Alexandre Daniel Pereira Brito

An Hypervolume Dichotomic Scheme for Multiobjective Optimization

1 2 9 0 UNIVERSIDADE D COIMBRA

1 2 9 0

UNIVERSIDADE D
COIMBRA

Alexandre Daniel Pereira Brito

An Hypervolume Dichotomic Scheme for Multiobjective Optimization

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems, advised by Professor Luís Paquete and presented to Faculty of Sciences and Technology / Department of Informatics Engineering.

June 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

An Hypervolume Dichotomic Scheme for Multiobjective Optimization

Alexandre Daniel Pereira Brito

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems advised by Prof. Luís Paquete and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering

June 2021

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Acknowledgements

First and foremost, I'd like to thanks my advisor Prof. Luís Paquete for inviting me to this project, helping and trusting in me to develop this framework. Would like to thank him, in particular, for his patience, understanding and continued availability during this last year, even when his schedule was clearly full. This project was an important and significant learning experience on one of the topics that I find most fun. I would also like to thank CISUC for the scholarship they granted to me.

A heartfelt thank you to all my colleagues who have been a significant part of my journey in this department. The last 5 years have not been easy, but all of you have made the late night rushes to deliver a project, the all-nighters that were sometimes required, the seemingly endless study sessions and of course, the all to fleeting free time a true joy to remember. In particular I'd like to thank my close friends Vera, Sofia, Martinho, Sara, Caio and Mariana. It is been five years and somehow, for some reason, you still put up with me and support me.

Last but not least, thank you to my parents for, not only giving me the opportunity to continue studying and expanding my horizons, but also strongly encourage it. You have allowed me to explore what I love. You have shown unwavering support for all I try to do and for that I am grateful. Thank you.

This page is intentionally left blank.

Abstract

In this dissertation, we propose methods to solve multiobjective optimization problems. In particular, these methods are based on the concept of hypervolume scalarization and extend previous work on the following manner: i) we describe a software framework that implements a dichotomic search to solve any biobjective optimization problem that can be formulated in terms of hypervolume scalarization; this framework can provide the complete set of optimal solution as well as a subset of it that has an approximation guarantee; ii) we present a parallel version of the dichotomic scheme to find the complete set of optimal solutions; iii) we extend the dichotomic approach for any number of objectives that works under some assumptions. In addition, we present numerical results on a wide range of instances.

Keywords

Multiobjective Optimization, Hypervolume Scalarization, Dichotomic Search, Integer Linear Programming.

This page is intentionally left blank.

Resumo

Nesta dissertação propomos métodos para resolver problemas de otimização multiobjectivo. Em particular, estes métodos são baseados no conceito de escalarização de hypervolume e estendem o trabalho anterior na seguinte forma: i) descrevemos uma software framework que implementa a procura dicotómica para resolver qualquer problema de otimização biobjetivo que pode ser formulado em termos de escalarização de hypervolume; esta ferramenta é capaz de encontrar o conjunto completo de soluções ótimas, assim como um deste que tem uma garantia de proximidade; ii) apresentamos uma versão do esquema dicotómico com paralelização para encontrar o conjunto completo de soluções ótimas; iii) estendemos a abordagem dicotómica para qualquer número de objetivos que funciona sob alguns pressupostos. Para além disso, apresentamos resultados numéricos numa ampla gama de instâncias.

Palavras-Chave

Optimização Multiobjectivo, Escalarização de Hypervolume, Procura Dicotomica, Programação Linear Inteira.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

ϵ-C ϵ-Constraint Scalarization. 5

HDS Hypervolume Dichotomic Scheme. xv, 10–12, 21, 27–29, 32, 33, 45

HS Hypervolume Scalarization. 9, 10, 33

ILP Integer Linear Programming. 2, 3, 6, 28, 29, 33

LP Linear Programming. 6

m-HDS Hypervolume Dichotomic Scheme for m-Dimensions. 23, 24, 27, 32, 45, 46

MILP Mixed-Integer Linear Programming. 2, 3, 6, 7, 28

MO Multiobjective Optimization. 3–6, 10

PHDS Parallelized Hypervolume Dichotomic Scheme. xv, 18, 27–29, 32, 45, 46

RoI Return on Investment. 1, 2

SCIP Solving Constraint Integer Programs. xiii, xiv, 2, 7, 8, 28, 33, 34, 38–42, 45

SCIPOPT SCIP Optimization Suite. 2, 7

WCS Weighted-Chebycheff Scalarization. 6, 10

WSS Weighted-Sum Scalarization. 5, 10

This page is intentionally left blank.

