauchy distribution with mean $\mu_F^{(k)}$ and standard deviation 0.1. Also, $\mu_F^{(k)}$ is an **internal** parameter at iteration k controlled by an **adaptive parameter control strategy** using the following equation:

$$\mu_F^{(k+1)} = \begin{cases} (1-c) \cdot \mu_F^{(k)} + c \cdot \frac{\sum_{F \in \boldsymbol{S}_F} F^2}{\sum_{F \in \boldsymbol{S}_F} F} & \text{if } |\boldsymbol{S}_F| \neq 0 \\ \mu_F^{(k)} & \text{otherwise} \end{cases}, \tag{4.42}$$

where c is the same **adaptation rate**, and $\boldsymbol{S}_F$ is a set of successful scaling factors from the previous iteration.

Both the successful sets $\boldsymbol{S}_{CR}$ and $\boldsymbol{S}_F$ start at every iteration empty and then they are populated whenever a trial vector $\boldsymbol{u}_i$ evaluation is better than its original individual $\boldsymbol{x}_i^{(k)}$ during the selection operation as described in Eq. 4.43 and Eq. 4.44.

$$\boldsymbol{S}_F \leftarrow \begin{cases} F_i^{(k)} & \text{, if } f(\boldsymbol{u}_i) < f(\boldsymbol{x}_i^{(k)}) \\ \emptyset & \text{, otherwise} \end{cases}. \tag{4.43}$$

$$\boldsymbol{S}_{CR} \leftarrow \begin{cases} CR_i^{(k)} & \text{if } f(\boldsymbol{u}_i) < f(\boldsymbol{x}_i^{(k)}) \\ \emptyset & \text{otherwise} \end{cases}. \tag{4.44}$$

4.3.11.2 Parameters Description for JADE

JADE contains two **external static** parameters: (a) p called **mutation greediness**, and (b) c called **adaptation rate**. Parameter $p \in (0, 1]$ controls the size of the set of $100p\%$ best solutions

are randomly selected. The value for p influences the exploratory behavior of the algorithm since a small value for p forces JADE individuals to move towards a small set of best individuals focusing in few local minima solutions. Meanwhile, a large value for p allows JADE to move towards a broad set of local optimum, increasing its exploratory behavior. The parameter $c \in (0, 1]$) with suggested value of 0.15 controls the rate in which the adaptation mechanisms for $\mu_F^{(k)}$ and $\mu_{CR}^{(k)}$ convert towards the averages of successful parameter values in the kth iteration.

JADE contains six **internal** parameters: (a) F_i, (b) $\mu_F^{(k)}$, (c) CR_i, (d) $\mu_{CR}^{(k)}$, (e) $\mu_{F_{initial}}$, and (f) $\mu_{CR_{initial}}$. Four of them are parameters that configure **parameter control strategies**. It illustrates that even though JADE is capable of adjusting its parameters during execution, the algorithm designer still has to decide values for some internal parameters that influence the adaptive meta-heuristic performance. JADE has four **parameter control strategies** with all of them fitting in the **adaptive** category present in Section 4.2.1.

The parameter F_i is the **scaling factor** for an individual i and it is controlled in an individual-scope by randomly re-sampling its values at every iteration using another **internal** parameter $\mu_F^{(k)}$. Meanwhile, $\mu_F^{(k)}$ is adjusted in a population-scope using as evidence for its changes the improvement in objective function f evaluation that is indicated by a set of successful values for $\boldsymbol{S}_F$. The adaptation mechanism for $\mu_F^{(k+1)}$ guides its value in the next iteration towards another region of the space of possible scaling factors by combining its current value $\mu_F^{(k)}$ and the averages of the best scaling factors in iteration k. The parameter CR_i is configured by an **adaptive parameter control** that works in a similar manner as the one for F_i to update the values of crossover rates in an individual scope together with the **adaptive parameter control strategy** for $\mu_{CR}^{(k)}$.

While $\mu_{F_{initial}} = 0.5$ and $\mu_{CR_{initial}} = 0.5$ are **internal static** parameters that defines the initial setting points for the scaling factors, and crossover rates respectively.

4.3.12 Crossover Strategy Adaptive Self-Adaptive DE (CSASADE)

4.3.12.1 Meta-Heuristic Description

CSASADE, as proposed by [77], is an adaptive version of the DE algorithm. CSASADE adds to DE adaptive parameter control strategies similar to those of JADE (Section 4.3.11) as well as **adaptive operator selection** mechanisms. The name CSASADE has the term "Self-Adaptive". However, CSASADE does not use any self-adaptive parameter control mechanisms, since no parameter evolves encoded together with the decision variable. Instead, its parameter control strategies fit on the adaptive strategy category. Algorithm 15 presents CSASADE heuristic in pseudo-code form.

Each ith individual in the population $\boldsymbol{P}_c$ holds a possible solution $\boldsymbol{x}_i$ for the objective function f as well as their own parameters F_i and CR_i and a pair of indices mut_i and cro_i that denotes, respectively, which mutation and crossover strategy the individual is using to generate its mutant vectors $\boldsymbol{v}_i$ and trial vectors $\boldsymbol{u}_i$. CSASADE uses two set of **internal** parameters $N_{mut}^{(k)}$ and $N_{cro}^{(k)}$ that holds, respectively, the number of individuals during a iteration k using a mutation strategy

Algorithm 15 Adaptive Differential Evolution (CSASADE)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of dimensions (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure CSASADE($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  Evaluate $f(\boldsymbol{x}_i^{(0)} (i = 1, ..., n)$
  Update $\boldsymbol{x}_{best}$
  Update $\boldsymbol{x}_{worst}$
  $N_{mut}^{(0)} = \lceil \frac{n}{5} \rceil$
  $N_{cro}^{(0)} = \lceil \frac{n}{2} \rceil$
  $\mu_{wF}^{(0)} = \mu_{wFinitial}$
  $\mu_{wCR}^{(0)} = \mu_{wCRinitial}$
  $I_{start} = 0.2I$
  for $k = 1$ to $I - 1$ **do**
    Update σ ▷ Eq. 4.61
    for $i = 1$ to n **do**
      Generate CR_i ▷ Eq. 4.64
      Generate F_i ▷ Eq. 4.60
      if $k < I_{start}$ **then**
        (*rand*/1) *Mutation* ▷ Eq. 4.45
        Binary Crossover ▷ Eq. 4.2
      else
        mut_i*th Mutation*
        cro_i*th Crossover*
      end if
      Selection ▷ Eq. 4.3
      Calculate weight w_i ▷ Eq. 4.66
    end for
    Update $\mu_{wF}^{(k)}$ ▷ Eq. 4.62
    Update $\mu_{wC}R^{(k)}$ ▷ Eq. 4.65
    if $k \geq I_{start}$ **then**
      Adapt Mutation Numbers ▷ Eq. 4.52
      Adapt Crossover Numbers ▷ Eq. 4.56
      for $i = 1$ to n **do**
        Reassign $mut_i^{(k)}$
        Reassign $cro_i^{(k)}$
      end for
    end if
    Update $\boldsymbol{x}_{best}$
    Update $\boldsymbol{x}_{worst}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

or crossover strategy in the pool of possible ones.

The symbolic value mut = { *rand/1, rand/2, current-to-best/1, current-to-best/2, best/2* } represents one of the possible mutation strategies used by CSASADE and for each ith individual mut_i correspond to an index in the pool of mutation strategies that ranges in $[1, 5]$. These mutation strategies and their indices is represented as follows: (a) *rand/1* mutation (Eq. 4.45) with index $mut_i = 1$ for individual i and $N_{rand/1}$ is the number of individuals that uses this operator; (b) *rand/2* mutation (Eq. 4.46) with index $mut_i = 2$ for individual i and $N_{rand/2}$ is the number of individuals that uses this operator; (c) *current-to-best/1* mutation (Eq. 4.47) with index $mut_i = 3$ for individual i and $N_{current\text{-}to\text{-}best/1}$ is the number of individuals that uses this operator; (d) *current-to-best/2* mutation (Eq. 4.48) with index $mut_i = 4$ for individual i and $N_{current\text{-}to\text{-}best/2}$ is the number of individuals that uses this operator; (e) *best/2* mutation (Eq. 4.49) with index $mut_i = 5$ for individual i and $N_{best/2}$ is the number of individuals that uses this operator. In Equations 4.45 to 4.49, r_1, r_2, r_3, r_4, and r_5 are different random integers that represent indices in $[1, n]$. Vector $\boldsymbol{x}_{best}$ is the best solution found so far, $\boldsymbol{v}_i$ is the mutant vector

generated by the ith individual, and F_i is the scaling factor used by individual i.

$$\boldsymbol{v}_i = \boldsymbol{x}_{r_1}^{(k)} + F_i(\boldsymbol{x}_{r_2}^{(k)} - \boldsymbol{x}_{r_3}^{(k)}). \tag{4.45}$$

$$\boldsymbol{v}_i = \boldsymbol{x}_{r_1}^{(k)} + F_i(\boldsymbol{x}_{r_2}^{(k)} - \boldsymbol{x}_{r_3}^{(k)} + \boldsymbol{x}_{r_4}^{(k)} - \boldsymbol{x}_{r_5}^{(k)}). \tag{4.46}$$

$$\boldsymbol{v}_i = \boldsymbol{x}_{i}^{(k)} + F_i(\boldsymbol{x}_{best} - \boldsymbol{x}_{i}^{(k)} + \boldsymbol{x}_{r_1}^{(k)} - \boldsymbol{x}_{r_2}^{(k)}). \tag{4.47}$$

$$\boldsymbol{v}_i = \boldsymbol{x}_{i}^{(k)} + F_i(\boldsymbol{x}_{best} - \boldsymbol{x}_{i}^{(k)} + \boldsymbol{x}_{r_1}^{(k)} - \boldsymbol{x}_{r_2}^{(k)} + \boldsymbol{x}_{r_3}^{(k)} - \boldsymbol{x}_{r_4}^{(k)}). \tag{4.48}$$

$$\boldsymbol{v}_i = \boldsymbol{x}_{best}^{(k)} + F_i(\boldsymbol{x}_{r_1}^{(k)} - \boldsymbol{x}_{r_2}^{(k)} + \boldsymbol{x}_{r_3}^{(k)} - \boldsymbol{x}_{r_4}^{(k)}). \tag{4.49}$$

Similar to mutation strategies, CSASADE also has a smaller pool of crossover strategies where $cro = \{bin, exp\}$. The integer value cro_i for the ith individual contains the index for one of these two crossover strategies where: (a) *bin* binary crossover (Eq. 4.38) with index $cro_i = 1$ for individual i and N_{bin} is the number of individuals that uses this operator; (b) *exp* exponential crossover (Algorithm 16) with index $cro_i = 2$ for individual i and N_{exp} is the number of individuals that uses this operator;

Algorithm 16 DE Exponential Crossover

INPUT: Decision variable $(\boldsymbol{x}_i^{(k)})$
Mutant vector $(\boldsymbol{v}_i)$
OUTPUT: Trial vector $(\boldsymbol{u}_i)$

1: **procedure** DEExpCrossover($\boldsymbol{x}_i^{(k)}, \boldsymbol{v}_i$)
2: $d = |\boldsymbol{x}_i^{(k)}|$
3: $\boldsymbol{u}_i = \boldsymbol{x}_i^{(k)}$
4: *Randomly select integer* j *in* $[1, N)$
5: $L = 1$
6: **do**
7: $u_{i,j} = v_{i,j}$
8: $j =$ mod $(j + 1, d)$*
9: $L = L + 1$
10: r *is a random real number in* $[0, 1)$
11: **while** $r < CR$ and $L \leq N$
12: **return** $\boldsymbol{u}_i$
13: **end procedure**

* $mod(\cdot)$ is a modulo operator that returns the remainder of a integer division..

During the initialization process, CSASADE assigns for each ith individual in population $\boldsymbol{P}_c$ a randomly generated solution for f in the search space defined by the upper and lower boundary vectors $\boldsymbol{ub}$ and $\boldsymbol{lb}$. The parameters for each ith individual is initialized as in JADE with $F_i = \mu_{wF}^0 = \mu_{wFinitial}$ and $CR_i = \mu_{wCR}^0 = \mu_{wCRinitial}$ where μ_F^k and μ_{CR}^k are two **internal** parameters adjusted by an **adaptive control strategy**. Both $\mu_{Finitial}$ and $\mu_{CRinitial}$ are **internal static** parameters. The number of individuals using each operator for mutation and crossover strategies are initialized as present in Eq, 4.50 and Eq. 4.51.

$$\begin{aligned} N_{rand/1}^{(0)} = N_{rand/2}^{(0)} = N_{current-to-best/1}^{(0)} = \\ N_{current-to-best/2}^{(0)} = N_{rand-to-best/1}^{(0)} = \left\lfloor \frac{n}{5} \right\rfloor. \end{aligned} \tag{4.50}$$

$$N_{bin}^{(0)} = N_{exp}^{(0)} = \left\lfloor \frac{n}{2} \right\rfloor. \tag{4.51}$$

Then, the mutation operator indices for the population are re-assigned by an **adaptive operator selector** that works in a process that divides the population $\boldsymbol{P}_c$ in five sets and for the first set of individuals in the population with indices $l_1 \in [1, N^{(0)}_{rand/1}]$ receives $mut_{l_1} = 1$, then the second set with indices $l_2 \in [N^{(0)}_{rand/1} + 1, N^{(0)}_{rand/1} + N^{(0)}_{rand/2}]$ receives $mut_{l_1} = 1$, and so on, until the last set with indices $l_5 \in N^{(0)}_{current-to-best/2} + 1, N^{(0)}_{current-to-best/2} + N^{(0)}_{best/2}]$ receives indices $mut_{l_5} = 5$. Other **adaptive operator selector** that works in the same fashion re-assigns for each individual their crossover operators in the pool $cro = \{bin, exp\}$.

Through the iterations, CSASADE generates for each ith individual their mutant vector $\boldsymbol{u}_i$ using the mutation strategy with index mut_i and trial vectors $\boldsymbol{v}_i$ using their crossover strategy with index mut_i.

The **internal** parameters that contain the number of individuals a mutation strategy $N^{(k)}_{mut}$ at iteration k is adjusted using an **adaptive control parameter strategy** that behaves as following:

$$N^{(k+1)}_{mut} = \begin{cases} N^{(k)}_{mut} + 1, \text{ if } \Delta_{N_{mut}} > 0 \\ N^{(k)}_{mut} - 1, \text{ if } \Delta_{N_{mut}} < 0 \\ N^{(k)}_{mut}, \text{ otherwise} \end{cases}, \tag{4.52}$$

where $mut = \{$ *rand/1, rand/2, current-to-best/1, current-to-best/2, best/2* $\}$, and $\Delta_{N_{mut}}$ is an expected variation for $N^{(k)}_{mut}$ and it is calculated using the following equation:

$$\Delta_{N_{mut}} = \left\lfloor N \frac{s_{mut}}{s_{all\_mut}} + 0.5 \right\rfloor - N^{(k)}_{mut}, \tag{4.53}$$

where s_{mut} is the sum of differences between evaluations of f for the solutions of individuals that uses a mutation strategy mut and the worst solution evaluation $f(\boldsymbol{x}_{worst})$, and s_{all_mut} is the sum of differences for all individuals against the worst evaluation $f(\boldsymbol{x}_{worst})$ is calculated using Eq. 4.55. The value s_{mut} is calculated as follows:

$$s_{mut} = \sum_{\forall \boldsymbol{x}^{(k)}_i \in \{\boldsymbol{x}_i \in \boldsymbol{P}_c \,:\, mut_i = mut\}} |f(\boldsymbol{x}^{(k)}_i) - f(\boldsymbol{x}^{(k)}_{worst})|. \tag{4.54}$$

$$s^{(k)}_{all\_mut} = s^{(k)}_{rand/1} + s^{(k)}_{rand/2} + s^{(k)}_{current-to-best/1} + s^{(k)}_{current-to-best/2} + s^{(k)}_{rand-to-best/1}. \tag{4.55}$$

In a similar fashion as the one used by mutation strategy, the set of **internal** parameters that controls the number of individuals using each crossover strategy is updated as following by an **adaptive parameter control strategy**:

$$N^{(k+1)}_{cro} = \begin{cases} N^{(k)}_{cro} + 1, \text{ if } \Delta_{N_{cro}} > 0 \\ N^{(k)}_{cro} - 1, \text{ if } \Delta_{N_{cro}} < 0 \\ N^{(k)}_{cro}, \text{ otherwise} \end{cases}, \tag{4.56}$$

where $cro = \{bin, exp\}$, and $\Delta_{N_{cro}}$ is an expected variation for $N^{(k)}_{mut}$ and it is calculated as follows:

$$\Delta_{N_{cro}} = \left\lfloor N \frac{s_{cro}}{s_{all\_cro}} + 0.5 \right\rfloor - N^{(k)}_{cro}, \tag{4.57}$$

where s_{mut} is the sum of differences between evaluations of f for the solutions of individuals that uses a mutation strategy mut and the worst solution evaluation $f(\boldsymbol{x}_{worst})$, and s_{all_mut} is the sum of differences for all individuals against the worst evaluation $f(\boldsymbol{x}_{worst})$ is calculated using Eq. 4.59. The value of s_{cro} is obtained as follows:

$$s_{cro} = \sum_{\forall \boldsymbol{x}_i^{(k)} \in \{\boldsymbol{x}_i \in \boldsymbol{P}_c \,:\, cro_i = cro\}} |f(\boldsymbol{x}_i^{(k)}) - f(\boldsymbol{x}_{worst}^{(k)})|. \tag{4.58}$$

$$s_{all\_mut}^{(k)} = s_{rand/1}^{(k)} + s_{rand/2}^{(k)} + s_{current-to-best/1}^{(k)} + s_{current-to-best/2}^{(k)} + s_{rand-to-best/1}^{(k)}. \tag{4.59}$$

CSASADE uses an **adaptive parameter control strategy** to configure the scaling factors for each ith individual by re-sampling the value F_i as follows:

$$\begin{aligned} \nu &= \mathcal{N}(\mu_{w_F}^{(k)}, \sigma) \\ F_i^{(k)} &= \begin{cases} \nu & \text{if } \nu \in (0,1), \\ 1 & \text{otherwise} \end{cases} \end{aligned} \tag{4.60}$$

where F_i is randomly generated scaling factor for individual i, $\mathcal{N}(\mu_{w_F}^{(k)}, \sigma)$ is a Normal distribution with average $\mu_{w_F}^{(k)}$ and standard deviation σ. Both $\mu_{w_F}^{(k)}$ and σ are **internal** parameters updated using **parameter control strategies**.

The standard deviation σ is adjusted using a **deterministic parameter control strategy** in which its values decreases between σ_{max} and σ_{min} throughout the iterations using equation:

$$\sigma = \sigma_{max} - (\sigma_{max} - \sigma_{min}) \left(1 - \left(\frac{k}{I-1}\right)^2\right), \tag{4.61}$$

where σ_{max} and σ_{min} are two **internal static** parameters.

Every iteration, μ_{w_F} is also adjusted using an **adaptive parameter control strategy** as follows:

$$\mu_{w_F}^{(k)} = \sum_{i=1}^{n} w_i^{(k)} F_i, \tag{4.62}$$

where F_i is the scaling factor for the ith individual, and $\boldsymbol{w}_i$ is a weighting factor calculated for individual i as follows:

$$w_i^{(k)} = \frac{|f(\boldsymbol{x}_i^{(k)}) - f(\boldsymbol{x}_{worst}^{(k)})|}{\sum_{j=1}^{n} |f(\boldsymbol{x}_j^{(k)}) - f(\boldsymbol{x}_{worst}^{(k)})|}, \tag{4.63}$$

where $\boldsymbol{x}_i$ is the solution for individual i, and $\boldsymbol{x}_{worst}^{(k)}$ is the worst solution found in the current iteration k.

In CSASADE, crossover rate CR_i for each ith individual in the population is adjusted using a adaptive parameter control as follows:

$$CR_i = \min\left(1, \max\left(0, \mathcal{N}\left(\mu_{w_{CR}}^{(k)}, \sigma\right)\right)\right) \tag{4.64}$$

where σ is the same deviation used in Eq. 4.61, and $\mu_{w_{CR}}^{(k)}$ is a **internal** parameter that holds the weighted average crossover rates in the population.

The average $\mu_{wCR}^{(k)}$ is adjusted every iteration by an adaptive parameter control strategy that works as follows:

$$\mu_{wCR}^{(k)} = \sum_{i=1}^{n} w_i \cdot CR_i, \tag{4.65}$$

where w_i is the weighting factors defined by Eq. 4.66, and CR_i is the crossover rate used by the ith individual in the population.

$$w_i^{(k)} = \frac{|f(\boldsymbol{x}_i^{(k)}) - f(\boldsymbol{x}_{worst}^{(k)})|}{\sum_{j=1}^{n} |f(\boldsymbol{x}_j^{(k)}) - f(\boldsymbol{x}_{worst})|} \tag{4.66}$$

In [77], CSASADE also contains an **internal static** I_{start} parameter that control in which iteration the adaptive operation selection mechanisms start to take place. So, during the first I_{start} iterations, CSASADE uses only $rand/1$ mutation and bin crossover strategies as of the DE (Section 4.3.2).

During the first I_{start} iterations, CSASADE uses only the strategies presented in $DE/rand/1/bin$ and only starts to apply different crossover and mutation strategies after these initial iterations.

4.3.12.2 Parameters Description for CSASADE

CSASADE has no **external static** parameters that need to set by the user. However, CSASADE has fourteen **internal** parameters. This large number of parameters that are "invisible", at least in the user point of view, indicates the complexity of CSASADE. CSASADE also has two **adaptive operation selection** mechanisms (Section 4.2.2) and seven **parameter control strategies** with six of them fitting in the adaptive category and one of them fitting the deterministic category in Section 4.2.1. The list of internal parameters are: (a) F_i, (b) μ_{wF}, (c) CR_i, (d) μ_{wCR}, (e) (f) σ, $\mu_{wFinitial}$, (f) $\mu_{wCRinitial}$, (h) σ_{max}, (i) σ_{min}, (j) N_{mut}, (k) N_{cro}, (l) mut_i, (m) cro_i, and (n) I_{start}.

Parameters F_i and CR_i are respectively the **scaling factors** and **crossover rates** for each ith individual in the population. Their values are controlled by **adaptive parameter controls strategies** that adjust them in an individual-scope using other **internal** parameters, namely, μ_{wF}, μ_{wCR} and σ. Both μ_{wF} and $\mu_{wC}R$ are two weighted averages controlled by their adaptive parameter control strategies in a population-scope using as evidence for change the objective function results. The weighted averages μ_{wF} and μ_{wCR} are influenced more by individuals that obtain evaluations with large differences for the worst evaluation found in an iteration, and less influenced by evaluations close to the worst evaluation.

The value for σ is reduced through the iterations by a **deterministic parameter control strategy**, and it influences the exploratory behavior of CSASADE searching for possible suitable values of F_i and CR_i. The reduction of σ forces CSASADE to have a broad spread of parameter values during the initial iterations and a small spread during the last iterations.

The **internal static** parameters $\mu_{wFinitial}$, $\mu_{wCRinitial}$ controls the initial value for *internal* **scaling factors** and **crossover rates** therefore defining the behaviour for CSASADE during

the first iteration. Their values are $\mu_{wF_{initial}} = 0.5$ and $\mu_{wCR_{initial}} = 0.5$. Both parameters $\sigma_{max} = 0.8$ and $\sigma_{min} = 0.35$ controls respectively the rate in which the spreading of scaling factors and crossover rates are reduced during CSASADE execution.

The **internal** parameters N_{mut} and N_{cro} are adjusted by **adaptive parameter control strategies** in a population-scope, and their evidence for change is similar to those used by μ_{wF} and μ_{wF} mechanisms using the differences between solution evaluations and the worst evaluation. Mutation strategies that regularly obtain solutions with evaluations closer to the worst evaluation, i.e., deemed bad solutions, ends up with $N_{mut} = 0$ and no individuals using them. The same applies to crossover strategies. In this way, the first iterations, after I_{start}, CSASADE explores which strategies work and which do not, and during the last iterations, CSASADE uses only the best strategies.

A pair of **adaptive operation selection** schemes directly use N_{mut} and N_{cro} to adjust the mutation and crossover operations used by individuals. These **adaptive operation selection** changes the operators by changing indices mut_i and cro_i for each ith individual.

At last, the **internal static** parameter controls how many iterations CSASADE should work only using $rand/1$ mutation and bin crossover before start to use **adaptive operator selection**. It affects how spread are the solutions in the search space when the process to explore and select strategies start. The proposed value for I_{start} is 20% of I (the maximum number of iterations).

4.3.13 Discrete Particle Swarm Optimization (DPSO)

4.3.13.1 Meta-Heuristic Description

DPSO [78] is a modified PSO-based meta-heuristic that uses a discrete representation for the positions of particles to exclusively solve COP. DPSO has a swarm $\boldsymbol{P}$ of n particles and each ith particle has a position $\boldsymbol{x}_i^{(k)}$ at iteration k in a d-dimensional discrete search space, and, analogous to PSO (Section 4.3.3), a copy of the best individual position $\boldsymbol{x}_{pbest_i}$ and reference for the best position $\boldsymbol{x}_{best}$ found so far by the DPSO. Algorithm 17 illustrates DPSO in pseudo-code form.

During the initialization, n random solutions where each solution for the COP with objective function f is generated inside of the search space given the integer vectors $\boldsymbol{ub}$ and $\boldsymbol{lb}$ that compose the upper and lower bounds values for the search space components. The **internal** parameter w, called **inertia weight**, is initialized during this stage using the value held by an **external static** parameter $w_{initial}$.

In the iterative stage, DPSO updates the position for each ith individual in the jth dimension using the following equation:

$$x_{i,j}^{(k+1)} = g(x_{best}, g(x_{pbest_{i,j}}, v_{i,j})), \tag{4.67}$$

where $g : \{\mathbb{Z}, \mathbb{Z}\} \to \mathbb{Z}$ is a function that given two integer numbers returns a third integer number randomly generated between them as expressed in Eq. 4.68, and $v_{i,j}$ is a d-dimensional vector that represents the particle i speed.

Algorithm 17 Discrete Particle Swarm Optimization (DPSO)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Initial inertial coefficient ($w_{initial}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

1: **procedure** DPSO($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}, w_{initial}$)
2: *Initialize* $\boldsymbol{x}_i^{(0)}$ $(i = 1, ..., N)$
3: *Evaluate* $f(\boldsymbol{x}_i^{(0)})$
4: $w = w_{initial}$
5: *Update* $\boldsymbol{x}_{pbest_i}$
6: *Update* $\boldsymbol{x}_{best}$
7: **for** $k = 1$ to $I - 1$ **do**
8: **for** $i = 1$ to n **do**
9: *Calculate* $\boldsymbol{x}_i^{(k)}$ ▷ Eq. 4.67
10: *Evaluate* $f(\boldsymbol{x}_i^{(k)})$
11: *Update* $\boldsymbol{x}_{pbest_i}$
12: **end for**
13: *Update* $\boldsymbol{x}_{best}$
14: $w = w\beta$
15: **end for**
16: **return** $\boldsymbol{x}_{best}$
17: **end procedure**

$$g(a,b) = \begin{cases} b + \lfloor r_1(a-b) + 0.5 \rfloor & \text{if } blea \\ a + \lfloor r_1(b-a) + 0.5 \rfloor & \text{otherwise} \end{cases} \qquad a, b \in \mathbb{Z}, \tag{4.68}$$

where r_1 are randomly generated using a uniform distribution in the interval $[0, 1]$. Different than in the canonical PSO, the speed vector $\boldsymbol{v}_i$ for a particle i does not contain information about its motion in the search space. Instead, $\boldsymbol{v}_i$ is a combination of a randomly generated vector with the position of the ith particle. The value for $v_{i,j}$ is a calculated using the following equation:

$$v_{i,j} = \begin{cases} r_{ej} & \text{if } r_2 < w^{(k)} \\ x_{i,j}^{(k)} & \text{otherwise} \end{cases}, \tag{4.69}$$

where r_2 is a random real number i n $[0, 1]$, r_{ej} is a random generated integer inside of the range $[lb_j, ub_j]$, and $w^{(k)}$ is the **inertia weight** that has its value configured by a **deterministic parameter control strategy** using the following equation:

$$w^{(k)} = \beta \; w^{(k-1)}, \tag{4.70}$$

where β is an **external static** parameter in the range $(0, 1)$.

4.3.13.2 Parameters Description for DPSO

DPSO has two **external static** parameters: (a) $w_{initial}$, and (b) β. The value for $w_{initial}$ influences DPSO by defining the initial value for w that in turn is used generating speed vectors for the particles, Parameter β controls the rate in which the **internal** parameter w changes

through the course of the iterations. The suggested value for these parameters are $\beta = 0.9$ and $w_{initial} = 0.8$.

DPSO has only one **internal** parameter: w that is called **inertia weight** and it controls the probability in which random solutions components are added into particle speed vectors. For this reason, a large value of w results in DPSO preferring a more exploratory behavior. Similar to the **inertia weight** in PSO, w decreases through the iterations using a deterministic **parameter controls strategy**.

4.3.14 Self-Adaptive Particle Swarm Optimization (SAPSO)

4.3.14.1 Meta-Heuristic Description

SAPSO [79] is a swarm intelligence PSO-based meta-heuristic that adds self-adaptive parameter control mechanisms to PSO intending to improve its performance. Algorithm 18 displays SAPSO using pseudo-code.

Algorithm 18 Self-Adaptive Particle Swarm Optimization (SAPSO)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure SAPSO($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize solutions $\boldsymbol{x}_i$ $(i = 1, ..., n)$
  Initialize parameters c_{1i} $(i = 1, ..., n)$
  Initialize parameters c_{2i} $(i = 1, ..., n)$
  Initialize parameters v_{nmaxi} $(i = 1, ..., n)$
  $\boldsymbol{z}_i = (\boldsymbol{x}_i, c_{1i}, c_{2i}, v_{nmaxi})$ $(i = 1, ..., n)$
  $\boldsymbol{C}_l = \emptyset$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{z}_{pbest_i}$
  Update $\boldsymbol{z}_{besti}$
  Update collision list $\boldsymbol{C}_l$
  Update successful particles set $\boldsymbol{S}_p$
  for $k = 1$ to $I - 1$ **do**
    $\boldsymbol{S}_p = \emptyset$
    Update $w^{(k)}$ ▷ Eq. 4.74
    for $i = 1$ to n **do**
      Calculate $\boldsymbol{v}_i^{(k)}$ ▷ Eq. 4.72
      Calculate $\boldsymbol{z}_i^{(k)}$ ▷ Eq. 4.73
    end for
    for $i = 1$ to n **do**
      Evaluate $f(\boldsymbol{x}_i^{(k)})$
      Update $\boldsymbol{z}_{pbest_i}$
      Update $\boldsymbol{z}_{best_i}$
      Update $\boldsymbol{S}_p$
      Update $\boldsymbol{C}_l$
      if $\boldsymbol{z}_{i,j_x}^{(k)} \notin \boldsymbol{S}_p \wedge \boldsymbol{z}_{i,j_x}^{(k)} \in \boldsymbol{C}_l$ **then**
        Regenerate $\boldsymbol{z}_{i,j_x}^{(k)}$
        Remove i from $\boldsymbol{C}_l$
      end if
    end for
  end for
  return $\boldsymbol{x}_{best}$
end procedure

SAPSO has a swarm of $\boldsymbol{P}_c$ with n particles and each ith particle keeps a position $\boldsymbol{z}_i^{(k)}$ represented by a $d+3$ dimensions real vector, a speed vector $\boldsymbol{v}_i^{(k)}$ with the same number of dimensions, a copy of the individual best position $\boldsymbol{z}_{pbest_i}$, and a reference for the best position vector $\boldsymbol{z}_{best}$

found so far. Each ith particle position has its first d dimensions forming a solution $\boldsymbol{x}_i$ for an single-objective function f, similar to the position vector of the canonical PSO, and the last three components are PSO parameters for that particle, respectively, ${c1_i}^{(k)}$, ${c2_i}^{(k)}$ and ${v_{nmax}}_i^{(k)}$ that represents the social and cognitive factors, and its normalized maximum speed per dimension. In other words, $\boldsymbol{z}_i^{(k)} = (\boldsymbol{x}_i^{(k)}, {c1_i}^{(k)}, {c2_i}^{(k)}, {v_{nmax}}_i^{(k)})$. Since $\boldsymbol{z}_i^{(k)}$ is composed of multiple parts, the search space for $\boldsymbol{z}_i^{(k)}$ is composed of the search-space for the problem at hand as well as the search-space for the parameters. The search space boundaries used by SAPSO are represented by two vectors $\boldsymbol{ub}_z = (\boldsymbol{ub}, 4, 4, 1)$ and $\boldsymbol{lb}_z = (\boldsymbol{lb}, 4, 4, 1)$ that are, respectively, the upper boundaries and lower boundaries.

SAPSO initialization process randomly generates its particles $\boldsymbol{z}_i$ using a random uniform distribution for each jth component in the interval $[lb_{zj}, ub_{zj}]$ with $j = \{1, ..., d+3\}$. In the initialization stage, the particles speeds are also generated for the first iteration using fixed values, where the first d elements are set as $v_{i,j}^{(0)} = \frac{||\boldsymbol{ub}-\boldsymbol{lb}||}{2}$ with $j = \{1, ..., d\}$. The other components are initialized as follows: $v_{i,d+1}^{(0)} = v_{i,d+2}^{(0)} = 2$ and $v_{i,d+3}^{(0)} = 0.5$.

During the iterative process, SAPSO update the speed for each jth component of each ith particle using the following equation for a placeholder variable with $j = \{1, ..., d\}$:

$$\nu_{i,j} = w^{(k)} v_{i,j}^{(k)} + r_1 z_{i,j_{d+1}}^{(k)} (z_{gbest}^{(k)} - z_{i,j}^{(k)}) + r_2 z_{i,j_{d+2}}^{(k)} (z_{i\_pbest,j}^{(k)} - z_{i,j}^{(k)}), \tag{4.71}$$

where $w^{(k)}$ is an **internal** parameter called the **inertia weight** during iteration k. Parameter $w^{(k)}$ is adjusted by a **deterministic parameter control strategy**, and $\nu_{i,j}$ is a placeholder variable.

Since each multiple components of a particle speed have different boundaries, the use of ν to update differently each part for a particle speed as described by the following equation:

$$v_{i,j} = \begin{cases} \min(z_{i,d+3}^{(k)} \frac{||\boldsymbol{ub}-\boldsymbol{lb}||}{2}, \max(-z_{i,d+3}^{(k)} \frac{||\boldsymbol{ub}-\boldsymbol{lb}||}{2}, \nu_{i,j})) & \text{if } j \in [1, d] \\ \min(4 z_{i,d+3}^{(k)}, \max(-4 z_{i,d+3}^{(k)}, \nu_{i,j})) & \text{if } j \in [d+1, d+2] \\ \min(z_{i,d+3}^{(k)}, \max(-z_{i,d+3}^{(k)}, \nu_{i,j})) & \text{otherwise} \end{cases}, \tag{4.72}$$

After the calculation of speed of a particle i its position for each jth component is updated as follows:

$$z_{i,j} = \max\left(ub_{zj}, \min\left(ub_{zj}, z_{i,j}^{(k)} + v_i^{(k+1)}\right)\right), \tag{4.73}$$

where $j = \{1, ..., d\}$.

SAPSO update its **inertial weight** $w^{(k)}$ by a **deterministic parameter control** using the following equation:

$$w^{(k)} = 0.5 + \frac{1}{2(ln(k)+1)}. \tag{4.74}$$

Since PSO is prone to have multiple particles to converge to a local minimum with a lower diversity of solutions, SAPSO implements a strategy of regeneration-on-collision in which particles that "collide"with the global best solution are reinitialized randomly to improve diversity. This process uses a pair of sets $\boldsymbol{S}_p$ and $\boldsymbol{C}_l$, respectively, successful particles indices and collision list

indices. SAPSO populates the $\boldsymbol{S}_p$ set whenever a particle i obtains a fitness evaluation that is equal or better than the one obtained by the global best solution $\boldsymbol{z}_{gbest}$ as present in Eq. 4.75. The same process is applied to populate the collision list $\boldsymbol{C}_l$ in Eq.4.76. The main difference between these two sets is that the successful set $\boldsymbol{S}_i$ life-time is for only an iteration, because $\boldsymbol{S}_p$ is reset at the beginning of every iteration, while $\boldsymbol{C}_l$ is kept for SAPSO execution and a particle index is present in $\boldsymbol{C}_l$ until it is removed by the regeneration process.

$$\boldsymbol{S}_p \leftarrow \begin{cases} i & \text{if } f(\boldsymbol{x}_i) \leq f(\boldsymbol{x}_{best}) \\ \emptyset & \text{otherwise} \end{cases}. \tag{4.75}$$

$$\boldsymbol{C}_l \leftarrow \begin{cases} i & \text{if } f(\boldsymbol{x}_i) \leq f(\boldsymbol{x}_{best}) \wedge i \notin \boldsymbol{C}_l \\ \emptyset & \text{otherwise} \end{cases}. \tag{4.76}$$

Whenever a particle index is present at the collision list set but not in the successful particles set, it indicates that this particle has collided with the best solution, but it is not one of the best particles anymore. In this case, SAPSO then regenerates this particle by reinitializing its solution with random components in the same fashion as the initialization process.

4.3.14.2 Parameters Description for SAPSO

SAPSO is a meta-heuristic that has no **external static** parameters. It means that SAPSO does not needs to be tuned before its execution. SAPSO contains a single **internal** parameter: the **inertia weight** $w^{(k)}$ that controls change between exploratory and exploitative behaviour for SAPSO when searching for solutions and parameters.

SAPSO contains four **parameter control strategies**. It includes three **self-adaptive parameter control strategies** that encode in the meta-heuristic decision variables the parameters for social and cognitive factors c_1 and c_2 as well as the maximum speed for each particle v_{max}. It results in the use of the operations in which PSO search for optimal solutions concurrently being applied to search for optimal parameters as well. SAPSO also includes a **deterministic parameter control strategy** that configure the values for inertial weight during the algorithm execution.

4.3.15 Hybrid Discrete Particle Swarm Optimization Makespan-based (HDPSO-M)

4.3.15.1 Meta-Heuristic Description

HDPSO-M is a hybrid between a discrete PSO and a GA meta-heuristic that is adapted to specifically solve the makespan problem when mapping jobs to be executed [80] in parallel machines. It is difficult to disassociate HDPSO-M with the problem that it was designed to solve since HDPSO-M search operations are intertwined with the problem. For this reason, in this work, the goal is to use HDPSO-M only in problems that involve the task mapping of real-time

application onto multiple processor systems as present in Chapter 3 more precisely in Section 3.8. Consequently, HDPSO-M is expected to be used with a objective function $f(\boldsymbol{M}, \boldsymbol{\Omega}, \boldsymbol{\Psi})$ that evaluates a mapping of a real-time application $\boldsymbol{\Omega}$ on a MPSoC platform $\boldsymbol{\Psi}$ using the task placement $\boldsymbol{M}$.

Algorithm 19 Hybrid Discrete Particle Swarm Optimization - Makespan based (HDPSO-M)

INPUT: Objective Function ($f(\cdot)$)
Application ($\boldsymbol{\Omega}$)
Platform ($\boldsymbol{\Psi}$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Bernoulli Probability (p)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

1: **procedure** HDPSO-M
2: *Initialize* $\boldsymbol{x}_i$ $(i = 1, ..., n)$
3: *Evaluate* $f(\boldsymbol{x}_i - \mathbf{1})$
4: *Update* $\boldsymbol{x}_{pbest_i}$
5: *Update* $\boldsymbol{x}_{best}$
6: **for** $k = 1$ to $I - 1$ **do**
7: **for** $i = 1$ to n **do**
8: *Calculate* $\boldsymbol{v}_i^k$ ▷ Eq. 4.77
9: *Calculate* $\boldsymbol{x}_i^k$ ▷ Eq. 4.78
10: *Evaluate* $f(\boldsymbol{M}_i, \boldsymbol{\Omega}, \boldsymbol{\Psi})$
11: *Update* $\boldsymbol{x}_{pbest_i}$
12: **end for**
13: *Update* $\boldsymbol{x}_{best}$
14: $\boldsymbol{M}_{best} = \boldsymbol{x}_{bbest} - \bar{\mathbf{1}}$
15: **end for**
16: **return** $\boldsymbol{M}_{best}$
17: **end procedure**

HDPSO-M has a swarm $\boldsymbol{P}$ with n particles and the ith particle at every algorithm iteration has its position $\boldsymbol{x}_i^{(k)}$, speed $\boldsymbol{v}_i^{(k)}$, best individual and global positions $\boldsymbol{x}_{pbest_i}$ and $\boldsymbol{x}_{best}$ that are d-dimensional vectors with integer elements varying between $[0, d]$. The particles positions are not directly a solution for the mapping problem, namely, $\boldsymbol{M}_i$, instead, it is represented as a possible mapping summed up element-by-element by one, i. e. $\boldsymbol{x}_i = \boldsymbol{M}_i + \bar{\mathbf{1}}$ with $\bar{\mathbf{1}}$ a d-dimension vector of ones.

The process to update a particle speed and position uses Eq. 4.77 and Eq. 4.78, respectively.

$$\boldsymbol{v}_i^{(k+1)} = \boldsymbol{v}_i^{(k)} \overset{cross}{+} \left(LPTM(v_{i,pbest}) \overset{cross}{+} LPTM(v_{gbest}) \right) \tag{4.77}$$

$$\boldsymbol{x}_i^{(k+1)} = \boldsymbol{x}_i^{(k)} \overset{cross}{+} \boldsymbol{v}_i^{(k+1)}, \tag{4.78}$$

where $\overset{cross}{+}$, $\overset{o}{\times}$ and $\overset{o}{-}$ are special operators of addition, multiplication, and subtraction, to make the interpretation of movement analogous to that of the original PSO. $LPTM(\cdot)$ is a greedy algorithm that uses the tasks makespan values to find mapping solution components and will be explained with more details further. $\boldsymbol{v}_{pbest_i}$ and $\boldsymbol{v}_{gbest}$ are terms that holds the influence caused by the individual best position found by the ith particle and the global best solution and they are calculated by Eq. 4.79 and 4.80.

$$\boldsymbol{v}_{pbest} = \boldsymbol{R}_1 \overset{o}{\times} (\boldsymbol{x}_{i,pbest} \overset{o}{-} \boldsymbol{x}_i^{(k)}). \tag{4.79}$$

$$\boldsymbol{v}_{gbest} = \boldsymbol{R}_2 \overset{o}{\times} (\boldsymbol{x}_{gbest} \overset{o}{-} \boldsymbol{x}_i^{(k)}), \tag{4.80}$$

where $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ are d-dimensional random integer vectors that contains elements with 0 or 1. Both these random vectors have their components generated using a Bernoulli distribution with a probability of receiving a 1 equals to p. p is an **external static** parameter.