List of Figures

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

Multiobjective optimization problems can easily be found in our daily life. In multiple cases we inadvertently find ourselves trying to optimize several objectives. Consider that you are a businessman that wishes to invest in several different projects. Each project has an involved up-front price and is characterized by the Return on Investment (RoI) and a ecological benefit score. In this scenario you wish to find the subset of projects to invest that would give you the maximum RoI and ecological benefit score while abiding to a limit of how much money you wish to invest. If you are environmentally conscious you would not ignore the ecological benefit and you would not choose only the subset with highest RoI. At the same time you wish to have a good RoI, therefore you cannot give up the profit completely. A balance between the two objectives may need to be found, depending of the investor profile.

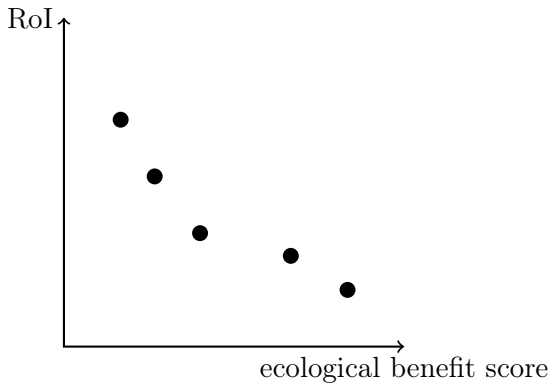


Figure 1.1: Possible investments with different ecological benefit scores and profits

We have to take this type of decisions on our daily life, most of the times without realizing it, but there is no perfect solution because the objectives may be conflicting, that is, some of the possible solutions are better for some objectives but worse for others, requiring choices to be made.

One way of finding the most preferable solution is to define preferences *a priori*, for example, using a weighted sum of the objectives where the most relevant objectives are assigned a larger weight. But in the example above, should we value the environment more than money? How much more important is the RoI versus the possible damage to the environment? How are we able to express our preferences in terms of weights? Given the

difficulty of defining preferences a priori, a possibility is to access all potentially interesting solutions, and by inspection, choose the most appropriate. A suitable notion of *interesting* solutions is those that are *nondominated*. Clearly, a dominated solution, for which, for all objectives, there exists at least another solution with better or equal objective values (with at least one strict inequality) is not interesting. For instance, we can think of a solution that has a marginal ecological benefit score but also negative RoI. This is not interesting as we would lose money and still have a big ecological impact. There are also others that have a better RoI and better ecological footprint and therefore are more interesting.

Our goal is to develop algorithms that are capable of finding all nondominated point of problems like the one presented above. As such, in this dissertation we develop an implementation of the approach described in [22] for solving biobjective optimization problems and make it available to both the research community and practitioners. The framework integrates SCIP `C++` libraries with the dichotomic approach and it can be easily adapted to other biobjective optimization problems. The code is available in github [1].
In addition, we propose the following extensions for this algorithm:

- A parallelized version, that solves multiple subproblems in parallel, capable of finding the nondominated set;
- An extension for any number of objectives that finds the nondominated set under some assumptions.

This document is structured as follows: In Chapter 2, we provide an overview of the concepts of optimality, scalarization and the basics of Integer Linear Programming and an MILP solver, SCIP Optimization Suite (SCIPOPT). Chapter 3 introduces the notion of hypervolume, its scalarization and the base algorithm of the framework developed, which was first presented in [22]. In Chapter 4 we introduce a parallelized version of the algorithm, while in Chapter 5 we present a generalized version of the algorithm capable of solving problems for any number of objectives. In Chapter 6 we present the experimental analysis of the approaches presented in this work. In Chapter 7 we present a guide to assist users in using the framework available in GitHub, as well as an example for an biobjective knapsack problem. Finally, Chapter 8 presents a general discussion and further work

[1] Frameworks code available at https://github.com/AlexdBrito/HVScalar

Chapter 2

Basic Concepts

This chapter introduces the basic concepts that are required to understand multiobjective optimization and other related notions of optimality.This material is mostly based on a textbook written by Ehrgott [7]. In Section 2.1 we introduce the basic concepts of multiobjective optimization and in Section 2.2 we present several methods for transforming the multiobjective optimization problem into a single objective problem, known as a *scalarization*, and corresponding properties. In Section 2.3 we introduce the concept of Integer Linear Programming (ILP) and in Section 2.4 we present a solver capable of solving MILP problems.

2.1 Multiobjective Optimization

In Multiobjective Optimization (MO) problems, the goal is to optimize m objective functions $f^i : \mathcal{X} \to \mathbb{R}$, $i = 1, \ldots, m$ with a set of feasible solutions $\mathcal{X} \subseteq \mathbb{R}^n$.Set $\mathcal{Y} := \{f(x) : x \in X\} \subseteq \mathbb{R}^m$ represents the image of the feasible set in the *objective space* $\mathbb{R}^{\mathrm{m}}$ and is called the set of *attainable outcomes*.

A MO problem can be formalized as follows (assuming maximization of m objectives):

$$\begin{aligned} \text{vmax} \quad & f(x) = (f^1(x), \ldots, f^m(x)) \\ \text{s.t.} \quad & x \in \mathcal{X} \end{aligned} \tag{2.1}$$

where vmax still needs to be defined. For these problems, generally, there is no optimal solution for all objective functions f^i. In this dissertation, we will rely on the concept of Pareto-optimal solutions, that is, solutions that cannot be improved in one of the objectives without worsening any other. The concept of Pareto optimality is based on the componentwise order relations in $\mathbb{R}^{\mathrm{m}}$. Let $p, q \in \mathbb{R}^m$:

$$p \geqq q \iff p^i \geq q^i \quad \text{for } i = 1, \ldots, m \tag{1.1}$$

$$p \geq q \iff p \geqq q \quad \text{and } p \neq q \tag{1.2}$$

$$p > q \iff p^i > q^i \quad \text{for } i = 1, \ldots, m \tag{1.3}$$

We say that a point p weakly dominates q if and only if $p \geqq q$ (see Expression(1.1)). We say that a point p dominates q if and only if $p \geqq q$ and $p \neq q$ (see Expression (1.2)). Finally, point p strictly dominates q if and only if $p > q$ (see Expression (1.3)). A solution $x \in \mathcal{X}$ is called *efficient* if there is no other solution $\bar{x} \in \mathcal{X}$ such that $f(\bar{x}) \geq f(x)$. If x is efficient, we say that its image in the objective space, $f(x)$, is a *nondominated point*. If $f(x) \geq f(\bar{x})$,

we say that solution x dominates $\overline{x}$ and the corresponding point $f(x)$ dominates the point $f(\overline{x})$. A solution $x \in \mathcal{X}$ is called weakly efficient, if there exists no other solution $\overline{x} \in \mathcal{X}$ such that $f(\bar{x}) > f(x)$.

Another order of special interest is the *lexicographic order*. Let $p, q \in \mathbb{R}^m$:

$$p \geq_\ell q \iff \begin{cases} p = q \quad \text{or} \\ p^i > q^i \;\; i = \min\{k : p^k \neq q^k\} \end{cases}$$

Let $\mathcal{S}^m$ be the symmetric group of order m, and let $\pi \in \mathcal{S}^m$ denote a permutation of the numbers $1, \ldots, m$,$x \in \mathcal{X}$ and $f^\pi(x) := (f^{\pi(1)}(x), \ldots, f^{\pi(m)}(x))$. A solution x is called *lexicographically optimal* with respect π if there exists no other solution $\overline{x} \in \mathcal{X}$ such that $f^\pi(\bar{x}) \geq^\ell f^\pi(x)$ with $f(x) \neq f(\overline{x})$.

The set of *nondominated points* (the nondominated set or Pareto front), is denoted by $\mathcal{Y}_N \subseteq \mathcal{Y}$ and the set of *efficient solutions* (the efficient set or Pareto set) by $\mathcal{X}_E \subseteq \mathcal{X}$. In this dissertation, we are particularly interested on combinatorial optimization problems, that is, the solution has some combinatorial property, such as path in a graph, or as a subset of a larger set. In multiobjective combinatorial optimization problems, the set of feasible points $\mathcal{Y}$ is finite, or countably infinite, and discrete. It is important to know the limits of the nondominated set, as provided by the *ideal point* and *nadir point*, which are illustrated in Figure 2.1.

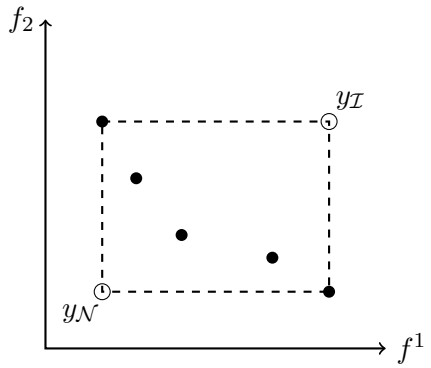


Figure 2.1: Illustration of Nadir point ($y_{\mathcal{N}}$) and Ideal point ($y_{\mathcal{I}}$)

The *ideal point* $y_{\mathcal{I}} = (y^1_{\mathcal{I}}, \ldots, y^m_{\mathcal{I}})$ is given by

$$y^i_{\mathcal{I}} := \max\{y^i : y \in \mathcal{Y}\} \quad i = 1, \ldots, m,$$

Generally, the ideal point is not an element of $\mathcal{Y}$, as if that was the case, i.e. if $y_{\mathcal{I}} \in \mathcal{Y}$, it would dominate all other feasible points in the set, making it the only nondominated point, $\mathcal{Y}_N = \{y_{\mathcal{I}}\}$. The ideal point is an upper bound on $\mathcal{Y}_N$.