The subtraction operator $(\overset{o}{-})$ is a function that takes two integer vectors $\boldsymbol{A}$ and $\boldsymbol{B}$, as left and right operands, and returns another vector of integers $\boldsymbol{C}$. This operator compares element-wise the vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ calculating vector $\boldsymbol{C}$ formed by elements C_j as expressed by Eq. 4.81.

$$C_j = \begin{cases} 0 & \text{if } A_j = B_j \\ A_j & \text{otherwise} \end{cases}. \tag{4.81}$$

This operator, as used in the algorithm, compares components of a particle's position and the best individual or global positions and keeps the values for the best positions when they are different or varies them if they are the same.

The multiplication operator $(\overset{o}{\times})$ works as the Hadamard product $(\circ)$ by multiplying element-by-element the components of two vectors $\boldsymbol{A}$ and $\boldsymbol{B}$ with fixed sizes d and generating a vector with the same size $\boldsymbol{C}$ as expressed by equation 4.82.

$$C_j = A_j B_j \text{ with } j = 1, ..., d. \tag{4.82}$$

This operator adds diversity to the heuristic by multiplying the results of the subtraction operators with random vectors.

The addition operator $(\overset{cross}{+})$ is a two-point crossover genetic operator from GA-based algorithms. This crossover genetic operator uses a pair of integer vectors representing solution components and then exchange information between them as illustrated in Fig. 4.6. Since the crossover operation results in two vectors, one of them are chosen randomly to be used as a result for the addition operator, meanwhile the other one is dismissed.

Due to the encoding used by HDPSO-M, the terms $\boldsymbol{v}_{pbest_i}$ and $\boldsymbol{v}_{gbest}$ are possible task mappings offset by one in which some of their components are equal to 0 and correspond to tasks not mapped yet. HDPSO-M applies over these terms $\boldsymbol{v}_{pbest}$ and $\boldsymbol{v}_{gbest}$ a local search method $LPTM$ that uses the LPT algorithm [81] to map tasks not mapped while keeping the other tasks already mapped. The LPT (Largest Processing Time first) algorithm sorts the tasks in descend order of execution cost and assign them successively to the minimally loaded processor, i. e. the processor with the minimal sum of execution costs of the tasks mapped into it. The pseudo-code present in Algorithm 20 illustrates the functionality of the method LPTM.

4.3.15.2 Parameters Description for HDPSO-M

HDPSO-M contains an **external static** parameter p that represents the probability used in a Bernouli distribution. The suggested value for p is 0.3. p influences the rate in which new solution components are added to the components generated by $LPTM$. For better algorithm performance, p value should be configured using a parameter tuning strategy before the algorithm execution.

Algorithm 20 Largest Processing Time First based Mapping (LPTM)

INPUT: Application ($\boldsymbol{\Omega}$)
Platform ($\boldsymbol{\Psi}$)
Offset Mapping ($\boldsymbol{x}$)
OUTPUT: Possible Offset Mapping ($\boldsymbol{x}_{out}$)

1: **procedure** LPTM($\boldsymbol{x}, \boldsymbol{\Omega}, \boldsymbol{\Psi}$)
2: $\boldsymbol{T}_{unmapped\_tasks} = \{\{\boldsymbol{\tau}_j, x_j\} \in \{\boldsymbol{\Gamma}, \mathbb{N}\} : x_j \in \boldsymbol{x} \wedge x_j = 0\}$
3: *Sort* $\boldsymbol{T}_{unmapped\_tasks}$ *cost executions in descend order*
4: $\boldsymbol{T}_{mapped\_tasks} = \{\{\boldsymbol{\tau}_k, x_k\} \in \{\boldsymbol{\Gamma}, \mathbb{N}\} : x_k \in \boldsymbol{x} \wedge x_k \neq 0\}$
5: **for** $i = 1$ to $|\boldsymbol{T}_{mapped\_tasks}|$ **do**
6: $\{\boldsymbol{\tau}_k, x_k\} = \boldsymbol{T}_{mapped\_tasks_i}$
7: Map task $\boldsymbol{\tau}_k$ to core with index $x_k - 1$
8: **end for**
9: **for** $i = 1$ to $|\boldsymbol{T}_{unmapped\_tasks}|$ **do**
10: $s_{min} = MAX^*$
11: $\pi_{min} = \emptyset$
12: **for** $n = 1$ to $|\Pi|$ **do**
13: $s_n = \sum_{\tau_m \in map(\pi_n)} C_m$
14: **if** $s_n < s_{min}$ **then**
15: $s_{min} = s_n$
16: $\pi_{min} = \pi_n$
17: **end if**
18: **end for**
19: $\{\boldsymbol{\tau}_j, x_j\} = \boldsymbol{T}_{unmapped\_tasks_i}$
20: Map task $\boldsymbol{\tau}_j$ to core π_{min}
21: $x_j = index(\pi_{min}) + 1$
22: **end for**
23: $\boldsymbol{T} = \boldsymbol{T}_{mapped\_tasks} \cup \boldsymbol{T}_{unmapped\_tasks}$
24: *Sort* $\boldsymbol{T}$ *based on tasks indices.*
25: $\boldsymbol{x}_{out} \leftarrow \emptyset$
26: **for** $i = 1$ to $|\boldsymbol{T}|$ **do**
27: $\{\boldsymbol{\tau}, x\} = \boldsymbol{T}_i$
28: $\boldsymbol{x}_{out} \leftarrow x$
29: **end for**
30: **return** $\boldsymbol{x}_{out}$
31: **end procedure**

* MAX represents an arbitrary very large value.

4.4 Single-Objective Optimization Bio-Inpired Meta-Heuristics Developed in this Work

4.4.1 Single-Objective Adaptive with Modified Selection Differential Evolution (SOAMSDE)

4.4.1.1 Meta-Heuristic Description

SOAMSDE is a DE-based adaptive meta-heuristic that apart from the common framework that uses **crossover** and **mutation strategies** of a DE algorithm, SOAMSDE also contains **adaptive parameter control strategies** that throughout the algorithm execution changes its parameter values. It also has **adaptive operator selection** mechanisms to control the operators used by the algorithm. Additionally, SOAMSDE has a modified selection that is different from the one used by DE (Section 4.3.2). Instead, it uses an elitist process to select and keep the best solutions similar to that present in GA (Section 4.3.1). SOAMSDE is a single-objective version of the MONSADE (Section 4.6.2). Algorithm 21 presents the pseudo-code for SOAMSDE.

Algorithm 21 Single-Objective Adaptive with Modified Selection DE (SOAMSDE)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure SOAMSDE($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, F_i, CR_i, mut_i, cro_i \rangle$ ($i = 1, ..., n$)
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{x}_{best}$
  Update $\boldsymbol{x}_{worst}$
  $N_{mut}^{(0)} = \lceil \frac{n}{5} \rceil$
  $N_{cro}^{(0)} = \lceil \frac{n}{2} \rceil$
  $\mu_{wF}^{(0)} = \mu_{wFinitial}$
  $\mu_{wCR}^{(0)} = \mu_{wCRinitial}$
  Assign $mut_i (i = 1, ..., n)$
  Assign $cro_i (i = 1, ..., n)$
  for $k = 1$ to $I - 1$ **do**
    Calculate σ ▷ Eq. 4.61
    Calculate μ_{wF} ▷ Eq. 4.60
    Calculate μ_{wCR} ▷ Eq. 4.64
    $\boldsymbol{P}_{cnew} = \emptyset$
    for $i = 1$ to n **do**
      Generate CR_i ▷ Eq. 4.64
      Generate F_i ▷ Eq. 4.60
      $mut_i^{(k)}$*th Mutation*
      $cro_i^{(k)}$*th Crossover*
      Generate new individual $\boldsymbol{t}_{new} = \langle \boldsymbol{u}_i, F_i^{(k)}, CR_i^{(k)}, mut_i^{(k)}, cro_i^{(k)} \rangle$
      $\boldsymbol{P}_{cnew} \leftarrow \boldsymbol{t}_{new}$
    end for
    $\boldsymbol{P} =$ *Sort* $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    Remove last individuals in $\boldsymbol{P}$ *until* $|\boldsymbol{P}| = n$
    Adapt Mutation Numbers ▷ Eq. 4.52
    Adapt Crossover Numbers ▷ Eq. 4.56
    for $i = 1$ to n **do**
      Reassign $mut_i^{(k)}$
      Reassign $cro_i^{(k)}$
    end for
    Update $\boldsymbol{x}_{best}$
    Update $\boldsymbol{x}_{worst}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

SOAMSDE has a population $\boldsymbol{P_c}$ containing n individuals an each ith is a tuple $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, F_i, CR_i, mut_i, cro_i \rangle$ composed of a d-dimensional vector $\boldsymbol{x}_i$ that is a possible solution for a single-objective function f, scaling factor F_i, crossover rate CR_i, index $mut_i^{(k)}$ for one of the possible mutation strategies, and index $cro_i^{(k)}$ for one of the possible crossover strategies.

SOAMSDE **adaptive operator selection** mechanisms adjust the mutation and crossover strategies used by each individual in the same manner as CSASADE in Section 4.3.12 using the indices mut_i and cro_i as well as a pair of sets of parameter indices N_{mut} and N_{cro} **internal** parameters. However, the pool of mutation strategies are larger for SOAMSDE. $mut = \{rand/1, rand/2, current\text{-}to\text{-}best/1, current\text{-}to\text{-}best/2, best/1, best/2, tournment\text{-}based/1\}$ represents one of the mutation strategies used and each ith individual holds an index $mut_i \in [1, 7]$ for one of these strategies: (a) *rand/1* mutation (Eq. 4.45) with index $mut_i = 1$ for the ith individual and $N_{rand/1}$ is the number of individuals that uses this operator; (b) *rand/2* mutation (Eq. 4.46) with index $mut_i = 2$ for the ith individual and $N_{rand/2}$ is the number of individuals that uses this operator; (c) *current-to-best*/1 mutation (Eq. 4.47) with index $mut_i = 3$ for the ith individual

and $N_{current\text{-}to\text{-}best/1}$ is the number of individuals that uses this operator; (d) *current-to-best*/2 mutation (Eq. 4.48) with index $mut_i = 4$ for the ith individual and $N_{current\text{-}to\text{-}best/2}$ is the number of individuals that uses this operator; (e) *best*/1 mutation (Eq. 4.83) with index $mut_i = 5$ for the ith individual and $N_{best/1}$ is the number of individuals that uses this operator; (f) *best*/2 mutation (Eq. 4.49) with index $mut_i = 6$ for the ith individual and $N_{best/2}$ is the number of individuals that uses this operator; (g) *tournment-based*/1 mutation (Eq. 4.84) with index $mut_i = 7$ for the ith individual and $N_{tournment\text{-}based/1}$ is the number of individuals that uses this operator; In Equations 4.45, 4.46, 4.47, 4.48, 4.83, and 4.49, r_1, r_2, r_3, r_4, and r_5 are different random integers in $[1, n]$, $\boldsymbol{x}_{best}$ is the best solution found so far, and $\boldsymbol{v}_i$ is the mutant vector generated by one of these strategies generated for the ith individual. F_i and CR_i are, respectively, the **scaling factor** and **crossover rate** for each individual. The **adaptive operator selection** scheme used in SOAMSDE is analogous to the one present in CSASADE (Section 4.3.12).

$$\boldsymbol{v}_i = \boldsymbol{x}_{best}^{(k)} + F_i(\boldsymbol{x}_{r_1}^{(k)} - \boldsymbol{x}_{r_2}^{(k)} + \boldsymbol{x}_{r_3}^{(k)} - \boldsymbol{x}_{r_4}^{(k)}). \tag{4.83}$$

The last mutation strategy used in SOAMSDE is present in 4.5.2 works in a similar manner as the **bynary tournament selection** present in GA (Section 4.3.1). In this tournament process, a pair of different individuals are selected randomly from the population, and the individual in this pair with the best objective evaluation has its solution selected as $\boldsymbol{x}_{twinner}$. Then *tournment-based*/1 mutation strategy uses the following equation:

$$\boldsymbol{v}_i = \boldsymbol{x}_{r_1}^{(k)} + F_i \left(\boldsymbol{x}_{twinner} - \boldsymbol{x}_i^{(k)}\right), \tag{4.84}$$

where $r_1 \in [1, n]$ is a random index for an individual in the population, and F_i is the scaling factor for the ith individual.

During its initialization, SOAMSDE generates randomly uniform solutions inside of the search space. In the first iteration, the parameters $N_{mut}^{(k)}$ and $N_{cro}^{(k)}$ represents the number of individuals using each possible mutation and crossover strategies, these values are initialized as $\lceil \frac{n}{7} \rceil$ and $\lceil \frac{n}{2} \rceil$. For each ith individual their scaling factors and crossover rates are initially assigned $F_i = \mu_{wF}^0 = \mu_{wFinitial}$ and $CR_i = \mu_{wCR}^0 = \mu_{wCRinitial}$ where μ_F^k and μ_{CR}^k are two **internal** parameters adjusted by **adaptive control strategies** similar to the ones present in CSASADE.

During the iterative stage, the **adaptive operation selector** responsible to assign the mutation and crossover strategies to the individuals in the population in a circular manner similarly to how a dealer distributes a deck of cards to multiple players, instead of "chopping" the population in multiple sets and giving strategies to these sets, as used in CSASADE and APMTMODE. By assigning strategies in this manner, SOAMSDE prevents that specific operators are only assigned to the best or worst individuals since the selection of individuals to be kept in the population uses a sort method. An example of this assignment on work is present in the example where given a population of 8 individuals and in an iteration that $N_{bin}^{(k)} = 3$ and $N_{exp}^{(k)} = 7$, the assignment of crossover strategies for each individuals as follows: $cro_1^{(k)} = cro_3^{(k)} = cro_5^{(k)} = 1$ and $cro_2^{(k)} = cro_4^{(k)} = cro_6^{(k)} = cro_7^{(k)} = cro_8^{(k)} = 2$.

SOAMSDE at every iteration and for each individual in the population generates a trial-vector

$\boldsymbol{u}_i$ using a mutation and a crossover strategy with indices held by the ith individual, respectively, mut_i and cro_i. Instead of using the selection process as in Eq. 4.3, SOAMSDE uses the same selection process of GA (Section 4.3.1) by actually using trial-vectors as new solutions for new individuals that are then added to a population of new individuals $\boldsymbol{P}_{cnew}$. Then at the end of an iteration, SOAMSDE joins both populations $\boldsymbol{P}_c$ and $\boldsymbol{P}_{cnew}$ sort their individuals based on their evaluation values in ascending order and then iteratively removes the last individuals until this new population has size n and substitute $\boldsymbol{P}_c$ for the next iteration. Each generated individual have their parameter values copied from their original i-th individual, it includes $F_{new} = F_i$, $CR_{new} = CR_i$, $mut_{new} = mut_i$, and $cro_{new} = cro_i$.

For each ith individual in the population its scaling factors F_i is calculated as follows:

$$F_i^{(k)} = \min(1, \max(0, \mathcal{N}(\mu_{w_F}^{(k)}, \sigma))), \tag{4.85}$$

where $\mu_{w_F}^{(k)}$ is an **internal** parameter that represents the weighted averages for scaling factors and its value is adjusted using an adaptive parameter control strategy shown in Eq. 4.62 in Section 4.3.12, and σ is an **internal** parameter for the standard deviation controlled by a **deterministic parameter control strategy** similar to the one that controls another σ in CSASADE (4.3.12).

In a similar approach, SOAMSDE updates the crossover rates CR_i using the following equation:

$$\begin{aligned} \nu &= \mathcal{C}(\mu_{w_{CR}}^{(k)}, \sigma) \\ CR_i^{(k)} &= \begin{cases} \nu & \text{if } \nu \in (0,1), \\ 1 & \text{otherwise} \end{cases} \end{aligned} \tag{4.86}$$

where $\mu_{w_{CR}}^{(k)}$ is another **internal** parameter representing weighted averages for crossover rates, and it is adjusted by an **adaptive parameter control strategy** shown in Eq. 4.65 in Section 4.3.12.

The number of individuals using each mutation and crossover strategies, respectively, $N_{mut}^{(k)}$ and $N_{cro}^{(k)}$, changes in the same manner as CSASADE using Eq. 4.52 and Eq. 4.56.

4.4.1.2 Parameters Description for SOAMSDE

SOAMSDE is a complex DE-based adaptive meta-heuristic with seven **parameter control strategies** used inspired by the CSASADE (Section 4.3.12). However, SOAMSDE has the following novelties when compared to CSASADE: (a) a larger pool of mutation strategies including a tournament mutation strategy based on [82]; (b) SOAMSDE leaves the selection strategy to an elitist process similar to the one present in GA that in turn alters the type of pressure imposed by the algorithm to select good solutions influencing the speed of convergence; (c) **Adaptive operation selection** schemes present in SOAMSDE circularly assign mutation and crossover strategies to the population in a way that resembles distributing cards to different players in a card game.

SOAMSDE contains no **external** parameters and does not need to be fined tuned by a user. However, SOAMSDE contains thirteen **internal** parameters analogous to CSASADE, listed as

follows: (a) F_i, (b) μ_{wF}, (c) CR_i, (d) μ_{wCR}, (e) σ, (f) $\mu_{wFinitial}$, (g) $\mu_{wCRinitial}$, (h) σ_{max}, (i) σ_{min}, (j) N_{mut}, (k) N_{cro}, (l) mut_i, and (m) cro_i.

Similar to CSASADE, two **adaptive parameter control strategies** work in an individual-scope to update the values for scaling factors F_i and crossover rates CR_i for each ith individual in the population based on the **internal parameters** representing weighted averages μ_{wF} and μ_{wCR} that are in turn controlled by other two **adaptive parameter control strategies**. σ has its value controlled by a **deterministic parameter control** that independently of the algorithm performance reduces the variation of generated scaling factors and crossover rates.

Internal parameters N_{mut} and N_{cro} update the number of individuals using specific mutation and crossover strategies in a population-scope. The values assumed by these parameters are controlled by a pair of **adaptive parameter control strategies** that use the difference in evaluation values of individuals using specific operators. These differences are then used to increase the number of individuals using operators that shows to have improvements while reducing the number of individuals using operators with poor performance. N_{mut} and N_{cro} are used by two **adaptive operator selection** that alters the values for indices mut_i and cro_i in a individual-scope. These adaptive mechanisms use the own selection process to reduce the number of strategies that obtain "bad" solutions and increase or maintain the number of strategies the obtain "good" solutions.

Apart from the **internal** parameters influenced by adaptive schemes, SOAMSDE uses **static internal parameters** that controls some characteristics of the adaptive techniques. These static parameters are: (a) the initial weighted averages $\mu_{wCRinitial}$ and $\mu_{wCRinitial}$ that sets the initial points for the parameter search with both of them set 0.5; (b) σ_{max} and σ_{min} represents the maximum values of σ assumes during the first iterations and then over the last iteration.

4.4.2 Adaptive Genetic Algorithm v1 (AGAv1)

4.4.2.1 Meta-Heuristic Description

AGAv1 is a GA-based meta-heuristic that have **adaptive parameter control strategies** inspired by the ones present in JADE. Algorithm 26 shows AGAv1 in pseudo-code form.

AGAv1 has a population $\boldsymbol{P}$ formed by n individuals. Each ith individual is represented as a tuple $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, p_{ci}, p_{mi} \rangle$ composed of a d-dimensional integer vector $\boldsymbol{x}_i$ as well as the **crossover** and **mutation rates** p_{ci} and p_{mi} that were used by the genetic operators to generate the vector $\boldsymbol{x}_i$.

AGAv1 initialization process randomly generates solutions for a COP with objective function f, analogous to GA. However, since AGAv1 contain adaptive techniques, it also initialize its **internal** parameters related to the adaptive mechanisms, namely, $p_{ci} = \mu_{p_c}^{(0)} = \mu_{p_c initial}$ and $p_{mi} = \mu_{p_m}^{(0)} = \mu_{p_c initial}$ for each ith individual.

During every iteration, AGAv1 generates a population of offsprings $\boldsymbol{P}_{new}$, and before the selection operator application, the algorithm creates for the new offspring individuals their **mutation**

Adaptive Genetic Algorithm v1 (AGAv1)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure AGAv1($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i,\ p_{mi},\ p_{ci} \rangle$
  Evaluate $f(\boldsymbol{x}_i)$
  Update $\boldsymbol{x}_{best}$
  for $k = 1$ to $I - 1$ **do**
    $\mathbf{P}_{new} = \emptyset$
    for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
      Generate p_{mnew} ▷ Eq. 4.87
      Generate p_{cnew} ▷ Eq. 4.89
      Binary Tournament Selection Operator
      One-Point Crossover Operator using p_{cnew}
      Mutation Operator using p_{mnew}
      Evaluate offsprings $\boldsymbol{x}_{new1}$ *and* $\boldsymbol{x}_{new2}$
      $\boldsymbol{t}_{new1} = \langle \boldsymbol{x}_{new1}, p_{mnew}, p_{cnew} \rangle$
      $\boldsymbol{t}_{new2} = \langle \boldsymbol{x}_{new2}, p_{mnew}, p_{cnew} \rangle$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new1}$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new2}$
    end for
    Sort $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    $\boldsymbol{P}$ *is replaced by Best* n *in sorted* $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    Update $\mu_{p_m}^{(k)}$ ▷ Eq. 4.88
    Update $\mu_{p_c}^{(k)}$ ▷ Eq. 4.90
    Update $\boldsymbol{x}_{best}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

and **crossover rates**, respectively, p_{mnew} and p_{cnew}.

p_{mnew} is generated using the following equation that defines an **adaptive parameter control strategy**:

$$p_{mnew} = \min\left(p_{mmax}, \max\left(p_{mmin}, \mathcal{N}(\mu_{p_m}^{(k)}, \sigma_{p_m})\right)\right), \tag{4.87}$$

where p_{mmax} and p_{mmin} are **internal static** parameters that defines the upper and lower boundaries for the mutation and crossover probabilities. σ_{p_m} is another **internal static** parameter that controls the spread of values for mutation probabilities. $\mu_{p_m}^{(k)}$ is an **internal** parameter configured by an **adaptive parameter control** as following:

$$\mu_{p_m}^{(k+1)} = \min\left(p_{mmax}, \max\left(p_{mmin}, (1-c)\mu_{p_m}^{(k)} + c\left(\frac{\sum_{i=1}^{n} p_{mi}}{n}\right)\right)\right), \tag{4.88}$$

where c is an **internal static** parameter that, analogous to JADE, controls the adaptation speed.

p_{cnew} is generated by a very similar **adaptive parameter control** as the one that generates p_{cnew}, as expressed in Eq. 4.89 that uses a second **internal** parameter $\mu_{p_c}^{(k)}$ controlled by another adaptive mechanism as well as using other **internal static** parameters, namely, p_{cmax}, p_{cmin} that work similarly to their counterparts for mutation, and the adaptation speed parameter c as shown in Eq. 4.88.

$$p_{cnew} = \min\left(p_{cmax}, \max\left(p_{cmin}, \mathcal{N}(\mu_{p_c}^{(k)}, \sigma_{p_c})\right)\right), \tag{4.89}$$

$$\mu_{p_c}^{(k+1)} = \min\left(p_{cmax}, \max\left(p_{cmin}, (1-c)\mu_{p_c}^{(k)} + c\left(\frac{\sum_{i=1}^{n} p_{ci}}{n}\right)\right)\right), \tag{4.90}$$

After the generation of $p_{m_{new}}$ and $p_{c_{new}}$, these parameter are used to generate the new chromosomes $\boldsymbol{x}_{new1}$ and $\boldsymbol{x}_{new2}$. These two chromosomes are then joined with $p_{m_{new}}$ and $p_{c_{new}}$ into two new offsprings $\boldsymbol{t}_{new1} = \langle \boldsymbol{x}_{new1}, p_{m_{new}}, p_{c_{new}} \rangle$ and $\boldsymbol{t}_{new2} = \langle \boldsymbol{x}_{new2}, p_{m_{new}}, p_{c_{new}} \rangle$ that are then added to the new population $\boldsymbol{P}_{new}$. The same elitist process present in GA is used to keep the population with the best solutions found so far.

4.4.2.2 Parameters Description for AGAv1

AGAv1 contains four **adaptive parameter control** strategies. Two of them work in an individual-scope controlling the parameter values for p_m and p_c for each ith individual of the population $\boldsymbol{P}$. The other two work in a population-scope controlling parameters $\mu_{p_m}^{(k)}$ and $\mu_{p_c}^{(k)}$ that in turn controls the adaptive mechanisms that generates by p_m and p_c values. All of these **adaptive parameter control strategies** use as evidence for change the improvement in the performance of the algorithm through the acquisition of better solution with smaller evaluation values for the objective function f since the elitist process keeps better solutions and consequently altering the average of parameter values.

AGAv1 contains thirteen **internal** parameters that influence the exploratory and exploitation behavior for the search of possible parameters and in turn, these parameter controls the meta-heuristic as a whole. These **internal** parameters are: (a) crossover probability p_{ci} for each individual, (b) mutation probability p_{mi} for each individual, (c) average crossover rate $\mu_{p_c}^{(k)}$ at iteration k, (d) average crossover rate $\mu_{p_c}^{(k)}$ at iteration k, (e) initial average crossover rate $\mu_{p_{c\,initial}} = 0.7$, (f) crossover rate standard deviation $\sigma_{p_c} = 0.25$, (g) minimum crossover rate $p_{cmin} = 0.1$, (h) maximum crossover rate $p_{cmax} = 0.95$, (i) average mutation rate $\mu_{p_m}^{(k)}$ at iteration k, (j) mutation rate standard deviation $\sigma_{p_m} = 0.025$, (k) minimum mutation rate $p_{mmin} = 0.01$, (l) maximum mutation rate $p_{mmax} = 0.1$, and (m) adaptation rate $c = 0.15$. The parameters that are set represent **static** ones.

4.4.3 Adaptive Genetic Algorithm v2 (AGAv2)

4.4.3.1 Meta-Heuristic Description

AGAv2 is the second GA-based adaptive meta-heuristic to optimize SOOP developed in this work. AGAv2 contains some similarities with AGAv1. However, when compared with AGAv1, the **adaptive parameter control strategies** developed for AGAv2 permit faster changes in the averages of mutation and crossover rates during the algorithm execution. Therefore, faster exploration for the mutation and crossover probabilities used by the **mutation** and **crossover** genetic operators.

Another additional feature in AGAv2 is an **adaptive parameter control strategy** that uses as evidence for change a success metric to configure standard deviation values for the generation of

mutation and **crossover rates**. These adaptation schemes have a population-scope since these standard deviations are used to generate parameter for every individual. Algorithm 22 shows AGAv2 is pseudo-code form.

Algorithm 22 Adaptive GA v2 (AGAv2)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure AGAv2($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i,\ p_{mi},\ p_{ci} \rangle$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{x}_{best}$
  for $k = 1$ to $I - 1$ **do**
    $\boldsymbol{A}_f$ = *evaluations of* $\boldsymbol{P}$
    $\boldsymbol{P}_{new} = \emptyset$
    for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
      Generate p_{mnew} ▷ Eq. 4.91
      Generate p_{cnew} ▷ Eq. 4.92
      Binary Tournament Selection Operator
      One-Point Crossover Operator using p_{cnew}
      Mutation Operator using p_{mnew}
      Evaluate offsprings $\boldsymbol{M}_{new1}$ *and* $\boldsymbol{M}_{new2}$
      $\boldsymbol{t}_{new1} = \langle \boldsymbol{x}_{new1}, p_{mnew}, p_{cnew} \rangle$
      $\boldsymbol{t}_{new2} = \langle \boldsymbol{x}_{new2}, p_{mnew}, p_{cnew} \rangle$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new1}$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new2}$
    end for
    Sort $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    $\boldsymbol{P}$ *is replaced by Best* n *in sorted* $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    $\boldsymbol{A}_{f_{new}}$ = *evaluations of* $\boldsymbol{P}$
    Count success s *comparing* $\boldsymbol{A}_{f_{new}}$ *and* $\boldsymbol{A}_f$
    Update $\sigma_{p_m}^{(k+1)}$ ▷ Eq. 4.94
    Update $\sigma_{p_c}^{(k+1)}$ ▷ Eq. 4.93
    Update $\mu_{p_m}^{(k+1)}$ ▷ Eq. 4.95
    Update $\mu_{p_c}^{(k+1)}$ ▷ Eq. 4.96
    Update $\boldsymbol{x}_{best}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

AGAv2 is composed of a population $\boldsymbol{P}$ with of n tuples and each ith tuple is represented as $t_i = \langle \boldsymbol{x}_i, p_{ci}, p_{mi} \rangle$. These tuples represent the individuals in AGAv2.

The parameters p_{mnew} and p_{mnew} are generated for the new offspring individuals using Eq. 4.91 and Eq. 4.92. Note that even though the process is analogous to that present in AGAv1, the **internal parameters** that represent the standard deviations σ_{p_m} and σ_{p_c} are controlled by a pair of **deterministic parameter control strategies**.

$$p_{mnew} = \min\left(p_{mmax}, \max\left(p_{mmin}, \mathcal{N}(\mu_{p_m}^{(k)}, \sigma_{p_m}^{(k)})\right)\right). \tag{4.91}$$

$$p_{cnew} = \min\left(p_{cmax}, \max\left(p_{cmin}, \mathcal{N}(\mu_{p_c}^{(k)}, \sigma_{p_c}^{(k)})\right)\right). \tag{4.92}$$

During every iteration, the processes to update the standard deviation values $\sigma_{p_c}^{(k)}$ and $\sigma_{p_m}^{(k)}$ use

sets containing copies of the objective evaluation results from the parent population at the beginning of the iteration $\boldsymbol{A}_f$ and the population after the elitist process has been applied by AGAv2 $\boldsymbol{A}_{f_{new}}$. Set $\boldsymbol{A}_f$ is populated and sorted from the best to worst evaluations at the beginning of an iteration k. After the genetic operators are applied, including the elitist process, the evaluation results for the new population composed of old and new individuals are copied and sorted as a set $\boldsymbol{A}_{f_{new}}$. $\boldsymbol{A}_{new}$ is then compared against $\boldsymbol{A}_f$ in an element-by-element basis generating a success value s that is computed as the number of times in which there was an improvement in evaluation obtained from an iteration to another between two individuals ranked in the same ith position in the population.

AGAv2 then uses s to indicate the improvement of the population solutions as a whole between two iterations. This metric is used to decide whether the algorithm should explore more the search space of parameters or exploit more the parameter values found so far. This change between exploration and exploitation is done by increasing or decreasing the standard deviations values $\sigma_{p_c}^{(k)}$ and $\sigma_{p_m}^{k}$ using Eq. 4.93 and Eq. 4.94.

$$\nu_c = \begin{cases} a\sigma_{p_c}^{(k)} & \text{if } s > \frac{n}{4} \\ a^{-1}\sigma_{p_c}^{(k)} & \text{otherwise}. \end{cases} \tag{4.93}$$
$$\sigma_{p_c}^{(k+1)} = \min\left(\sigma_{p_c max}, \max\left(\sigma_{p_c min}, \nu_c\right)\right)$$

$$\nu_m = \begin{cases} a\sigma_{p_m}^{(k)} & \text{if } s > \frac{n}{4} \\ a^{-1}\sigma_{p_m}^{(k)} & \text{otherwise}. \end{cases} \tag{4.94}$$
$$\sigma_{p_m}^{(k+1)} = \min(\sigma_{p_m max}, \max(\sigma_{p_c min}, \nu_m))$$

The **internal** parameters $\mu_{p_m}^{(k)}$ and $\mu_{p_c}^{(k)}$ are updated directly as the arithmetic averages for the mutation and crossover rates in each individual that were kept by the elitist process, as if $c = 1$ in AGAv1, as present in Eq. 4.95 and Eq. 4.96.

$$\mu_{p_m}^{(k+1)} = \min\left(p_{m max}, \max\left(p_{m min}, \frac{\sum_{j=1}^{n} p_{m j}}{n}\right)\right). \tag{4.95}$$

$$\mu_{p_c}^{(k+1)} = \min\left(p_{c max}, \max\left(p_{c min}, \frac{\sum_{j=1}^{n} p_{c j}}{n}\right)\right). \tag{4.96}$$

4.4.3.2 Parameters Description for AGAv2

AGAv2 contains six **parameter control strategies**. The first four are **adaptive parameter control strategies** that work in a similar fashion as the ones present in AGAv1 for the parameters p_{m_i}, p_{c_i}, $\mu_{p_m}^{(k)}$, and $\mu_{p_c}^{(k)}$. However, AGAv2 also contains two **deterministic parameter control strategies** that adjust the values of the internal parameters σ_{p_m} and σ_{p_c} based on the rate of success, i.e., improvement of individuals, during consecutive iterations.

AGAv2 contains nineteen **internal** parameters listed as follows: (a) crossover probability p_{ci} for each individual, (b) mutation probability p_{mi} for each individual, (c) average crossover rate $\mu_{p_c}^{(k)}$ at iteration k, (d) initial average crossover rate $\mu_{p_c initial} = 0.8$, (e) average mutation rate $\mu_{p_m}^{(k)}$, (f) initial average mutation rate $\mu_{p_m initial} = 0.01$, (g) crossover rate standard deviation $\sigma_{p_c}^{(k)}$, (h) initial crossover rate standard deviation $\sigma_{p_c initial} = 0.1$, (i) mutation rate standard deviation $\sigma_{p_m}^{(k)}$, (j) initial mutation rate standard deviation $\sigma_{p_m initial} = 0.005$, (k) minimum crossover rate $p_{cmin} = 0.1$, (l) maximum crossover rate $p_{cmax} = 0.95$, (m) minimum mutation rate $p_{mmin} = 0.01$, (n) maximum mutation rate $p_{mmax} = 0.1$, (o) minimum crossover rate standard deviation $\sigma_{p_c min} = 0.01$, (p) maximum crossover rate standard deviation $\sigma_{p_c max} = 1$, (q) minimum mutation rate standard deviation $\sigma_{p_m min} = 0.001$, (r) maximum mutation rate standard deviation $\sigma_{p_m max} = 0.5$, and (s) standard deviation adaptation rate $a = 1.2$. Since from these parameters, all thirteen of them are **static** ones, the parameter tuning for AGAv2 internal parameters is its major disadvantage.

4.4.4 Adaptive Genetic Algorithm v3 (AGAv3)

4.4.4.1 Meta-Heuristic Description

AGAv3 is the third GA-based adaptive meta-heuristic developed in this work. It combines adaptive mechanisms featured in AGAv2 and AGAv3 with schemes present in CSASADE (Section 4.3.12). For this reason, AGAv3 has **parameter control strategies** and **adaptive operator selection** techniques. Algorithm 23 illustrates AGAv3 in pseudo-code.

Algorithm 23 Adaptive GA v3 (AGAv3)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure AGAv3($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, p_{mi}, p_{ci}, cro_i, mut_i \rangle$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{x}_{best}$
  Update $\boldsymbol{x}_{worst}$
  for $k = 1$ to $I - 1$ **do**
    Calculate σ_{p_m} ▷ Eq. 4.109
    Calculate σ_{p_x} ▷ Eq. 4.110
    Calculate μ_{wp_m} ▷ Eq. 4.106
    Calculate μ_{wp_c} ▷ Eq. 4.106
    Calculate p_{mi} ▷ Eq. 4.104
    Calculate p_{ci} ▷ Eq. 4.105
    $\boldsymbol{P}_{new} = \emptyset$
    for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
      Binary Tournament Selection Operator
      if $f(\boldsymbol{x}_{p_1}) < f(\boldsymbol{x}_{p_2})$ **then**
        $p_c = p_{c_{p_1}}$
        $p_m = p_{m_{p_1}}$
        $cro = cro_{p_1}$
        $mut = mut_{p_1}$
      else
        $p_c = p_{c_{p_2}}$
        $p_m = p_{m_{p_2}}$
        $cro = cro_{p_2}$
        $mut = mut_{p_2}$
      end if
      cro th Crossover Operator using p_c
      mut th Mutation Operator using p_m
      Evaluate offsprings $\boldsymbol{x}_{new1}$ *and* $\boldsymbol{x}_{new2}$
      $\boldsymbol{t}_{new1} = \langle \boldsymbol{x}_{new1}, p_m, p_c, cro, mut \rangle$
      $\boldsymbol{t}_{new2} = \langle \boldsymbol{x}_{new2}, p_m, p_c, cro, mut \rangle$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new1}$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new2}$
    end for
    Sort $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    $\boldsymbol{P}$ *is replaced by Best* n *in sorted* $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    Adapt Mutation Numbers ▷ Eq. 4.52
    Adapt Crossover Numbers ▷ Eq. 4.56
    for $i = 1$ to n **do**
      Reassign $mut_i^{(k)}$
      Reassign $cro_i^{(k)}$
    end for
    Update $\boldsymbol{x}_{best}$
    Update $\boldsymbol{x}_{worst}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

AGAv3 has a population $\boldsymbol{P}$ composed of n individuals that are represented as tuples. Each ith individual is a tuple $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, p_{ci}, p_{mi}, cro_i, mut_i \rangle$, where $\boldsymbol{x}_i$ is d-dimensional integer vector that is a solution for a COP with objective f, p_{ci} is the mutation rate used to generate this individual solution, p_{mi} is the crossover rate is used to generate $\boldsymbol{x}_i$, cro_i is the index for one of the operators present in the pool of genetic crossover operators, and mut_i is the index for one of the genetic mutation operators.

AGAv3 has a pool of possible crossover operators represented as $cro = \{one\text{-}point, sbx, two\text{-}point, uniform\}$ and a set of **internal** parameters $N_{cro}^{(k)}$ that represents the number of individuals using one of the crossovers. For each ith individual cro_i can assume

integer values in the range $[1, 4]$ representing the following list of crossover strategies: (a) *one-point* crossover with index $cro_i = 1$ for individual i and $N_{one\text{-}point}$ is the number of individuals that uses this operator; (b) *sbx* (simulated binary) crossover with index $cro_i = 2$ for individual i and N_{sbx} is the number of individuals that uses this operator; (c) *two-point* crossover with index $cro_i = 3$ for individual i and $N_{two\text{-}point}$ is the number of individuals that uses this operator; (d) *uniform* crossover with index $cro_i = 4$ for individual i and $N_{uniform}$ is the number of individuals that uses this operator;

The one-point crossover genetic operator functionality is explained in the GA meta-heuristic present in Section 4.3.1. The two-point crossover is similar to the one-point, however, it uses two random crossover points, respectively, r_{d1} and r_{d2}, in the parent solutions $\boldsymbol{x}_{p_1}$ and $\boldsymbol{x}_{p_2}$ and then exchange genetic material between these crossover points as illustrated in Fig. 4.6. Meanwhile, the uniform crossover operator does a random test for each chromosome and, if this test is passed, that chromosome is swapped between both parents generating two offsprings $\boldsymbol{x}_{o1}$ and $\boldsymbol{x}_{o2}$. Figure 4.7 illustrates an application of a uniform crossover where r_j is a random value to be used by the jth solution component.

A simulated binary crossover, as present in [83], uses Eq. 4.97, Eq. 4.98 and Eq. 4.99 to transfer genetic material between two parent solutions $\boldsymbol{x}_{p_1}$ and $\boldsymbol{x}_{p_2}$ generating two offspring solutions $\boldsymbol{x}_{o1}$ and $\boldsymbol{x}_{o2}$, where r_1 and r_2 are random real numbers in $[0, 1]$.

$$\beta = \begin{cases} (2r_1)^{\left(\frac{1}{d+1}\right)} & \text{if } r_1 > 0.5 \\ (2(1-r_1))^{\left(\frac{1}{d+1}\right)} & \text{otherwise} \end{cases}. \tag{4.97}$$

$$x_{o1_j} = \begin{cases} \left(\max\left(lb_j, \min\left(ub_j, \left\lfloor 0.5 + 0.5\left((1+\beta)x_{p1_j} + (1-\beta)x_{p2_j}\right\rfloor\right)\right)\right)\right) & \text{if } r_2 < p_c \\ x_{p1_j} & \text{otherwise} \end{cases}. \tag{4.98}$$

$$x_{o2_j} = \begin{cases} \left(\max\left(lb_j, \min\left(ub_j, \left\lfloor 0.5 + 0.5\left((1+\beta)x_{p2_j} + (1-\beta)x_{p1_j}\right\rfloor\right)\right)\right)\right) & \text{if } r_2 < p_c \\ x_{p2_j} & \text{otherwise} \end{cases}. \tag{4.99}$$

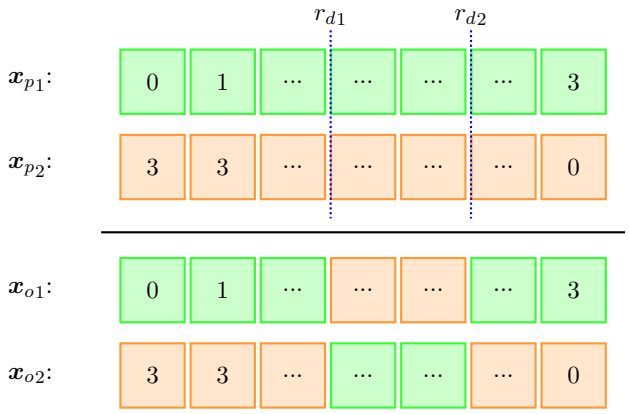


Figura 4.6 – Graphical representation of a two-point crossover operator.

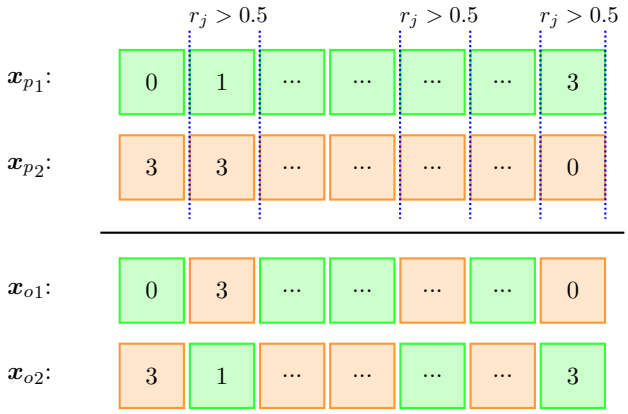


Figura 4.7 – Graphical representation of a uniform crossover operator.

Similar to crossover operators, AGAv3 has a pool for mutation genetic operators $mut = \{bnormalr, cauchyr, normalr, poly, unifr\}$ and a set of **internal** parameters $N_{mut}^{(k)}$ that represents the number of individuals using one of the mutation operators. Indices mut_i can assume integer values $[1, 5]$ representing of the possible following mutation strategies: (a) *bnormalr* represents a biased random gene flip mutation that uses a normal distribution. The ith individual has index $mut_i = 1$, and $N_{bnormalr}$ is the number of individuals that uses this operator; (b) *cauchyr* represents a random gene flip mutation that uses a Cauchy distribution. The ith individual has index $mut_i = 2$, and $N_{cauchyr}$ is the number of individuals that uses this operator; (c) *normalr* is a random gene flip mutation that uses a normal distribution. The ith individual has index $mut_i = 3$, and $N_{normalr}$ is the number of individuals that uses this operator; (d) *poly* is a polynomial mutation. The ith individual has index $mut_i = 4$, and N_{poly} is the number of individuals that uses this operator; (e) *unifr* is a random gene flip mutation the same used by GA in Section 4.3.1. The ith individual has index $mut_i = 5$, and N_{unifr} is the number of individuals that uses this operator;

The biased gene flip mutation operator generates new chromosomes values that use its old value as the mean in a normal distribution with a standard deviation of 10% of the mean between the boundaries of the search space whenever a random test passes with p_m for every component. In other words, given a solution $\boldsymbol{x}$, it generates a new value for the jth chromosome x_{newj} as follows:

$$x_{newj} = \left\lfloor \min\left(ub_j, \max\left(lb_j, \mathcal{N}(x_j, \frac{ub_j - lb_j}{10})\right)\right)\right\rfloor. \tag{4.100}$$

AGAv3 uses two new types of random gene flip mutation: (a) *cauchyr* generates new chromosomes x_{newj} using a Cauchy distribution as follows:

$$x_{newj} = \left\lfloor \min\left(ub_j, \max\left(lb_j, \mathcal{C}\left(lb_j + \frac{ub_j - lb_j}{2}, \frac{ub_j - lb_j}{10}\right)\right)\right) + 0.5\right\rfloor. \tag{4.101}$$

(b) *cauchyr* generates new chromosomes x_{newj} using a Cauchy distribution as follows:

$$x_{newj} = \left\lfloor \min\left(ub_j, \max\left(lb_j, \mathcal{N}\left(lb_j + \frac{ub_j - lb_j}{2}, \frac{ub_j - lb_j}{10}\right)\right)\right) + 0.5\right\rfloor. \tag{4.102}$$

In mutations operators *bnormalr*, *normalr*, and *cauchyr*, the generated values need to be truncated back to the search space, since both distribution may generate values outside of the search space, and the results also need to be rounded to the closest integer number because solutions in AGAv3 are d-dimensional integer vectors.

The polynomial mutation (*poly*) generates a new chromosome $x_{j_{new}}$ given the previous chromosome x_j. This process uses the following equation:

$$x_{j_{new}} = \begin{cases} \max\left(lb_j, \min\left(ub_j, \left\lfloor x_j + (2r)^{\left(\frac{1}{d+1}\right)} - 0.5 \right\rfloor\right)\right) & \text{if } r < 0.5 \\ \max\left(lb_j, \min\left(ub_j, \left\lfloor x_j + (2(1-r))^{\left(\frac{1}{d+1}\right)} - 0.5 \right\rfloor\right)\right) & \text{otherwise} \end{cases}, \tag{4.103}$$

where r is a random real number $r \in [0, 1]$.

The **adaptive operator selection** scheme in AGAv4 assigns each individual its mutation and crossover operator in the same fashion as SOAMSDE (Section 4.4.1), in other words, in a circular, depending on the values of $N_{cro}^{(k)}$ and $N_{mut}^{(k)}$.

During AGAv3 initialization, random solutions for f are generated inside of the search space and assigned as chromosomes to each of the individuals in AGAv3. The parameters for each individual are initialized as in CSASADE and SOAMSDE, but instead of scaling factors and crossover rates, the parameters are related to crossover probabilities and mutation probabilities. This process assigns to each individual i $p_{mi} = \mu_{w_{p_m}}^0 = \mu_{w_{p_m initial}}$ and $p_{ci} = \mu_{w_{p_c}}^0 = \mu_{w_{p_c initial}}$ where μ_p^k and $\mu_{p_c}^k$ are two **internal** parameters adjusted by **adaptive control strategies**.

Another stage of the initialization is to assign to each genetic operator equal number of individuals in such a way that $N_{one\text{-}point} = N_{sbx} = N_{two\text{-}point} = N_{uniform} = \lfloor \frac{n}{4} \rfloor$ and $N_{bnormalr} = N_{cauchyr} = N_{normalr} = N_{poly} = N_{unifr} = \lfloor \frac{n}{5} \rfloor$.

During I iterations, two **adaptive parameter control strategies** dynamically change the values for crossover and mutation probabilities of the population in an individual-scope by generating new mutation and crossover rates using Eq. 4.104 and Eq. 4.105.

$$p_{mi} = \min\left(p_{m_{max}}, \max\left(p_{m_{min}}, \mathcal{N}\left(\mu_{w_{p_m}}^{(k)}, \sigma_{p_m}\right)\right)\right). \tag{4.104}$$

$$p_{ci} = \min\left(p_{c_{max}}, \max\left(p_{c_{min}}, \mathcal{N}\left(\mu_{w_{p_c}}^{(k)}, \sigma_{p_c}\right)\right)\right). \tag{4.105}$$

These adaptive parameter control schemes use weighted averages to generate new parameter values, namely $\mu_{w_{p_m}}$ and $\mu_{w_{p_c}}$. Both averages are adjusted by others **adaptive parameter control strategies** calculated using Eq. 4.106 and Eq. 4.107 with the normalized weight for each ith individual calculated using Eq. 4.108.

$$\mu_{w_{p_m}}^{(k)} = \sum_{i=1}^{n} w_i p_{mi}. \tag{4.106}$$

$$\mu_{w_{p_c}}^{(k)} = \sum_{i=1}^{n} w_i p_{mi}. \tag{4.107}$$

$$w_i = \frac{|f(\boldsymbol{x}_i) - f(\boldsymbol{x}_{worst})|}{\sum_{j=1}^{n} |f(\boldsymbol{x}_j) - f(\boldsymbol{x}_{worst})|}. \tag{4.108}$$

The values for standard deviations for both mutation and crossover probabilities are expressed as **internal** parameters σ_{p_m} and σ_{p_c}. Both of them are updated using their own **deterministic parameter control strategies** that uses Eq. 4.109 and Eq. 4.110.

$$\sigma_{p_m} = \sigma_{p_m min} + (\sigma_{p_m max} - \sigma_{p_m min}) \left(1 - \left(\frac{k}{I}\right)^2\right). \tag{4.109}$$

$$\sigma_{p_c} = \sigma_{p_c min} + (\sigma_{p_c max} - \sigma_{p_c min}) \left(1 - \left(\frac{k}{I}\right)^2\right). \tag{4.110}$$

During AGAv3 reproductive stage, when generating offspring individuals, the parent individuals are selected using a **binary tournament selection** as GA (Section 4.3.1). In this manner, given two parents selected from the population with indices p_1 and p_2, the crossover and mutation genetic operators and parameter values used to generate offsprings between these parents are obtained from the parent with the best evaluation of the objective function f. Both new offspring solutions inherit their parameters from the parent with the best evaluation results.

4.4.4.2 Parameters Description for AGAv3

AGAv3 has no **external static** parameters. However, it contains *nineeen* internal parameters listed as follows: (a) mutation probability for the ith individual p_{mi}, (b) crossover probability for the ith individual p_{ci}, (c) average crossover probability $\mu_{p_c}^{(k)}$ at iteration k, (d) initial average crossover probability $\mu_{p_c initial} = 0.7$, (e) average mutation probability $\mu_{p_m}^{(k)}$, (f) initial average mutation probability $\mu_{p_m initial} = 0.05$, (g) minimum crossover probability $p_{c min} = 0.1$, (h) maximum crossover probability $p_{c max} = 0.95$, (i) minimum mutation probability $p_{m min} = 0.01$, (j) maximum mutation probability $p_{m max} = 0.1$, (l) minimum crossover probability standard deviation $\sigma_{p_c min} = 0.1$, (m) maximum crossover probability standard deviation $\sigma_{p_c max} = 0.4$, (n) minimum mutation probability standard deviation $\sigma_{p_m min} = 0.01$, (o) maximum mutation probability standard deviation $\sigma_{p_m max} = 0.05$. (p) number of individuals using a specific mutation operator N_{mut}, (q) number of individuals using a specific crossover operator N_{cro}, (r) index for the mutation operator used by individual i mut_i, and (s) index for the mutation operator used by individual i cro_i. All parameters in this list that are set are **static** ones. There are ten one of them in AGAv3.

Similar to SOAMSDE, AGAv3 has two **adaptive operation selection** mechanisms that controls the use of operators by the individuals based on the values of the **internal** parameters N_{mut} and N_{cro} in a similar fashion as SOAMSDE.

AGAv3 contains eight **parameter control strategies**. Six of them are **adaptive parameter control strategies** to control the internal parameters p_{mi}, p_{ci}, $\mu_{p_m}^{(k)}$, $\mu_{p_c}^{(k)}$, N_{mut} and N_{cro}. Finally, two of these adaptive mechanisms are **deterministic parameter control strategies** that adjust the values of the internal parameters σ_{p_m} and σ_{p_c}.

4.4.5 Adaptive Genetic Algorithm v4 (AGAv4)

4.4.5.1 Meta-Heuristic Description

AGAv4 is an adaptive meta-heuristic that is an expansion of AGAv3 that adds an **adaptive parameter control strategy** to configure the size of the tournament selection in a population-scope as present in [84]. Algorithm 24 shows AGAv4 in pseudo-code.

Algorithm 24 Adaptive GA v4 (AGAv4)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure AGAv4($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, p_{mi}, p_{ci} cro_i, mut_i, p_{ti} \rangle$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{x}_{best}$
  Update $\boldsymbol{x}_{worst}$
  for $k = 1$ to $I - 1$ **do**
    Calculate σ_{p_m} ▷ Eq. 4.109
    Calculate σ_{p_x} ▷ Eq. 4.110
    Calculate μ_{wp_m} ▷ Eq. 4.106
    Calculate μ_{wp_c} ▷ Eq. 4.106
    Calculate p_{mi} ▷ Eq. 4.104
    Calculate p_{ci} ▷ Eq. 4.105
    Calculate T_{size} ▷ Eq. 4.111
    $\boldsymbol{P}_{new} = \emptyset$
    for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
      T_{size} Tournament Selection Operator
      if $f(\boldsymbol{x}_{p_1}) < f(\boldsymbol{x}_{p_2})$ **then**
        $p_c = p_{c_{p_1}}$
        $p_m = p_{m_{p_1}}$
        $cro = cro_{p_1}$
        $mut = mut_{p_1}$
        $p_t = p_{t_{p_1}}$
      else
        $p_c = p_{c_{p_2}}$
        $p_m = p_{m_{p_2}}$
        $cro = cro_{p_2}$
        $mut = mut_{p_2}$
        $p_t = p_{t_{p_2}}$
      end if
      cro th Crossover Operator using p_c
      mut th Mutation Operator using p_m
      Evaluate offsprings $\boldsymbol{x}_{new1}$ and $\boldsymbol{x}_{new2}$
      Calculate $p_{t_{new1}}$ and $p_{t_{new2}}$ ▷ Eq. 4.114
      $\boldsymbol{t}_{new1} = \langle \boldsymbol{x}_{new1}, p_m, p_c, cro, mut, p_{t_{new1}} \rangle$
      $\boldsymbol{t}_{new2} = \langle \boldsymbol{x}_{new2}, p_m, p_c, cro, mut, p_{t_{new2}} \rangle$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new1}$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new2}$
    end for
    Sort $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    $\boldsymbol{P}$ *is replaced by Best* n *in sorted* $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    Adapt Mutation Numbers ▷ Eq. 4.52
    Adapt Crossover Numbers ▷ Eq. 4.56
    for $i = 1$ to n **do**
      Reassign $mut_i^{(k)}$
      Reassign $cro_i^{(k)}$
    end for
    Update $\boldsymbol{x}_{best}$
    Update $\boldsymbol{x}_{worst}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

AGAv4 has a population $\boldsymbol{P}$ with n individuals and each ith individual is a tuple $\boldsymbol{t}_i$ =

$\langle \boldsymbol{x}_i, p_{ci}, p_{mi}, cro_i, mut_i, p_{ti} \rangle$ that is similar to the tuple of AGAv3 with the addition of p_{ti} that represents the pressure for this individual when deciding the size of the tournament selection T_{size}. During the initialization, as the population are generated, each individual receives a random pressure value inside of the range $[p_{t_{min}}, p_{t_{max}}]$.

The pressure value for each individual contributes to the size for the tournament selection T_{size} using the following equation:

$$T_{size} = \min\left(T_{max}, \max\left(T_{min}, \left\lfloor \sum_{i=1}^{n} {p_w}^{(k)} p_{ci} \right\rfloor\right)\right), \tag{4.111}$$

where T_{min} and T_{max} are internal static parameters that represent the minimum and maximum tournament size, respectively. ${p_w}^{(k)}$ is a weighting factor that is configured by a deterministic parameter control that uses Eq. 4.112 and forces the tournament to be smaller during the last iterations of the algorithm.

$${p_w}^{(k)} = 0.25 - 0.15 \max\left(0, 2\left(\frac{k}{I} - 0.5\right)\right) \tag{4.112}$$

A tournament selection of size T_{size} selects T_{size} random individuals from the population and, from these individuals, it returns the individual with the best objective function evaluation f to be the first parent. Then it obtains another T_{size} individuals from the population without the first parent and selects the best individual to be the second parent. Notice that the size of the tournament selection decides the convergence speed of AGAv4. A big tournament size forces a fast convergence but reduces the diversity of solutions in the population and a small tournament size maintain population diversity but has slower convergence.

During the reproductive process, after the generation of a pair of offspring solutions $\boldsymbol{x}_{new1}$ and $\boldsymbol{x}_{new2}$, a new pressure values are calculated to the offsprings using Eq. 4.113 and Eq. 4.114 where given that p_t is the pressure tournament from the parent with the best objective evaluation $f(\boldsymbol{x}_p)$ and r is random real number generated using a normal distribution with $r = \mathcal{N}(0, 1)$.

$$\delta p_t = \frac{1}{1 + \left(\frac{1-p_t}{p_t}\right)\exp\left(-\gamma r\right)}. \tag{4.113}$$

$$p_{t_{new}} = \begin{cases} \min\left(p_{t_{max}}, \max\left(p_{t_{min}}, p_t + \delta p_t\right)\right) & \text{if } f(\boldsymbol{x}_{new}) < f(\boldsymbol{x}_p) \\ \min\left(p_{t_{max}}, \max\left(p_{t_{min}}, p_t - \delta p_t\right)\right) & \text{otherwise} \end{cases}. \tag{4.114}$$

4.4.5.2 Parameters Description for AGAv4

Since AGAv4 is an extension of AGAv3 it has all of its internal parameters with the addition of the following parameters: (a) pressure for individual i p_t, (b) tournament size T_{size}, (c) minimum possible tournament size $T_{min} = 2$, (d) maximum possible tournament size $T_{max} = \lfloor \frac{n}{3} \rfloor$, (e) minimum pressure value for individuals $p_{t_{min}} = 0.08$, (f) maximum pressure value for individuals

$p_{t_{max}} = 1.0$, and (g) learning rate for the pressure adaptation $\gamma = 0.3$. AGAv4 has a total of twenty-six **internal** parameters.

AGAv4 contains ten **parameter control strategies**. Eight of them are the same used by AGAv3. The added **parameter control strategies** related to the adaptation of tournament size are: (a) an **adaptive parameter control strategy** configures the pressures for tournament in each iteration i, and (b) a **deterministic parameter control strategy** that configures the tournament size for the algorithm in a specific iteration.

4.4.6 Adaptive Particle Swarm Optimization (APSO)

4.4.6.1 Meta-Heuristic Description

APSO is an adaptive PSO-based meta-heuristic developed in this work that has the following additions to the PSO (Section 4.3.12): a) **adaptive parameter control strategies** that adjust **cognitive** and **social** factors, and b) alteration in the **deterministic parameter control strategy** for the **inertia weight**. The pseudo-code for APSO meta-heuristic can be seen in Algorithm 25.

Algorithm 25 Adaptive Particle Swarm Optimization (APSO)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Initial weight ($w_{initial}$)
Final weight (w_{final})
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure APSO
  Initialize $\boldsymbol{x}_i^{(0)}$ $(i = 1, ..., n)$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Calculate v_{max} ▷ Eq. 4.118
  Update particles $\boldsymbol{x}_{pbest_i}$
  Update $\boldsymbol{x}_{best}$
  $\mu_{c_1}^1 = 2$
  for $k = 1$ to $I - 1$ **do**
    $\boldsymbol{S}_{c_1} \leftarrow \emptyset$
    Calculate $w^{(k)}$ ▷ Eq. 4.117
    for $i = 1$ to n **do**
      Calculate c_{1i} *and* c_{2i} ▷ Eq. 4.115
      Calculate $\boldsymbol{v}_i^k$ ▷ Eq. 4.4
      Calculate $\boldsymbol{x}_i^k$ ▷ Eq. 4.5
      Evaluate $f(\boldsymbol{x}_i^{(k)})$
      if $f(\boldsymbol{x}_i^k) < f(\boldsymbol{x}_i^{k-1})$ **then**
        $\boldsymbol{S}_{c_1} \leftarrow c_{1i}$
      end if
      Update $\boldsymbol{x}_{pbest_i}$
    end for
    Update $\boldsymbol{x}_{best}$
    Update μ_{c_1} ▷ Eq. 4.116
  end for
  return $\boldsymbol{x}_{best}$
end procedure

The proposed modification adds a pair of **adaptive parameter control strategies** to change both social and cognitive factors influenced by the adaptive mechanism from JADE. This modification gives each ith particle their own unique social and cognitive factors, respectively, c_{1i} and c_{2i}, with values that can dynamically vary as the algorithm progress based on the swarm ex-

ploration of the search space. This mechanism works in an individual-scope by re-sampling the parameters using an average value for the cognitive factor μ_{c_1} that in turn uses as evidence for change parameter that improves the algorithm performance. Both c_{1i} and c_{2i} are tied together restricted to $c_{1i}, c_{2i} \in (0, 4)$ and $c_{1i} + c_{2i} = 4$, so whenever a factor increases the other decreases.

At every k iteration, the cognitive and social factors are generated for every particle using a normal distribution as presented in Eq. 4.115. The choice to use a normal in favor of the Cauchy distribution is because Cauchy distribution has long flat tails is more likely to generate a value further from its mean than the normal distribution.

$$\begin{aligned} {c_1}_i^k &= \mathcal{N}(\mu_{c_1}^k, 0.1) \\ {c_2}_i^k &= 4 - {c_1}_i^k \end{aligned}, \tag{4.115}$$

where $\mu_{c_1}^k$ is the average of social factors and its value is calculated as follows:

$$\mu_{c_1}^k = \begin{cases} 0.8 \cdot \mu_{c_1}^{k-1} + 0.2 \cdot \sum_{c_{1l} \in \boldsymbol{S_{c_1}}} \frac{c_{1l}}{|\boldsymbol{S}_{c_1}|} & \text{if } |\boldsymbol{S}_{c_1}| \neq 0 \\ \mu_{c_1}^{k-1} & \text{c.c.} \end{cases}, \tag{4.116}$$

where $\boldsymbol{S}_{c_1}$ is a set of successful values of cognitive factors c_{1i} obtained from the previous iterations. This successful set is populated in the same manner as the successful sets for mutation and crossover rates present in JADE.

The **inertial weight** modification alters the importance given by APSO to explore or refine its information about the search space. In PSO [61], the **inertial weight** w changes linearly through the iterations. It softly changes PSO behavior between an exploratory to an exploitative one. This change in values of w is done using a deterministic parameter control strategy. The proposed change modifies this linear behavior to a non-linear sigmoid function, as shown in Fig. 4.8 with the reasoning to allow APSO to spend more iterations on the exploratory stage using high weight values and, at the end of its execution, use more iterations in exploitative stages. The **deterministic parameter control strategy** used for w is expressed by the following equation:

$$w^k = w_{initial} + \frac{w_{final} - w_{initial}}{1 + e^{\left(-12\left(\frac{k}{I} - c\right)\right)}}, \tag{4.117}$$

where I is the number of iterations allowed by the algorithm to run, k is the current iteration and c is the real **static internal** parameter inside $(0, 1)$ that adjusts in which percentage $100c\%$ of iterations the algorithm changes its behavior from exploratory to exploitative, a suggested value to c is 0.45. Lastly, the initial weight $w_{initial} \in (0, 1)$ with a suggested value of 0.85 and the final weight $w_{final} \in (0, 1)$ with a recommended value of 0.05. Reminding that $w_{initial} > w_{final}$.

Even though not added as an adaptive parameter, the maximum speed v_{max} it is fixed as an **internal static** parameter that depends on the search space size. Equation 4.118 is used to obtain the v_{max}.

$$v_{max} = \frac{||\boldsymbol{ub} - \boldsymbol{lb}||}{2} \tag{4.118}$$

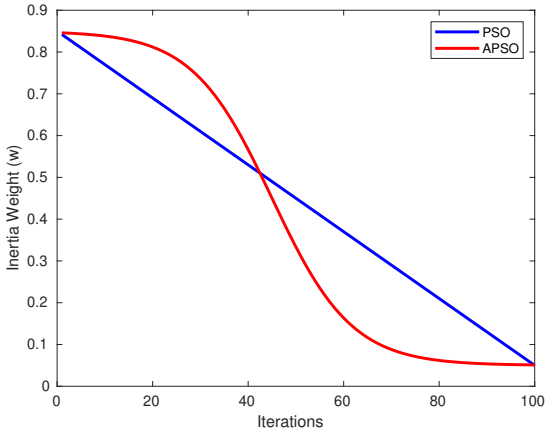


Figura 4.8 – Graphical example of the inertia weight evolution during execution of APSO (red) and PSO (blue) with both having $I = 100$, $w_{initial} = 0.85$ and $w_{final} = 0.05$.

4.4.6.2 Parameters Description for APSO

APSO has the following six **internal** parameters: (a) social factor for particle i c_{1i}, (b) cognitive factor for particle i c_{2i}, (c) initial average of social factors $\mu_{c_1\,initial} = 2$, (d) ratio of iterations in which the weight changes from exploratory to exploitative $c = 0.45$, (e) maximum velocity v_{max}, and (f) initial velocity $v_{initial} = 0.25v_{max}$.

APSO has an **adaptive parameter control strategy** that controls simultaneously the social and individual factors for each particle in a way that when one increases the other decreases. It forces each individual to assume an exploratory or exploitative behavior. Similar to PSO, APSO has a **deterministic parameter control strategy** for the inertia weight parameter.

APSO also contain two **external static** parameters: (a) initial weight value $w_{initial}$ and (b) final weight value w_{final}.

4.4.7 Adaptive Particle Swarm Optimization v2 (APSOv2)

4.4.7.1 Meta-Heuristic Description

APSOv2 is the second implementation of an adaptive PSO-based that have similar features from APSO. The additions of APSOv2 compared with APSO is the change of the parameter control the **inertial weight** for the algorithm from a **deterministic** approach that that work in a population scope to an **adaptive** one that works in an individual-scope. Algorithm 26 presents APSOv2 in pseudo-code format.

APSOv2 has a swarm $\boldsymbol{P}_c$ of n particles and each of them contains a particle position as d-dimensional vector $\boldsymbol{x}_i$, speed $\boldsymbol{v}_i$, copy of the best individual position $\boldsymbol{x}_{pbest_i}$, the particle inertia weight w_i and individual social and cognitive factors c_{1i} and c_{2i}.

During the initialization, APSOv2 generates random solutions in the search space for function

Algorithm 26 Adaptive Particle Swarm Optimization v2 (APSOv2)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure APSOv2($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{x}_i^{(0)}$ $(i = 1, ..., n)$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Calculate v_{max} ▷ Eq. 4.118
  Update particles $\boldsymbol{x}_{pbest_i}$
  Update $\boldsymbol{x}_{best}$
  $\mu_{c_1}^1 = 2$
  for $k = 1$ to $I - 1$ **do**
    Update $\boldsymbol{x}_{best}^{(k)}$ *and* $\boldsymbol{x}_{worst}^{(k)}$
    Update $\boldsymbol{w}_i^{(k)}$ $(i = 1, ..., n)$ ▷ Eq. 4.119
    $\boldsymbol{S}_{c_1} \leftarrow \emptyset$
    for $i = 1$ to n **do**
      Calculate c_{1i} *and* c_{2i} ▷ Eq. 4.115
      Calculate $\boldsymbol{v}_i^k$ ▷ Eq. 4.4
      Calculate $\boldsymbol{x}_i^k$ ▷ Eq. 4.5
      Evaluate $f(\boldsymbol{x}_i^{(k)})$
      if $f(\boldsymbol{x}_i^k) < f(\boldsymbol{x}_i^{k-1})$ **then**
        $\boldsymbol{S}_{c_1} \leftarrow c_{1i}$
      end if
      Update $\boldsymbol{x}_{pbest_i}$
    end for
    Update $\boldsymbol{x}_{best}$
    Update μ_{c_1} ▷ Eq. 4.116
  end for
  return $\boldsymbol{x}_{best}$
end procedure

f to set as positions for the particles. APSOv2 similar to APSO initialize the particles speeds, and social and cognitive factors c_{1i} and c_{2i}.

During the iterative process, for every ith particle, the social and cognitive parameters are updated using the same scheme as APSO using Eq. 4.115.

APSOv2 calculates each particle weight using w_i the following equation:

$$w_i = w_{min} + \frac{(w_{max} - w_{min})}{1 + exp\left(-14\frac{f(\boldsymbol{x}_i) - f(\boldsymbol{x}_{best}^{(k)})}{f(\boldsymbol{x}_{worst}^{(k)}) - f(\boldsymbol{x}_{best}^{(k)})}\right)}, \tag{4.119}$$

where w_{min} and w_{max} are **internal static** parameter that are, respectively, the minimum and maximum inertial weight values allowed for the particles. $\boldsymbol{x}_{best}^{(k)}$ and $\boldsymbol{x}_{worst}^{(k)}$ are the best and worst solutions found in iteration k, instead of the best and worst solution found for the whole APSOv2 execution. This **adaptive parameter control strategy** results in individuals close to the best solution having small weight values that consequently produce exploitative behavior and individuals far to the best solution to behave in an exploratory manner. Since each particle has its weight w_i, APSOv2 has during every iteration particles concurrently in an exploitative stage while others are in an explorative one. Different than PSO and APSO, in which all particles in the swarm behave in an exploratory or exploitative manner.

4.4.7.2 Parameters Description for APSOv2

APSOv2 does not contain **external static** parameters. APSOv2 has eight **internal** parameters. These parameters are listed as follows: (a) social factor for particle i c_{1i}, (b) cognitive factor for particle i c_{2i}, (c) initial average of social factors $\mu_{c_1 initial} = 2$, (d) initial average of cognitive factors $\mu_{c_2}^{(0)} = 2$, (e) maximum velocity v_{max}, (f) initial velocity $v_{initial} = 0.25 v_{max}$, (g) maximum weight value $w_{max} = 0.9$, and (h) minimum weight value $w_{final} = 0.05$.

APSOv2 has two **adaptive parameter control strategies** that adjust the **social** and **cognitive** factors for a particle i c_{1i} and c_{2i}, respectively, and the inertia weight for each particle i w_i.

4.4.8 Hybrid Discrete Particle Swarm Optimization Utilization-based (HDPSO-U)

4.4.8.1 Meta-Heuristic Description

HDPSO-U is a GA and PSO hybrid meta-heuristic specialized to optimize the task mapping problem of RTA into RTNoC-based SoC platforms. The problem present in Chapter 3.8. HDPSO-U is a version of HDPSO-M with a local search algorithm different than the LPT algorithm. Instead, it uses a method that uses the utilization factors of both the not mapped tasks and the processors in a platform $\mathbf{\Pi}$ in an approach to map tasks similar to the *MappingToContainer* phase on the constructive task mapping algorithm present on [85]. Algorithm 27 shows HDPSO-U in pseudo-code form.

Algorithm 27 Hybrid Discrete Particle Swarm Optimization - Utilization based (HDPSO-U)

INPUT: Objective Function ($f(\cdot)$)
Application ($\mathbf{\Omega}$)
Platform ($\mathbf{\Psi}$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Bernoulli Probability (p)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

1: **procedure** HDPSO-U
2: *Initialize* $\boldsymbol{x}_i$ $(i = 1, ..., n)$
3: *Evaluate* $f(\boldsymbol{x}_i - \mathbf{1})$
4: *Update* $\boldsymbol{x}_{pbest_i}$
5: *Update* $\boldsymbol{x}_{best}$
6: **for** $k = 1$ to $I - 1$ **do**
7: **for** $i = 1$ to n **do**
8: *Calculate* $\boldsymbol{v}_i^k$ ▷ Eq. 4.120
9: *Calculate* $\boldsymbol{x}_i^k$ ▷ Eq. 4.78
10: ***Evaluate*** $f(\boldsymbol{x}_i - \mathbf{1})$
11: *Update* $\boldsymbol{x}_{pbest_i}$
12: **end for**
13: *Update* $\boldsymbol{x}_{best}$
14: **end for**
15: **return** $\boldsymbol{x}_{best} - \mathbf{1}$
16: **end procedure**

Similar to HDPSO-M, the algorithm has a swarm of n particles and each of one them at iteration k has a position $x_i^{(k)}$, a speed $v_i^{(k)}$, and copies for individual and global best $\boldsymbol{x}_{pbest_i}$ and

$\boldsymbol{x}_{best}$. The process to update positions and speed use the same operators for addition, subtraction and multiplication, respectively, $\overset{cross}{+}$, $\overset{o}{-}$ and $\overset{o}{\times}$, used by HDPSO-M as well as the Eq. 4.78 to update particles positions.

HDPSO-U updates its particles speeds using the following equation:

$$v_i^{(k+1)} = v_i^{(k)} \overset{cross}{+} \Big(LUM(v_{i,pbest}) \overset{cross}{+} LUM(v_{gbest}) \Big), \tag{4.120}$$

where $LUM(\cdot)$ is a deterministic algorithm that has as input and output variables equal to those of $LPTM$ in HDPSO-M (see Algorithm 20). In other words, it receives a vector $\boldsymbol{x}$ containing components $x_j = 0$ representing tasks not mapped yet and elements $x_k \neq 0$ symbolizing mapped tasks to processors with indices $x_k - 1$. LUM takes this vector $\boldsymbol{x}$ and then sorts the unmapped tasks in descending order of utilization factors to then sequentially allocate them to the least used processors in $\boldsymbol{\Pi}$, i.e., the processors with the least utilization factor. LUM algorithm is represented in pseudo-code form in Algorithm 28.

Algorithm 28 Largest Utilization first based Mapping (LUM)

INPUT: Application ($\boldsymbol{\Omega}$)
Platform ($\boldsymbol{\Psi}$)
Offset Mapping ($\boldsymbol{x}$)
OUTPUT: Possible Offset Mapping ($\boldsymbol{x}_{out}$)

procedure LUM($\boldsymbol{\Omega}, \boldsymbol{\Psi}, \boldsymbol{x}$)
  $\boldsymbol{T}_{unmapped\_tasks} = \{\{\boldsymbol{\tau}_j, x_j\} \in \{\boldsymbol{\Gamma}, \mathbb{N}\} : x_j \in \boldsymbol{x} \wedge x_j = 0\}$
  Sort $\boldsymbol{T}_{unmapped\_tasks}$ utilization factors in descend order
  $\boldsymbol{T}_{mapped\_tasks} = \{\{\boldsymbol{\tau}_k, x_k\} \in \{\boldsymbol{\Gamma}, \mathbb{N}\} : x_k \in \boldsymbol{x} \wedge x_k \neq 0\}$
  for $i = 1$ to $|\boldsymbol{T}_{mapped\_tasks}|$ **do**
    $\{\boldsymbol{\tau}_k, x_k\} = \boldsymbol{T}_{mapped\_tasks_i}$
    Map task $\boldsymbol{\tau}_k$ to core with index $x_k - 1$
  end for
  for $i = 1$ to $|\boldsymbol{T}_{unmapped\_tasks}|$ **do**
    $s_{min} = MAX^*$
    $\pi_{min} = \emptyset$
    for $n = 1$ to $|\Pi|$ **do**
      $s_n = \sum_{\tau_m \in map(\pi_n)} \frac{C_m}{T_m}$
      if then $s_n < s_{min}$
        $s_{min} = s_n$
        $\pi_{min} = \pi_n$
      end if
    end for
    $\{\boldsymbol{\tau}_j, x_j\} = \boldsymbol{T}_{unmapped\_tasks_i}$
    Map task $\boldsymbol{\tau}_j$ to core π_{min}
    $x_j = index(\pi_{min}) + 1$
  end for
  $\boldsymbol{T} = \boldsymbol{T}_{mapped\_tasks} \cup \boldsymbol{T}_{unmapped\_tasks}$
  Sort $\boldsymbol{T}$ based on tasks indices.
  $\boldsymbol{x}_{out} \leftarrow \emptyset$
  for $i = 1$ to $|\boldsymbol{T}|$ **do**
    $\{\boldsymbol{\tau}, x\} = \boldsymbol{T}_i$
    $\boldsymbol{x}_{out} \leftarrow x$
  end for
  return $\boldsymbol{x}_{out}$
end procedure

* MAX represents an arbitrary very large value.

Analogous to HDPSO-M, HDPSO-U has a single **external static** parameter that controls its diversity of possible solutions.

4.4.8.2 Parameters Description for HDPSO-U

HDPSO-U do not have any **parameter control strategy** neither any **internal** parameters. HDPSO-U only relies on a single **external static** parameter p that represents the rate of addition of random components into the task mapping obtained by LUM.

4.4.9 Adaptive Hybrid Discrete Particle Swarm Optimization Utilization-based (AHDPSO-U)

4.4.9.1 Meta-Heuristic Description

AHDPSO-U is an adaptive version of HDPSO-U inspired by the **adaptive parameter control strategies** present in AGAv2. The added adaptive mechanism dynamically change the Bernoulli probability used to generate the random vectors $\boldsymbol{R}_1$ and $\boldsymbol{R}_2$ that are used to decide which task should be mapped by LUM in Algorithm 28. AHDPSO-U is displayed in pseudo-code form in Algorithm 29.

Algorithm 29 Adaptive Hybrid Discrete Particle Swarm Optimization - Utilization based (AHDPSO-U)

INPUT: Objective Function ($f(\cdot)$)
Application ($\boldsymbol{\Omega}$)
Platform ($\boldsymbol{\Psi}$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

1: **procedure** AHDPSO-U
2: *Initialize* $\boldsymbol{x}_i$ $(i = 1, ..., n)$
3: *Initialize* $p_i = 0.5$ $(i = 1, ..., n)$
4: *Evaluate* $f(\boldsymbol{x}_i - \mathbf{1})$
5: *Update* $\boldsymbol{x}_{pbest_i}$
6: *Update* $\boldsymbol{x}_{best}$
7: **for** $k = 1$ to $I - 1$ **do**
8: $\boldsymbol{S}_p = \emptyset$
9: **for** $i = 1$ to n **do**
10: *Generate* p_i $(i = 1, ..., n)$ ▷ Eq. 4.121
11: *Calculate* $\boldsymbol{v}_i^k$ ▷ Eq. 4.120
12: *Calculate* $\boldsymbol{x}_i^k$ ▷ Eq. 4.78
13: *Evaluate* $f(\boldsymbol{x}_i - \mathbf{1})$
14: *Update* $\boldsymbol{S}_p$ ▷ Eq. 4.124
15: *Update* $\boldsymbol{x}_{pbest_i}$
16: **end for**
17: *Update* $\mu_p^{(k+1)}$ ▷ Eq. 4.122
18: *Update* $\sigma_p^{(k+1)}$ ▷ Eq. 4.123
19: *Update* $\boldsymbol{x}_{best}$
20: **end for**
21: **return** $\boldsymbol{x}_{best} - \mathbf{1}$
22: **end procedure**

Similar to HDPSO-U and HDPSO-M, AHDPSO-U has a swarm of n particles and each ith particle position is a d-dimensional integer vector $\boldsymbol{x}_i^{(k)}$ that represents an offset task mapping. These particles also have individual probabilities values p_i that is updated during every k iteration using Eq. 4.121. The adaptive parameter control strategy configures p_i in an individual scope using as evidence for change objective function f evaluation by creating a set of successful values of

parameters and then re-sampling them based on the average of these successful parameter values. AHDPSO initializes p_i probabilities with values equal to 0.5.

$$p_i = \min(p_{max}, \max(p_{min}, \mathcal{N}(\mu_p^{(k)}, \sigma_p^{(k)}))), \tag{4.121}$$

where p_{max} and p_{min} are the maximum and minimum probabilities allowed by the algorithm and their suggested values are 0.95 and 0.05. $\mu_p^{(k)}$ and $\sigma_p^{(k)}$ are the mean value and the standard deviation value for the Bernoulli probabilities generation during the kth iteration. Equation 4.122 and Eq. 4.123 adjust $\mu_p^{(k)}$ and $\sigma_p^{(k)}$ values using information gathered during AHDPSO-U current iteration k.

$$\mu_p^{(k+1)} = \begin{cases} \frac{\sum_{p \in \boldsymbol{S}_p}(p)}{|\boldsymbol{S}_p|} & \text{if } \boldsymbol{S}_p \neq \emptyset \\ \mu_p^{(k)} & \text{otherwise} \end{cases}. \tag{4.122}$$

$$\begin{gathered} \nu = \begin{cases} a\sigma_p^{(k)} & \text{if } |\boldsymbol{S}_p| > \lceil \frac{N}{5} \rceil \\ a^{-1}\sigma_p^{(k)} & \text{if } |\boldsymbol{S}_p| < \lceil \frac{N}{5} \rceil \\ \sigma_p^{(k)} & \text{otherwise} \end{cases}, \\ \sigma_p^{(k+1)} = \min(0.5, \max(0.05, \nu)) \end{gathered} \tag{4.123}$$

where a is an adaptation rate for the standard deviation value, and its suggested value is 1.2. Also, $\boldsymbol{S}_p$ is a set that contains successful values of probabilities, and it is populated whenever there is an improvement in the evaluation of a particle position between two iterations according to Eq. 4.124.

$$\boldsymbol{S}_p \leftarrow \begin{cases} p_i & \text{if } |unsch(\boldsymbol{x}_i^{(k+1)} - \bar{\mathbf{1}}, \boldsymbol{\Omega}, \boldsymbol{\Psi})| < |unsch(\boldsymbol{x}_i^{(k)} - \bar{\mathbf{1}}, \boldsymbol{\Omega}, \boldsymbol{\Psi})| \\ \emptyset & \text{otherwise} \end{cases} \tag{4.124}$$

4.4.9.2 Parameters Description

AHDPSO-U has an **adaptive parameter control strategy** to dynamically configure the Bernoulli probabilities used by each particle p_i.

AHDPSO-U relies on the following seven internal static parameters: (a) Bernoulli probability for the ith particle p_i, (b) maximum Bernoulli probability allowed $p_{max} = 0.95$, (c) minimum Bernoulli probability allowed $p_{min} = 0.05$, (d) standard deviation adaptation rate $a = 1.2$, (e) initial average for Bernoulli probabilities $= \mu_p^{(0)} = 0.5$, (f) maximum standard deviation $\sigma = 0.5$, and (g) minimum standard deviation $\sigma = 0.05$.

4.4.10 Adaptive Gray Wolf Optimization (AGWO)

4.4.10.1 Meta-Heuristic Description

AGWO is an adaptive version for GWO that adds an adaptive parameter control strategy, similar to the one present in JADE (Section 4.3.11), to modify the scheme in which wolves update their positions. AGWO is illustrated in pseudo-code form in Algorithm 30.