The *nadir point* $y_{\mathcal{N}} = (y^1_{\mathcal{N}}, \ldots, y^m_{\mathcal{N}})$ is given by

$$y^i_{\mathcal{N}} := \min\{y^i : y \in \mathcal{Y}\} \quad i = 1, \ldots, m,$$

No efficient method to determine $y_{\mathcal{N}}$ for a general MO problem is known, except when $m = 2$. The *nadir point* is a lower bound on $\mathcal{Y}_N$.

2.2 Scalarization

There are several techniques for solving MO problems that consist of transforming them into a single-objective problem. These are known as scalarization methods and we will introduce some in this section: *Weighted Sum Scalarization*, *ε-Constraint Scalarization*, and the *Weighted Chebycheff Scalarization*. These scalarizations can be solved using single-objective methods. The optimal solution(s) to the scalarized optimization problem, depending on the scalarization parameters, usually allow to compute a subset of $\mathcal{X}_E$ of the original MO problem.

2.2.1 Weighted-Sum Scalarization

The Weighted-Sum Scalarization (WSS) method was introduced by Gass and Saaty [9] in 1955 for linear programming problems with two objectives. In this method, a weighting coefficient is associated with each objective function. The scalarized function consists of aggregating the m objective functions into a weighted sum. This way, the feasible set remains the same and with the same number of constraints. The Weighted-Sum Scalarization is formulated as follows:

$$\max_{x \in \mathcal{X}} \sum_{i=1}^{m} \lambda^i \cdot f^i(x), \qquad \text{WSS}$$

where λ represents the *weights* with $\lambda \in \mathbb{R}^m$.

The WSS method has some well-known properties [10]: For the set of weights $\{\lambda \in \mathbb{R}^m : \lambda^i \geq 0, \text{ for } i = 1, \ldots, m\}$ every optimal solution is a weakly-efficient solution of the related MO problem. For the set of weights $\{\lambda \in \mathbb{R}^m : \lambda^i > 0, \text{ for } i = 1, \ldots, m\}$ every solution of WSS is an efficient solution of the MO problem. However, the opposite is not true, that is, there are efficient solutions that are not optimal for any weight taken from this set. These solutions are called *unsupported* and the remaining are called *supported*. Every supported efficient solution of the original problem can be found as an optimal solution of the WSS with appropriate weights $\lambda \in \mathbb{R}^m_{>0}$.

2.2.2 ε-Constraint Scalarization

The ϵ-Constraint Scalarization (ϵ-C) method was introduced by Haimes et al. in 1971 [14]; see also [3]. In this method, one objective function f^k, $k \in \{1, \ldots, m\}$ of the MO problem is selected to be optimized, while the other objectives are converted into inequality constraints. Let $\varepsilon = (\varepsilon^1, \ldots, \varepsilon^m) \in \mathbb{R}^m$, where each ε^i is associated with an objective function $f^i(x)$ for $i = 1, \ldots, m$, the *ε-Constraint Scalarization* of a MO problems formalized as follows:

$$\begin{aligned} \max_{x \in \mathcal{X}} \quad & f^k(x) \\ \text{s.t} \quad & f^i(x) \geq \varepsilon^i \quad i = 1, \ldots, m, \ i \neq k \end{aligned} \qquad \varepsilon\text{-C}$$

For this method it is known that, for any $\varepsilon \in \mathbb{R}^m$, an optimal solution of the ε-Constraint problem is a weakly efficient solution of the MO problem. If the set of solutions of ε-C only has one element, then this solution is an efficient solution of the MO problem. Lastly, by choosing the appropriate vector, it is possible to find every efficient solution of the MO problem in the ε-Constraint method for any $k \in \{1, \ldots, m\}$.

2.2.3 Weighted Chebycheff Scalarization

The Weighted-Chebycheff Scalarization (WCS) method was originally proposed by Bowman in 1976 [2] (see also [15] and [16]). In order to find the efficient set $\mathcal{Y}_N$, this method uses a weight vector $\lambda = (\lambda^1, \ldots, \lambda^m) \in \mathbb{R}^m_{\geq 0}$ and a point $z = (z^1, \ldots, z^m)$, that is defined as follows for each component i:

$$z^i = \max_{x \in \mathcal{X}} \quad f^i(x)$$

The Weighted Chebycheff scalarization problem for a MO problem can then be formalized as follows:

$$\min_{x \in \mathcal{X}} \quad \max_{i=1,\ldots,m} \{\lambda^i (z^i - f^i(x))\} \qquad \text{WCS}$$

where $\max_{i=1,\ldots,m} \lambda^i (z^i - f^i(x))$ represents the weighted Chebycheff distance between the ideal point z and the objective function $f(x)$, and $0 < \lambda^i < 1$.