Algorithm 30 Adaptive Gray Wolf Optimization (AGWO)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Scaling factor (F)
Crossover rate (CR)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure AGWO($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize wolves $\boldsymbol{x}_i^{(0)}$ $(i = 1, ..., n)$
  Initialize coefficients a, A and C
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{x}_\alpha$ *Best Solution*
  Update $\boldsymbol{x}_\beta$ *Second best Solution*
  Update $\boldsymbol{x}_\delta$ *Third best solution*
  for $k = 1$ to $I - 1$ **do**
    $\boldsymbol{S}_\alpha = \emptyset$
    $\boldsymbol{S}_\beta = \emptyset$
    $\boldsymbol{S}_\delta = \emptyset$
    for $i = 1$ to N **do**
      Update wolf $\boldsymbol{x}_i^{(k)}$ ▷ Eq. 4.125
      Evaluate $f(\boldsymbol{x}_i^{(k)})$
      Update $\boldsymbol{x}_\alpha$ *Best Solution*
      Update $\boldsymbol{x}_\beta$ *Second best Solution*
      Update $\boldsymbol{x}_\delta$ *Third best solution*
      Update $\boldsymbol{S}_\alpha$ ▷ Eq. 4.129
      Update $\boldsymbol{S}_\beta$ ▷ Eq. 4.130
      Update $\boldsymbol{S}_\delta$ ▷ Eq. 4.131
    end for
    Update coefficients a, A and C
    Update $w_\alpha^{(k+1)}$ ▷ Eq. 4.126
    Update $w_\beta^{(k+1)}$ ▷ Eq. 4.127
    Update $w_\beta^{(k+1)}$ ▷ Eq. 4.127
  end for
  return $\boldsymbol{x}_\alpha$
end procedure

Similar as the canonical GWO, AGWO has a pack of omega wolves $\boldsymbol{P}_c$ of size N with their positions represented as vectors $\boldsymbol{x}_i^{(k)}$ for every ith wolf at iteration k of the algorithm. AGWO also has three vectors holding the positions for the leaders $\boldsymbol{x}_\alpha$, $\boldsymbol{x}_\beta$ and $\boldsymbol{x}_\delta$ that are used to update the omega wolves positions.

Each ith wolf updates its position following the three leader wolves α, β and δ giving the same level of importance to them. This behavior does not reflect the description of wolf hierarchical social behavior in nature accurately since the α wolf should be selected to be followed over β and δ. Even in particular occasions in which this hierarchy does not entirely apply, for example, when a β or δ wolf is closer in the hunting ground than an α, a ω wolf does not follow the three classes of leaders equally. AGWO takes this difference into consideration to apply a different update mechanism for the ω wolves positions using weight parameter values for each leader, respectively, $w_\alpha^{(k)}$, $w_\beta^{(k)}$ and $w_\alpha^{(k)}$ as shown in Equation 4.125.

$$x_{i,j}^{(k+1)} = \frac{w_\alpha^{(k)} x_{f_\alpha,j} + w_\beta^{(k)} x_{f_\beta,j} + w_\delta^{(k)} x_{f_\delta,j}}{w_\alpha^{(k)} + w_\beta^{(k)} + w_\delta^{(k)}}, \tag{4.125}$$

where x_{f_α}, x_{f_β} and x_{f_δ} are the positions that the ith wolf would be if it would follow only one of the wolf leaders as calculated by Eq. 4.10.

All omega wolves in the pack share the leaders weighting factors parameters. For this reason, the parameter control mechanism for these parameters works in a population-scope. The adaptive parameters $w_\alpha^{(k+1)}$, $w_\beta^{(k+1)}$, and $w_\delta^{(k+1)}$ are updated at every iteration using Eq. 4.126, Eq. 4.127 and Eq. 4.128.

$$w_\alpha^{(k+1)} = \begin{cases} (1-c)w_\alpha^{(k)} + c\frac{\sum_{w_\alpha \in \boldsymbol{S}_\alpha}(w_\alpha)}{|\boldsymbol{S}_\alpha|} & \text{if } \boldsymbol{S}_\alpha \neq \emptyset \\ w_\alpha^{(k)} & \text{otherwise} \end{cases}. \tag{4.126}$$

$$w_\beta^{(k+1)} = \begin{cases} (1-c)w_\beta^{(k)} + c\frac{\sum_{w_\beta \in \boldsymbol{S}_\beta}(w_\beta)}{|\boldsymbol{S}_\beta|} & \text{if } \boldsymbol{S}_\beta \neq \emptyset \\ w_\beta^{(k)} & \text{otherwise} \end{cases}. \tag{4.127}$$

$$w_\delta^{(k+1)} = \begin{cases} (1-c)w_\delta^{(k)} + c\frac{\sum_{w_\delta \in \boldsymbol{S}_\delta}(w_\delta)}{|\boldsymbol{S}_\delta|} & \text{if } \boldsymbol{S}_\delta \neq \emptyset \\ w_\delta^{(k)} & \text{otherwise} \end{cases}, \tag{4.128}$$

where c is a internal static parameter that controls the adaptation speed, suggested value used is $c = 0.15$, and $\boldsymbol{S}_\alpha$, $\boldsymbol{S}_\beta$ and $\boldsymbol{S}_\delta$ are sets that hold successful values of weights that are updated whenever a omega wolf improves one of the leaders evaluation result.

The sets $\boldsymbol{S}_\alpha$, $\boldsymbol{S}_\beta$ and $\boldsymbol{S}_\delta$ are updated process follows equations 4.129, 4.130, and 4.131, respectively.

$$\boldsymbol{S}_\alpha \leftarrow \begin{cases} w_\alpha^{(k)} & \text{if } f(\boldsymbol{x}_i) < f(\boldsymbol{x}_\alpha) \\ \emptyset & \text{otherwise} \end{cases}. \tag{4.129}$$

$$\boldsymbol{S}_\beta \leftarrow \begin{cases} w_\beta^{(k)} & \text{if } f(\boldsymbol{x}_i) < f(\boldsymbol{x}_\beta) \wedge f(\boldsymbol{x}_i) > f(\boldsymbol{x}_\alpha) \\ \emptyset & \text{otherwise} \end{cases}. \tag{4.130}$$

$$\boldsymbol{S}_\delta \leftarrow \begin{cases} w_\delta^{(k)} & \text{if } f(\boldsymbol{x}_i) < f(\boldsymbol{x}_\delta) \wedge f(\boldsymbol{x}_i) > f(\boldsymbol{x}_\beta) \\ \emptyset & \text{otherwise} \end{cases}. \tag{4.131}$$

4.4.10.2 Parameters Description for AGWO

AGWO contains three **adaptive parameter control mechanisms** to configure the values for weights used by each wolf leader ($\boldsymbol{x}_\alpha$, $\boldsymbol{x}_\beta$ and $\boldsymbol{x}_\delta$). These schemes work in a population-scope and use as evidence for changing the improvement of objective function f evaluation using successful sets of weight values as intermediate metrics. AGWO also contains **internal** both **static** and dynamic. These parameters are listed as follows: (a) weighting factor for alpha wolf w_α, (b) weighting factor for beta wolf w_β, (c) weighting factor for delta wolf w_δ, (d) adaptation rate $c = 0.15$, (e) initial weight for alpha wolf $w_\alpha^{(0)} = \frac{1}{3}$, (f) initial weight for beta wolf $w_\beta^{(0)} = \frac{1}{3}$, and (g) initial weight for delta wolf $w_\delta^{(0)} = \frac{1}{3}$.

4.5 Multi-Objective Optimization Bio-Inpired Meta-Heuristics from Literature

4.5.1 Non-dominated Sorting Genetic Algorithm II (NSGA-II)

4.5.1.1 Meta-Heuristic Description

Non-dominated Sorting Genetic Algorithm (NSGA-II) [86] is a multi-objective GA-based bio-inspired meta-heuristic that is used to solve COP by searching for Pareto Optimal (PO) solutions inside of the search space that forms a Pareto Frontier (PF). NSGA-II is shown in pseudo-code form in Algorithm 31.

Algorithm 31 Non-dominant Sorting Genetic Algorithm (NSGA-II)

INPUT: Objective Function ($\boldsymbol{F}(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of chromosomes (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Mutation rate (p_m)
Crossover rate (p_c)
OUTPUT: Pareto Front ($\boldsymbol{PF}$)

1: **procedure** NSGAII($\boldsymbol{F}, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}, p_m, p_c$)
2: *Initialize* $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci} \rangle$
3: *Evaluate* $\boldsymbol{F}(\boldsymbol{x}_i)$
4: *Obtain Dominance Ranks* d_{ri} ▷ Alg. 32
5: *Obtain Crowding Distances* d_{ci} ▷ Alg. 33
6: *Sort* $\boldsymbol{P}$ *using* ***non-dominatant sorting***
7: $\boldsymbol{PF} = \boldsymbol{S}_{F1}$
8: **for** $k = 1$ to $I - 1$ **do**
9: $\mathbf{P}_{new} = \emptyset$
10: **for** $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
11: *Binary Tournament Selection*
12: *One-Point Crossover*
13: *Uniform Gene-Flip Mutation*
14: *Evaluate generated offsprings*
15: $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new} = \langle \boldsymbol{x}_{new1}, 0, 0 \rangle$
16: $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new} = \langle \boldsymbol{x}_{new1}, 0, 0 \rangle$
17: **end for**
18: $\boldsymbol{P} = \boldsymbol{P} \cup \boldsymbol{P}_{new}$
19: *Obtain Dominance Ranks* d_{ri} ▷ Alg. 32
20: *Obtain Crowding Distances* d_{ci} ▷ Alg. 33
21: *Sort* $\boldsymbol{P}$ *using* ***non-dominatant sorting***
22: *Remove last individuals in* $\boldsymbol{P}$ *until* $|\boldsymbol{P}| = n$
23: $\boldsymbol{PF} = \boldsymbol{S}_{F1}$
24: **end for**
25: **return** $\boldsymbol{PF}$
26: **end procedure**

NSGA-II has a population $\boldsymbol{P}$ with n individuals and each ith individual is represented by a tuple $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci} \rangle$, where $\boldsymbol{x}_i$ is a d-dimensional integer vector that is a possible solution for the multi-objective function $\boldsymbol{F}$ with o objectives. The value d_{ri} is the dominance rank for the solution held by individual i and d_{ci} is the crowding distance between this solution and its farthest neighbor that shares the same dominance rank d_{ri}.

A dominance rank d_{ri} for a solution $\boldsymbol{x}_i$ represents the rank for a front in the objective space generated by all solutions in the population that does not dominate $\boldsymbol{x}_i$ including $\boldsymbol{x}_i$. In other words, $\{\boldsymbol{x}_j : \boldsymbol{t}_j \in \boldsymbol{P} \wedge \boldsymbol{F}(\boldsymbol{x}_i) \not\prec \boldsymbol{F}(\boldsymbol{x}_j) \wedge \boldsymbol{F}(\boldsymbol{x}_j) \not\prec \boldsymbol{F}(\boldsymbol{x}_i)\}$. If the front generated by $\boldsymbol{x}_i$ dominates

all other fronts, then $\boldsymbol{x}_i$ is PO in the population and its front is the Pareto front with $d_{ri} = 1$.

NSGA-II uses *ndFronts* defined in Algorithm 32 to obtain the list of non-dominated front solutions $\boldsymbol{L}_F$ that is composed of sets that in turn contains solutions for fronts with different ranks. In other words, $\boldsymbol{L}_F = \{\boldsymbol{S}_{F1}, \boldsymbol{S}_{F2}, ..., \boldsymbol{S}_{Fm}\}$ where $\boldsymbol{S}_{F1}$ is composed of solutions with evaluations that form the Pareto Front (PF) or front with rank 1 and $\boldsymbol{S}_{Fm}$ is formed by solutions that forms the front with dominance rank m. $ndFronts(\cdot)$ uses a set $\boldsymbol{X}$ that is composed of solutions $\boldsymbol{x}_i$ for all individuals $\boldsymbol{t}_i \in \boldsymbol{P}$.

Algorithm *ndFronts* first identifies for each solution $\boldsymbol{x}_i \in \boldsymbol{X}$ the set of solutions that are dominated by it $\boldsymbol{S}_{di}$ and the number of solutions that dominates it n_{di}. Meanwhile, filing the set of solutions that results in the PF $\boldsymbol{S}_{F1}$. Knowing n_{di} and $\boldsymbol{S}_{di}$, *ndFronts* then iteratively populates the other fronts dominated by PF until a last empty front is found and the iterative process ends.

Algorithm 32 Non-dominant Fronts Calculator (ndFronts)

INPUT: Objective Function ($\boldsymbol{F}(\cdot)$)
Set of solutions ($\boldsymbol{X}$)
Number of individuals (n)
Number of components (d)
OUTPUT: List of Fronts ($\boldsymbol{L}_F$)

1: **procedure** $ndFronts(\boldsymbol{F}, \boldsymbol{X}, n, d)$
2: $\quad\boldsymbol{L}_F = \emptyset$
3: $\quad\boldsymbol{S}_{F1} = \emptyset$
4: $\quad$**for** $\boldsymbol{x}_i \in \boldsymbol{X}$ **do**
5: $\quad\quad\boldsymbol{S}_{di} = \emptyset$
6: $\quad\quad n_{di} = 0$
7: $\quad\quad$**for** $\boldsymbol{x}_j \in \{\boldsymbol{x} \in \boldsymbol{X} : \boldsymbol{x} \neq \boldsymbol{x}_i\}$ **do**
8: $\quad\quad\quad$**if** $\boldsymbol{F}(\boldsymbol{x}_i) \prec \boldsymbol{F}(\boldsymbol{x}_j)$ **then**
9: $\quad\quad\quad\quad\boldsymbol{S}_{di} \leftarrow \boldsymbol{x}_j$
10: $\quad\quad\quad$**else**
11: $\quad\quad\quad\quad$**if** $\boldsymbol{F}(\boldsymbol{x}_j) \prec \boldsymbol{F}(\boldsymbol{x}_i)$ **then**
12: $\quad\quad\quad\quad\quad n_{di} = n_{di} + 1$
13: $\quad\quad\quad\quad$**end if**
14: $\quad\quad\quad$**end if**
15: $\quad\quad$**end for**
16: $\quad\quad$**if** $n_{di} = 0$ **then**
17: $\quad\quad\quad\boldsymbol{S}_{F1} \leftarrow \boldsymbol{x}_i$
18: $\quad\quad$**end if**
19: $\quad$**end for**
20: $\quad\boldsymbol{L}_F \leftarrow \boldsymbol{S}_{F1}$
21: $\quad k = 1$
22: $\quad$**while** $\boldsymbol{S}_{Fk} \neq \emptyset$ **do**
23: $\quad\quad\boldsymbol{A} = \emptyset$
24: $\quad\quad$**for** $\boldsymbol{x}_i \in \boldsymbol{S}_{Fk}$ **do**
25: $\quad\quad\quad$**for** $\boldsymbol{x}_j \in \boldsymbol{S}_{di}$ **do**
26: $\quad\quad\quad\quad n_{dj} = n_{dj} - 1$
27: $\quad\quad\quad\quad$**if** $n_{dj} = 0$ **then**
28: $\quad\quad\quad\quad\quad\boldsymbol{A} \leftarrow \boldsymbol{x}_j$
29: $\quad\quad\quad\quad$**end if**
30: $\quad\quad\quad$**end for**
31: $\quad\quad$**end for**
32: $\quad\quad k = k + 1$
33: $\quad\quad\boldsymbol{S}_{Fk} = \boldsymbol{A}$
34: $\quad\quad\boldsymbol{L}_F \leftarrow \boldsymbol{S}_{Fk}$
35: $\quad$**end while**
36: $\quad$**return** $\boldsymbol{L}_F$
37: **end procedure**

After acquiring $\boldsymbol{L}_F$ by using *ndFronts*, the process to assign dominance rank for each $\boldsymbol{t}_i$ is to designate the rank for the front in which the solution $\boldsymbol{x}_i$ is situated. In the case that $\boldsymbol{x}_i \in \boldsymbol{S}_{Fm}$ then $d_{ri} = m$.

NSGA-II uses *crwdDist* defined in Algorithm 33 to calculate the crowding distance for each solution $\boldsymbol{x}_i$ in a front $\boldsymbol{S}_{Fm}$. Algorithm *crwdDist* first initialize all crowding distances as 0. Then

for each k objective of the possible o objectives in $\boldsymbol{F}$, *crwdDist* sots the solutions in an ascending order based on their evaluations of f_k and assigns for first and last solutions in $\boldsymbol{S}_{Fm}$, namely $\boldsymbol{x}_1$ and $\boldsymbol{x}_l$, the crowding distance with an infinity large value, i.e. $\eta_1 = \eta_l = \infty$. The reasoning for this is that solutions with results in the boundaries of their fronts have their farthest neighbor infinity far (since there are none). Figure 4.9 illustrates the crowding distance for solutions $\boldsymbol{x}_i$ with an evaluation $\boldsymbol{F}(\boldsymbol{x}_i)$ that is part of a front $\boldsymbol{S}_{Fm}$.

Algorithm 33 Crowding Distance Calculator (crwdDist)

INPUT: Objective Function ($\boldsymbol{F}(\cdot)$)
Number of Objectives in $\boldsymbol{F}$ (o)
Solutions in Front m ($\boldsymbol{S}_{Fm}$)
OUTPUT: Vector of Crowding Distances ($\boldsymbol{\eta}$)

1: **procedure** *crwdDist*
2: $l = |\boldsymbol{S}_{Fm}|$
3: *Generate vector* $\boldsymbol{\eta}$ *with size* l
4: **for** $\boldsymbol{x}_i \in \boldsymbol{S}_{Fm}$ **do**
5: $\eta_i = 0$
6: **end for**
7: **for** $k = 1$ to o **do**
8: *Sort* $\boldsymbol{S}_{Fm}$ *based on* f_k
9: $\eta_1 = \eta_l = \infty$
10: **for** $j = 2$ to $(l-1)$ **do**
11: $\eta_j = \eta_j + (f_k(\boldsymbol{x}_{(j+1)}) - f_k(\boldsymbol{x}_{(j-1)}))$
12: **end for**
13: **end for**
14: **return** $\boldsymbol{\eta}$
15: **end procedure**

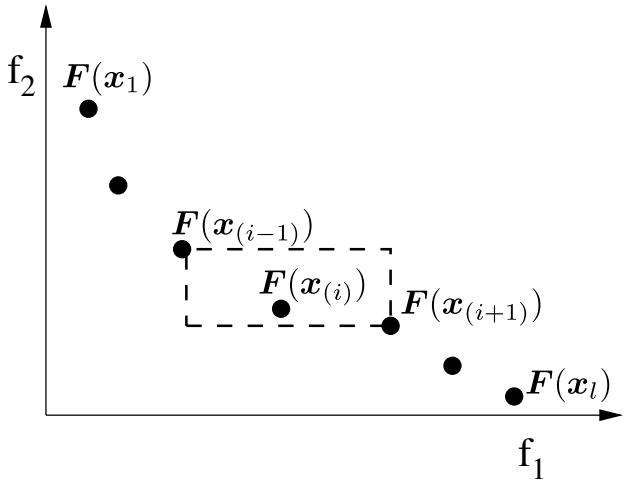


Figura 4.9 – Graphical example of the crowding distance for a solution $\boldsymbol{x}_i$.

As in other bio-inspired meta-heuristics, during the initialization stage, NSGA-II generates random solutions inside of the search space with upper and lower boundaries defined by the vectors $\boldsymbol{ub}$ and $\boldsymbol{lb}$. After this solution generation step, the population has their domination ranks and crowding distances calculated. Then, NSGA-II sorts the population in an ascending order using a comparison operator called **crowded comparison operator** by Deb [86]. The sorting process executed by NSGA-II is called **non-dominant sorting**

The **crowded comparison operator** compares an in individual based on two criterions: a) the rank for a front generated by its solution, and b) the crowding distance for this individual.

This operator is defined as $>_c$ follows the following equation for two individuals $\boldsymbol{t}_i$ and $\boldsymbol{t}_j$:

$$\boldsymbol{t}_i >_c \boldsymbol{t}_j \text{ if } d_{ri} < d_{rj} \vee (d_{ri} = d_{rj} \wedge d_{ci} > d_{cj}) \tag{4.132}$$

Crowded comparison operator ($>_c$) considers that solutions in fronts with lower dominance ranks are considered better than those in fronts with higher ranks. If one or more solutions have the same dominance rank, i.e., make part of the same front, then these solutions are compared based on their crowding distance.

After the initialization stage, NSGA-II generates offspring individuals by using a binary tournament selection, a one-point crossover, and a random flip mutation genetic operators in the same manner in which GA (Section 4.3.1). However, the selection operation instead of only comparing a single objective, it compares individuals in the same way as the **non-dominant sorting** by using the **crowded comparison operator** $>_c$.

The population of offspring individuals $\boldsymbol{P}_{new}$ are then unified with their parent population $\boldsymbol{P}$ as in $\boldsymbol{P} = \boldsymbol{P} \cup \boldsymbol{P}_{new}$. The combined population has its individuals reassigned with new dominance ranks and crowding distances by using Algorithm 32 and Algorithm 33.

Since $|\boldsymbol{P}| > n$, $\boldsymbol{P}$ is sorted using the **non-dominate sorting** approach and has their individuals with higher dominance ranks and smaller crowding distances removed from the population until $|\boldsymbol{P}| = n$. This elitist process leaves solution that results in real vectors close to those of the PF for the multi-objective function at hand at the same time spread across in their resulting fronts.

As stated, NSGA-II uses the same genetic operators used by GA. For this reason, NSGA-II has the same two external static parameters as GA, namely, the crossover and the mutation rates p_c and p_m.

4.5.1.2 Parameters Description for NSGA-II

NSGA-II relies on **two external static** parameters p_m and p_c similar to GA.

4.5.1.3 Continuous NSGA-II

NSGA-II can be modified to be applied to continuous problems as well. This adjusted version, called CNSGA-II has the following differences: a) the solutions held by individuals are d-dimensional real vectors instead of integers in a population $\boldsymbol{P}_c$. b) the crossover over operations are changed to attend continuous solutions. NSGA-II is shown in pseudo-code form in Algorithm 34.

CNSGA-II uses a simulated binary crossover as presented in AGAv3 (Section 4.4.4) and it uses the following equations to generate its offsprings:

$$\beta = \begin{cases} (2r_1)^{\left(\frac{1}{d+1}\right)} & \text{if } r_1 > 0.5 \\ (2(1-r_1))^{\left(\frac{1}{d+1}\right)} & \text{otherwise} \end{cases}, \tag{4.133}$$

Algorithm 34 Continuous Non-dominant Sorting Genetic Algorithm (CNSGA-II)

INPUT: Objective Function ($\boldsymbol{F}(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of chromosomes (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
Mutation rate (p_m)
Crossover rate (p_c)
OUTPUT: Pareto Front ($\boldsymbol{PF}$)

procedure CNSGAII($\boldsymbol{F}, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}, p_m, p_c$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci} \rangle$
  Evaluate $\boldsymbol{F}(\boldsymbol{x}_i)$
  Obtain Dominance Ranks d_{ri} ▷ Alg. 32
  Obtain Crowding Distances d_{ci} ▷ Alg. 33
  Sort $\boldsymbol{P}$ *using* ***non-dominatant sorting***
  $\boldsymbol{PF} = \boldsymbol{S}_{F1}$
  for $k = 1$ to $I - 1$ **do**
    $\boldsymbol{P}_{cnew} = \emptyset$
    for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
      Binary Tournament Selection
      Simulated Binary Crossover
      Polynomial Mutation
      Evaluate generated offsprings
      $\boldsymbol{P}_{cnew} \leftarrow \boldsymbol{t}_{new} = \langle \boldsymbol{x}_{new1}, 0, 0 \rangle$
      $\boldsymbol{P}_{cnew} \leftarrow \boldsymbol{t}_{new} = \langle \boldsymbol{x}_{new1}, 0, 0 \rangle$
    end for
    $\boldsymbol{P}_c = \boldsymbol{P}_c \cup \boldsymbol{P}_{cnew}$
    Obtain Dominance Ranks d_{ri} ▷ Alg. 32
    Obtain Crowding Distances d_{ci} ▷ Alg. 33
    Sort $\boldsymbol{P}_c$ *using* ***non-dominatant sorting***
    Remove last individuals in $\boldsymbol{P}_c$ *until* $|\boldsymbol{P}_c| = n$
    $\boldsymbol{PF} = \boldsymbol{S}_{F1}$
  end for
  return $\boldsymbol{PF}$
end procedure

$$x_{o1j} = \begin{cases} \left(\max\left(lb_j, \min\left(ub_j, 0.5\left((1+\beta)x_{p1_j} + (1-\beta)x_{p2_j}\right)\right)\right)\right) & \text{if } r_2 < p_c \\ x_{p1_j} & \text{otherwise} \end{cases}, \tag{4.134}$$

$$x_{o2j} = \begin{cases} \left(\max\left(lb_j, \min\left(ub_j, 0.5\left((1+\beta)x_{p2_j} + (1-\beta)x_{p1_j}\right)\right)\right)\right) & \text{if } r_2 < p_c \\ x_{p2_j} & \text{otherwise} \end{cases}. \tag{4.135}$$

In the same manner, CNSGA-II uses a mutation operator suitable for continuous decision variables. CNSGA-II uses the polynomial mutation as present in AGAv3 (Section 4.4.4). This mutation operates in each component of an offspring solution using the following equation:

$$x_{j_{new}} = \begin{cases} \max\left(lb_j, \min\left(ub_j, x_j + (2r)^{\left(\frac{1}{d+1}\right)} - 0.5\right)\right) & \text{if } r < 0.5 \\ \max\left(lb_j, \min\left(ub_j, x_j + (2(1-r))^{\left(\frac{1}{d+1}\right)} - 0.5\right)\right) & \text{otherwise} \end{cases} \tag{4.136}$$

CNSGA-II has the same two external static parameters as NSGA-II, namely, the crossover and the mutation rates p_c and p_m.

4.5.2 Adaptive Parameter with Mutation Tournament Multi-Objective DE (APMTMODE)

4.5.2.1 Meta-Heuristic Description

APMTMODE, as defined by Santos [82] (in Portuguese), is a multi-objective DE-based algorithm that has **adaptive parameter control strategies** for the scaling factors and crossover rates. It also has **adaptive operator selection** mechanisms for dynamically select mutation and crossover operators, and in the pool of mutation operators, it contains a tournament-based mutation. APMTMODE is shown in pseudo-code form in Algorithm 35.

Algorithm 35 Adaptive Parameter Mutation Tournament Multi-Objective DE (APMTMODE)

INPUT: Objective Function ($\boldsymbol{F}(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of chromosomes (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Pareto Front ($\boldsymbol{PF}$)

procedure APMTMODE($\boldsymbol{F}, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci}, F_i, CR_i, mut_i, cro_i \rangle$ $(i = 1, ..., n)$
  Evaluate $\boldsymbol{F}(\boldsymbol{x}_i)$
  Obtain Dominance Ranks d_{ri} ▷ Alg. 32
  Obtain Crowding Distances d_{ci} ▷ Alg. 33
  $N_{mut}^{(0)} = \lceil \frac{n}{5} \rceil$
  $N_{cro}^{(0)} = \lceil \frac{n}{2} \rceil$
  $\mu_{wF}^{(0)} = \mu_{wF initial}$
  $\mu_{wCR}^{(0)} = \mu_{wCR initial}$
  Assign $mut_i (i = 1, ..., n)$
  Assign $cro_i (i = 1, ..., n)$
  Sort $\boldsymbol{P}$ *using* ***non-dominatant sorting***
  $\boldsymbol{S}_{PF} = \boldsymbol{S}_{F1}$
  for $k = 1$ to $I - 1$ **do**
    if $mod(k, I_f) = 0$ **then**
      Randomly generate m
    end if
    Update $\boldsymbol{x}_{worst}$ *the worst solution for* f_m
    $\boldsymbol{P}_{cnew} = \emptyset$
    Calculate σ ▷ Eq. 4.61
    Calculate μ_{wF} ▷ Eq. 4.141
    Calculate μ_{wCR} ▷ Eq. 4.142
    for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
      Generate CR_i
      Generate F_i
      mut_i*th Mutation*
      cro_i*th Crossover*
      $\boldsymbol{t}_{new} = \langle \boldsymbol{u}_i, 0, 0, F_i, CR_i, mut_i, cro_i \rangle$
      $\boldsymbol{P}_{cnew} \leftarrow \boldsymbol{t}_{new}$
    end for
    $\boldsymbol{P}_c = \boldsymbol{P}_c \cup \boldsymbol{P}_{cnew}$
    Obtain Dominance Ranks d_{ri} ▷ Alg. 32
    Obtain Crowding Distances d_{ci} ▷ Alg. 33
    Sort $\boldsymbol{P}_c$ *using* ***non-dominatant sorting***
    Remove last individuals in $\boldsymbol{P}_c$ *until* $|\boldsymbol{P}_c| = n$
    $\boldsymbol{S}_{PF} = \boldsymbol{S}_{F1}$
    Obtain Dominance Ranks d_{ri} *for* $\boldsymbol{S}_{PF}$ ▷ Alg. 32
    Obtain Crowding Distances d_{ci} *for* $\boldsymbol{S}_{PF}$ ▷ Alg. 33
    Sort $\boldsymbol{S}_{PF}$ *using* ***non-dominatant sorting***
    Remove last individuals in $\boldsymbol{S}_{PF}$ *until* $|\boldsymbol{S}_{PF}| = n$
  end for
  return $\boldsymbol{S}_{PF}$
end procedure

APMTMODE has a population of $\boldsymbol{P}_c$ of n individuals and each individual i is represented as a tuple $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci}, F_i, CR_i, mut_i, cro_i \rangle$ where $\boldsymbol{x}_i$ is a d-dimensional real vector that is a solution for a multi-objective function $\boldsymbol{F}$, d_{ri} is the dominance rank for the front formed by this solution image in the objective space, d_{ci} is the crowding distance for this solution image in the objective space, F_i is the scaling factor used by this individual, CR_i is this individual crossover rate, $mut_i^{(k)}$ and $cro_i^{(k)}$ are, respectively, the indices for a mutation and crossover strategies in the pool of mutation and crossover strategies used by APMTMODE.

APMTMODE design was inspired by CSASADE (Section 4.3.12) and, for this reason, APMTMODE adaptive mechanisms for mutation and crossover strategy selection and adaptive parameter control are identical for those present in CSASADE. However, it has modifications in its selection strategy to deal with multi-objective functions.

The pool of mutation strategies used in APMTMODE is defined as
$mut = \{rand/1, rand/2, current\text{-}to\text{-}best/2, tournment\text{-}based/1\}$ with indices mut_i varying in the range $[1, 4]$. These different mutation strategies are listed in the following order: (a) *rand/1* mutation (4.137) with index $mut_i = 1$ for individual i and $N_{rand/1}$ is the number of individuals using this operator; (b) *rand/2* mutation (4.138) with index $mut_i = 2$ for individual i and $N_{rand/2}$ is the number of individuals using this operator; (c) *current-to-best/2* mutation (4.139) with index $mut_i = 3$ for individual i and $N_{current\text{-}to\text{-}best/2}$ is the number of individuals using this operator; (d) *tournment-based/1* mutation (4.140) with index $mut_i = 4$ for individual i and $N_{tournment\text{-}based/1}$ is the number of individuals using this operator;

$$\boldsymbol{v}_i = \boldsymbol{x}_{r_1}^{(k)} + F_i^{(k)}(\boldsymbol{x}_{r_2}^{(k)} - \boldsymbol{x}_{r_3}^{(k)}). \tag{4.137}$$

$$\boldsymbol{v}_i = \boldsymbol{x}_{r_1}^{(k)} + F_i^{(k)}(\boldsymbol{x}_{r_2}^{(k)} - \boldsymbol{x}_{r_3}^{(k)} + \boldsymbol{x}_{r_4}^{(k)} - \boldsymbol{x}_{r_5}^{(k)}). \tag{4.138}$$

$$\boldsymbol{v}_i = \boldsymbol{x}_{r_1}^{(k)} + F_i^{(k)} \left(\boldsymbol{x}_{best}^{(k)} - \boldsymbol{x}_{r_2}^{(k)} + \boldsymbol{x}_{r_3}^{(k)} - \boldsymbol{x}_{r_4}^{(k)}\right). \tag{4.139}$$

$$\boldsymbol{v}_i = \boldsymbol{x}_{r_1}^{(k)} + F_i^{(k)} \left(\boldsymbol{x}_{twinner} - \boldsymbol{x}_i^{(k)}\right). \tag{4.140}$$

Similar to CSASADE and SOAMSDE (Section 4.4.1), APMTMODE pool of crossover strategies are $cro = \{bin, exp\}$ and.the integers $cro_i^{(k)}$ can assume the value $[1, 2]$ representing, respectively, the use of a **binary crossover strategy** as in Eq. 4.2 when the value is 1 and **exponential crossover strategy** as in Algorithm 16 when the value is 2.

In the initialization, APMTMODE generates, using random uniform distribution, n solutions $\boldsymbol{x}_i$ with components defined by the boundary vectors $\boldsymbol{ub}$ and $\boldsymbol{lb}$ and assign them to their respective individual. Afterwards, APMTMODE evaluates these initial solutions using the objective function $\boldsymbol{F}$ for then assign their dominance ranks d_{ri} and crowding distance d_{ci} using $ndFronts$ (defined in Algorithm 32) and $crwdDist$ (defined Algorithm 33) in the same fashion as NSGA-II (Section 4.5.1).

Still during the initialization, APMTMODE assign for each individual a crossover strategy index in a way that the first $\lfloor \frac{n}{2} \rfloor$ individuals use **binary crossover strategy** with $cro_i = 1$, and the last $\lceil \frac{n}{2} \rceil$ use **exponential crossover strategy** with $cro_i = 2$. The same approach is used to assign

the mutation indices where the first set of $\lfloor \frac{n}{4} \rfloor$ individuals using $rand/1$ mutation with $mut_i = 1$ until the fourth, and last, set of $\lceil \frac{n}{4} \rceil$ individuals using *tournament-based/1* mutation with $mut_i = 4$. At the end of the initialization stage, each ith individual is $\boldsymbol{t}_i = \langle \boldsymbol{x}^{(0)}, d_{ri}, d_{ci}, 0.5, 0.5, mut_i, cro_i \rangle$.

The **adaptative operation selection** for mutation strategies in APMTMODE reassign during each iteration the indices $mut_i^{(k)}$ by updating the number of individuals using each strategy in the same fashion as CSASADE by using Eq. 4.52. Another analogous **adaptative operation selection** in APMTMODE is used for crossover strategies selection in APMTMODE and it used in Eq. 4.56.

During the iterations, APMTMODE uses for each individual the mutation and crossover strategies assigned to them using the indices mut_i and cro_i. However, similar to SOAMSDE, instead of using the DE selection strategy to substitute an individual solution $\boldsymbol{x}_i$ with its trial-vector $\boldsymbol{u}_i$, APMTMODE creates a new individual represented by the tuple $\boldsymbol{t}_i = \langle \boldsymbol{u}_i, d_{ri}, d_{ci}, F_i, CR_i, mut_i, cro_i \rangle$ and add this tuple to the population of offsprings $\boldsymbol{P}_{cnew}$. After generate an offspring to each individual, APMTMODE merge both populations $\boldsymbol{P}_c = \boldsymbol{P}_c \cup \boldsymbol{P}_{cnew}$, and calculate for the new population their dominance ranks d_r and crowded distances d_c. To at last, sort this new population using the **crowded comparison operator** ($>_c$). After the **non-dominant sorting**, APMTMODE removes the last individual of $\boldsymbol{P}_c$ iteratively until $|\boldsymbol{P}_c| = n$.

APMTMODE also contains a set $\boldsymbol{S}_{PF}$ that holds the values for Pareto optimal solutions found by the algorithm. At the end of every iteration, APMTMODE adds the solutions with rank 1 from the population to $\boldsymbol{S}_{PF}$ calculates the non-dominant rank and crowded distances for solutions in $\boldsymbol{S}_{PF}$. Then APMTMODE sorts $\boldsymbol{S}_{PF}$ and remove the last individuals until $\boldsymbol{S}_{PF} = n$.

APMTMODE **adaptive parameter control strategies** to adjust the individuals parameters F_i and CR_i works in a similar fashion as CSASAMODE using Eq. 4.60 and Eq. 4.64 that in turn is controlled by **internal** parameters μ_{wF} and μ_{wCR}. However, since the goal is to optimize a multi-objective $\boldsymbol{F}$ composed of o objectives $\boldsymbol{F} = (f_1, ..., f_o)$, APMTMODE uses one of the objectives $m \in [1, o]$ chosen randomly, where the index m changes at every I_f iterations that is an **internal static** parameter.

The **adaptive parameter control strategy** to adjust μ_{wF} uses the following equation:

$$\mu_{wF}^{(k)} = \sum_{i=1}^{n} F_i \left(\frac{|f_m(\boldsymbol{x}_i^{(k)}) - f_m(\boldsymbol{x}_{worst}^{(k)})|}{\sum_{j=1}^{n} |f_m(\boldsymbol{x}_j^{(k)}) - f_m(\boldsymbol{x}_{worst}^{(k)})|} \right), \tag{4.141}$$

where F_i is the scaling factor for the ith individual, $f_m(\boldsymbol{x}_i)$ is the evaluation of objective $f_m \in \boldsymbol{F}$ for the individual i and $f_m(\boldsymbol{x}_{worst})$ is the evaluation for the same function but for the worst evaluation for function m. A similar approach is used for μ_{wCR} represented as follows:

$$\mu_{wCR}^{(k)} = \sum_{i=1}^{n} CR_i \left(\frac{|f_m(\boldsymbol{x}_i^{(k)}) - f_m(\boldsymbol{x}_{worst}^{(k)})|}{\sum_{j=1}^{n} |f_m(\boldsymbol{x}_j^{(k)}) - f_m(\boldsymbol{x}_{worst}^{(k)})|} \right). \tag{4.142}$$

4.5.2.2 Parameters Description for APMTMODE

APMTMODE has no **external static** parameters. APMTMODE contains the following fourteen **internal** parameters: (a) mutation rate for each ith individual F_i, (b) average weighting mutation rate μ_{wF}, (c) initial average weighting mutation rate $\mu_{wF_{initial}}$, (d) crossover rate for each ith individual CR_i, (e) average weighting crossover rate μ_{wCR}, (f) initial average weighting crossover rate $\mu_{wCR_{initial}}$, (g) standard deviation σ, (h) maximum standard deviation $\sigma_{max} = 0.5$, (i) minimum standard deviation $\sigma_{min} = 0.1$, (j) number of individuals using a crossover strategy N_{cro}, (k) number of individuals using a mutation strategy N_{mut}, (l) index for the ith individual crossover strategy cro_i, (m) index for the ith individual mutation strategy mut_i, and (n) number of iterations until the $I_f = 5$. APMTMODE has seven **adaptive parameter control strategies** that works analogous to those present in CSASADE.

4.6 Multi-Objective Optimization Bio-Inspired Meta-Heuristics Developed in this Work

4.6.1 Non-dominant Sorting Adaptive Genetic Algorithm (NSAGA)

4.6.1.1 Meta-Heuristic Description

NSAGA is a multi-objective adaptive GA meta-heuristic that combine features present in AGAv4 (Section 4.4.5) with components present in NSGA-II (Section 4.5.1). The contributions for NSAGA, when compared with NSGA-II, is that it does not need any form of parameter tuning from a user since it contains **parameter control strategies**. NSAGA also contains **adaptive operation selection** schemes to change the use of genetic operators during its execution dynamically. NSAGA contains the two **adaptive operator selection** from AGAv4 as well as all of its ten **parameter control strategies** present in AGAv4. The pseudo-code for NSAGA is present in Algorithm 36.

Algorithm 36 Non-dominant Sorting Adaptive GA(NSAGA)

INPUT: Objective Function ($f(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of components (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Best solution ($\boldsymbol{x}_{best}$)

procedure NSAGA($f, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci}, p_{mi}, p_{ci} cro_i, mut_i, p_{ti} \rangle$
  Evaluate $f(\boldsymbol{x}_i^{(0)})$
  Update $\boldsymbol{x}_{best}$
  Update $\boldsymbol{x}_{worst}$
  for $k = 1$ to $I - 1$ **do**
    Calculate σ_{p_m} ▷ Eq. 4.109
    Calculate σ_{p_x} ▷ Eq. 4.110
    Calculate μ_{wp_m} ▷ Eq. 4.106
    Calculate μ_{wp_c} ▷ Eq. 4.106
    Calculate p_{mi} ▷ Eq. 4.104
    Calculate p_{ci} ▷ Eq. 4.105
    Calculate T_{size} ▷ Eq. 4.111
    $\boldsymbol{P}_{new} = \emptyset$
    for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
      T_{size} *Tournament Selection Operator*
      if $f(\boldsymbol{x}_{p_1}) < f(\boldsymbol{x}_{p_2})$ **then**
        $p_c = p_{c p_1}$
        $p_m = p_{m p_1}$
        $cro = cro_{p_1}$
        $mut = mut_{p_1}$
        $p_t = p_{t p_1}$
      else
        $p_c = p_{c p_2}$
        $p_m = p_{m p_2}$
        $cro = cro_{p_2}$
        $mut = mut_{p_2}$
        $p_t = p_{t p_2}$
      end if
      cro th Crossover Operator using p_c
      mut th Mutation Operator using p_m
      Evaluate offsprings $\boldsymbol{x}_{new1}$ *and* $\boldsymbol{x}_{new2}$
      Calculate $p_{t_{new1}}$ *and* $p_{t_{new2}}$ ▷ Eq. 4.114
      $\boldsymbol{t}_{new1} = \langle \boldsymbol{x}_{new1}, p_m, p_c, cro, mut, p_{t_{new1}} \rangle$
      $\boldsymbol{t}_{new2} = \langle \boldsymbol{x}_{new2}, p_m, p_c, cro, mut, p_{t_{new2}} \rangle$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new1}$
      $\boldsymbol{P}_{new} \leftarrow \boldsymbol{t}_{new2}$
    end for
    Sort $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    $\boldsymbol{P}$ *is replaced by Best* n *in sorted* $\boldsymbol{P} \cup \boldsymbol{P}_{new}$
    Adapt Mutation Numbers ▷ Eq. 4.52
    Adapt Crossover Numbers ▷ Eq. 4.56
    for $i = 1$ to n **do**
      Reassign $mut_i^{(k)}$
      Reassign $cro_i^{(k)}$
    end for
    Update $\boldsymbol{x}_{best}$
    Update $\boldsymbol{x}_{worst}$
  end for
  return $\boldsymbol{x}_{best}$
end procedure

NSAGA has a population $\boldsymbol{P}$ that contains n individuals and each ith individual is represented as a tuple $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci}, p_{ci}, p_{mi}, cro_i, mut_i, p_{ti} \rangle$, where $\boldsymbol{x}_i$ is a d-dimensional integer vector that represents a solution for a COP multi-objective function $\boldsymbol{F}$, d_{ri} is the dominance rank for the front formed by this solution image in the objective space, d_{ci} is the crowding distance for this solution image in the objective space, p_{ci} is the crossover rate used in this individual chromosome, p_{mi} is the mutation rate used in this individual chromosome, cro_i is the index for one of the operators present in the pool of genetic crossover operators, mut_i is the index for one of the genetic mutation

operators, and p_{ti} is the tournament pressure for this individual.