For the (WCS) method it is well known that every optimal solution is a weakly efficient solution of the MO problem, and the set of all optimal solutions of the WCS method contains, at least, one efficient solution of the original problem. Furthermore, if the set of optimal solutions has only one element, that solution is efficient for the corresponding MO problem. Lastly, if $\overline{x}$ is an efficient solution of the original problem, then there exists a $\lambda > 0$ such that $\overline{x}$ is optimal for the WCS problem.

2.3 Integer Linear Programming

A Linear Programming (LP) problem is the problem of optimizing a linear function subject to linear constraints. The linear function is called *objective function* in the literature, and the constraints can be either equalities or inequalities. The set of feasible solutions forms a polytope, a generalization of the three-dimensional polyhedron, which is a convex, connected set with flat, polygonal faces [20].
The standard form of a LP is formalized as the following:

$$\begin{aligned} \max_{x \in \mathbb{Z}^n} \quad & c^T x \\ \text{s.t} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

And the canonical form is formalized as follows:

$$\begin{aligned} \max_{x \in \mathbb{Z}^n} \quad & c^T x \\ \text{s.t} \quad & Ax + w = b \\ & x, w \geq 0 \end{aligned}$$

where A denotes a matrix, b and w are vectors [13]. Vector w is often called a vector of slack variables [20] and is introduced to convert the inequality constraints to equalities. In an Integer Linear Programming (ILP) formulation, the variables take only integer values. They are used to model discrete optimization problems where the objective function and the constraints are linear. In a Mixed-Integer Linear Programming (MILP), we allow both continuous and integer variables. Usually, an MILP solvers reads an MILP model and solves it using implicit enumeration techniques, such as branch and bound.

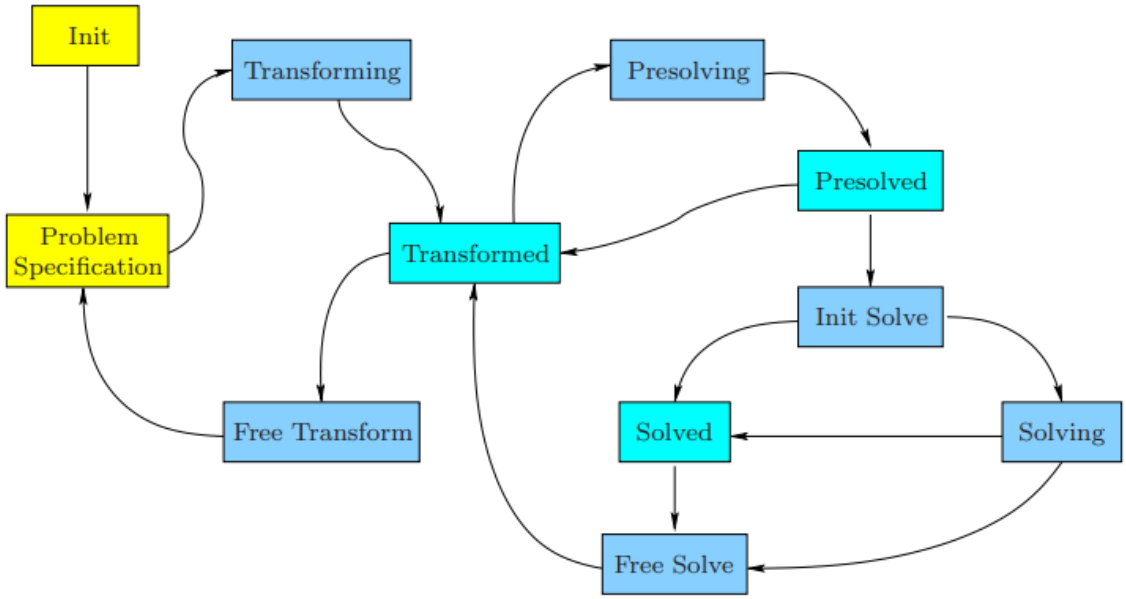


Figure 2.2: Operational stages of SCIP with the arrows representing possible transitions between stages

2.4 MILP Solver

In order to solve a MILP problem we can use a solver such as SCIP Optimization Suite (SCIPOPT). SCIPOPT is a toolbox that contains Solving Constraint Integer Programs (SCIP), a solver framework that we will use to build and solve each instance of the problem. A detailed description can be found in [1].