During the initialization stage, NSAGA generates n random solutions inside of $\boldsymbol{F}$ search space and assign them as chromosomes $\boldsymbol{x}_i$ for each ith individual $\boldsymbol{t}_i$. For each $\boldsymbol{t}_i$ populates its components intially as $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, 0, 0, \mu_{p_c initial}, \mu_{p_m initial}, cro_i, mut_i, p_{ti} \rangle$, where p_{ti} is randomly generated during the initialization inside of the range $[p_{tmin}, p_{tmax}]$. The integer values cro_i, and mut_i are assigned as the indices of one of the crossover and mutation genetic operators present in NSAGA. NSAGA assignment for indices cro_i and mut_i work in the same fashion as AGAv4 and so is the method to initialize the parameters N_{mut} and N_{cro} for the pool of genetic operators $cro = \{one\text{-}point, sbx, two\text{-}point, uniform\}$ and $mut = \{bnormalr, cauchyr, normalr, poly, unifr\}$.

During the iterations, NSAGA updates the **internal** parameters that represent the mutation and crossover standard deviation for the generation of p_m and p_c, respectively, σ_{p_m} and σ_{p_c} using **deterministic parameter control strategies** as expressed in Eq. 4.109 and Eq. 4.109.

The process to update both p_m and p_c is expressed by Eq. 4.104 and Eq. 4.105 that uses respectively the internal parameters $\mu_{w_{p_m}}$ and $\mu_{w_{p_c}}$. The adaptive parameter strategies that calculate the values for $\mu_{w_{p_m}}$ and $\mu_{w_{p_c}}$ every iteration and for the usage of all population is expressed in Eq. 4.106 Eq. 4.107 that calculates a weighting factor w_i for each individual in the population. The first difference in the adaptive mechanisms of NSAGA and AGAv4 is in the process to calculate w_i that is calculated by NSAGAv4 as follows:

$$w_i = \frac{||\boldsymbol{F}(\boldsymbol{x}_i) - (f_1(\boldsymbol{x}_{worst\_1}), ..., f_o(\boldsymbol{x}_{worst\_1}))||}{\sum_{l=1}^{n} ||\boldsymbol{F}(\boldsymbol{x}_l) - (f_1(\boldsymbol{x}_{worst\_1}), ..., f_o(\boldsymbol{x}_{worst\_1}))||}, \tag{4.143}$$

where $\boldsymbol{F}(\boldsymbol{x}_i)$ is the vector of objective evaluations $\boldsymbol{F}$ for $\boldsymbol{x}_i$, and $f_1(\boldsymbol{x}_{worst\_1})$ to $f_o(\boldsymbol{x}_{worst\_o})$ represents the worst evaluations for solutions of each one of the o objectives that forms $\boldsymbol{F}$. The weighting factors take into consideration the values for all objectives to estimate the improvement in the performance of different mutation and crossover probabilities.

Every iteration during the adjustment of N_{mut} and N_{cro}, NSAGA uses Eq. 4.52 and Eq. 4.56. The expected variation $\Delta_{N_{mut}}$ and $\Delta_{N_{cro}}$ in both this equations use s_{mut} and s_{cro} that are both sums of the distances in the objective space of individuals that uses a class of operator and the worst point possible composed by the worst evaluation for each one of the o objectives used. Both $\Delta_{N_{mut}}$ and $\Delta_{N_{cro}}$ use, respectively, s_{mut_all} and s_{cro_all} that represents the total sum of all distances.

The values for s_{mut} is calculated as follows:

$$s_{mut} = \sum_{\forall \boldsymbol{x}_i^{(k)} \in \{\boldsymbol{x}_i \in \boldsymbol{P} \,:\, mut_i = mut\}} ||\boldsymbol{F}(\boldsymbol{x}_i) - (f_1(\boldsymbol{x}_{worst\_1}), ..., f_o(\boldsymbol{x}_{worst\_1}))||. \tag{4.144}$$

Similarly, s_{cro} is calculated using the following equation:

$$s_{cro} = \sum_{\forall \boldsymbol{x}_i^{(k)} \in \{\boldsymbol{x}_i \in \boldsymbol{P} \,:\, cro_i = cro\}} ||\boldsymbol{F}(\boldsymbol{x}_i) - (f_1(\boldsymbol{x}_{worst\_1}), ..., f_o(\boldsymbol{x}_{worst\_1}))||. \tag{4.145}$$

After the calculation of each s_{mut} and s_{cro}, it is possible to obtain the values for s_{all_mut} and s_{all_cro} in the following equations:

$$s_{all\_mut} = s_{bnormalr} + s_{cauchyr} + s_{normalr} + s_{poly} + s_{unifr}. \tag{4.146}$$

$$s_{all\_cro} = s_{one\text{-}point} + s_{two\text{-}point} + s_{sbx} + s_{uniform} + s_{unifr}. \tag{4.147}$$

During the reproductive process, NSAGA uses a tournament with size of T_{size} to select a pair of parents $\boldsymbol{t}_{p1}$ and $\boldsymbol{t}_{p2}$. Then the parent that with a solution that dominates the other has its parameters used to generate the offsprings. In other words, if $\boldsymbol{F}(\boldsymbol{x}_{p1}) \preceq \boldsymbol{F}(\boldsymbol{x}_{p2})$, then the genetic operators and probabilities held by the parent $\boldsymbol{t}_{p1}$ is used. Otherwise, parent $\boldsymbol{t}_{p2}$ is the one selected.

4.6.1.2 Parameters Description for NSAGA

NSAGA has no **external static** parameters. However, it contains *twenty-six* internal parameters listed as follows: (a) mutation probability for the ith individual p_{m_i}, (b) crossover probability for the ith individual p_{c_i}, (c) average crossover probability $\mu_{p_c}^{(k)}$ at iteration k, (d) initial average crossover probability $\mu_{p_c initial} = 0.7$, (e) average mutation probability $\mu_{p_m}^{(k)}$, (f) initial average mutation probability $\mu_{p_m initial} = 0.05$, (g) minimum crossover probability $p_{c_{min}} = 0.1$, (h) maximum crossover probability $p_{c_{max}} = 0.95$, (i) minimum mutation probability $p_{m_{min}} = 0.01$, (j) maximum mutation probability $p_{m_{max}} = 0.1$, (l) minimum crossover probability standard deviation $\sigma_{p_c min} = 0.1$, (m) maximum crossover probability standard deviation $\sigma_{p_c max} = 0.4$, (n) minimum mutation probability standard deviation $\sigma_{p_m min} = 0.01$, (o) maximum mutation probability standard deviation $\sigma_{p_m max} = 0.05$. (p) number of individuals using a specific mutation operator N_{mut}, (q) number of individuals using a specific crossover operator N_{cro}, (r) index for the mutation operator used by individual i mut_i, (s) index for the mutation operator used by individual i cro_i. (t) pressure for individual i p_t, (u) tournament size T_{size}, (v) minimum possible tournament size $T_{min} = 2$, (w) maximum possible tournament size $T_{max} = \lfloor \frac{n}{3} \rfloor$, (x) minimum pressure value for individuals $p_{t_{min}} = 0.08$, (y) maximum pressure value for individuals $p_{t_{max}} = 1.0$, and (z) learning rate for the pressure adaptation $\gamma = 0.3$. From this list of **internal** parameters fifteen of them are **static**.

NSAGA has two **adaptive operation selection** mechanisms that controls the use of operators by the individuals based on the values of the **internal** parameters N_{mut} and N_{cro}. NSAGA uses ten **parameter control strategies**. In the same fashion as AGAv4, NSAGA has seven **adaptive parameter control strategies** to configure the values for the internal parameters p_{m_i}, p_{c_i}, $\mu_{p_m}^{(k)}$, $\mu_{p_c}^{(k)}$, N_{mut}, N_{cro}, and p_{t_i}. NSGA also has three deterministic parameter control strategies to adjust the following parameters σ_{p_m}, σ_{p_c} and T_{size}.

4.6.2 Multi-Objective Non-dominant Sorting Adaptive DE (MONSADE)

4.6.2.1 Meta-Heuristic Description

MONSADE is a multi-objective version of SOAMSDE (Section 4.4.1) that contains modifications to be able to optimize a multi-objective function $\boldsymbol{F}$ that contains o objectives. MONSADE contains the same adaptive mechanisms present in SOAMSDE including six **parameter control strategies** to adjust the crossover rate and scaling factor for each individual, and two **adaptive operator selection** schemes to modify the crossover and mutation strategies used independen-

tly by each individual. The difference between MONSADE and SOAMSDE is the presence of a **non-dominant sorting** process instead of the typical sorting using a single-objective function evaluation as well as the adaptive mechanisms that alter the values for μ_{wF} and μ_{wCR}.

Algorithm 37 Multi-Objective Non-dominant Sorting DE (MONSADE)

INPUT: Objective Function ($\boldsymbol{F}(\cdot)$)
Iterations (I)
Number of individuals (n)
Number of chromosomes (d)
Lower-bound vector ($\boldsymbol{lb}$)
Upper-bound vector ($\boldsymbol{ub}$)
OUTPUT: Pareto Front ($\boldsymbol{PF}$)

procedure MONSADE($\boldsymbol{F}, I, n, d, \boldsymbol{lb}, \boldsymbol{ub}$)
  Initialize $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci}, F_i, CR_i, mut_i, cro_i \rangle$ $(i = 1, ..., n)$
  Evaluate $\boldsymbol{F}(\boldsymbol{x}_i)$
  Obtain Dominance Ranks d_{ri} ▷ Alg. 32
  Obtain Crowding Distances d_{ci} ▷ Alg. 33
  $N_{mut}^{(0)} = \lceil \frac{n}{7} \rceil$
  $N_{cro}^{(0)} = \lceil \frac{n}{2} \rceil$
  $\mu_{wF}^{(0)} = \mu_{wFinitial}$
  $\mu_{wCR}^{(0)} = \mu_{wCRinitial}$
  Assign $mut_i (i = 1, ..., n)$
  Assign $cro_i (i = 1, ..., n)$
  Sort $\boldsymbol{P}$ *using* ***non-dominatant sorting***
  $\boldsymbol{PF} = \boldsymbol{S}_{F1}$
  for $k = 1$ to $I - 1$ **do**
    Update $\boldsymbol{x}_{worst\_m}$ *the worst solution for* f_m
    $m = (1, ..., o)$
    $\boldsymbol{P}_{cnew} = \emptyset$
    Calculate σ ▷ Eq. 4.61
    Calculate μ_{wF} ▷ Eq. 4.62
    Calculate μ_{wCR} ▷ Eq. 4.65
    for $i = 1$ to $\lfloor \frac{n}{2} \rfloor$ **do**
      Generate CR_i
      Generate F_i
      mut$_i$*th Mutation*
      cro$_i$*th Crossover*
      $\boldsymbol{t}_{new} = \langle \boldsymbol{u}_i, 0, 0, F_i, CR_i, mut_i, cro_i \rangle$
      $\boldsymbol{P}_{cnew} \leftarrow \boldsymbol{t}_{new}$
    end for
    $\boldsymbol{P}_c = \boldsymbol{P}_c \cup \boldsymbol{P}_{cnew}$
    Obtain Dominance Ranks d_{ri} ▷ Alg. 32
    Obtain Crowding Distances d_{ci} ▷ Alg. 33
    Sort $\boldsymbol{P}_c$ *using* ***non-dominatant sorting***
    Remove individuals in $\boldsymbol{P}_c$ *until* $|\boldsymbol{P}_c| = n$
    $\boldsymbol{PF} = \boldsymbol{S}_{F1}$
  end for
  return $\boldsymbol{PF}$
end procedure

Similar to APMTMODE (Section 4.5.2), MONSADE has its individuals represented as tuples and each ith tuple is expressed as $\boldsymbol{t}_i = \langle \boldsymbol{x}_i, d_{ri}, d_{ci}, F_i, CR_i, mut_i, cro_i \rangle$ where $\boldsymbol{x}_i$ is a d-dimensional real vector that is a solution for a multi-objective function $\boldsymbol{F}$, d_{ri} is the dominance rank for the front formed by this solution image in the objective space, d_{ci} is the crowding distance for this solution image in the objective space, F_i is the scaling factor used by this individual, CR_i is this individual crossover rate, $mut_i^{(k)}$ and $cro_i^{(k)}$ are, respectively, the indices for a mutation and crossover strategies in the pool of mutation and crossover strategies used by MONSADE and SOAMSDE.

The adaptive parameter control strategies to modify μ_{wF} μ_{wCR} as expressed in Eq. 4.60 and Eq. 4.64 uses a weighting factor w_i for each ith individual $\boldsymbol{t}_i$ calculated using the following

equation:

$$w_i = \frac{||\boldsymbol{F}(\boldsymbol{x}_i) - (f_1(\boldsymbol{x}_{worst\_1}), ..., f_o(\boldsymbol{x}_{worst\_1}))||}{\sum_{l=1}^{n} ||\boldsymbol{F}(\boldsymbol{x}_l) - (f_1(\boldsymbol{x}_{worst\_1}), ..., f_o(\boldsymbol{x}_{worst\_1}))||}, \tag{4.148}$$

where $\boldsymbol{F}(\boldsymbol{x}_i)$ is the vector of objective evaluations $\boldsymbol{F}$ for $\boldsymbol{x}_i$, and $f_1(\boldsymbol{x}_{worst\_1})$ to $f_o(\boldsymbol{x}_{worst\_o})$ represents the worst evaluations for solutions for all o objectives that composes $\boldsymbol{F}$. So instead of using only an objective, MONSADE considers all objectives being optimized to calculate the weighting factors to update the crossover rates and scaling factors.

4.6.2.2 Parameters Description for MONSADE

Since MONSADE is just the multi-objective version of SOAMSDE, it contains the same thirteen **internal** parameters, five **parameter control strategies** and two **adaptive operator selection** schemes. MONSADE parameters are listed as follows: (a) F_i, (b) μ_{wF}, (c) CR_i, (d) μ_{wCR}, (e) σ, (f) $\mu_{wFinitial}$, (g) $\mu_{wCRinitial}$, (h) σ_{max}, (i) σ_{min}, (j) N_{mut}, (k) N_{cro}, (l) mut_i, and (m) cro_i.

4.7 Conclusions of the Chapter

This chapter presents the reasoning for the use of bio-inspired meta-heuristics, including a taxonomy about these methods. Beyond that, this work also presents the theory and nomenclature used to define adaptive techniques used to control the exploratory and exploitative behavior of different classes of meta-heuristics.

At last, as an essential part of this chapter, there is an intensive presentation for all bio-inspired meta-heuristics used in this work, including the ones developed by the author.

In the list of meta-heuristics present in this chapter the following ones were implemented directly from their descriptions present in related literature works: (a) GA (Section 4.3.1), (b) DE (Section 4.3.2), (c) PSO (Section 4.3.3), (d) SSA (Section 4.3.4), (e) GWO (Section 4.3.5), (f) EHO (Section 4.3.6), (g) DA (Section 4.3.7), (h) MFO (Section 4.3.8), (i) WOA (Section 4.3.9), (j) BA (Section 4.3.10), (k) JADE (Section 4.3.11), (l) CSASADE (Section 4.3.12), (m) DPSO (Section 4.3.13), (n) SAPSO (Section 4.3.14), (o) HDPSO-M (Section 4.3.15), (p) NSGA-II (Section 4.5.1), and (q) APMTMODE (Section 4.5.2).

The following meta-heuristics were developed and create in this work: (a) SOAMSDE (Section 4.4.1), (b) AGAv1 (Section 4.4.2), (c) AGAv2 (Section 4.4.3), (d) AGAv3 (Section 4.4.4), (e) AGAv4 (Section 4.4.5), (f) APSO (Section 4.4.6), (g) APSOv2 (Section 4.4.7), (h) HDPSO-U (Section 4.4.8), (i) AHDPSO-U (Section 4.4.9), (j) AGWO (Section 4.3.5), (k) NSAGA (Section 4.6.1), and (l) MONSADE (Section 4.6.2).

The meta-heuristics algorithms present in this chapter are used in Chapter 7 that includes the experimental setup for the application of these multiple meta-heuristics and compare their performance when optimizing benchmark problems present in Chapter 2 as well as the task mapping of real-applications onto MPSoC platforms based on NoCs as present in Chapter 3.

5 RELATED WORKS

This chapter introduces some of the related works for this dissertation from the rich literature of both NoC and bio-inspired meta-heuristics with emphasis with the ones that uses meta-heuristics algorithms to solve the task mapping problem of real-time applications onto MPSoC baed on NoCs. The chapter is divided as follows: Section 5.1 introduces the chapter, Section 5.2 presents related works in the field of meta-heuristics highlighting some of the similar adaptive techniques present in literature, Section 5.3 presents related works for the task mapping problem of RTA onto MPSoC platforms, Section 5.4 concludes this chapter.

5.1 Introduction

Both fields of bio-inspired meta-heuristic algorithms and network-on-a-chip (NoC) are very active with a regular stream of works that adds new methodologies for their advancement. This chapter intends to present the reader with an overview of both fields without being an extensive survey. It exhibits to the reader related works while pointing to an interested reader other sources that could be used along with this work. Themes focus upon in related works are similar meta-heuristics as the ones developed in this work, as well as similar alternative approaches for task mapping problem in the context of RTNoC.

5.2 Search-Based Meta-Heuristics Algorithms

The field of search-based meta-heuristic is current a popular one, and researches are continually presenting new meta-heuristics or modified versions of others that already exist. The primary theoretical basis for this intense interest by the research community is the NFL theorem [58], since it states that algorithms there is no universal optimal performance meta-heuristic. Instead, their good performance is limited to a family of problems. So new methods always have at least its own niche of problems that it responds better than other algorithms.

Search-space optimization meta-heuristics are not limited to metaphors based on biological phenomena. The survey [18] briefly presents meta-heuristic based on natural non-linear phenomena such as vortex in fluids or fractal formation. This survey also singles out different classes of heuristic operators shared by many of these non-linear phenomena based meta-heuristics.

The survey in [7] has an intensive study to classify both the current trends as well as the possible future ones for the field of bio-inspired meta-heuristics. In the list of topics emphasized as open

areas in the meta-heuristics field, the use of adaptive techniques such as the one extensively studied in this work is one of them. This survey also presents a call for standardization of benchmark problems and methodologies used for comparison between new and old meta-heuristics. For this reason, the survey also presents the necessity of statistical tests to overcome the random-nature of these meta-heuristics. Figure 5.1 was extracted from the survey in [7], and it shows the timeline history for the development of meta-heuristic in the past decade.

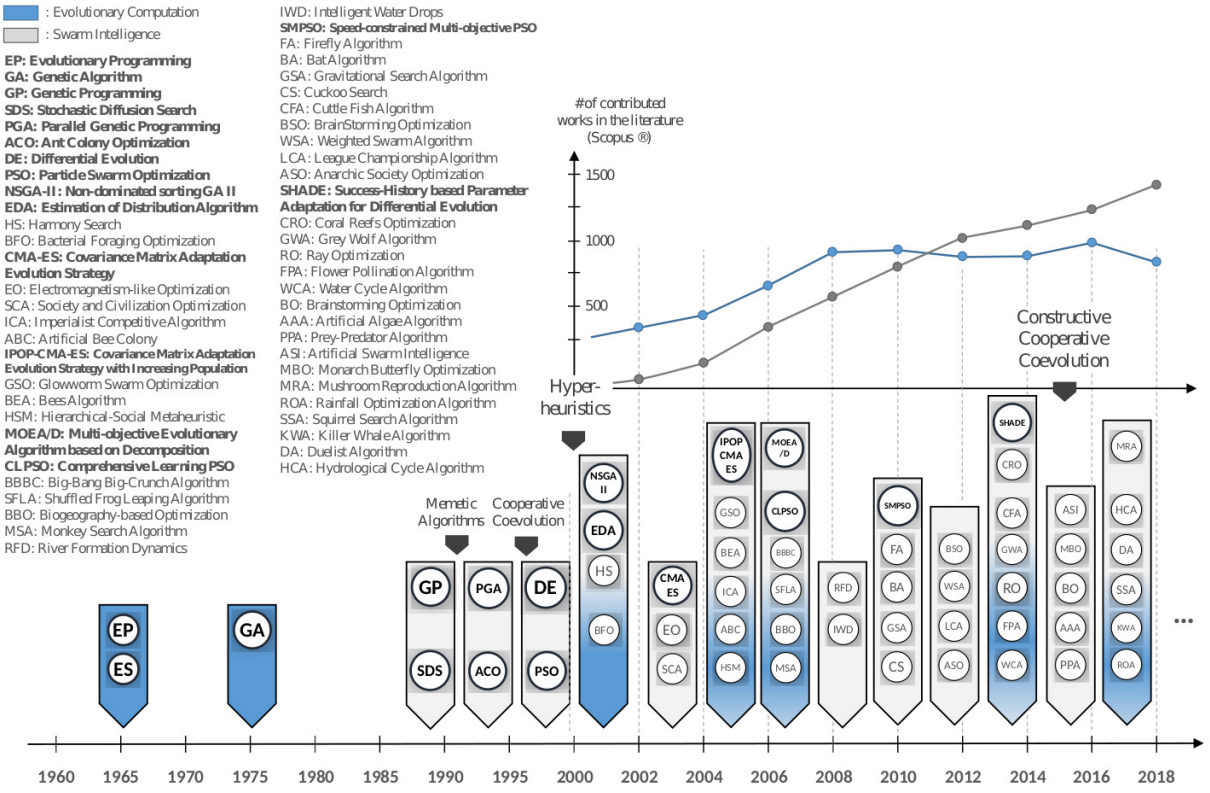


Figura 5.1 – Illustration of the timeline history for meta-heuristics developed in the past years (extracted from [7]).

5.2.1 Adaptive Techniques

The concepts of adaptive techniques for parameter control in meta-heuristics started with a focus on Evolution Strategies (ES), and in a way always existed in the field of bio-inspired meta-heuristics since the critical role that parameters values play in these algorithms performance. The concepts of adaptive techniques start to take shape and be classified as presented in Section 4.2 in the work [59], and more recently revisited in the [60]. In both these works, there is a classification for adaptive techniques. It also presents the possible advantages that meta-heuristics that contain adaptive mechanisms have that their counterparts do not include, for example, the fact these meta-heuristics does not need parameter tuning and their improvement convergence in many cases.

The following works contain bio-inspired meta-heuristics that incorporate adaptive techniques

to control their parameter values and heuristic operations other than the ones present in Chapter 4.

- The work in [87] presents a comparative experiment of different variations of DE against multiple variants of a "self-adaptive" DE meta-heuristic called SADE that contains adaptive parameter control strategies. The results obtained in this work show that SADE variants are capable of obtaining better results than those present by DE with lower convergence speeds, due to the exploration of possible parameters by SADE variants.
- The work in [65] presents a version for the Success-History Adaptive DE (SHADE) that apart for its adaptive parameter control mechanisms also contain a deterministic parameter control strategy to reduce the size of the population as the algorithm executes.
- The work in [88] presents an adaptive version of DE called SLADE that controls both parameter values and mutation strategies in a similar fashion as CSASADE [77] (Section 4.3.12). However, this algorithm also contains a different technique to randomly initialize the population of search agents in the search space.
- The work in [89] presents an adaptive version of PSO that contains an adaptive parameter control mechanism only for the inertial weight and works similarly as the one present in APSOv2 (Section 4.4.7). Another addition in this version of APSO is an adaptive operation selection to decide the neighborhood topology used by PSO. For example, in this work, PSO only uses the All topology in which all particles have access to all other particles information for the best solution through the use of $\boldsymbol{x}_{best}$ solution.
- The work in [90] presents an adaptive version of PSO that uses a fuzzy logic as adaptive parameter control for the cognitive and social factors in PSO. The decision for these parameter values uses as evidence for change the difference of evaluation between different iterations.

Table 5.1 presents information about the quantity of external static parameters (ESP) and adaptive techniques for the single-objective adaptive bio-inspired meta-heuristics present in these related works compared to the meta-heuristics developed in this work. The acronyms for adaptive techniques are: (**AOS**) Adaptive Operator Selection, (**DPCS**) Deterministic Parameter Control Strategy, (**APCS**) Adaptive Parameter Control Strategy, and (**SAPCS**) Self-Adaptive Parameter Control Strategy.

5.3 Task Mapping Problem for RTNoC-based MPSoC

The survey in [4] presents a definition for RTNoC as well as a classification for the different design choices that such a system may contain. It also presents a section discussing techniques of evaluation of RTNoC design, pointing out the differences and advantages obtained in static analytical such as the ones used in this work when compared with simulation-based ones. Since these types of analyses allow quick evaluation of multiple designs in the same period in which a simulation-based evaluation would only perform one.

Tabela 5.1 – Quantity of adaptation schemes in these related works.

Reference	Meta-Heuristic	ESP	DPCS	APCS	SAPCS	AOS
[87]	SADE	0	0	2	0	0
[65]	LSHADE	0	0	5	0	0
[88]	SLADE	0	0	5	0	1
[89]	NTAPSO	3	0	1	0	1
[90]	AFPSO	3	1	2	0	0
Section 4.4.1	SOAMSDE	0	1	6	0	2
Section 4.4.2	AGAv1	0	0	4	0	0
Section 4.4.3	AGAv2	0	2	4	0	0
Section 4.4.4	AGAv3	0	2	6	0	2
Section 4.4.5	AGAv3	0	2	8	0	2
Section 4.4.6	APSO	2	1	2	0	0
Section 4.4.7	APSOv2	0	0	3	0	0
Section 4.4.8	HDPSO-U	1	0	0	0	0
Section 4.4.9	AHDPSO-U	0	0	2	0	0
Section 4.4.10	AGWO	0	0	3	0	0

The work in [12] presents a survey for the problem of placing tasks from an application onto their respective PEs inside of an MPSoC platform. In this survey, the authors categorized different approaches for efficient task mapping emphasizing the multiple goals expected to be met by these task placement solutions including, for example, resources usage, timing performance, and power consumption. The categories are two: (a) design-time approaches in which the task mapping is performed during the system's design, and (b) run-time approaches in which the task placement is performed during the system's usage. Another interesting point drawn in this work is that this classification targets both homogeneous and heterogeneous systems.

The work in [91] presents another survey for the task mapping problem in NoC-based MPSoCs. However, the type of task mapping problems is closer to the problem of defining PEs placement onto NoC tiles than the placement of tasks onto processors. The authors also present their classification for possible approaches for this problem, including static and dynamic ones. However, this work also highlights in static task mapping problems the possibility of using meta-heuristics algorithms as an approach to tackle the task mapping, including GA, PSO, and Ant Colony Optimization (ACO). This work also contains a comparison for the results of the different task mapping approaches using a set of benchmark applications being mapped onto MPSoC systems.

The work in [92] present an approach that uses a multi-objective GA meta-heuristic to search for solutions mapping of applications onto NoC-based MPSoCs. These solutions improve both the system performance as well as the consumed power. The evaluation of different mappings is performed using a simulation-based approach.

The work in [93] presents an approach that uses an NSGA-II meta-heuristic allied with a simulation-based evaluation of task mapping of a real-time application into an RTNoC-based MPSoC. The goal is to find non-dominant solutions that are capable of optimizing both the

timing restrictions, dissipated energy by system's modules, and router buffer sizes.

The work in [85] presents a heuristic to map real-time application onto MPSoCs based on NoC that has operators that exploit knowledge about the problem to quickly find solutions as good as the ones obtained by the GA. This heuristic first maps tasks into bins and then map these bins to processor cores.

The work in [52] presents an NSGA-II based multi-objective bio-inspired meta-heuristic to search for non-dominant solutions for task mappings that at the same time optimize the time requirements for the system in the form of the schedulability of tasks as well the communication architecture power dissipation.

5.4 Conclusions of the Chapter

This chapter briefly presents a list of related works that include at least one of the main two topics in this work: (a) Bio-inspired meta-heuristic, and (b) Mapping of applications on RTNoCs.

In the works related to bio-inspired meta-heuristic, the focus is upon the ones that contain information about adaptive techniques and meta-heuristics that use them.

In the context of mapping application on RTNoCs, the list of works present either alternative for the evaluation of the system or present different approaches to tackle the problem such as the use of specialized heuristics or search-based meta-heuristics as done in this work.

6 SEARCH-BASED OPTIMIZATION META-HEURISTIC FRAMEWORK

This chapter briefly introduced the software implemented in this work. This chapter is divided as follows: Section 6.1 introduces this chapter including a contextualization of other similar works; Section 6.2 presents the designed object-oriented for the framework; Section 6.3 presents a short end-user tutorial for some of the modules in the framework; Section 6.4 concludes this chapter contextualizing it to other parts of this work.

6.1 Introduction

At the beginning of this work development, as the author studied meta-heuristics and possible experimental setup for their usage, the author noticed the necessity for a software framework for algorithms of these type. A framework capable of reliably and quickly execute meta-heuristics and perform static analysis for multiple RTNoCs task mappings. A characteristic that MATLAB options at the time were not capable of having. For this reason, during this work development, the author has implemented a software framework in C++11 that contains implementations for the recent bio-inspired meta-heuristics as well as modules necessary to process RTNoC analysis.

It was also important to consider other alternatives to software frameworks in the literature that works similarly. These other alternatives are:
(a) JMetal [94]: it is a meta-heuristic object-oriented framework in Java that contains the implementation of single-/multi-objective meta-heuristics including for example NSGA-II, GA, PSO, and DE. However, it lacks many of the new swarm intelligence based meta-heuristics, and since the goal is to use these meta-heuristics high-performance experimentation setting, a C++ framework is more desired;
(b) Shark Machine Learning [95]: it is a machine-learning framework in C++ that contains implementations of evolutionary strategy (ES) that is a type of bio-inspired evolutionary meta-heuristics;
(c) GA Lib [96]: it is a framework in C++ that contains implementation for GA.

6.2 Developed Software Architecture and Design

This work focuses on employing bio-inspired meta-heuristics in identifying a task mapping placement for real-time applications onto RTNoC-based MPSoC. This project software was implemented in C++11 only using the Standard Template Library (STL) to easily employ the software

in platforms based on Linux and Windows.

The software platform design uses an object-oriented architecture to facilitate the development and addition of new meta-heuristics as well as new objective functions.

Figure 6.1 and Figure 6.2 displays a simplistic overview in UML diagrams depicting, respectively, the architecture for the meta-heuristics and objective functions. Class *Algorithm* is responsible to solve an objective function *CostCalculator*. Both of these classes are abstract ones and contain the structure in which possible meta-heuristics and objective function have to have in the software framework architecture.

Class *Algorithm* has a virtual function to execute the meta-heuristic optimizing the objective function defined by function *setCostCalculator*. In order to be called, *execute* needs for its *Algorithm* to be configured via function *configuration*, and, once "everything" is ready for the execution, function *isReady* returns true. Once *execute* has been called and the optimization has been done, *execute* returns a Boolean value, and if true, *Algorithm* can save the results in a file via function *saveResultsInFile*. These saved results are related to the best solution found by the meta-heuristic and its convergence history of optimization. In other words, it saves the best solution, and evaluation found during each iteration.

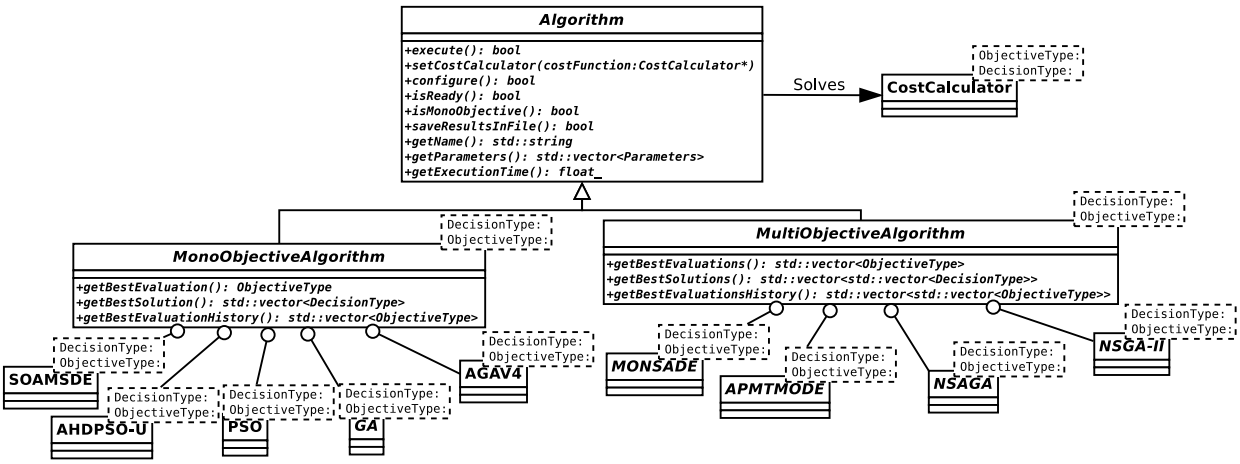


Figura 6.1 – Illustration of meta-heuristics object-oriented architecture.

Class *Algorithm* is inherited by two other abstract classes, namely, *MonoObjectiveAlgorithm* and *MultiObjectiveAlgorithm*. These two classes specialize in the optimization of single- and multi-objective problems, respectively. All of the classes representing meta-heuristics implements and inherit from either one of these two classes, *MonoObjectiveAlgorithm* and *MultiObjectiveAlgorithm*, depending on if the meta-heuristic is single- or multi-objective focused. To facilitate and introduce flexibility for the framework and developers, most of classes implementation uses templates to alternate between the types used for decision variables and objectives, respectively, *DecisionType* and *ObjectiveType*.

Class *CostCalculator* is responsible to implement a single-/multi-objective function to evaluate a solution that is implemented as a vector of components of the type *DecisionType* returning a vector of objective functions results with the type *ObjectiveType*. If the optimization problem is

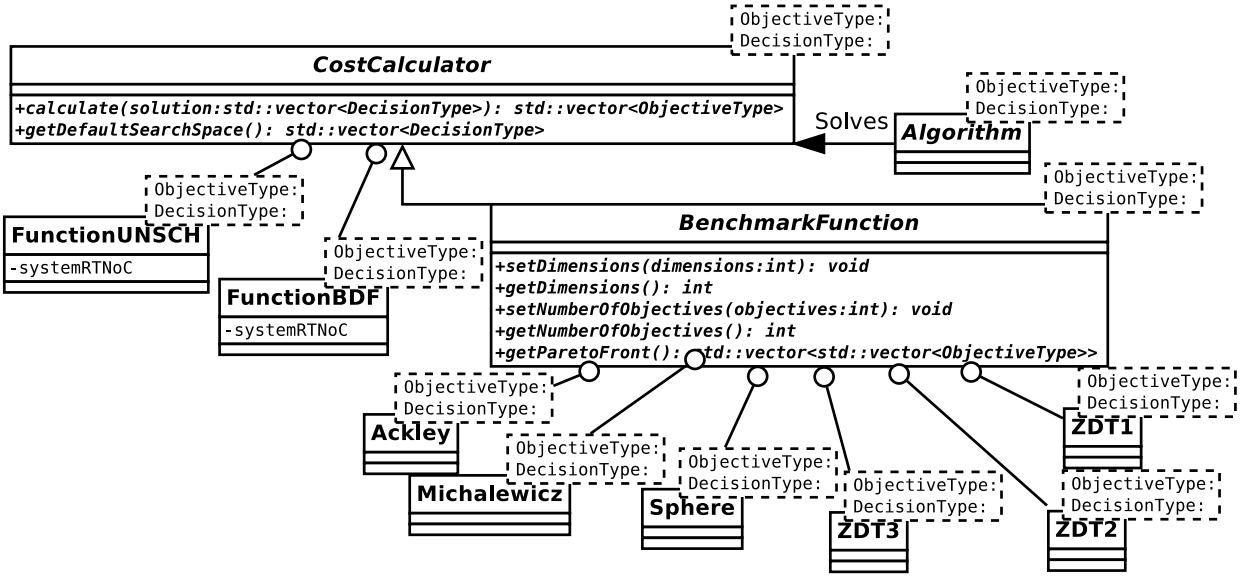


Figura 6.2 – Illustration of objective optimization problem object-oriented architecture.

single-objective, the vector of objective results has only one result.

Since benchmark functions have the unique property to have its best evaluation or PF known, they are implemented in its specialized class *BenchmarkFunction*. Most of the implemented benchmark functions also can alter their dimensions and objective functions, for example, *ZDT1*. For this reason, *BenchmarkCostCalculator* has *setDimensions* and *setNumberOfObjectives* to modify the number of dimensions and objectives.

The utilization of the framework for the objective function enables the use of the metaheuristics to optimize specific problems by wrapping them using a class that implements the abstract class *CostCalculator*. An example of that is present in the objective functions *FunctionUNSCH* and *FunctionBDF*. These functions contain a class *SystemRTNoC* that is responsible for performing a static analysis of an MPSoC platform given a task placement for an RTA model. This class is represented in Figure 6.3, together with its component classes in a simplistic UML diagram.

6.3 End-user Tutorial

For an end-user, it is crucial to demonstrate how to compile the framework in a Linux-based platform, the examples demonstrate here, a computer running Ubuntu 18.04. First, it is required that system to run a C++ compiler that supports the use of C++11. An example of C++ compiler that provides this feature is the GCC C++ compiler [97] widely used in Linux-based platforms for being an open-source and free software. In a platform running Ubuntu 18.04 that has *apt* package manager, the user needs to install GCC using the following command in terminal:

```
\$ sudo apt install gcc libstdc++
```

The other requirement is the use of CMake 3.0 [98] that is a cross-platform software to aid the

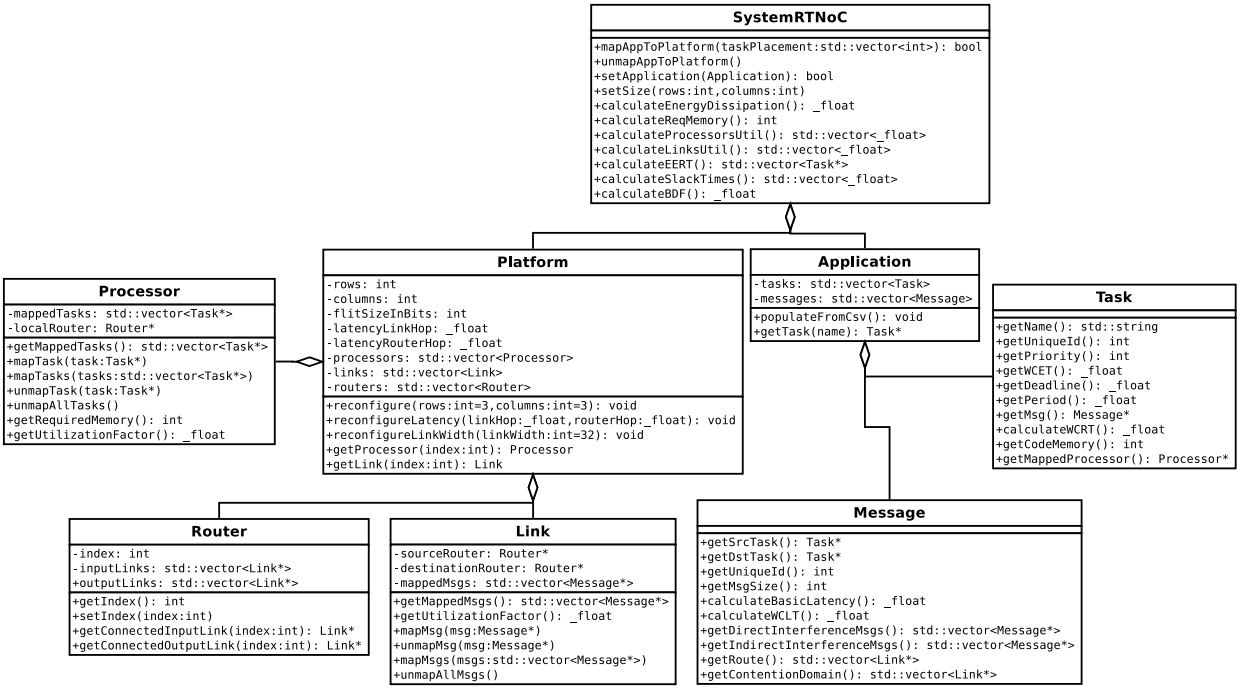


Figura 6.3 – Illustration of the RTNoC system framework in object-oriented architecture.

generation of a *Makefile* specific for the target platform. In the example platform running Ubuntu 18.04, CMake can either be installed from its source code or directly from the package manager repository using the following command:

```
\$ sudo apt install cmake
```

To generate the *Makefile* using *cmake*, the user needs to go to the folder in which the framework is present and run the following command:

```
\$ cmake .
```

If there are no errors, CMake must have generated a *Makefile* that can be used to compile the source code using the following command:

```
\$ make
```

If the compilation has been completed without any errors, the folder should now contain the following binary files holding the multiple experiment modules for the software developed:
(a) *benchmarkexperiment*,
(b) *mobenchmarkexperiment*,
(c) *appgenerator*,
(d) *nocexperiment*,
(e) *monocexperiment*.
These binary files are responsible for the experiment modules in the software.

6.3.1 Perform Single-Objective Meta-Heuristic Experiment using Benchmark Functions

To run experiments using single-objective benchmark functions, the user needs to run the binary *benchmarkexperiment*. If executed without any parameters, it displays the following help message:

```
\$./benchmarkexperiment
Parameters:
------------------------------------------------------------------------
-r R          - Number of repetitions for the algorithm.
If not given R=32 is used.
-p P          - Number of agents used in the algorithm.
-i I          - Number of iterations used in the algorithm.
-d D          - Number of dimensions used in the cost functions.
If not given D=3 is used.
-c <costname> - Name of the cost function used.
<ACKLEY> Ackley Function - Mulimodal and Non-separable
<GRIEWANK> Griewank Function - Multimodal and Non-separable
<MICHALEWICZ> Michalewicz Function - Steep Ridges/Drops, Multimodal and Separable
<RASTRIGIN> Rastrigin Function - Multimodal and Separable
<ROSENBROCK> Rosenbrock Function - Valley-Shaped, Non-Separable and Unimodal
<ROTATEDHYPERELLIPSOID> Rotated Hyper Ellipsoid Function - Convex and Unimodal
<SCHWEFEL> Schwefel Function - Multimodal and Separable
<SPHERE> Sphere Function - Separable, Convex and Unimodal
<STEP> Step Function - Separable and Unimodal
<STEPINT> StepInt Function - Separable and Unimodal
<SUMSQUARES> SumSquares Function - Separable and Unimodal
<TRID> Trid Function - Non-Separable and Unimodal
<ZAKHAROV> Zakharov Function - Plate-Shaped, Unimodal and Non-separable
-a <algname>  - Name of the algorithm used.
<AGWO> Adaptive Gray Wolf Optimization
<APSO> Adaptive Particle Swarm Optimization
<APSOV2> Adaptive Particle Swarm Optimization v2
<CSASADE> Crossover Strategy Adaptive - Self-Adaptive Differential Evolution
<JADE> Adaptive Differential Evolution
<SAPSO> Self-Adaptive Particle Swarm Optimization
<SIMPLEBA> Bat Algorithm
<SIMPLEDA> Dragonfly Algorithm
<SIMPLEDE> Differential Evolution (DE/rand/1/bin)
<SIMPLEEHO> Elephant Herd Optimization
<SIMPLEGWO> Gray Wolf Optimization
<SIMPLEMFO> Moth-Flame Optimization
<SIMPLEPSO> Particle Swarm Optimization
<SIMPLESSA> Salp Swarm Optimization
<SIMPLEWOA> Whale Optimization Algorithm
<SOADE> Single-Objective Adaptive Differential Evolution
-h            - Prints this help message.
```

For example, imagine an experiment using this module that would execute for 64 repetitions the meta-heuristic GWO optimizing the benchmark function *Ackley* with 25 dimensions, and in each repetition, GWO runs for 100 iterations and has 40 wolves. For this specific case, *benchmarkexperiment* should be executed with the following parameters:

```
\$./benchmarkexperiment -r 64 -p 40 -i 100 -d 25 -c ACKLEY -a SIMPLEGWO
```

During the execution, *benchmarkexperiment* prints in the terminal console a progress bar based on the percentage of repetitions executed so far as shown below:

```
Progress: |XXXXXXXXXX----------------------| 33.33% Completed
```

At the end of its execution, *benchmarkexperiment* saves in a folder *data/results* in the same folder of the binary the results for the experiment in a text file containing the parameters for the algorithm and experiment, for example, the internal static parameter for a meta-heuristic that contain any, and number of executions, iterations, and search agents. This report text file additionally includes the best evaluation and solution for each one of the repeated executions. It also saves the history for the best evaluation during each of executions iterations.

6.3.2 Perform Multi-Objective Meta-Heuristic Experiment using Benchmark Functions

Experiments using multi-objective benchmark optimization functions are performed using the binary execution file *mobenchmarkexperiment*. If executed without any parameters, the following message is displayed:

```
\$./mobenchmarkexperiment
Parameters:
------------------------------------------------------------------------
-r R          - Number of repetitions for the algorithm.
If not given R=32 is used.
-p P          - Number of agents used in the algorithm.
-i I          - Number of iterations used in the algorithm.
-d D          - Number of dimensions used in the cost functions.
If not given D=8 is used.
Note that this parameter is not applied in some problems.
-m M          - Number of objectives used in the cost functions.
If not given M=3 is used.
Note that this parameter is not applied in some problems.
-c <costname> - Name of the cost function used.
<DTLZ1> DTLZ1 - Multi-Objective Benchmark
<DTLZ2> DTLZ2 - Multi-Objective Benchmark
<DTLZ7> DTLZ7 - Multi-Objective Benchmark
<KURSAWE> KURSAWE - Multi-Objective Benchmark
<ZDT1> ZDT1 - Multi-Objective Benchmark
<ZDT2> ZDT2 - Multi-Objective Benchmark
<ZDT3> ZDT3 - Multi-Objective Benchmark
-a <algname>  - Name of the algorithm used.
```

```
<APMTMODE> Adaptive Parameter Mutation Tournment MODE
<MONSADE> Multi-Objective Non-dominant Sorting Adaptive DE
<NSGAII> Non-dominant Sorting Genetic Algorithm II
-h             - Prints this help message.
```

In the same fashion as the single-objective counterpart, *mobenchmarkexperiment* during its execution displays a progress bar in terminal. Also, when it finishes, it saves the results in a file in the folder *data/results* containing the non-dominant solutions and their evaluations for each experiment as well as the solutions and evaluations obtained by each search agents throughout the iterations.

An example case would be to experiment with NSGA-II optimizing *DTLZ1* for 55 repetitions and each repetition containing a population with size 250 and 300 iterations. This experiment is performed using the following parameters:

```
\$ ./mobenchmarkexperiment -r 55 -i 250 -p 300 -c DTLZ1 -a NSGAII
```

6.3.3 Generate Synthetic Real-Time Application

The software framework also contains a module to generate possible synthetic real-time applications to be used by the RTNoC-based task mapping experiments. First, it is necessary to highlight that applications are represented as CSV files that could either be visualized or edited by any text editor software. In these files, the first row adds the table headers, and the subsequent ones represent each task in the application. The possible values for the header and what these columns represented are the following (note: case-sensitive): (a) *NAME*: Task's name, (b) *COST*: Task's WCET in seconds, (c) *DEADLINE*: Task's deadline time in seconds, (d) *PERIOD*: Task's period time in seconds, (e) *DEST_NAME*: Task's message destination task name (blank if there are none), (f) *PAYLOAD*: Task's communication load in bits (blank if there are none), (g) *PRIORITY*: Task's priority value, and (h) *MEMORY*: Task's required code in local memory in bytes.

To generate a new synthetic real-time application, execute *appgenerator*, and if executed without any parameters, the following help message is displayed:

```
\$./appgenerator
Parameters:
-------------------------------------------------------------------------
-a <appName>  - Generated application testbench name.
-t N          - Number of tasks
If not given the application, total utilization factor will be used.
-u U          - Application total utilization factor.
If not given the number of tasks will be used.
-p <priority> - Priority assignment scheme.
* RATEMONOTONIC - Rate Monotonic Priority Assignment.
* DEADLINEMONOTONIC - Deadline Monotonic Priority Assignment.
If not given the Rate Monotonic priority assigner will be used.
-d <uniform>  - Distribution for tasks' utilization.
```

```
* UNIFORM - [Default] Uniform Distribution.
* EXPONENTIAL - Exponential Distribution.
* NORMAL - Normal Distribution.
* CAUCHY - Cauchy Distribution.
* CHISQUARED - Chi-Squared Distribution.
-m <uniform>  - Distribution for payloads' sizes.
* UNIFORM - [Default] Uniform Distribution.
* EXPONENTIAL - Exponential Distribution.
* NORMAL - Normal Distribution.
* CAUCHY - Cauchy Distribution.
* CHISQUARED - Chi-Squared Distribution.
-r <MIN-MAX>  - Application tasks min-max required memory in bytes.
* If not given MIN-MAX=2048-16384 required memory values.
Note:           If both utilization and number of tasks are given,
it will use the number of tasks.
-h            - Prints this help message.
```

This module comes with a default testbench application in the file *AppIndrusiak2014.csv* representing an Autonomous Vehicle Application (AVA). More information about this specific testbench application is presented in Section 7.6.1.

For example, to generate a synthetic application in the file *ExampleApp.csv* that has a total utilization factor of 15, *appgenerator* should be executed with the following parameters:

```
\$./appgenerator -a ExampleApp.csv -u 15
```

6.3.4 Perform Single-Objective Task Mapping onto RTNoC-based MPSoC Experiment

The software framework contains a module specialized in using meta-heuristics to optimize single-objective functions related to task mapping of RTA onto RTNoC-based MPSoC platforms. This module works similarly as the one for single-objective benchmark functions as present in Section 6.3.1. If executed without any parameters, it displays the following message:

```
        \$./nocexperiment
Parameters:
-------------------------------------------------------------------------
-s NxM        - Platform number of rows <N> and number of columns <M>
If not given NxM=4x4 is used.
-r R          - Number of repetitions for the algorithm.
If not given R=32 is used.
-p P          - Number of agents used in the algorithm.
-i I          - Number of iterations used in the algorithm.
-t <filename> - Application testbench file name (a .csv file) inside
of the folder data/testbench/ .
If none given <filename>=AppIndrusiak2014.csv
-c <costname> - Name of the cost function used.
<BDF> Breakdown Frequency Analysis
```

```
<EERT> End-to-End Response Time Analysis
<ENED> Estimated Normalized Total Energy Dissipation by Messages
<MRMC> Local Memory Requirement Model C Analysis
<EERTNEGATIVEMINSLACKRATIO> EERT + Negative Minimum Slack Ratio
<UTILIZATION> Processors and Links Utilization Analysis
-a <algname>  - Name of the algorithm used.
<AHDPSOU> Adaptive HDPSO-U
<AGAV1> Adaptive Genetic Algorithm v1
<AGAV2> Adaptive Genetic Algorithm v2
<AGAV3> Adaptive Genetic Algorithm v3
<AGAV4> Adaptive Genetic Algorithm v4
<AGWO> Adaptive Gray Wolf Optimization
<APSO> Adaptive Particle Swarm Optimization
<APSOV2> Adaptive Particle Swarm Optimization v2
<CSASADE> Crossover Strategy Adaptive - Self-Adaptive Differential Evolution
<DPSO> Discrete Particle Swarm Optimization (Kang2008)
<HDPSOM> Hybrid Discrete Particle Swarm Optimization Makespan-based
<HDPSOU> Hybrid Discrete Particle Swarm Optimization Utilization-based
<JADE> Adaptive Differential Evolution
<SAPSO> Self-Adaptive Particle Swarm Optimization
<SIMPLEBA> Bat Algorithm
<SIMPLEDA> Dragonfly Algorithm
<SIMPLEDE> Differential Evolution (DE/rand/1/bin)
<SIMPLEEHO> Elephant Herd Optimization
<SIMPLEGA> Genetic Algorithm
<SIMPLEGWO> Gray Wolf Optimization
<SIMPLEMFO> Moth-Flame Optimization
<SIMPLEPSO> Particle Swarm Optimization
<SIMPLESCA> Sine Cosine Algorithm
<SIMPLESSA> Salp Swarm Optimization
<SIMPLEWOA> Whale Optimization Algorithm
<SOADE> Single-Objective Adaptive Differential Evolution
-h            - Prints this help message.
```

An example of experiment with this module would be to perform for 33 times the optimization of f_{unsch} (Section 3.8.3) mapping an application represented in a CSV file *exampleApp.csv* onto a $2x3$ mesh-grid platform using AGWO (Section 4.3.5) with 50 wolves running for 120 iteration each time. This experiment case would be performed using the following command:

```
\$./nocexperiment -s 2x3 -t exampleApp.csv -c EERT -a AGWO -r 33 -p 50 -i 120
```

After the experiment execution, its results are saved in a file containing information for the meta-heuristic parameters along with the solutions and evaluations for each one of the optimization repetitions.

6.3.5 Perform Multi-Objective Task Mapping onto RTNoC-based MPSoC Experiment

At last, the last module *monocexperiment* was developed to perform experiments using multi-objective meta-heuristics to optimize multiple characteristics for the design of an MPSoC based on RTNoC related to the task placement of its real-time application. Similar to previous modules, this one was implemented with a help display if it is executed without any parameters as follows:

```
\$./monocexperiment
Parameters:
-------------------------------------------------------------------------
-s NxM        - Platform number of rows <N> and number of columns <M>
If not given NxM=4x4 is used.
-r R          - Number of repetitions for the algorithm.
If not given R=32 is used.
-p P          - Number of agents used in the algorithm.
-i I          - Number of iterations used in the algorithm.
-t <filename> - Application testbench file name (a .csv file) inside
of the folder data/testbench/ .
If none given <filename>=AppIndrusiak2014.csv
-c <costname> - Name of the cost function used.
<EERTMULTI> EERT with Slack, Memory, Energy and Utiization Awareness
-a <algname>  - Name of the algorithm used.
<APMTMODE> Adaptive Parameter Mutation Tournment MODE
<NSAGA> Non dominant Sorting Adaptive GA
<MONSADE> Multi-Objective Non-dominant Sorting Adaptive DE
<NSGAII> Non-dominant Sorting Genetic Algorithm II
-h            - Prints this help message.
```

This module after its execution saves a file containing the non-dominant solutions and their evaluations for each one of the repeated executions for a multi-objective meta-heuristic optimizing the multiple design metrics of $\mathbf{F}_{noc}$ (Section 3.8.8).

6.4 Conclusions of the Chapter

This chapter briefly introduces the software framework developed in this work, including its designed object-oriented architecture, similar software present in the literature, and an end-user tutorial with a few example cases for the modules implemented.

The modules implemented by this software framework is based on the meta-heuristics present in Chapter 4, single-/multi-objective benchmark functions present in Chapter 2. The modules are responsible for performing static analysis of RTNoC-based MPSoC platforms as part of objective functions, as present in Section 3. The experimental setup and the results obtained in Chapter 7 uses the modules present in this chapter.

7 EXPERIMENTAL SETUP AND RESULTS

This chapter introduces the experimental setup used to evaluate the bio-inspired meta-heuristics implemented used in a variety of optimization problems, including their employment in the optimization of design metrics of RTNoC-based MPSoCs. It also presents the results obtained together with their statistical analysis to indicate the most competitive meta-heuristics for the problems at hand. The chapter is divided as follows: Section 7.1 introduces this chapter by explaining its goals; Section 7.2 presents the statistical framework used to analyze the gathered data; Section 7.3 presents characteristics shared by all experiments; Section 7.4 presents the experimental setup and results obtained when applying continuous single-objective meta-heuristics in a set of benchmark problems; Section 7.5 presents the experimental setup and results obtained when applying continuous multi-objective meta-heuristics in a set of benchmark problems; Section 7.6 presents the experimental setup and results obtained when applying single-/multi-objective meta-heuristics in a range of task mapping problems for the optimization of RTNoC-based systems; Section 7.7, concludes the chapter and contextualizes it with other parts of this work.

7.1 Introduction

The different experimental setup proposed here has the goal to make a fair comparison between multiple bio-inspired meta-heuristic to optimize a range of optimization problems including single-/multi-objectives benchmark problems, and the task mapping of real-time applications onto NoC-based MPSoC platforms. The last one is the emphasized problem, and, for this reason, it is focused upon in this work by including multiple objective functions related to different aspects of the design of NoCs that can be optimized, by a search-based meta-heuristic, given its task mapping assignment.

7.2 Statistical Framework

Since bio-inspired meta-heuristics are stochastic, each algorithm optimizing a problem may produce different results in distinct executions. For this reason, it is common practice to use statistical non-parametric procedures [99] as tools to compare meta-heuristics performances because they do not need to comply with any assumptions about the data collected except that is enough statistical information about the meta-heuristics results. These procedures can infer whether a meta-heuristic is statically better than others and what is the significance of this assertion. In this work, the statistical framework used to compare the results between the multiple meta-heuristics used is the Friedman test with posthoc procedures.

7.2.1 Friedman Test with post-hoc procedures

The Friedman test [99] [100] is intended to be used in situations where multiple meta-heuristics are being compared for a family of problems, and their results are used as samples. For this reason, this work uses the Friedman test allied with posthoc tests. These tests verify whether in a set of used meta-heuristics there are statistical inequalities between them, and, if there are significant differences, compare the algorithm that acquired the best results in regard with the others to estimate how different they are pairwise. The procedure to use the Friedman test with posthoc tests is shown in pseudo-code form in Algorithm 38.

Algorithm 38 Non-parametric analysis using Friedman test

INPUT: Data for algorithm/problem pairs.
OUTPUT: Test results

1: **for all** problem results **do**
2: *Rank algorithms from best to worst.*
3: **end for**
4: **for all** algorithms **do**
5: *Calculate average ranks.* ▷ Equation 7.1
6: **end for**
7: *Calculate Friedman Statistic.* ▷ Equation 7.2
8: **if** *Friedman statistic H_0 is rejected.* **then**
9: *Use post-hoc procedures.*
10: **return** *Ranked list of algorithms*
11: **return** *Inferences between best algorithm and others.*
12: **else**
13: **return** *All algorithms are statistically equal.*
14: **end if**

The Friedman test null hypothesis H_0 is that all used meta-heuristics have the same median of performance metrics, i.e., they are equivalent, and the alternative hypothesis H_1 negates the H_0 with an inference level of α. Friedman test operates by calculating ranks for each meta-heuristic $Algorithm_i$, where $(i = 1, ..., n)$ and n is the number of used meta-heuristics based on the median of their performance metrics (obtained results) when optimizing a problem $Problem_j$ where $(j = 1, ..., m)$ with m is the number of problems used. For a given problem, the meta-heuristic with the best result receives rank 1, while the second-best one receives rank 2, and so on, until the meta-heuristic with the worst results receives rank n. If there are ties between two or

more meta-heuristics, they receive an average for their ranks. For example, if two meta-heuristics would receive rank 2, they both receive 2.5 as rank and the next algorithm worse than them receives rank 3. Given that $r_{i,j}$ is the rank of the ith meta-heuristic when optimizing the problem j, the average rank for this algorithm is $rank_i$ and it is calculated using the following equation:

$$rank_i = \frac{1}{n}\sum_{j=1}^{m} r_{i,j}, \tag{7.1}$$

where n is the number of meta-heuristics being compared, and m is the number of optimization problems used by these meta-heuristics.

After calculate the average ranks, the Friedman statistic χ_f^2 is calculated using Equation 7.2 to inquire whether the null hypothesis of all meta-heuristics being statistically identical with the same average rank is accepted or not. χ_f^2 follows a χ^2 distribution with $n-1$ degrees of freedom.

$$\chi_f^2 = \frac{12m}{n(n+1)}\left[\sum_{i=1}^{n} rank_i^2 - \frac{n(n+1)^2}{4}\right]. \tag{7.2}$$

If the Friedman test null hypothesis H_0 is rejected, then it is possible to use posthoc tests to indicate how distinct is the best meta-heuristic when compared in a pairwise-fashion against the other algorithms with smaller average ranks. The posthoc methods perform these comparisons as multiple hypothesis tests. In each one of these hypothesis tests, each null hypothesis H_0 indicates whether both algorithms are statistically equal by having the same rank distribution. Meanwhile, the alternative hypotheses indicate that the algorithms are different and consequently the algorithm with the best average rank is better than the one with smaller average rank. In other words, the p-values obtained by these hypothesis tests can be used as a metric to indicate the difference between pairs of algorithms. The statistic z using a normal distribution for comparing the best algorithm with average rank $rank_{best}$ against another ith meta-heuristic with smaller average rank $rank_i$ is presented as follows:

$$z = \frac{rank_{best} - rank_i}{\sqrt{\frac{n(n+1)}{6m}}}. \tag{7.3}$$

Even though p-values obtained during each comparison could be used as a metric. They are not suitable because as they are, because they do not take into consideration the Family-Wise Error Rate (FWER) that comes from other algorithms comparisons and may influence the tests by varying their chosen significance level α and degrading the test.

Methods that adjust the p-values solve this problem compensating the effects due to FWER and obtaining Adjusted p-values (APV) that can be directly used to compare with the significance level desired [99]. In this work we use two p-value adjustment methods: (a) Finner's procedure [101], and (b) Li's procedure [102].

Finner's procedure adjusts its α value in a step-down manner by analyzing all hypothesis from the largest p-value p_1 to the smallest one p_{n-1} sequentially, and rejecting those alternative hypotheses where $p_i > 1-(1-\alpha)^{\frac{n-1}{i}}$. Each algorithm i has its APV equal to the minimum between 1 and ν with $\nu = \max(1, 1-(1-p_j)^{(n-1)/j})$ with $j = 1, ..., i$.

Meanwhile, Li's procedure is a two-step rejection method for multiple comparisons. In the first step, it identifies if all null hypotheses are rejected by comparing if p_{n-1} is bellow the chosen significance level α. Otherwise, it applies the second step, by comparing and rejecting null hypothesis with $p_i \leq (1 - p_{n-1})/(\alpha(1-\alpha))$. The APV for each ith algorithm is $p_i/(p_i + 1 - p_{n-1})$.

The rule of thumb to obtain significant results with the Friedman test allied with posthoc methods is to perform experiments where the number of problems m is at least twice as big as the number of meta-heuristics n. In other words, $m > 2n$.

7.3 Shared Experimental Setup Characteristics

For each experiment in this work, each one of the meta-heuristics was executed for 50 times per optimization problem. This large amount of execution has as a goal the obtaining of a significant number of samples about the meta-heuristics performance, and, consequently, having enough data for each meta-heuristic in each problem for the statistical analysis to draw trustworthy conclusions about the results obtained.

In each experimental setup, the meta-heuristics were only allowed to evaluate the objective function for 10000 times each execution. Since two types of bio-inspired meta-heuristics are being used, the author exploits their characteristics to improve their performance by dividing differently how these evaluations are performed by changing their number of iterations and their amount of search space agents. Evolutionary algorithms were configured to run with populations of 100 individuals running with 100 iterations. Meanwhile, swarm intelligence algorithms were configured to run with 25 search agents and 400 iterations.

Another experimental characteristic that is shared by multiple experiments is the metric used to evaluate performance. Single-objective bio-inspired meta-heuristics return a single solution that in turn can have its evaluation result used as a numerical criterion of performance. This work deals with minimization problems, the smaller the evaluation result, the better the algorithm performance that is, in turn, closer to the optimal solution. So the criterion for performance is how small is the median of results for an algorithm optimizing a problem. However, multi-objective meta-heuristics returns a set of solutions that possibly forms the Pareto front for the problem. A single numeric metric of performance for the results should take into consideration all the non-dominated solutions resulting from a meta-heuristic execution and evaluate how closer to the Pareto optimal they are. In this work, the numeric metric chosen to evaluate the quality of the results of multi-objective meta-heuristics is the Inverted Generational Distance (IGD).

IGD [103] measures the distance of the non-dominated solutions in the objective space to the known Pareto front for a problem. IGD metric can be calculated by the following equation:

$$IGD = \frac{1}{n} \sum_{i=1}^{n} dist_i^2, \tag{7.4}$$

where n is the number of non-dominant solution found by a meta-heuristic, and $dist_i$ is the euclidean distance between the ith non-dominant solution in the objective space and its closest

solution in the known Pareto Front for the problem. In this manner, the closer the (near-)Pareto optimal solutions resulted from a meta-heuristic to the true Pareto optimal ones, the smaller the IGD metric. For this reason, the IGD is used as a performance metric for multi-objective meta-heuristics, and in the same manner, as the evaluation results for single-objective algorithms, the median for performance metrics for the different runs of a meta-heuristic optimizing a problem is used to calculate the ranks in a Friedman non-parametric statistical test.

7.4 Experimental Setup - Single-Objective Benchmark Functions

In this experiment, the goal is to compare the results for the single-objective meta-heuristics in a group of continuous benchmark optimization problems. Table 7.1 displays the single-objective meta-heuristics used that contains external parameters, and how these parameters were tuned, where N is the number of search agents, and I is the number of iterations. Note that the values for N and I matches the shared configuration since meta-heuristics with $N = 100$ and $I = 100$ are evolutionary.

Tabela 7.1 – External parameters for the continuous single-objective meta-heuristics.

Algorithm	N	I	Parameters				
DE	100	100	$F = 0.5$		$CR = 0.9$		
PSO	25	400	$w_0 = 0.85$	$w_f = 0.05$	$v_{max} = \|\|\boldsymbol{ub} - \boldsymbol{lb}\|\|/2$	$c_1 = 2.1$	$c_2 = 1.9$
EHO	25	400	$N_{clans} = 5$		$N_{kept} = 5$	$\alpha = 0.5$	$\beta = 0.1$
BA	25	400	$A = 0.5$		$p_r = 0.5$		
JADE	100	100	$c = 0.15$		$p = 0.2$		

The set of problems in this experiment are single-objective benchmark problems present in Section 2.3.2.2 with one of the multiple dimensions $[5, 10, 25, 50, 75, 100]$. In this manner, each the meta-heuristics were executed with 78 problems since it was used thirteen continuous benchmark functions times their six configurations of dimensions.

The list of calculated average ranks for each meta-heuristic is presented in Table 7.2. The meta-heuristics with smaller ranks are considered to be better than the ones with bigger ranks.

Tabela 7.2 – Average Rankings for the continuous single-objective meta-heuristics.

Meta-Heuristic	Ranking	Meta-Heuristic	Ranking
WOA	**4.3654**	APSOV2	7.8077
SOAMSDE	5.2372	DE	9.8141
EHO	5.3269	BA	9.9295
JADE	5.5064	APSO	10.0449
SSA	5.9167	SAPSO	10.3526
AGWO	6.1795	PSO	13.3141
GWO	6.25	DA	13.6346
CSASADE	6.3205	MFO	16

The calculated Friedman statistic χ_f^2 is 634.898 resulting in a p-value of ≈ 0 and consequently rejecting the null hypothesis using a significance level α of 0.05. Notice that a small obtained p-value when compared with the significance level implies more considerable evidence against the null hypothesis.

Since the null hypothesis was rejected, it means that indeed the meta-heuristics are statistically different, and for this set of problems, WOA (Section 4.3.9) in average is better than the others, followed by SOAMSDE (Section 4.4.1) and EHO (Section 4.3.6).

The notion of how different the performance of WOA is against the other meta-heuristics is estimated using posthoc procedures. Posthoc procedures compare WOA pairwise against the other algorithms with smaller average ranks. The null hypothesis for each pairwise comparison is that WOA has the same performance as the other meta-heuristics used in this experiment. The calculated values for p-values for these multiple comparison in the posthoc methods used is present in Table 7.3, where unadjusted p represents the p-value before the family-wise error correction, p_{Finner} is the p-value adjusted using Finner's method, and p_{Li} is the p-value adjusted using Li's post-hoc method. To aid the reader to quickly identify which meta-heuristic is statistically equivalent to EHO and which are not using a significance level of 5%, cells in Table 7.3 according to its possible hypothesis analysis. Meta-heuristics colored green are statistically worse than EHO with p-values below 0.05, the ones colored yellow are not significantly different at least in one of the p-values above 0.05, and the ones colored red are statistically equivalent to EHO with p-values above 0.05.

Tabela 7.3 – Adjusted p-values for pair-wise comparison against WOA

Index	Meta-Heuristic	Unadjusted p	p_{Finner}	p_{Li}
1	MFO	0	0	0
2	DA	0	0	0
3	PSO	0	0	0
4	SAPSO	0	0	0
5	APSO	0	0	0
6	BA	0	0	0
7	DE	0	0	0
8	APSOV2	0.000006	0.000012	0.000008
9	CSASADE	0.010331	0.017158	0.013637
10	GWO	0.013433	0.020082	0.017661
11	AGWO	0.017332	0.02356	0.022671
12	SSA	0.041867	0.052058	0.05306
13	JADE	0.134472	0.15349	0.152522
14	EHO	0.207215	0.220255	0.217115
15	SOAMSDE	0.252813	0.252813	0.252813

The p-values obtained by the posthoc methods indicates, with a significance level α of 5%, that WOA is not significantly different than four meta-heuristics: (a) SOAMSDE, (b)EHO, (c)JADE, and (d) SSA. Note that SSA only under the Li's adjusted p-value is higher than 0.5. However, in this work, we treat it as a sign that indeed, WOA is not statistically different than SSA.

A vital point drawn from this analysis is that even though WOA obtained better results during the experiments than these four meta-heuristics, they are not significantly worse than WOA and their results are competitive. Using the p-values to indicate as a level of similarity, it is possible to see that SOAMSDE and WOA are different only under a significance level α of 26%.

7.5 Experimental Setup - Multi-Objective Benchmark Functions

In this experiment the goal is to compare the quality of the PO solutions obtained by the continuous multi-objective meta-heuristics when optimizing continuous benchmark problems present in Section 2.3.2.3 with multiple dimensions, respectively, $[10, 50, 100]$. The list of benchmark MOOP are: (a) DTLZ1, (b) DTLZ2, (c) DTLZ3, (d) Kursawe, (e) DTLZ1, (f) DTLZ1,(g) ZDT1, (h) ZDT2, and (i) ZDT3. In this manner, the experiments were performed with 21 problems since each one of these seven benchmark function times three configurations of dimensions.

The meta-heuristics used in this experiment are the following: (a) CNSGA-II, (b) APMTMODE, and (c) MONSADE. All three meta-heuristics fall onto the evolutionary classification, and for this reason, they were executed with population $N = 100$ and number of iterations $I = 100$. Note that only CNSGA-II contains external parameters that need to be configured by the user. The tuned parameters for CNSGA-II were, respectively, $p_m = 0.1$ and $p_c = 0.8$.

Since the meta-heuristics are multi-objective, the metric of performance used for the Friedman test is the median of IGD values between the generated non-dominated solutions and the known PF of the problems for the fifty executions calculated for each meta-heuristic using each one of the problems The list of average ranks calculated using the median of IGD for each meta-heuristic is presented in Table 7.4.

The Friedman statistic calculated for this experiment was χ_f^2 is 14.095 resulting in a p-value of 0.0009 and consequently rejecting the null hypothesis using a significance level α of 0.05. It indicates that these three meta-heuristics are statically different. In this experiment MONSADE (Section 4.6.2) obtained the best average performance for the 21 problems, and since the Friedman null hypothesis was rejected, it is possible to perform posthoc statistical tests to compare pairwise the differences between MONSADE against APMTMODE and CNSGA-II.

Tabela 7.4 – Average Rankings for the continuous multi-objective meta-heuristics.

Meta-Heuristic	Ranking
MONSADE	**1.3333**
APMTMODE	2.2857
CNSGAII	2.381

Tabela 7.5 – Adjusted p-values for pair-wise comparison against MONSADE

Index	Meta-Heuristic	Unadjusted p	p_{Finner}	p_{Li}
1	CNSGAII	0.000687	0.001374	0.000688
2	APMTMODE	0.002028	0.002028	0.002028

.

The p-values obtained by the posthoc methods indicates with a significance level of $\alpha = 5\%$ that SOAMSDE is significantly better than APMTMODE and CNSGA-II optimizing the problems in this experiment.

7.6 Experimental Setup - Real-Time Application Mapping onto RTNoC-based MPSoC

In these experiments, the goal is to assess in different scenarios which meta-heuristic is capable of obtaining competitive results when optimizing different aspects of RTNoC-based MPSoC based on the task placement of the real-time application mapped onto it.

To maintain generality as well as to obtain statistically significant results, the experiments for task mapping problems use a large number of platforms and applications.

In the experiments, the characteristics of the NoC-based MPSoCs platforms fit the characteristics described in Section 3.7. The central aspect that varies for these platforms is their mesh-grid sizes that correspond to the number of homogeneous processor elements inside of the system. The characteristics for these platforms are present on Table 7.6 and their sizes are respectively: (a) 3x3; (b) 3x4; (c) 4x4; (d) 4x5, and (e) 5x5;

Tabela 7.6 – Characteristics of MPSoCs with wormhole-based RTNoCs used as platforms for the experiments.

Platform	$\mathbf{\Psi}_1$	$\mathbf{\Psi}_2$	$\mathbf{\Psi}_3$	$\mathbf{\Psi}_4$	$\mathbf{\Psi}_5$
Type	Homogeneous PEs				
Global Clock Speed	50 MHz				
Mesh-Grid Sizes	3x3	3x4	4x4	4x5	5x5
Number of Cores	9	12	16	20	25
Number of Links	42	58	80	102	130
Link Width	32 bits				
Link Latency	1 cycle				
Routing Algorithm	XY-Algorithm				
Arbitration Policy	Priority-Preemptive				
Routers Buffers	Have Virtual-Channels				
Buffer Depth per VC	2 *flits*				
Router Latency	10 cycles				
Normalized Energy Dissipated Transmitting 1 *flit* Router per Link	1				
Normalized Energy Dissipated Transmitting 1 *flit* NI per Link	1				

For the applications, these experiments use the Autonomous Vehicle Application (AVA) benchmark present on [45], that is based on a real-time application for autonomous vehicle control

as well as synthetic randomly generated applications. The set of real-time applications and their characteristics are present in Table 7.7.

Tabela 7.7 – Applications used for the experiments where $\bar{Z}_{payload}$ is the average size in flits of messages payloads, $|\mathbf{\Gamma}|$ is the task set size, $|\mathbf{\Phi}|$ is the number of messages, and U_{total} is the application total utilization.

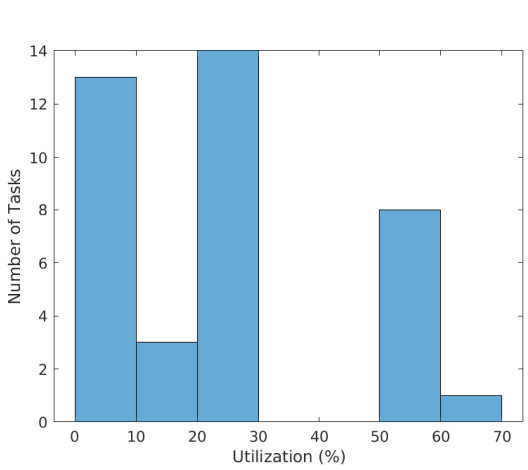
Application	Name	$\|\mathbf{\Gamma}\|$	$\|\mathbf{\Phi}\|$	$\bar{Z}_{payload}$	U_{total} (%)
$\mathbf{\Omega}_1$	AVA	39	39	6294.97	891.05
$\mathbf{\Omega}_2$	Uniform1	53	53	7953.62	1800.84
$\mathbf{\Omega}_3$	Uniform2	46	46	6694.09	1703.10
$\mathbf{\Omega}_4$	Normal1	37	37	8309.95	1306.02
$\mathbf{\Omega}_5$	Normal2	40	40	5589.93	1248.55
$\mathbf{\Omega}_6$	Cauchy1	39	39	8139.41	1398.53
$\mathbf{\Omega}_7$	Cauchy2	47	47	6143.17	1479.47
$\mathbf{\Omega}_8$	Chi-Squared1	55	55	8103.24	1111.23
$\mathbf{\Omega}_9$	Chi-Squared2	38	38	5821.24	1010.48
$\mathbf{\Omega}_{10}$	Exponential1	46	46	9420.89	900.71
$\mathbf{\Omega}_{11}$	Exponential2	40	40	6148.28	754.77

7.6.1 Synthetic Real-time Application Generation

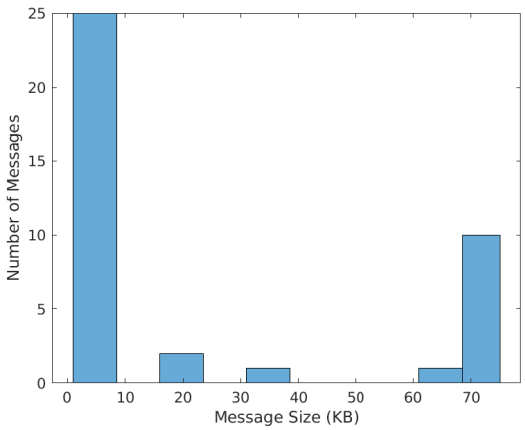
The benchmark application AVA, as explained by [104], was designed for a vehicle that recognizes obstacles using multiple stereo cameras and estimates these obstacles distance, and guide the vehicle during the navigation process. The benchmark contains tasks that manage the car sensing system, for example, cameras frame buffers and tire pressure sensors, as well as tasks that perform navigation and control computations, for example, visual odometry and stability control. This RTA is relevant due to its application context. It also has features that make it an ideal candidate to be mapped into an MPSoC with RTNoC platform. For example, it has a large volume of data in its messages, and its distributed subsystems (represented as tasks) has high computing execution times, and, finally, due to its embedded characteristics, it needs to have low power usage.

This work uses the AVA benchmark presented on [45] that has slight changes from the one presented on [104] (attach as appendix II to this document). AVA contains the following characteristics present in its tasks and messages: (a) AVA has 39 tasks that in total emits 39 messages. (b) tasks execution costs have an average and standard deviation of $\bar{C} = 19.62$ ms and $\sigma_C = 31.39$ ms ranging from 0.5 to 150 ms (milliseconds); (c) tasks periods ranges from 40 to 1000 ms; (d) tasks utilization factors ranges from 0.05 to 70% with an average and standard deviation of $\bar{U} = 22.85\%$ and $\sigma_U = 19.13\%$ respectively; (e) messages payload sizes varies from 1 to 76 KB (kilobytes) with an average of $\bar{Z} = 24.59$ KB and standard deviation of $\sigma_Z = 31.89$ KB. Figure 7.1 presents the

frequency histograms for AVA tasks utilization factors and messages payload sizes to visualize the application characteristics better.



(a) Frequency histogram of tasks utilization factors. in flits.



(b) Frequency histogram of messages payload sizes

Figura 7.1 – Frequency histograms for utilization factors of AVA tasks as well as their messages sizes in flits.

The synthetic application generation takes into consideration the range of values assumed by the tasks on AVA and generates different tasks sets using multiple random distributions for both tasks and messages. The process to generate synthetic RTA create each task with a utilization factor U generated between 0.1 and 0.75 using a given distribution function, and if the value is outside this boundaries, it is substituted with the violated boundary. These synthetic tasks are generated until the sum of their utilization factor is greater than a target total utilization factor $U_{\text{target\_total}}$. After a task utilization factor is generated, their inter-arrival period T is assigned using a uniform distribution ranging in $[0.04, 1.0]$ seconds. Then, both T and U are used to calculate the synthetic task cost execution C. Since each task also contains the amount of memory code required for its execution, M is randomly generated using a uniform distribution in the range $[2, 16]$ KB.

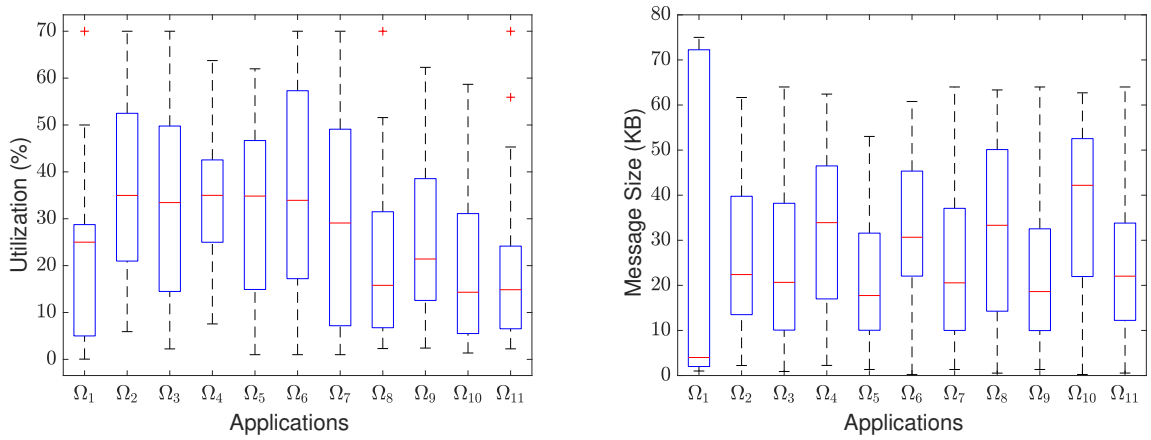
The process to create synthetic applications performs the Rate Monotonic algorithm (Section 3.6.3) to attach priorities to each one of the randomly generated tasks, and then, the messages for each of these tasks are generated. Tasks messages are created to be transmitted from a task with higher priority to a randomly selected synthetic task with lower priority, and their payload sizes are generated using a distribution function in the same fashion as the utilization factors generation for tasks. Since in the model used each task can only receive messages from one other task if a task can not find a lower priority task that other tasks are not sending messages to, a "sink"task is generated to receive that task message. A "sink"task represents a placeholder that denotes a message being sent to a processor core and not generating any processing execution cost. It would be the equivalent to a direct memory access process of a task to a processing core local memory, as described in [45].

The random application generation method is explained in pseudo-code form in algorithm 39.

The distribution functions to generate each one of the synthetic applications used during this work experiments is specified based on their names. Synthetic applications that have the same name prefix share the same distribution to generate their tasks utilization factors. Meanwhile, the ones with suffix 1 and 2 use different distributions to generate their messages sizes, respectively, a uniform distribution $\mathcal{U}(0.125, 64)$ and a scaled chi-squared distribution $\frac{64}{10}\chi^2(3)$ with 3 degrees of freedom and a scaling factor to generate random numbers closer to the interval of allowed message sizes that is between $[0.125, 64]$ KB.

Applications *Uniform1* and *Uniform2* use a uniform distribution function $\mathcal{U}(0.1, 0.75)$ to generate their tasks utilization factors and their target total utilization factors are, respectively, $U_{\text{target\_total}} = 18$ and $U_{\text{target\_total}} = 17$. *Normal1* and *Normal2* use a Normal distribution function $\mathcal{N}(\frac{0.75}{2}, \frac{0.75}{4})$ with target total utilization factors are $U_{\text{target\_total}} = 13$ and $U_{\text{target\_total}} = 12$, *Cauchy1* and *Cauchy2* use a Cauchy distribution function $\mathcal{C}(\frac{0.75}{2}, \frac{0.75}{4})$ with $U_{\text{target\_total}} = 13$ and $U_{\text{target\_total}} = 14$, *ChiSquared1* and *ChiSquared2* use a scaled chi-squared distribution, similar to that used to generate message payload sizes, with 3 degrees of freedom $0.1\chi^2(3)$ with target total utilization factors $U_{\text{target\_total}} = 11$ and $U_{\text{target\_total}} = 10$, and *Exponential1* and *Exponential2* use a scaled Exponential distribution function with parameter $\lambda = 1$ $0.2f_{exp}(1)$ with target total utilization factors $U_{\text{target\_total}} = 9$ and $U_{\text{target\_total}} = 7$.