SCIP is a Mixed-Integer Linear Programming (MILP) solver and constraint programming framework implemented as a C callable library. It gives complete control of the solution process and access to detailed information. The central objects of SCIP are the *constraint handlers*. Their primary task is to check a given solution for feasibility with respect to all constraints of its type that exist in the problem instance. To improve the performance of the solving process, constraint handlers may provide additional algorithms and information about their constraints to the framework, such as: presolving methods to simplify the problem's representation; propagation methods to tighten the variables' domains; a linear relaxation, which can be generated in advance or on the fly, which strengthens the linear programming relaxation of the problem; and branching decisions to split the problem into smaller subproblems, using structural knowledge of the constraints in order to generate a well-balanced branching tree. During the execution of SCIP, the formalized optimization problem goes through several stages, limiting the operations the user may perform in each of them. Figure 2.2 shows a flow chart of the *operational stages* that are traversed. A problem can be allocated and initiated using the method **SCIPcreate()**. With the problem created we must now include the plugins, for example, the default plugins with **SCIPcreateProbBasic()**. Afterwards we create the problem with **SCIPcreateProbBasic()**. This transitions the solver to the *Problem Specification Stage*, illustrated as the the lower yellow rectangle in Figure 2.2.

During the specification stage we can set the objective direction, as well as add variables

and constraints. The objective direction of the problem is set using **SCIPsetObjsense()**, setting it to either SCIP_OBJSENSE_MAXIMIZE or SCIP_OBJSENSE_MINIMIZE. A basic variable can be created using **SCIPcreateVarBasic()** and then we add it to the problem with **SCIPaddVar()**. With the variable added to the problem we must now define its coefficients with **SCIPaddCoefLinear()**. A linear constraint can be created using **SCIPcreateConsBasicLinear()** and added to the problem with **SCIPaddCons()**.

With the problem defined we can solve it with **SCIPsolve()**. Upon calling this function the solver goes through several stages. Firstly, it transitions to the *Transforming Stage*, where it creates a copy of the data of variables and constraints to a separate memory area. The copy is called the transformed problem and protects the original problem instance from modifications applied during *Presolving* or *Solving* stages. When this is completed the solver transitions to the *Transformed Stage*. This state is only an intermediate state, as it transitions immediately to the *Presolving Stage*.

During the presolving stage the solver detects fixings, i.e. variables whose value can be changed in order to gain some advantage in terms of performance or speed of solving, aggregations of variables that will be deleted from the transformed problem and replaced by their fixed or their representing active variables, respectively. The solver also checks if any of the constraints can be upgraded into a more specific type, as by doing this it can store the data in a more compact form and employ specialized, more efficient algorithms. For example, a linear constraint could be converted into a knapsack constraint, which is a more specific type of a linear constraint. After finishing all the aforementioned tasks the solver transitions to the *Presolved Stage*, that, much like the *Transformed Stage*, is an intermediate stage, as it immediately switch to the *Init Solve Stage*.

If the problem was solved during the *Presolving Stage*, then the solver automatically switches to the *Solved Stage*. Otherwise, the solving process begins as it transitions to the *Solving Stage*. When the solving process ends, we can obtain the result with **SCIPgetBestSol()**, which returns the best feasible primal solution of the problem. If we wish to know the result in more detail, including which variables were chosen, we can use **SCIPprintBestSol()** to print the best solution.

Before closing the problem we must release all the variables and constraint with the help of **SCIPreleaseCons()** and **SCIPreleaseVar()**. After releasing them we can use **SCIPfree()** to release the SCIP environment.

Chapter 3

Hypervolume

In this chapter we review the notion of hypervolume, based on the survey of Guerreiro et al. [12]. In Section 3.1, we introduce the hypervolume indicator and in Section 3.2 we present the Hypervolume Scalarization (HS). In Section 3.3, we present a Hypervolume Dichotomic Scheme as proposed in [22], which is the basis of this work.

3.1 Hypervolume Indicator

The hypervolume indicator was introduced by Zitzler and Thiele in 1998 [28] and it has become a versatile tool in multiobjective optimization for the evaluation of the performance of multiobjective evolutionary optimization algorithms. The hypervolume indicator is the measure of the region weakly dominated by a given point set $\mathcal{S} \subset \mathbb{R}^{\mathrm{m}}$ and the reference point $r \in \mathbb{R}^{\mathrm{m}}$ with $r \leqq p$ for all $p \in \mathcal{S}$. This can be represented as:

$$H(\mathcal{S}) := vol(\{q \in \mathbb{R}^d \mid \exists p \in \mathcal{S} : p \geq q \textbf{ and } q \geq r\})$$

The hypervolume contribution of a point p with respect to a point set $\mathcal{S}$, $H(p, \mathcal{S})$, is the region that is dominated by a point p and not dominated by any other point in $\mathcal{S}$, and is defined as

$$H(p, \mathcal{S}) := H(\{p\} \cup \mathcal{S}) - H(\mathcal{S}).$$

The hypervolume indicator has some interesting properties, namely, it is a submodular function, that is, the contribution of a point p, with respect to a set $\mathcal{S}$, decreases as the number of elements in the point set $\mathcal{S}$ increases. Furthermore, the nondominated set has maximal hypervolume value among all subsets of feasible points. This is illustrated in Figure 3.1, plot (a) illustrated the hypervolume indicator of a set of points ($H(\{p_1, \ldots, p_5\})$), and plot (b) illustrated the hypervolume contribution of point p_2 with respect to a point set ($H(p_2, \{p_1, p_3, p_4, p_5\})$).

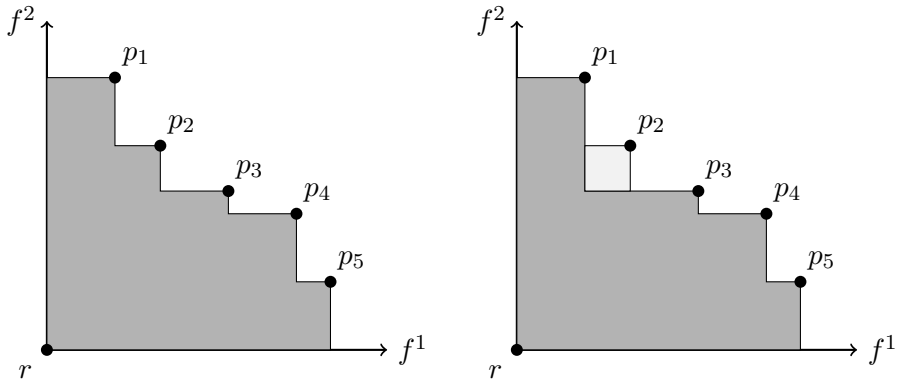


Figure 3.1: Illustration of (a) hypervolume indicator of a set of points, given a reference point, and (b) hypervolume contribution of a point p_2 with respect to a point set (light gray region)

3.2 Hypervolume Scalarization

A recent concept of scalarization is to consider the concept of hypervolume to aggregate the m objective functions. This is explored in [22] for solving biobjective problems and it will be the basis of this work. The hypervolume scalarization of a MO problem is defined as follows:

$$\begin{aligned} \max_{x \in \mathcal{X}} \quad & \prod_{i=1}^{m} f^i(x) - r^i \\ \text{s.t.} \quad & f^i(x) \geq r^i \quad \forall i = 1, \ldots, m \end{aligned} \tag{HS(MOP)}$$

Important to note that an optimal solution of the HS(MOP) is weakly efficient for the MO problem, and all efficient solutions of the MO problem can be determined as an optimal solution of a corresponding HS(MOP) for a given reference point r [22]. The constraint guarantees that the optimal point found dominates the reference point.

Unlike the Weighted-Sum Scalarization, the Hypervolume Scalarization can be applied without being restricted to supported efficient solutions. In this sense, it shares similar properties to that of the Weighted-Chebycheff Scalarization, mentioned in Section 2.2.3.

3.3 Hypervolume Dichotomic Scheme

In this section, we describe the Hypervolume Dichotomic Scheme (HDS) proposed in [22], which allows to find the efficient set of a MO problem for $m = 2$.

The pseudocode is shown in Algorithm 1. This approach requires an initial reference point, a list S that keeps a sequence of nondominated points found, ordered with respect to the first objective, and a priority queue P that stores candidate points in which the element in its top has the highest hypervolume contribution. The algorithm chooses, iteratively, a point s from the top of the priority queue P and solves two subproblems associated to that point. Each of the two subproblems to be solved are defined by a reference point, which