The synthetic generated utilization factors and message payloads are presented, together with the AVA application, in box plot format in figure 7.2 showing how these characteristics are distributed.



(a) Distribution of utilization factors for every application used on experiments.

(b) Distribution of messages payload sizes for every application used on experiments.

Figura 7.2 – Distributions of utilization factors and messages payload sizes for applications used on experiments.

Table 7.8 and Table 7.9 presents the parameters used for each meta-heuristic in the experiments regarding task mapping of RTA onto MPSoCs. In these tables, N is the number of search agents used, I is the maximum number of iterations, and $-$ is used for algorithms external without parameters All the meta-heuristics that have external static parameter had their parameters set as suggested by their original articles, except for the GA algorithm that had their parameters set

Algorithm 39 Synthetic Application Generation

INPUT: Desired total utilization ($U_{\text{target\_total}}$)
Distribution for utilization factor ($Dist_{util}$)
Distribution for messages sizes ($Dist_{msg}$)
OUTPUT: Synthetic application ($\mathbf{\Psi}$)

procedure GENERATEAPP
  $\mathbf{\Gamma} \leftarrow \emptyset$ ▷ Generate tasks
  $U_{total} = 0$
  while $U_{total} < U_{target\,total}$ **do**
    Generate T_i *using* $\mathcal{U}(40, 1000)$ *ms.*
    $D_i = T_i$
    Generate U_i *using* $Dist_{util}$ *in [1, 75] %*
    Generate M_i *in* $[2, 16]$ *KB*
    Calculate $C_i = U_i T_i$
    if $C_i < 0.5$ ms **then**
      $C_i = 0.5$ *ms*
      $U_i = C_i / T_i$
    end if
    $\boldsymbol{\tau}_i = \langle C_i, T_i, D_i, \sim, \sim, M_i, \sim \rangle$ *
    $\mathbf{\Gamma} \leftarrow \boldsymbol{\tau}_i$
    $U_{total} = U_{total} + U_i$
  end while
  Sort $\mathbf{\Gamma}$ *based on their periods.*
  Assign priorities using Rate Monotonic.
  $\mathbf{\Phi} \leftarrow \emptyset$ ▷ Generate messages.
  for $i = 1$ to $|\mathbf{\Gamma}|$ **do**
    $\boldsymbol{\tau}_d \leftarrow$ *Random task between* $\boldsymbol{\tau}_{i+1}$, *and* $\boldsymbol{\tau}_{min(i+6,|\Gamma|)}$
    if $\boldsymbol{\tau}_d = \emptyset$ **then**
      $\boldsymbol{\tau}_d = \langle -, -, -, -, -, \emptyset \rangle$ ▷ "sink"task.
    end if
    Generate Z_i using $Dist_{msg}$ in $[0.125, 64]$ KB
    $Z_i = \lfloor Z_i + 0.5 \rfloor$ ▷ Messages have integer sizes in bits.
    $\boldsymbol{\phi}_i = \langle \boldsymbol{\tau}_d, \sim, Z_i, \sim, \sim \rangle$ *
    $\boldsymbol{\tau}_i = \langle C_i, T_i, D_i, P_i, \sim, M_i, \boldsymbol{\phi}_i \rangle$ *
    $\mathbf{\Phi} \leftarrow \boldsymbol{\phi}_i$
  end for
  $\mathbf{\Psi} = \{\mathbf{\Gamma}, \mathbf{\Phi}\}$
  return $\mathbf{\Psi}$
end procedure

* $\sim$ denotes a temporary placeholder for components not present yet.

following [14].

Tabela 7.8 – List of single-objective meta-heuristics used for the experiments and their parameters regarding single-objective functions f_{unsch}, f_{bdf}, and f_{umsr}.

Algorithm	N	I	Parameters				
GA	100	100	$p_c = 0.8$		$p_m = 0.01$		
DE	100	100	$F = 0.5$		$CR = 0.9$		
PSO	25	400	$w_{initial} = 0.95$	$w_{final} = 0.05$	$v_{max} = \lvert\mathbf{\Pi} - 1\rvert/4$	$c_1 = 2.05$	$c_2 = 1.95$
EHO	25	400	$N_{clans} = 5$		$N_{kept} = 5$	$\alpha = 0.5$	$\beta = 0.1$
BA	25	400	$A = 0.5$		$p_r = 0.5$		
JADE	100	100	$c = 0.15$		$p = 0.2$		
DPSO	25	400	$w_{initial} = 0.8$				
HDPSO-M	25	400	$p = 0.3$				
HDPSO-U	25	400	$p = 0.3$				

Tabela 7.9 – List of multi-objective meta-heuristics used for the experiments and their parameters regarding the multi-objective function $\boldsymbol{F}_{noc}$.

Algorithm	N	I	Parameters	
NSGA-II	100	100	$p_m = 0.05$	$p_c = 0.9$
APMTMODE	100	100	-	
NSAGA	100	100	-	
MONSADE	100	100	-	

7.6.2 End-to-End Response Time Scheduling

When a bio-inspired meta-heuristic is used for task mapping problems that use the objective-function f_{unsch} (Section 3.8.3), its goal is to work as a schedulability algorithm that searches for task mappings solutions. In this case, the best solution found by the meta-heuristic is only deemed schedulable when its evaluation using f_{unsch} is equal to 0, i.e., there are 0 unschedulable tasks. In light of this information, this experiment aims to compare the results obtained by single-objective meta-heuristics optimizing multiple instances of the objective function f_{unsch} that was defined in Section 3.8.3 when mapping the RTA in Table 7.7 into the platforms present in Table 7.6. The analysis of this experiment point, which meta-heuristics are more suitable for the problem f_{unsch}.

In total, this experiment performs each meta-heuristic 50 times for each combination of applications and platforms. Since there are eleven applications, and five SoC platforms, the analysis in this experiment covers 55 task mapping problems.

The list of average ranks calculated for the meta-heuristics is present in Table 7.10.

The calculated Friedman statistic for the ranks in Table 7.10 is $\chi_f^2 = 813.557$ resulting in a p-value of ≈ 0, and consequently, rejecting with a significance level $\alpha = 5\%$ the Friedman null hypothesis that all meta-heuristics are in average statistically equal. It also means that indeed,

Tabela 7.10 – Average Rankings for single-objective meta-heuristics optimizing different instances of f_{unsch}.

Meta-Heuristic	Ranking	Meta-Heuristic	Ranking
AGAV4	**4.9636**	SSA	12.9545
AGAV3	5.0091	BA	13.2545
SOAMSDE	6.2545	HDPSO-U	15.0727
AGAV1	6.8909	AGWO	16.6818
AGAV2	6.9364	GWO	16.6909
GA	7.1	DE	17.0455
CSASADE	7.9	WOA	17.0545
APSOV2	10.0545	DA	19.0909
ADPSOUTIL	10.2364	SAPSO	21.0727
APSO	10.5091	MFO	21.7909
PSO	10.5364	EHO	22.0636
JADE	10.5455	DPSO	24.3
HDPSO-U	10.9909		

AGAv4 (Section 4.4.5) is, on average, the best algorithm for the analyzed problems. However, its difference compared in a pairwise manner against the other meta-heuristics is obtained using posthoc methods that compare with a null hypothesis whether AGAv4 is in average equal to the other algorithm being analyzed. This analysis generates the p-values present in Table 7.11. These p-values indicate with a significance level of 5% that AGAV4 is not significantly different than six: (a) AGAV3, (b) SOAMSDE, (c) AGAV1, (d) AGAV2, (e) GA, and (f) CSASADE. Therefore these six algorithms are competitive results between them. Note that only under Li's posthoc method that CSASADE is different than AGAv4.

A point that is possible to draw from these results is that evolutionary algorithms are especially suitable for problems that use f_{unsch}, specially GA-based ones. These results agree with results obtained by related works such as [45] and [14].

By using the values of the adjusted p-values to indicate how different these six meta-heuristics are from AGAv4, it is possible to confirm that since AGAv4 is an expansion of AGAv3, both algorithms, in this experiment, have close to no differences. Another point that is possible to be noticed is that the adaptive meta-heuristics obtained better results than its counterparts including the multiple adaptive GA-based meta-heuristics developed in this work, adaptive DE-based ones, adaptivePSO-based ones, and AHDPSO-U against HDPSO-U. The only exception is SAPSO that obtained worse results than PSO. However, even this exception can be seen as an effect of the NFL theorem, since SAPSO when optimizing the continuous benchmark problems obtained better results than PSO.

For a convergence comparison for each meta-heuristic using f_{unsch}, we present a visual representation using box-plot for the algorithms in three instances of f_{unsch} mapping the application AVA into platforms with size $3x3$, $3x4$, and $4x4$. The reasoning for the selection of AVA as the main RTA in these convergence comparisons is due to its use in related works in the area. So it becomes a shared benchmark for comparison for the results obtained in this work and the ones

Tabela 7.11 – Adjusted p-values for pair-wise comparison against AGAV4

Index	Meta-Heuristic	Unadjusted p	p_{Finner}	p_{Li}
1	DPSO	0	0	0
2	EHO	0	0	0
3	MFO	0	0	0
4	SAPSO	0	0	0
5	DA	0	0	0
6	WOA	0	0	0
7	DE	0	0	0
8	GWO	0	0	0
9	AGWO	0	0	0
10	HDPSO-M	0	0	0
11	BA	0	0	0
12	SSA	0	0	0
13	HDPSO-U	0.000018	0.000032	0.000677
14	JADE	0.00007	0.00012	0.002692
15	PSO	0.000072	0.00012	0.002766
16	APSO	0.000078	0.00012	0.003
17	AHDPSO-U	0.000172	0.000243	0.006613
18	APSOV2	0.000286	0.000382	0.010959
19	CSASADE	0.036418	0.045779	0.584979
20	GA	0.127956	0.151511	0.832001
21	AGAV2	0.159838	0.180483	0.860848
22	AGAV1	0.169681	0.183598	0.867853
23	SOAMSDE	0.357674	0.369919	0.93263
24	AGAV3	0.974163	0.974163	0.974163

present in similar works. Meanwhile, the argument for the selection of these three platforms is the fact that they contain a similar processor element count to the the U_{total} for AVA (not considering communication architecture resources). Since AVA has a U_{total} of 891.05%, it means that AVA needs at least 8.9 processor cores to only its tasks to pass the utilization factor test (Section 3.6.4.1). So in these cases, the search for a task mapping of AVA into a $3x3$ platform is improbable to find a schedulable solution. The search for a mapping of AVA into a $3x4$ platform has more chances to find schedulable solutions. At last, the search in a $4x4$ is very likely to find a complete schedulable solution. In light of this information, the spread of evaluation results of a meta-heuristic during its 50 executions in these mapping problems exhibits its performance. Figures 7.3, 7.4, 7.5 illustrates, in box-plots, the spread of convergence for each meta-heuristics mapping AVA into platforms with sizes $3x3$, $3x4$, and $4x4$, respectively. In these box-plots, the medians are represented as black dots, and the outliers are represented as red crosses.

By analyzing the box-plots, it is possible to visualize that it is more difficult to differentiate the meta-heuristics for the cases where multiple of them find schedulable solutions, as shown in Figure 7.4 and Figure 7.5. This small sample of the experiment problems confirms the results obtained by the statistical analysis. Another interesting observation is the behavior for meta-heuristics AHDPSO-U and HDPSO-U that has improved performance in cases where there is enough processing capability in the platform to support the application execution. This improvement in performance for these specific cases is due to their local-search method that permits these meta-heuristics to randomize solutions that are generated by distributing tasks to the processors with lower utilization. However, this same local-search method is also the reason for the lower performance in cases where there are not enough processors for the application, because it limits the exploratory behavior for these meta-heuristics and they end up converging prematurely.

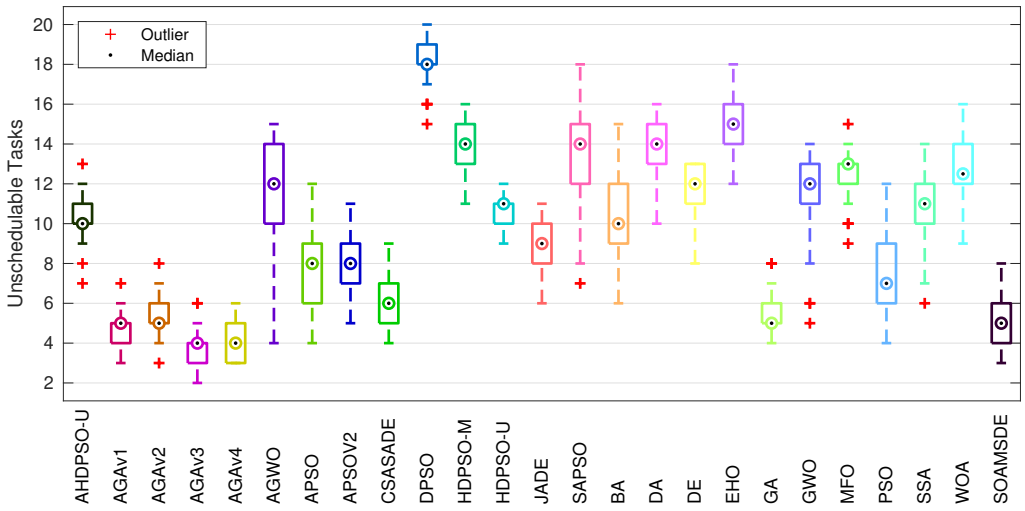


Figura 7.3 – $(3x3)$ Box-plot for $f_{unsch}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_1})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

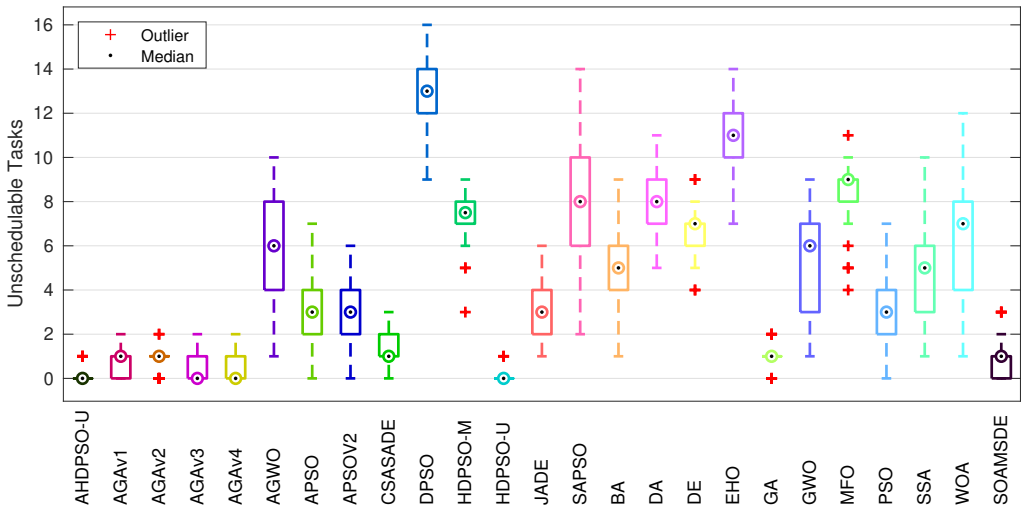


Figura 7.4 – $(3x4)$ Box-plot for $f_{unsch}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_2})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

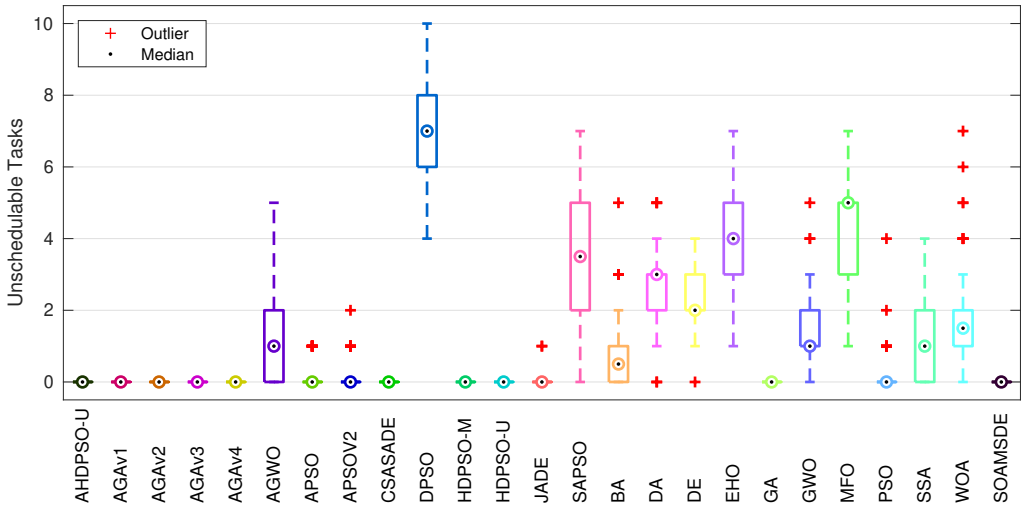


Figura 7.5 – $(4x4)$ Box-plot for $f_{unsch}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_3})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

7.6.3 Breakdown Frequency Optimization

In this experiment the goal is to compare the multiple single-objective meta-heuristics used in this work when optimizing task mapping problems using f_{bdf} (Section 3.8.6) as objective using the applications in Table 7.7 being mapped onto the platforms present in Table 7.6. A bio-inspired meta-heuristic searching for optimal results for f_{bdf} is obtaining a (near-)optimal task placement

that is reducing the frequency for the system.

In total, this experiment uses 55 task mapping problems. The average ranks calculated for each meta-heuristic using the median for their results is present in Table 7.12.

Tabela 7.12 – Average Rankings for single-objective meta-heuristics optimizing different instances of f_{bdf}.

Meta-Heuristic	Ranking	Meta-Heuristic	Ranking
AGAV4	**5.2727**	JADE	12.0273
AGAV3	5.3818	BA	14.7727
AGAV2	6.7727	AGWO	14.9
AGAV1	7.0455	GWO	15.1364
SOAMSDE	7.5909	SSA	16.5909
AHDPSO-U	7.7818	DE	16.8727
HDPSO-U	8.4	DA	19.1818
GA	9.3182	WOA	19.3182
HDPSO-M	9.4182	MFO	20.2
APSOV2	9.6455	SAPSO	20.7909
CSASADE	10.3	EHO	23.1364
PSO	10.3545	DPSO	24.0727
APSO	10.7182		

The calculate Friedman static for the ranks in Table 7.12 is $\chi_f^2 = 794.988$ resulting in a p-value of ≈ 0. It means that the null hypothesis that all meta-heuristics are statically different in this test is rejected with a significance of 5%. It also has a consequence that since these meta-heuristics are different, AGAv4 is on average better than the other meta-heuristics. By using posthoc methods to assess how different, and, consequently, better is AGAv4 when compared to each other meta-heuristics, posthoc methods are used, and the calculated p-values for each null hypothesis that AGAv4 is different than another meta-heuristics are present in Table 7.13. The adjusted p-values generated indicates that AGAV4 is not significantly better than the following seven meta-heuristics with a significance level $\alpha = 5\%$: (a) AGAv3, (b) AGAv2, (c) AGAv1, (d) SOAMSDE, (e) AHDPSO-U, (f) HDPSO-U, and (g) GA. Notice that AHDPSO-U and GA are not statistically worse than AGAv4 on average only when using the adjusted p-value obtained by Li's method.

Another point that is possible to be drawn is that the gap between AGAv4 and GA results in average has increased if compared with problems that use f_{unsch}. Also, the difference in p-values of the null hypothesis that both algorithms are statistically the same decreased. The reason for this effect is because AGA (v1, v2, v3, and v4) and GA find schedulable task placements in many problems of the set of task mapping problems used. Also, in these cases, f_{unsch} is unable to differentiate between schedulable solutions and both GA and AGA can not further optimize solutions, and further differentiate their behavior, hence a more significant similarity between their results in average.

However, by using f_{bdf} the meta-heuristics are capable to further optimize the design by finding a task mapping not only schedulable but also with even lower frequencies. In this scenario, AGAv4

can further optimize the designs when compared with GA and obtain average better results. The results for this experiment demonstrates that the developed adaptive GA-based algorithms, SOAMSDE, and AHDPSO-U, are all algorithms that obtain competitive results compared with GA.

Tabela 7.13 – Adjusted p-values for pair-wise comparison against AGAV4

Index	Meta-Heuristic	Unadjusted p	p_{Finner}	p_{Li}
1	DPSO	0	0	0
2	EHO	0	0	0
3	SAPSO	0	0	0
4	MFO	0	0	0
5	WOA	0	0	0
6	DA	0	0	0
7	DE	0	0	0
8	SSA	0	0	0
9	GWO	0	0	0
10	AGWO	0	0	0
11	BA	0	0	0
12	JADE	0.000001	0.000003	0.000024
13	APSO	0.000104	0.000193	0.001683
14	PSO	0.000294	0.000503	0.004716
15	CSASADE	0.000341	0.000545	0.005472
16	APSOV2	0.001835	0.002752	0.028769
17	HDPSO-M	0.003139	0.004429	0.048228
18	GA	0.003945	0.005257	0.059868
19	HDPSO-U	0.025863	0.032557	0.2945
20	AHDPSO-U	0.07381	0.087905	0.54365
21	SOAMSDE	0.098583	0.111849	0.61407
22	AGAV1	0.206549	0.223063	0.769252
23	AGAV2	0.285165	0.295523	0.821512
24	AGAV3	0.938043	0.938043	0.938043

Similarly to experiments using f_{eert}, the dispersion of results provided by each meta-heuristic optimizing f_{bdf} when mapping AVA into platforms with sizes $3x3$, $3x4$, and $4x4$, this work uses box-plots as displayed in Figure 7.6, Figure 7.7, and Figure 7.8, respectively.

By observing these box-plots, it confirms the results observed in the statistic analysis that shows the small difference observed between AGAv1, AGAv2, AGAv3, AGAv4. Another interesting observation that can be made is, once again, related to AHDPSO-U and HDPSO-U. These two meta-heuristics are capable of obtaining, in the problems shown by these box-plots, very competitive results due to their local-search capabilities that reduce the use of overloaded processor elements and ends up overall reducing the breakdown frequency necessary to schedule all tasks. However, this same local-search method also causes these meta-heuristics to obtain worse results than AGAv4, for example, due to their limited exploration capability.

By analyzing the box-plots, it is possible to visualize that the meta-heuristics are more difficult

to differentiate between themselves for the cases where multiple of them find schedulable solutions, as shown in Figure 7.4 and Figure 7.5. Another interesting observation is the behavior for meta-heuristics AHDPSO-U and HDPSO-U that has improved performance in cases where there is enough processing capability in the platform to support the application execution.

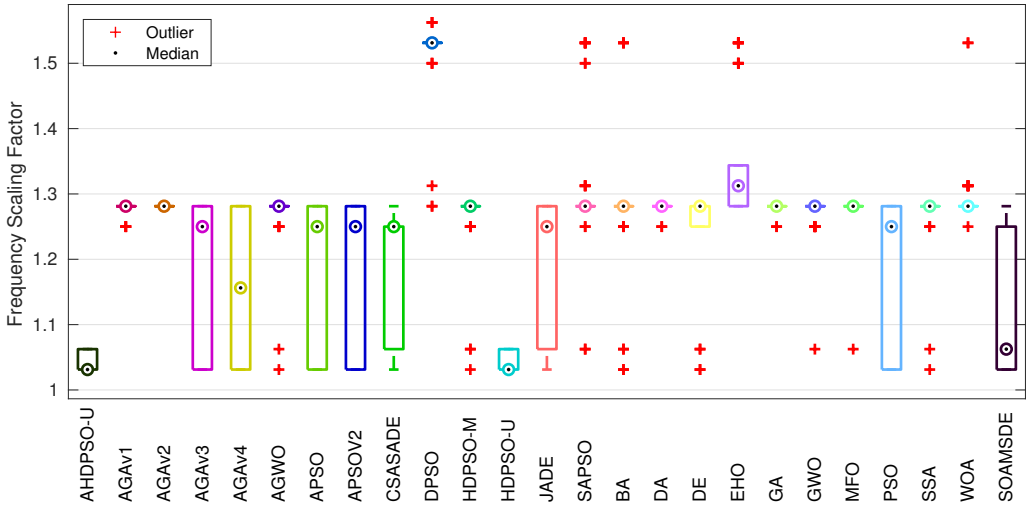


Figura 7.6 – (3x3) Box-plot for $f_{bdf}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_1})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

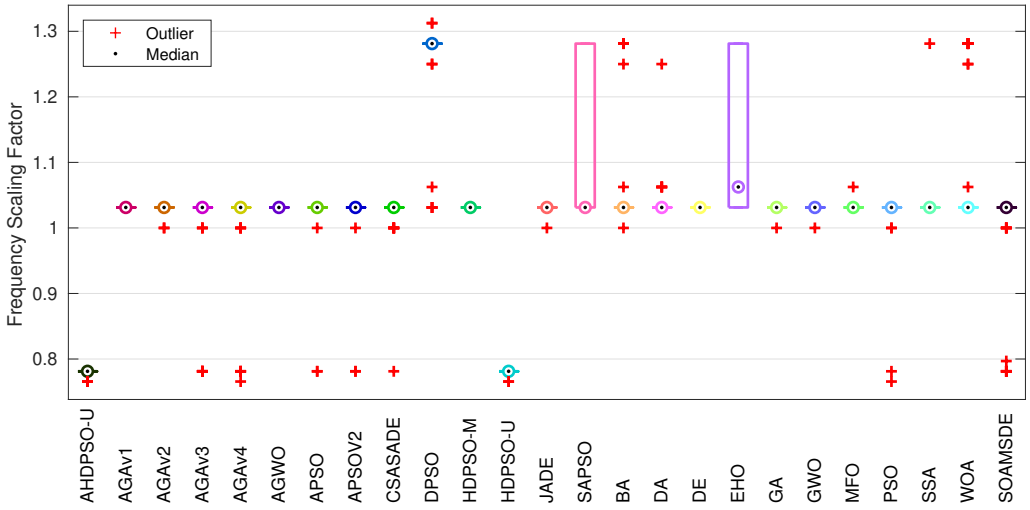


Figura 7.7 – (3x4) Box-plot for $f_{bdf}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_2})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

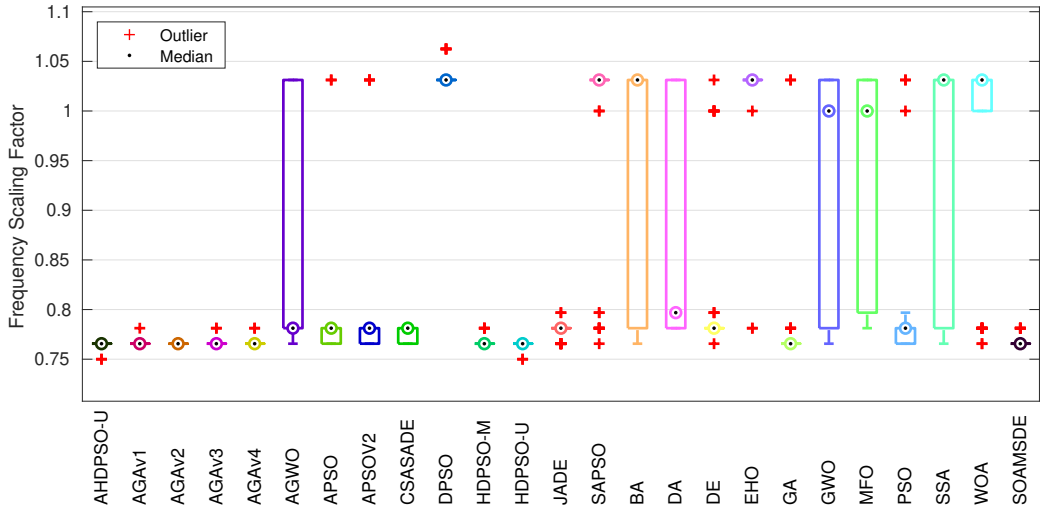


Figura 7.8 – (4x4) Box-plot for $f_{bdf}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_3})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

7.6.4 End-to-End Response Time Scheduling with Slack Awareness

This experiment aims to assess which single-objective meta-heuristic obtains best results when optimizing a set of task mapping problems using f_{umsr} (Section 3.8.7) as the objective function. The set of RTA and platforms used in this experiment is present in Table 7.7 and Table 7.6 resulting in a total of 55 task mapping problems. A bio-inspired meta-heuristic optimizing f_{umsr} is capable of either obtain an unschedulable task placement solution with the fewest number of unschedulable tasks or, if the task placement is schedulable, the minimum slack and deadline ratio for all tasks. Similarly to f_{bdf}, f_{umsr} is capable of differentiating schedulable task mapping solutions and further optimizing them. In the case of f_{umsr}, this criterion is the minimum slack deadline ratio.

The average ranks calculated for the medians of the results obtained by each meta-heuristic is shown in Table 7.14. These average ranks result in a calculated Friedman statistic of $\chi_f^2 = 1186.05$ resulting in a p-value of ≈ 0 annulling the null hypothesis with a significance level of 5%.

Since the null hypothesis was rejected, it means that all meta-heuristics are significantly different and AGAv4 is considered to be the best algorithm, on average, for this experiment. It also allows the use of posthoc methods to determine how different is each algorithm when compared to AGAv4.

Table 7.15 presents the p-values and adjusted p-values for the comparison for the medians of the results of each meta-heuristic against AGAv4. By using a significance level $\alpha = 5\%$, it is possible to observe that AGAv4 is not statistically different than the following five meta-heuristics: (a) AGAv3, (b) GA, (c) AGAv1, (d) SOAMSDE, and (e) AGAv2. Note that for these problems, AGAv4 performance was even better than those using f_{bdf}, and GA, AGAv1, SOAMSDE, and

Tabela 7.14 – Average Rankings for single-objective meta-heuristics optimizing different instances of f_{umsr}.

Meta-Heuristic	Ranking	Meta-Heuristic	Ranking
AGAV4	**1.6455**	SSA	13.9182
AGAV3	1.8909	BA	14.6
GA	4.7091	HDPSO-M	15.0091
AGAV1	4.8636	AGWO	17.0545
SOAMSDE	5.0364	GWO	17.5364
AGAV2	5.2909	WOA	18.5818
CSASADE	7.5818	DE	18.7182
AHDPSO-U	9.3909	DA	20.3455
HDPSO-U	10.7364	SAPSO	22.2909
APSOV2	10.7909	MFO	22.9273
JADE	10.9909	EHO	23.3364
PSO	11.3182	DPSO	24.9364
APSO	11.5		

Tabela 7.15 – Adjusted p-values for pair-wise comparison against AGAV4

Index	Meta-Heuristic	Unadjusted p	p_{Finner}	p_{Li}
1	DPSO	0	0	0
2	EHO	0	0	0
3	MFO	0	0	0
4	SAPSO	0	0	0
5	DA	0	0	0
6	DE	0	0	0
7	WOA	0	0	0
8	GWO	0	0	0
9	AGWO	0	0	0
10	HDPSO-M	0	0	0
11	BA	0	0	0
12	SSA	0	0	0
13	APSO	0	0	0
14	PSO	0	0	0
15	JADE	0	0	0
16	APSOV2	0	0	0
17	HDPSO-U	0	0	0
18	AHDPSO-U	0	0	0
19	CSASADE	0.000023	0.00003	0.000168
20	AGAV2	0.009391	0.011259	0.063356
21	SOAMSDE	0.015687	0.017908	0.101521
22	AGAV1	0.021846	0.023808	0.135958
23	GA	0.029042	0.030285	0.172994
24	AGAV3	0.861164	0.861164	0.861164

AGAv2 are only not significantly different under Li's adjusted p-value.

A possible observation is that the gap between GA and AGAv4 in performance is similar to the one observed when using f_{bdf}. The reasoning is the same as f_{bdf}, i.e., AGAv4 is capable of optimizing even further schedulable task mappings when compared to GA.

A point that is possible to observe in this experiment is that GA performance was better than its adaptive versions that do not contain **adaptive operation selection mechanisms**, namely, AGAv1, and AGAv2. The reasoning behind it may be a premature convergence in the part of these adaptive versions due to its adaptive mechanisms getting stuck in a parameter search region that behaves worse than the initially set parameters from GA ($p_m = 0.8$, and $p_c = 0.01$). Another similarity with the experiment using f_{bdf} is that APSO obtained worse results on average than PSO, while APSOv2 in both cases obtained better results on average. Since the only difference between both APSO and APSOv2 is the substitution of a deterministic parameter control to an adaptive one to control the particles inertial weight parameters. This improvement in performance is attributed to this change.

Similarly to experiments using f_{eert} and f_{bdf}, Figure 7.9, Figure 7.10, and Figure 7.11, respectively, shows as box-plots the dispersion of results form each meta-heuristic optimizing f_{umsr} when mapping AVA into platforms with sizes $3x3$, $3x4$, and $4x4$. These box-plots agrees with the statistic analysis by showing, in these smaller set of problems, that AGAv3 and AGAv4 have almost the same performance.

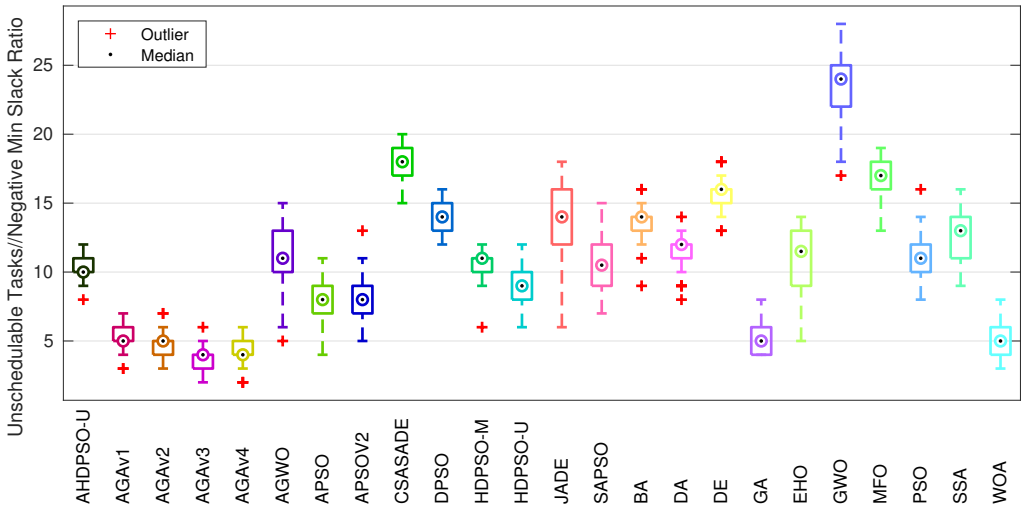


Figura 7.9 – ($3x3$) Box-plot for $f_{umsr}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_1})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

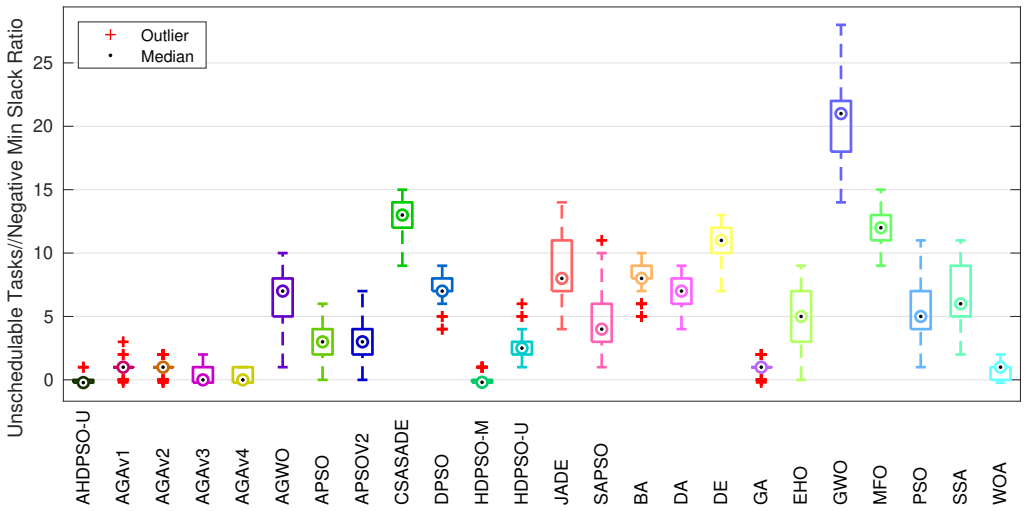


Figura 7.10 – $(3x4)$ Box-plot for $f_{umsr}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_2})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

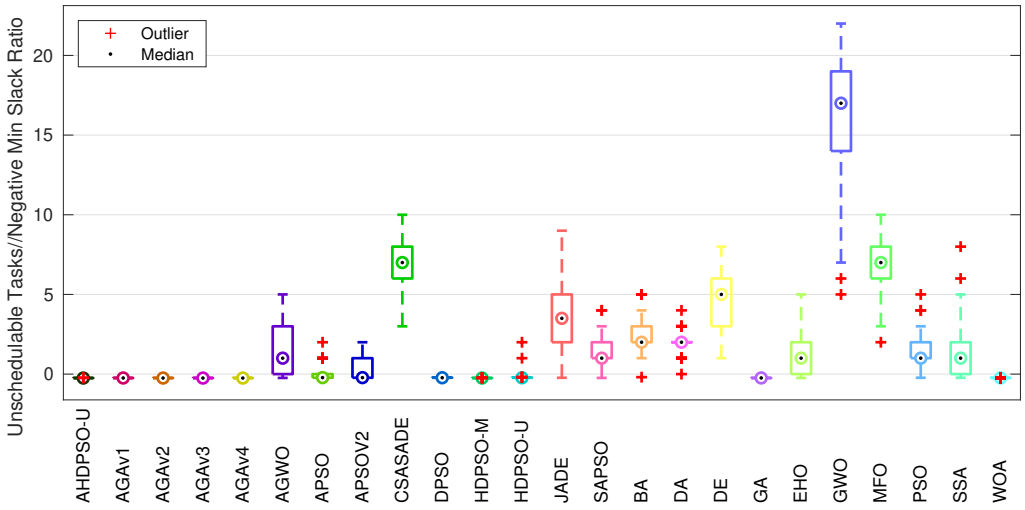


Figura 7.11 – $(4x4)$ Box-plot for $f_{umsr}(\boldsymbol{x}, \boldsymbol{\Omega_1}, \boldsymbol{\Psi_3})$ where $\boldsymbol{x} \in \boldsymbol{B}$, and $\boldsymbol{B}$ are the set of best results generated by each meta-heuristic during their 50 executions.

7.6.5 Multi-Objective Scheduling with Slack, Energy Dissipation, and Memory Requirement Awareness

This experiment aims to assess which multi-objective meta-heuristic performs on average better than the other when optimizing the multi-objective function $\boldsymbol{F}_{noc}$ (Section 3.8.8). A multi-objective meta-heuristic optimizing $\boldsymbol{F}_{noc}$ searches for non-dominant task mapping solutions. These solutions

are capable of optimizing at the same time (a) the number of unschedulable tasks/minimum slack deadline ratio, (b) memory requirement, (c) communication architecture dissipated energy and (d) the number of processor cores and links overburdened.

Similarly to other NoC related experiments, this experiment used 55 task mapping problems using the RTAs and Platforms present at Table 7.7 and Table 7.6, respectively. Since this problem is COP, it is possible to use NSAGA (Section 4.6.1) along with NSGA-II, MONSADE, and APMTMODE.

Similar to the multi-objective experiment that used benchmark functions, this experiment uses the median of the IGD metric to evaluate the quality of the PO solutions resulted from each meta-heuristic to use the Friedman test then. However, these problems Pareto front is not known, and it limits the use of IGD. In this work, the solution for the lack of PF to use the IGD metric is to generate a "fake" PF using all evaluations performed by each one of the meta-heuristics in use to generate a non-dominated set of solutions to be used as PF. In total, this "fake" PF is generated using the $500,000 \times 4$ evaluations of $\boldsymbol{F}_{noc}$, since this experiment uses four meta-heuristics: (a) NSGA-II, (b) NSAGA, (c) APMTMODE, and (d) MONSADE. The metric of performance is the median IGD of each non-dominant solution generated by each meta-heuristic using this "fake" PF.

The list of average ranks calculated using the IGD medians is present in Table 7.16. The Friedman statistic calculated for this experiment was χ_f^2 is 31.647 resulting in a p-value of 1×10^{-6} and consequently rejecting the null hypothesis using a significance level α of 0.05. Since the Friedman null hypothesis was rejected, it shows that these four meta-heuristics are statistically different. Afterward, it is possible to use posthoc methods to assess how different are the other meta-heuristics performance compared to NSAGA. Table 7.17 shows the calculated p-values for the hypothesis test for each pair of meta-heuristic vs. NSAGA where the null hypothesis is that both are equal, and the alternative hypothesis is that they are different.

Tabela 7.16 – Average Rankings for the multi-objective meta-heuristics optimizing different instances of $\boldsymbol{F}_{noc}$.

Meta-Heuristic	Ranking
NSAGA	**1.8182**
MONSADE-L	2.2909
APMTMODE	2.7818
NSGAII	3.1091

Tabela 7.17 – Adjusted p-values for pair-wise comparison against NSAGA

Index	Meta-Heuristic	Unadjusted p	p_{Finner}	p_{Li}
1	NSGAII	0	0	0
2	APMTMODE	0.000091	0.000136	0.000096
3	MONSADE	0.054829	0.054829	0.054829

The p-values obtained by the posthoc methods indicates with a significance level of α of 5%

that NSAGA is significantly better than APMTMODE and NSGA-II optimizing the problems in this experiment. But not significantly different when compared with MONSADE.

7.7 Conclusions

This chapter presents the experiments and their respective results for the multiple meta-heuristics implemented in this work optimizing a variety of single-/multi-objective problems, such as, for example, the continuous single-/multi-objective benchmark functions present in Chapter 2, and RTNoC related optimization functions defined in Chapter 3. This chapter emphasis is on the latter set of problems.

For statistical analysis for the results obtained by each meta-heuristic, the Friedman non-parametric test allied with post-hoc methods are used to determine the significance of differences between the multiple algorithms. Also, the results of obtained in the experiment show statistical evidence that, in average adaptive meta-heuristics results in better performance than their counterparts with static parameters for a myriad of optimization problems. However, it is essential to remind the reader that due to the NFL theorem, the use of an adaptive meta-heuristic instead of one with static parameters is no guarantee of obtaining the best results. The selection of meta-heuristics, in general, requires an analysis to identify which meta-heuristic responds on average better than the others to a family of problems.

8 CONCLUSIONS

The problem of searching for task placements of a real-time application onto a Many-/Multi-Processor System-on-a-Chip (MPSoC) components complies with its application time requirements is NP-hard. For this reason, this problem has no exact algorithm that can find task placements with guaranteed schedulability. In this context, the use of bio-inspired meta-heuristic is capable of achieving solutions that are both schedulable as well as optimized in other features of the system design, by using multi-objective functions, by using metaphors based on biological systems. These meta-heuristics search for promising task mapping solutions via the total possible solutions exploration while maintaining and exploiting the known ones. Between the set of bio-inspired meta-heuristics are those that are capable of searching for task mapping solutions and at the same time optimize its internal parameters to specialize its heuristic operators to the problem at hand. This type of meta-heuristics uses adaptive techniques on both its parameters and internal heuristic mechanisms.

This work goal was the study, implementation, and development of bio-inspired meta-heuristics that use these adaptive techniques applied to the problem of task placement of real-time applications onto MPSoC that uses RTNoC as their communication architecture. To fulfill this goal, the author has also made the following contributions:
(a) the development of a software framework to quickly and reliably experiment on different meta-heuristics at the same time being capable of being easily expanded due to its object-oriented nature and also allows use in a range of systems due to its use only on standard libraries of C++11. The framework source code is openly available in the repository:
`https://gitlab.com/jesseh.barreto/brasbomf`;
(b) the development of meta-heuristics that use adaptive techniques based on multiple bio-inspired existing in the literature;
(c) the conception of single-multi-objective functions that through the use of static analysis of the system is capable of evaluating timing characteristics such as the slack for schedulable tasks allied with other design features such as resource utilization and energy dissipated.
(d) the experimental comparison for all implemented meta-heuristics, including developed in this work and the ones from the literature, applied to benchmark functions and the problem of task placement for MPSoCs using NoC.

The experiment results show evidence that, for the set of problems used, adaptive bio-inspired meta-heuristics are in average statistically more prone to obtain equal or better results without the necessity of a parameter tuning when compared with the ones that do no contain adaptive mechanisms. These results are promising to the application of meta-heuristics to find suitable solutions for the task mapping problem used. They also open possible future additions for this work, as shown in the following Section 8.1.

The statical analysis in the problems used shows that, on average, the use of AGAv4 (Section 4.4.5) obtain competitive results when optimizing the following single-objective functions: (a)

f_{unsch} that considers the number of unschedulable tasks (Section 3.8.3); (b) f_{bdf} that considers the breakdown frequency scaling factor (Section 3.8.6); (c) f_{umsr} that considers the number of schedulable tasks with slack awareness (Section 3.8.7). The experiments also display that on average meta-heuristics that fall onto the evolutionary algorithm category are more capable of obtaining better performance than its swarm intelligence counterparts.

For the multi-objective case, the analysis of the experimental results suggests the use of NSAGA (Section 4.6.1) when optimizing multiple aspects of the NoC design at the same time. However, both NSAGA and MONSADE (Section 4.6.2) obtained statistically comparable results on average, i.e., there was no evidence whether both of them are different.

8.1 Future Works

This work has the following possible future works:

- **Extension of the experimental setup for larger task mapping problems**: Addition of an experimental setup in which analyze the meta-heuristics performance in a setting with applications containing hundreds of tasks being mapped to MPSoC platforms that have hundreds of processor cores. This additional experimental setup would be capable of showing whether the results obtained in our experiments are scalable with the problem.
- **Modification (possibly a reduction) of the static analysis of the system to be used in a dynamic task mapping setting**: Modification of the static analysis used to use the adaptive bio-inspired meta-heuristic based approaches on a dynamic scheduler for MPSoCs based on NoC where tasks are added or removed continuously and need to be mapped on the system while it is executing these tasks.
- **Extension of the experimental setup using more benchmark real-time application**: Apart from the real-time application synthetically generated, future works may add for the experiments using task mapping problems the use of benchmark RTA used in specific real-world problems such as, for example, a real-time encoding of MP4 videos, fast Fourier/-cosine transformations or matrix related operations. These example problems are commonly used in image processing related applications.
- **Extension of the static analysis used to cover heterogeneous systems**: Addition of features for the model to consider cases in which processor cores run with different frequencies or specialized for types of tasks, for example, processor elements specialized in video coding/encoding having more affinity to tasks of this nature.
- **Exploration of hyper-heuristic based approaches**: Besides the exploration of adaptive techniques, future works may explore the combination of adaptive parameter control and adaptive selection of operations not limited to a single meta-heuristic, but instead, incorporating multiple completely different bio-inspired meta-heuristics in a class of approximate

algorithm called hyper-heuristic [105]. For example, a hyper-heuristic that combines operators of both WOA (Section 4.3.9) with SOAMSDE (4.4.1) would possibly cover problem cases in which WOA have better performance and cases where SOAMDE works better.

- **Addition of more multi-objective meta-heuristics**: This work contains only two multi-objective meta-heuristics from literature, namely, NSGA-II (Section 4.5.1) and APMT-MODE (Section 4.5.2). It would be interesting to compare whether other meta-heuristics such as SPEA2 [106] and MOPSO [107] are capable of achieving results even better than those obtained by the developed meta-heuristic NSAGA (Section 4.6.1).

- **Parameter convergence study for adaptive techniques**: Conclusions drawn in this work coming from the experimental results are mainly based on a non-parametric static study for the meta-heuristics used. It would also be useful to study the behavior of the search process of parameters obtained by the adaptive techniques using, for example, Markov processes. This approach would have a two-fold advantage with a possible better understanding of the search mechanisms as well as the possible development of meta-heuristics with more effective adaptation mechanisms.

- **Inclusion of a simulation analysis**: As previously mentioned in Chapter 3, the advantages of static analysis is the speed of evaluation for the system that enables search-based meta-heuristics to cover a broad set of possible task mapping solutions in a short amount of time. However, as a natural continuation for this work, it would be interesting to develop a simulation platform using an RTNoC with the characteristics required in the models used to confirm the task mapping solutions obtained. This simulation platform would be capable of evaluating characteristics not evaluated in our model and also integrating the optimization process from the static analysis to the generation of hardware description of the systems used.

BIBLIOGRAPHIC REFERENCES

[1] PASRICHA, S.; DUTT, N. *On-chip communication architectures: system on chip interconnect*. [S.l.]: Morgan Kaufmann, 2010.

[2] MARCULESCU, R. et al. Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 28, n. 1, p. 3–21, jan 2009. ISSN 0278-0070. Available from Internet: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4723644 http://ieeexplore.ieee.org/document/4723644/>.

[3] ARM. *AMBA 2 Overview*. ARM Developer - AMBA 2 Specification, `https://developer.arm.com/architectures/system-architectures/amba/amba-2`. Retrieved July 18, 2019.

[4] HESHAM, S. et al. Survey on Real-Time Networks-on-Chip. *IEEE Transactions on Parallel and Distributed Systems*, v. 28, n. 5, p. 1500–1517, may 2017. ISSN 1045-9219.

[5] DALLY, W. J. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, v. 3, n. 2, p. 194–205, 1992. ISSN 10459219.

[6] BUTTAZZO, G. C. *Hard real-time computing systems: predictable scheduling algorithms and applications*. [S.l.]: Springer Science & Business Media, 2011. v. 24.

[7] Del Ser, J. et al. Bio-inspired computation: Where we stand and what's next. *Swarm and Evolutionary Computation*, v. 48, p. 220–250, aug 2019. ISSN 22106502. Available from Internet: <https://linkinghub.elsevier.com/retrieve/pii/S2210650218310277>.

[8] BENINI, L.; De Micheli, G. *Networks on Chips*. [S.l.]: Elsevier, 2006. 408 p. ISBN 9780123705211.

[9] QUALCOMM. *Qualcomm Snapdragon 855 Mobile Processor*. Qualcomm Products, `https://www.qualcomm.com/products/snapdragon-855-mobile-platform`. Retrieved July 18, 2019.

[10] PASRICHA, S.; DUTT, N. *On-Chip Communication Architectures : System On Chip Interconnect*. [S.l.]: Morgan Kaufmann, 2008. 544 p. ISBN 9780123738929.

[11] De Micheli, G.; BENINI, L. Networks on Chips: 15 Years Later. *Computer*, v. 50, n. 5, p. 10–11, may 2017. ISSN 0018-9162.

[12] SINGH, A. K. et al. Mapping on multi many core systems: Survey of current and emerging trends. *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, p. 1–10, 2013. ISSN 0738-100X.

[13] GAREY, M. R.; JOHNSON, D. S. *Computers and intractability*. [S.l.]: wh freeman New York, 2002. v. 29. 236–241 p.

[14] RACU, A.; INDRUSIAK, L. S. Using genetic algorithms to map hard real-time on NoC-based systems. In: *7th International Workshop on Reconfigurable and Communication-Centric*

Systems-on-Chip (ReCoSoC). IEEE, 2012. p. 1–8. ISBN 978-1-4673-2572-1. Available from Internet: <http://ieeexplore.ieee.org/document/6322893/>.

[15] AMD. *AMD, The "Zen"Core Architecture.* AMD, `https://www.amd.com/en/technologies/zen-core`. Retrieved July 22, 2019.

[16] INTEL. *Intel, Second Generation Intel Xeon Scalable Processors.* Intel, `https://www.intel.com/content/www/us/en/products/docs/processors/xeon/2nd-gen-xeon-scalable-processors-brief.html`. Retrieved July 22, 2019.

[17] BARROS, J. B.; QUINTERO, C. H. L.; SAMPAIO, R. C. An adaptive discrete particle swarm optimization for mapping real-time applications onto network-on-chip based mpsocs. In: ACM. *32nd Symposium on Integrated Circuits and Systems Design (SBCCI '19), August 26–30, 2019, Sao Paulo, Brazil*. [S.l.], 2019.

[18] SALCEDO-SANZ, S. Modern meta-heuristics based on nonlinear physics processes: A review of models and design procedures. *Physics Reports*, Elsevier, v. 655, p. 1–70, 2016.

[19] MIETTINEN, K. et al. *Multiobjective Optimization: Interactive and Evolutionary Approaches*. 1. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 2008. (Lecture Notes in Computer Science 5252 : Theoretical Computer Science and General Issues). ISBN 9783540889076,3540889078.

[20] LAWLER, E. L.; WOOD, D. E. Branch-and-bound methods: A survey. *Operations research*, INFORMS, v. 14, n. 4, p. 699–719, 1966.

[21] EDDY, S. R. What is dynamic programming? *Nature Biotechnology*, v. 22, n. 7, p. 909–910, jul 2004. ISSN 1087-0156. Available from Internet: <http://www.nature.com/articles/nbt0704-909>.

[22] GENT, I. P.; JEFFERSON, C.; NIGHTINGALE, P. Complexity of n-Queens Completion. *Journal of Artificial Intelligence Research*, v. 59, p. 815–848, aug 2017. ISSN 1076-9757. Available from Internet: <https://jair.org/index.php/jair/article/view/11079>.

[23] STONE, H. S.; STONE, J. M. Efficient search techniques-an empirical study of the n-queens problem. *IBM Journal of Research and Development*, v. 31, n. 4, p. 464–474, July 1987. ISSN 0018-8646.

[24] LAZAROVA, M. Efficiency of parallel genetic algorithm for solving n-queens problem on multicomputer platform. In: *the 9th wseas international conference on evolutionary computing*. [S.l.: s.n.], 2008. p. 51–56.

[25] XIAOHUI, H.; EBERHART, R. C.; YUHUI, S. Swarm intelligence for permutation optimization: a case study of n-queens problem. In: *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No.03EX706)*. IEEE, 2015. p. 243–246. ISBN 0-7803-7914-4. Available from Internet: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1202275 http://ieeexplore.ieee.org/document/1202275/>.

[26] HEINEMAN, G. T.; POLLICE, G.; SELKOW, S. *Algorithms in a nutshell: A practical guide*. [S.l.]: "O'Reilly Media, Inc.", 2016. 105-142 p.

[27] AWAD, N. H. et al. *Problem Definitions and Evaluation Criteria for the CEC 2017 Special Session and Competition on Single-Objective Real-Parameter Numerical Optimization*. 2017.

[28] CHENG, R. et al. *Benchmark Functions for the CEC2017 Competition on Evolutionary Many-Objective Optimization*. 2017. Available from Internet: <http://www.cercia.ac.uk/news/cec2017maooc/CEC2017-MaOO-Tech-Report.pdf>.

[29] CADENA, C. et al. Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age. *IEEE Transactions on Robotics*, v. 32, n. 6, p. 1309–1332, dec 2016. ISSN 1552-3098. Available from Internet: <http://ieeexplore.ieee.org/document/7747236/>.

[30] WOLF, W.; JERRAYA, A.; MARTIN, G. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 27, n. 10, p. 1701–1713, oct 2008. ISSN 0278-0070.

[31] BORKAR, S. Thousand core chips. In: *Proceedings of the 44th annual conference on Design automation - DAC '07*. New York, New York, USA: ACM Press, 2007. p. 746. ISBN 9781595936271. ISSN 0738100X. Available from Internet: <http://portal.acm.org/citation.cfm?doid=1278480.1278667>.

[32] XILINX. *ADAS Solutions Powered by Xilinx*. Xilinx Application in Autonomous Driver Assist, `https://www.xilinx.com/applications/megatrends/automotive-driver-assist.html`. Retrieved July 18, 2019.

[33] KAVALDJIEV, N.; SMIT, G. J. M. A Survey of Efficient On-Chip Communications for SoC. *Distributed and Embedded Security*, v. 41, 2003.

[34] BOLOTIN, E. et al. Cost considerations in network on chip. *Integration*, v. 38, n. 1, p. 19–42, oct 2004. ISSN 01679260. Available from Internet: <https://linkinghub.elsevier.com/retrieve/pii/S0167926004000343>.

[35] GOOSSENS, K. et al. Guaranteeing the quality of services in networks on chip. In: *Networks on chip*. [S.l.]: Springer, 2003. p. 61–82.

[36] MELLO, A. et al. Virtual channels in networks on chip: Implementation and evaluation on hermes NoC. *SBCCI 2005 - 18th Symposium on Integrated Circuits and Systems Design*, p. 178–183, 2005.

[37] BURNS, A.; WELLINGS, A. J. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. [S.l.]: Pearson Education, 2001.

[38] DAVIS, R. I.; BURNS, A. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, v. 43, n. 4, p. 1–44, oct 2011. ISSN 03600300. Available from Internet: <http://dl.acm.org/citation.cfm?doid=1978802.1978814>.

[39] CARPENTER, J. et al. A categorization of real-time multiprocessor scheduling problems and algorithms. In: *Handbook on Scheduling Algorithms, Methods, and Models*. [S.l.]: Chapman Hall/CRC, Boca, 2004.

[40] LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, ACM, v. 20, n. 1, p. 46–61, 1973.

[41] DEVILLERS, R.; GOOSSENS, J. Liu and layland's schedulability test revisited. *Information Processing Letters*, Elsevier, v. 73, n. 5-6, p. 157–161, 2000.

[42] HORN, W. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, Wiley Online Library, v. 21, n. 1, p. 177–185, 1974.

[43] AUDSLEY, N. et al. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, v. 8, n. 5, p. 284–292, Sep. 1993. ISSN 0268-6961.

[44] INDRUSIAK, L. S.; BURNS, A.; NIKOLIĆ, B. Buffer-aware bounds to multi-point progressive blocking in priority-preemptive nocs. In: IEEE. *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. [S.l.], 2018. p. 219–224.

[45] INDRUSIAK, L. S. End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. *Journal of Systems Architecture*, Elsevier B.V., v. 60, n. 7, p. 553–561, 2014. ISSN 13837621. Available from Internet: <http://dx.doi.org/10.1016/j.sysarc.2014.05.002>.

[46] SHI, Z.; BURNS, A. Priority assignment for real-time wormhole communication in on-chip networks. *Proceedings - Real-Time Systems Symposium*, p. 421–430, 2008. ISSN 10528725.

[47] XIONG, Q. et al. Extending real-time analysis for wormhole nocs. *IEEE Transactions on Computers*, IEEE, v. 66, n. 9, p. 1532–1546, 2017.

[48] XIONG, Q. et al. Real-time analysis for wormhole noc: Revisited and revised. In: IEEE. *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*. [S.l.], 2016. p. 75–80.

[49] STILL, L. R.; INDRUSIAK, L. S. Memory-aware genetic algorithms for task mapping on hard real-time networks-on-chip. In: IEEE. *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. [S.l.], 2018. p. 601–608.

[50] Byungjae Kim et al. A real-time communication method for wormhole switching networks. In: *Proceedings. 1998 International Conference on Parallel Processing (Cat. No.98EX205)*. IEEE Comput. Soc, 1998. p. 527–534. ISBN 0-8186-8650-2. Available from Internet: <http://ieeexplore.ieee.org/document/708526/>.

[51] INDRUSIAK, L. S.; BURNS, A.; NIKOLIC, B. Analysis of buffering effects on hard real-time priority-preemptive wormhole networks. *arXiv preprint arXiv:1606.02942*, 2016.

[52] SAYUTI, M. N. S. M.; INDRUSIAK, L. S. *Real-time low-power task mapping in Networks-on-Chip*. IEEE, 2013. 14–19 p. Available from Internet: <http://ieeexplore.ieee.org/document/6654616/>.

[53] HANSSON, A.; GOOSSENS, K.; RĂDULESCU, A. A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic. *VLSI design*, Hindawi, v. 2007, 2007.

[54] SAYUTI, M. N. S. M.; INDRUSIAK, L. S. A Function for Hard Real-Time System Search-Based Task Mapping Optimisation. In: *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, 2015. v. 18, p. 66–73. ISBN 978-1-4799-8781-8. Available from Internet: <http://ieeexplore.ieee.org/document/7153791/>.

[55] MEJIA-ALVAREZ, P.; LEVNER, E.; MOSSÉ, D. Adaptive scheduling server for power-aware real-time tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, ACM, v. 3, n. 2, p. 284–306, 2004.

[56] DESALE, S. et al. Heuristic and meta-heuristic algorithms and their relevance to the real world: a survey. *Int. J. Comput. Eng. Res. Trends*, Citeseer, v. 351, n. 5, p. 2349–7084, 2015.

[57] MACREADY, W. G.; WOLPERT, D. H. What makes an optimization problem hard? *Complexity*, v. 1, n. 5, p. 40–46, 1996. ISSN 10990526.

[58] WOLPERT, D. H.; MACREADY, W. G. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, v. 1, n. 1, p. 67–82, 1997. ISSN 1089778X.

[59] EIBEN, A.; HINTERDING, R.; MICHALEWICZ, Z. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, v. 3, n. 2, p. 124–141, jul 1999. ISSN 1089778X. Available from Internet: <http://ieeexplore.ieee.org/document/6322893/The paramete>.

[60] KARAFOTIAS, G.; HOOGENDOORN, M.; EIBEN, A. E. Parameter Control in Evolutionary Algorithms: Trends and Challenges. *IEEE Transactions on Evolutionary Computation*, v. 19, n. 2, p. 167–187, 2015. ISSN 1089778X.

[61] SHI, Y.; EBERHART, R. A modified particle swarm optimizer. In: *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*. IEEE, 1998. p. 69–73. ISBN 0-7803-4869-9. Available from Internet: <https://doi.org/10.1109/icec.1998.699146 http://ieeexplore.ieee.org/document/699146/>.

[62] HE, J.; YAO, X. From an individual to a population: An analysis of the first hitting time of population-based evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, v. 6, n. 5, p. 495–511, 2002. ISSN 1089778X.

[63] DIAZ-GOMEZ, P. A.; HOUGEN, D. Initial Population for Genetic Algorithms: A Metric Approach. *Proceedings of the 2007 International Conference on Genetic and Evolutionary Methods*, p. 43–49, 2007.

[64] CHEN, T. et al. A large population size can be unhelpful in evolutionary algorithms. *Theoretical Computer Science*, Elsevier B.V., v. 436, p. 54–70, 2012. ISSN 03043975. Available from Internet: <http://dx.doi.org/10.1016/j.tcs.2011.02.016>.

[65] TANABE, R.; FUKUNAGA, A. S. Improving the Search Performance of SHADE Using Linear Population Size Reduction. In: *IEEE Congress on Evolutionary Computation*. Beijing, China: IEEE, 2014.

[66] Razali N M; Geraghty J. Genetic algorithm performance with different selection strategies in solving TSP. *Proceedings of the world congress on engineering*, p. 1134–1139, 2011. Available from Internet: <https://pdfs.semanticscholar.org/010b/545848cfd29fe6e83987d494fdd00b486229.pdf>.

[67] STORN, R.; PRICE, K. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, v. 11, n. 4, p. 341–359, 1997. Available from Internet: <https://doi.org/10.1023/A:1008202821328>.

[68] KENNEDY, J.; EBERHART, R. Particle Swarm Optimization. *Proceedings of the 1995 IEEE International Conference on Neural Networks*, v. 4, p. 1942–1948, 1995. ISSN 1935-3812. Available from Internet: <http://www.scopus.com/inward/record.url?eid=2-s2.0-0029535737&partnerID=40&md5=e6bf04ae50f3268ae545d88ed91d1fc5>.

[69] CLERC, M.; KENNEDY, J. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, v. 6, n. 1, p. 58–73, 2002. ISSN 1089-778X.

[70] MIRJALILI, S. et al. Salp Swarm Algorithm: A bio-inspired optimizer for engineering design problems. *Advances in Engineering Software*, Elsevier Ltd, v. 114, p. 163–191, 2017. ISSN 18735339. Available from Internet: <http://dx.doi.org/10.1016/j.advengsoft.2017.07.002>.

[71] MIRJALILI, S.; MIRJALILI, S. M.; LEWIS, A. Grey Wolf Optimizer. *Advances in Engineering Software*, Elsevier Ltd, v. 69, p. 46–61, 2014. ISSN 09659978. Available from Internet: <http://dx.doi.org/10.1016/j.advengsoft.2013.12.007>.

[72] WANG, G. G. et al. A new metaheuristic optimisation algorithm motivated by elephant herding behaviour. *International Journal of Bio-Inspired Computation*, v. 8, n. 6, p. 394, 2016. ISSN 1758-0366. Available from Internet: <http://www.inderscience.com/link.php?id=10002274>.

[73] MIRJALILI, S. Dragonfly algorithm: a new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems. *Neural Computing and Applications*, Springer London, v. 27, n. 4, p. 1053–1073, may 2016. ISSN 0941-0643. Available from Internet: <http://link.springer.com/10.1007/s00521-015-1920-1>.

[74] MIRJALILI, S. Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm. *Knowledge-Based Systems*, Elsevier B.V., v. 89, p. 228–249, 2015. ISSN 0950-7051. Available from Internet: <https://doi.org/10.1016/j.knosys.2015.07.006>.

[75] YANG, X. S.; GANDOMI, A. H. Bat algorithm: A novel approach for global engineering optimization. *Engineering Computations (Swansea, Wales)*, v. 29, n. 5, p. 464–483, 2012. ISSN 02644401.

[76] ZHANG, J.; SANDERSON, A. C. JADE: Adaptive Differential Evolution with Optional External Archive. *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, v. 13, n. 5, p. 945–958, 2009.

[77] FAN, Q.; ZHANG, Y. Self-adaptive differential evolution algorithm with crossover strategies adaptation and its application in parameter estimation. *Chemometrics and Intelligent Laboratory Systems*, Elsevier B.V., v. 151, n. 1550, p. 164–171, 2016. ISSN 18733239. Available from Internet: <http://dx.doi.org/10.1016/j.chemolab.2015.12.020>.

[78] KANG, Q. et al. A Novel Discrete Particle Swarm Optimization Algorithm for Job Scheduling in Grids. n. 60534060, p. 401–405, 2008.

[79] MONTALVO, I. et al. Improved performance of PSO with self-adaptive parameters for computing the optimal design of Water Supply Systems. *Engineering Applications of Artificial Intelligence*, v. 23, n. 5, p. 727–735, 2010. ISSN 09521976.

[80] KASHAN, A. H.; KARIMI, B. A discrete particle swarm optimization algorithm for scheduling parallel machines. *Computers & Industrial Engineering*, Elsevier Ltd, v. 56, n. 1, p. 216–223, 2009. ISSN 0360-8352. Available from Internet: <http://dx.doi.org/10.1016/j.cie.2008.05.007>.

[81] GRAHAM, R. L. Bounds On Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*, v. 17, n. 2, p. 416–429, 1969. Available from Internet: <https://doi.org/10.1137/0117039>.

[82] SANTOS, C. *SELEÇÃO DE PARÂMETROS DE MÁQUINAS DE VETORES DE SUPORTE USANDO OTIMIZAÇÃO MULTIOBJETIVO BASEADA EM METAHEURÍSTICAS*. Tese (Doutorado) — Universidade de Brasília, 2019.

[83] DEB, K.; AGRAWAL, R. B. Simulated binary crossover for continuous search space. *Complex systems*, [Champaign, IL, USA: Complex Systems Publications, Inc., c1987-, v. 9, n. 2, p. 115–148, 1995.

[84] EIBEN, A.; SCHUT, M. C.; WILDE, A. de. Is self-adaptation of selection pressure and population size possible?–a case study. In: *Parallel problem solving from nature-PPSN IX*. [S.l.]: Springer, 2006. p. 900–909.

[85] SAYUTI, M. N. S. M.; INDRUSIAK, L. S. A Constructive Task Mapping Algorithm for Hard Real-Time Embedded NoCs. *Proceedings - 2015 IEEE Conference on System, Process and Control, ICSPC 2015*, n. December, p. 123–128, 2016.

[86] Deb, K. et al. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, v. 6, n. 2, p. 182–197, April 2002. ISSN 1089-778X.

[87] GOUDOS, S. K. et al. A comparative study of common and self-adaptive Differential Evolution strategies on numerical benchmark problems. *Procedia Computer Science*, Elsevier, v. 3, p. 83–88, 2011. ISSN 18770509. Available from Internet: <http://dx.doi.org/10.1016/j.procs.2010.12.015>.

[88] ZHAO, Z. et al. A differential evolution algorithm with self-adaptive strategy and control parameters based on symmetric Latin hypercube design for unconstrained optimization problems. *European Journal of Operational Research*, Elsevier B.V., v. 250, n. 1, p. 30–45, 2016. ISSN 03772217. Available from Internet: <http://dx.doi.org/10.1016/j.ejor.2015.10.043>.

[89] WANG, J. et al. The model of chaotic sequences based on adaptive particle swarm optimization arithmetic combined with seasonal term. *Applied Mathematical Modelling*, Elsevier Inc., v. 36, n. 3, p. 1184–1196, 2012. ISSN 0307904X. Available from Internet: <http://dx.doi.org/10.1016/j.apm.2011.07.089>.

[90] JUANG, Y.-T.; TUNG, S.-L.; CHIU, H.-C. Adaptive fuzzy particle swarm optimization for global optimization of multimodal functions. *Information Sciences*, Elsevier, v. 181, n. 20, p. 4539–4549, 2011.

[91] SAHU, P. K.; CHATTOPADHYAY, S. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture*, Elsevier B.V., v. 59, n. 1, p. 60–76, 2013. ISSN 13837621. Available from Internet: <http://dx.doi.org/10.1016/j.sysarc.2012.10.004>.

[92] ASCIA, G.; CATANIA, V.; PALESI, M. Multi-objective Mapping for Mesh-based NoC Architectures. 2004.

[93] BRUCH, J. V. et al. Deadline, Energy and Buffer-Aware Task Mapping Optimization in NoC-Based SoCs Using Genetic Algorithms. In: *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*. IEEE, 2017. v. 2017-Novem, p. 86–93. ISBN 978-1-5386-3590-2. ISSN 23247894. Available from Internet: <http://ieeexplore.ieee.org/document/8116564/>.

[94] DURILLO, J. J.; NEBRO, A. J. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, v. 42, p. 760–771, 2011. ISSN 0965-9978. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0965997811001219>.

[95] IGEL, C.; HEIDRICH-MEISNER, V.; GLASMACHERS, T. Shark. *Journal of Machine Learning Research*, v. 9, p. 993–996, 2008.

[96] GALIB. *GAlib - A C++ Library of Genetic Algorithm Components*. GAlib, `http://lancet.mit.edu/ga/`. Retrieved July 22, 2019.

[97] GCC. *GNU, the GNU Compiler Collection*. GCC, `https://gcc.gnu.org/`. Retrieved July 22, 2019.

[98] CMAKE. *CMake - A cross-platform tool to build, test and package software.* CMake, `https://cmake.org/`. Retrieved July 22, 2019.

[99] DERRAC, J. et al. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, Elsevier B.V., v. 1, n. 1, p. 3–18, 2011. ISSN 22106502. Available from Internet: <http://dx.doi.org/10.1016/j.swevo.2011.02.002>.

[100] SHESKIN, D. J. *Handbook of parametric and nonparametric statistical procedures.* [S.l.]: Chapman and Hall/CRC, 2003.

[101] FINNER, H. On a Monotonicity Problem in Step-Down Multiple Test Procedures. *Source Journal of the American Statistical Association*, v. 88, n. 423, p. 920–923, 1993. ISSN 0162-1459.

[102] LI, J. D. A two-step rejection procedure for testing multiple hypotheses. *Journal of Statistical Planning and Inference*, v. 138, n. 6, p. 1521–1527, 2008. Available from Internet: <https://doi.org/10.1016/j.jspi.2007.04.032>.

[103] COELLO, C. A. C.; SIERRA, M. R. A study of the parallelization of a coevolutionary multi-objective evolutionary algorithm. In: SPRINGER. *Mexican International Conference on Artificial Intelligence.* [S.l.], 2004. p. 688–697.

[104] SHI, Z.; BURNS, A.; INDRUSIAK, L. S. Schedulability Analysis for Real Time On-Chip Communication with Wormhole Switching. *International Journal of Embedded and Real-Time Communication Systems*, v. 1, n. 2, p. 1–22, 2010. ISSN 1947-3176.

[105] BURKE, E. K. et al. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, Taylor & Francis, v. 64, n. 12, p. 1695–1724, 2013.

[106] ZITZLER, E.; LAUMANNS, M.; THIELE, L. Spea2: Improving the strength pareto evolutionary algorithm. *TIK-report*, Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische ..., v. 103, 2001.

[107] SIERRA, M. R.; COELLO, C. A. C. Improving pso-based multi-objective optimization using crowding, mutation and epsilon-dominance. In: SPRINGER. *International conference on evolutionary multi-criterion optimization.* [S.l.], 2005. p. 505–519.

APPENDICES

I. BREAKDOWN FREQUENCY SCALING VALUES

List of 255 breakdown frequency values used in this work ordered from smallest to greatest starting with index $i = 1$ until $i = 255$ and values ranging from 0.01 to 100.

Tabela I.1 – List with the 255 frequency scaling values used.

i	Scale	i	Scale	i	Scale
1	1.0000000e-02	86	4.2187500e-01	171	2.6875000e+00
2	1.0685484e-02	87	4.2968750e-01	172	2.7500000e+00
3	1.1370968e-02	88	4.3750000e-01	173	2.8125000e+00
4	1.2056452e-02	89	4.4531250e-01	174	2.8750000e+00
5	1.2741935e-02	90	4.5312500e-01	175	2.9375000e+00
6	1.3427419e-02	91	4.6093750e-01	176	3.0000000e+00
7	1.4112903e-02	92	4.6875000e-01	177	3.0625000e+00
8	1.4798387e-02	93	4.7656250e-01	178	3.1250000e+00
9	1.5483871e-02	94	4.8437500e-01	179	3.1875000e+00
10	1.6169355e-02	95	4.9218750e-01	180	3.2500000e+00
11	1.6854839e-02	96	5.0000000e-01	181	3.3125000e+00
12	1.7540323e-02	97	5.1562500e-01	182	3.3750000e+00
13	1.8225806e-02	98	5.3125000e-01	183	3.4375000e+00
14	1.8911290e-02	99	5.4687500e-01	184	3.5000000e+00
15	1.9596774e-02	100	5.6250000e-01	185	3.5625000e+00
16	2.0282258e-02	101	5.7812500e-01	186	3.6250000e+00
17	2.0967742e-02	102	5.9375000e-01	187	3.6875000e+00
18	2.1653226e-02	103	6.0937500e-01	188	3.7500000e+00
19	2.2338710e-02	104	6.2500000e-01	189	3.8125000e+00
20	2.3024194e-02	105	6.4062500e-01	190	3.8750000e+00
21	2.3709677e-02	106	6.5625000e-01	191	3.9375000e+00
22	2.4395161e-02	107	6.7187500e-01	192	4.0000000e+00
23	2.5080645e-02	108	6.8750000e-01	193	4.8750000e+00
24	2.5766129e-02	109	7.0312500e-01	194	5.7500000e+00
25	2.6451613e-02	110	7.1875000e-01	195	6.6250000e+00
26	2.7137097e-02	111	7.3437500e-01	196	7.5000000e+00
27	2.7822581e-02	112	7.5000000e-01	197	8.3750000e+00
28	2.8508065e-02	113	7.6562500e-01	198	9.2500000e+00
29	2.9193548e-02	114	7.8125000e-01	199	1.0125000e+01
30	2.9879032e-02	115	7.9687500e-01	200	1.1000000e+01

Continued on next page

Tabela I.1 – *Continued from previous page*

i	Scale	i	Scale	i	Scale
31	3.0564516e-02	116	8.1250000e-01	201	1.1875000e+01
32	3.1250000e-02	117	8.2812500e-01	202	1.2750000e+01
33	3.8085938e-02	118	8.4375000e-01	203	1.3625000e+01
34	4.4921875e-02	119	8.5937500e-01	204	1.4500000e+01
35	5.1757812e-02	120	8.7500000e-01	205	1.5375000e+01
36	5.8593750e-02	121	8.9062500e-01	206	1.6250000e+01
37	6.5429688e-02	122	9.0625000e-01	207	1.7125000e+01
38	7.2265625e-02	123	9.2187500e-01	208	1.8000000e+01
39	7.9101562e-02	124	9.3750000e-01	209	1.8875000e+01
40	8.5937500e-02	125	9.5312500e-01	210	1.9750000e+01
41	9.2773438e-02	126	9.6875000e-01	211	2.0625000e+01
42	9.9609375e-02	127	9.8437500e-01	212	2.1500000e+01
43	1.0644531e-01	128	1.0000000e+00	213	2.2375000e+01
44	1.1328125e-01	129	1.0312500e+00	214	2.3250000e+01
45	1.2011719e-01	130	1.0625000e+00	215	2.4125000e+01
46	1.2695312e-01	131	1.0937500e+00	216	2.5000000e+01
47	1.3378906e-01	132	1.1250000e+00	217	2.5875000e+01
48	1.4062500e-01	133	1.1562500e+00	218	2.6750000e+01
49	1.4746094e-01	134	1.1875000e+00	219	2.7625000e+01
50	1.5429688e-01	135	1.2187500e+00	220	2.8500000e+01
51	1.6113281e-01	136	1.2500000e+00	221	2.9375000e+01
52	1.6796875e-01	137	1.2812500e+00	222	3.0250000e+01
53	1.7480469e-01	138	1.3125000e+00	223	3.1125000e+01
54	1.8164062e-01	139	1.3437500e+00	224	3.2000000e+01
55	1.8847656e-01	140	1.3750000e+00	225	3.4193548e+01
56	1.9531250e-01	141	1.4062500e+00	226	3.6387097e+01
57	2.0214844e-01	142	1.4375000e+00	227	3.8580645e+01
58	2.0898438e-01	143	1.4687500e+00	228	4.0774194e+01
59	2.1582031e-01	144	1.5000000e+00	229	4.2967742e+01
60	2.2265625e-01	145	1.5312500e+00	230	4.5161290e+01
61	2.2949219e-01	146	1.5625000e+00	231	4.7354839e+01
62	2.3632812e-01	147	1.5937500e+00	232	4.9548387e+01
63	2.4316406e-01	148	1.6250000e+00	233	5.1741935e+01
64	2.5000000e-01	149	1.6562500e+00	234	5.3935484e+01
65	2.5781250e-01	150	1.6875000e+00	235	5.6129032e+01
66	2.6562500e-01	151	1.7187500e+00	236	5.8322581e+01
67	2.7343750e-01	152	1.7500000e+00	237	6.0516129e+01
68	2.8125000e-01	153	1.7812500e+00	238	6.2709677e+01
69	2.8906250e-01	154	1.8125000e+00	239	6.4903226e+01

Continued on next page

Tabela I.1 – *Continued from previous page*

i	Scale	i	Scale	i	Scale
70	2.9687500e-01	155	1.8437500e+00	240	6.7096774e+01
71	3.0468750e-01	156	1.8750000e+00	241	6.9290323e+01
72	3.1250000e-01	157	1.9062500e+00	242	7.1483871e+01
73	3.2031250e-01	158	1.9375000e+00	243	7.3677419e+01
74	3.2812500e-01	159	1.9687500e+00	244	7.5870968e+01
75	3.3593750e-01	160	2.0000000e+00	245	7.8064516e+01
76	3.4375000e-01	161	2.0625000e+00	246	8.0258065e+01
77	3.5156250e-01	162	2.1250000e+00	247	8.2451613e+01
78	3.5937500e-01	163	2.1875000e+00	248	8.4645161e+01
79	3.6718750e-01	164	2.2500000e+00	249	8.6838710e+01
80	3.7500000e-01	165	2.3125000e+00	250	8.9032258e+01
81	3.8281250e-01	166	2.3750000e+00	251	9.1225806e+01
82	3.9062500e-01	167	2.4375000e+00	252	9.3419355e+01
83	3.9843750e-01	168	2.5000000e+00	253	9.5612903e+01
84	4.0625000e-01	169	2.5625000e+00	254	9.7806452e+01
85	4.1406250e-01	170	2.6250000e+00	255	1.0000000e+02

II. REAL-TIME APPLICATION TASK SET

List of task set for the model of real-time applications used in this work. Table II.1 contains the Autonomous Vehicle Application (AVA) task set where the fields *Cost*, *Deadline*, and *Period* are expressed in seconds, the field *Message Size* is expressed in bits, and the *Code Memory* field is expressed in bytes.

Tabela II.1 – Autonomous Vehicle Application (AVA) Benchmark.

Name	Cost (s)	Deadline (s)	Period (s)	Destination	Message Size (b)	Priority	Code Memory (B)
POSI-A	0.005	0.5	0.5	NAVC-A	16384	31	55048
NAVC-A	0.01	0.5	0.5	OBDB-A	32768	32	25088
OBDB-A	0.15	0.5	0.5	NAVC-X	262144	33	81624
OBDB-B	0.15	1	1	OBMG-B	524288	34	81624
NAVC-C	0.02	0.1	0.1	DIRC-X	8192	24	25088
SPES-C	0.005	0.1	0.1	NAVC-C	8192	25	34992
NAVC-D	0.01	0.1	0.1	DIRC-X	16384	26	25088
FBU3-E	0.01	0.04	0.04	VOD1-X	614400	1	102552
FBU8-F	0.01	0.04	0.04	VOD2-X	614400	2	102552
VOD1	0.02	0.5	0.5	NAVC-X	8192	3	78128
VOD2	0.02	0.5	0.5	NAVC-X	8192	4	78128
FBU1	0.01	0.04	0.04	BFE1	614400	5	102552
FBU2	0.01	0.04	0.04	BFE2	614400	6	102552
FBU3	0.01	0.04	0.04	BFE3	614400	7	102552
FBU4	0.01	0.04	0.04	BFE4	614400	8	102552
FBU5	0.01	0.04	0.04	BFE5	614400	9	102552
FBU6	0.01	0.04	0.04	BFE6	614400	10	102552
FBU7	0.01	0.04	0.04	BFE7	614400	11	102552
FBU8	0.01	0.04	0.04	BFE8	614400	12	102552
BFE1	0.02	0.04	0.04	FDF1	32768	13	72560
BFE2	0.02	0.04	0.04	OBMG-X	32768	14	72560
BFE3	0.02	0.04	0.04	OBMG-X	32768	15	72560
BFE4	0.02	0.04	0.04	OBMG-X	32768	16	72560
BFE5	0.02	0.04	0.04	FDF2	32768	17	72560
BFE6	0.02	0.04	0.04	THRC-X	32768	18	72560
BFE7	0.02	0.04	0.04	THRC-X	32768	19	72560
BFE8	0.02	0.04	0.04	THRC-X	32768	20	72560
FDF1	0.01	0.04	0.04	STPH	131072	21	45648
FDF2	0.01	0.04	0.04	STPH-X	131072	22	45648

Continued on next page

Tabela II.1 – *Continued from previous page*

Name	Cost (s)	Deadline (s)	Period (s)	Destination	Message Size (b)	Priority	Code Memory (B)
STPH	0.03	0.04	0.04	OBMG-X	65536	23	77008
POSI-Q	0.005	0.5	0.5	OBMG-X	16384	35	55048
USOS	0.005	0.1	0.1	OBMG-X	16384	27	26000
OBMG-B	0.02	1	1	OBDB-X	65536	37	17752
TPMS	0.005	0.5	0.5	STAC-S	32768	36	35384
VIBS	0.005	0.1	0.1	STAC-T	8192	28	33864
STAC-S	0.01	1	1	TPMS-X	32768	38	52080
SPES-U	0.005	0.1	0.1	STAC-X	16384	29	34992
STAC-T	0.01	0.1	0.1	THRC-X	16384	30	52080
OBMG-V	0.0005	1	1	OBDB-X	32768	39	17752
DIRC-X						40	
OBDB-X						41	
NAVC-X						42	
OBMG-X						43	
THRC-X						44	
STAC-X						45	
TPMS-X						46	
VOD1-X						47	
VOD2-X						48	
FDF1-X						49	
FDF2-X						50	
STPH-X						51	