results from a combination of coordinates of s and of its predecessor or successor in set S. This reference point is known as *local lower bound* in the literature [6]. These subproblems then originate, at most, two new nondominated points, each of which are added to set P, if their hypervolume contribution with respect to S is positive. A null hypervolume contribution means that the solver obtained a point whose coordinates were either equal to the ones that define the reference point, or equal to ones that define a point already in set S. At the end of each iteration, a point s is removed from P and inserted into S. This process continues until P is empty, which implies that S corresponds to the nondominated set. This process can be stopped early, which, in this case, implies that a subset of the nondominated set is returned. Let $(\overleftarrow{s^1}, \overleftarrow{s^2})$ and $(\overrightarrow{s^1}, \overrightarrow{s^2})$ (see Line 7 and Line 13 in Algorithm 1) be the points immediately before and after point s, respectively. To ensure that, in the main loop, a point always has a predecessor and a successor, we consider two dummy points, $(r_0^1, +\infty)$ and $(+\infty, r_0^2)$ (see Lines 2-3 in Algorithm 1), where (r_0^1, r_0^2) is the initial reference point, which are inserted into S in a preprocessing step. The two reference points that define the next two hypervolume scalarized problems are $(r^1, r^2) := (s^1 \overrightarrow{s^2})$ and $(r^1, r^2) := (\overleftarrow{s^1}, s^2)$.

Algorithm 1: Hypervolume Dichotomic Scheme (HDS)

```
   Data: (r_0^1, r_0^2)
 1 S = P = ∅
 2 insert((r_0^1, +∞), S)
 3 insert((+∞, r_0^2), S)
 4 s ← solve(r_0^1, r_0^2)
 5 insert(s, S)
 6 do
 7 |  (s^1→, s^2→) ← succ(s, S)
 8 |  (r^1, r^2) ← (s^1, s^2→)
 9 |  p ← solve(r^1, r^2)
10 |  if H(p, S) > 0 then
11 |  |  queue(p, P)
12 |  end
13 |  (s^1←, s^2←) ← pred(s, S)
14 |  (r^1, r^2) ← (s^1←, s^2)
15 |  p ← solve(r^1, r^2)
16 |  if H(p, S) > 0 then
17 |  |  queue(p, P)
18 |  end
19 |  insert(s, S)
20 |  s ← dequeue(P)
21 while |P| > 0;
22 remove((r_0^1, +∞), S)
23 remove((+∞, r_0^2), S)
24 return S
```

According to the authors in [22], the algorithm should follow a greedy principle, that is, at each iteration the point s with the largest hypervolume contribution with respect to the current subset S is chosen to be added to S. The authors in [22] show that this strategy allows to find a subset S with an hypervolume that is, in the worst case, an $(1 - 1/e)$-

approximation to the optimal hypervolume of a subset of the same size, if stopped early. This is an advantage over other approaches as the algorithm can terminate and be able to find a representative subset of the nondominated set with an approximation quality guarantee. The changes on Algorithm 1 to allow finding such a subset of size J are shown in Algorithm 2. By modifying the termination condition and changing the loop to a while do (see Lines 7-21 in Algorithm 2), it is possible to define, a priori, a maximum number of points to find (J). We also need to move $insert(s, S)$ out of the loop, placing it before the while loop and, finally, add a second $insert(s, S)$ as the last instruction of the loop (see Line 20 in Algorithm 2). Then, if the number of elements in S, excluding the two initial dummy points, is equal to J, the algorithm terminates and returns the contents of S. The algorithm performs $J - 1$ iterations of the while loop, which means $2J - 1$ hypervolume scalarized problems are solved. The efficiency of this dichotomic search depends on how efficient the scalarized problems can be solved (see Line 9 and Line 15 in Algorithm 1).

Algorithm 2: Hypervolume Dichotomic Scheme (HDS)

```
   Data: (r_0^1, r_0^2), J
 1 ...
 6 insert(s, S)
 7 while (S − 2) < J do
 8 |   (s^1→, s^2→) ← succ(s, S)
 9 |   ...
20 |   insert(s, S)
21 end
22 remove((r_0^1, +∞), S)
23 remove((+∞, r_0^2), S)
24 return S
```

Algorithm 2 explores the following invariant at each iteration of the while loop (see Lines 6-21 in Algorithm 1): for each point s that is removed from the priority list P, with s being the point with highest hypervolume contribution in the queue, the next point to be chosen by the algorithm can only be one of the following:

1. the point that is on top of the priority queue P;
2. the point that maximizes the hypervolume contribution above s;
3. the point that maximizes the hypervolume contribution to the right of s.

This invariant is illustrated in Figure 3.2. In plot (a) (see also plot (b)), the next point to be chosen after p_0 can either be above it with reference point r_1, to its right with reference point r_2 or, in the case that none of these reference points yield an optimal point that has a larger hypervolume contribution than the one that is currently on the top of the queue, the latter is chosen.

Figure 3.2 also shows another property of this algorithm: whenever a point is calculated, for example when, in plot (a) p_0 is found, a curve can be traced representing all points with hypervolume equal to that of p_0 from the reference point r_0. This curve is an upper bound on the search region associated to the reference point. Then, any further nondominated point will be found under this curve, as shown in plots (b) (see also plots (c) and (d)).

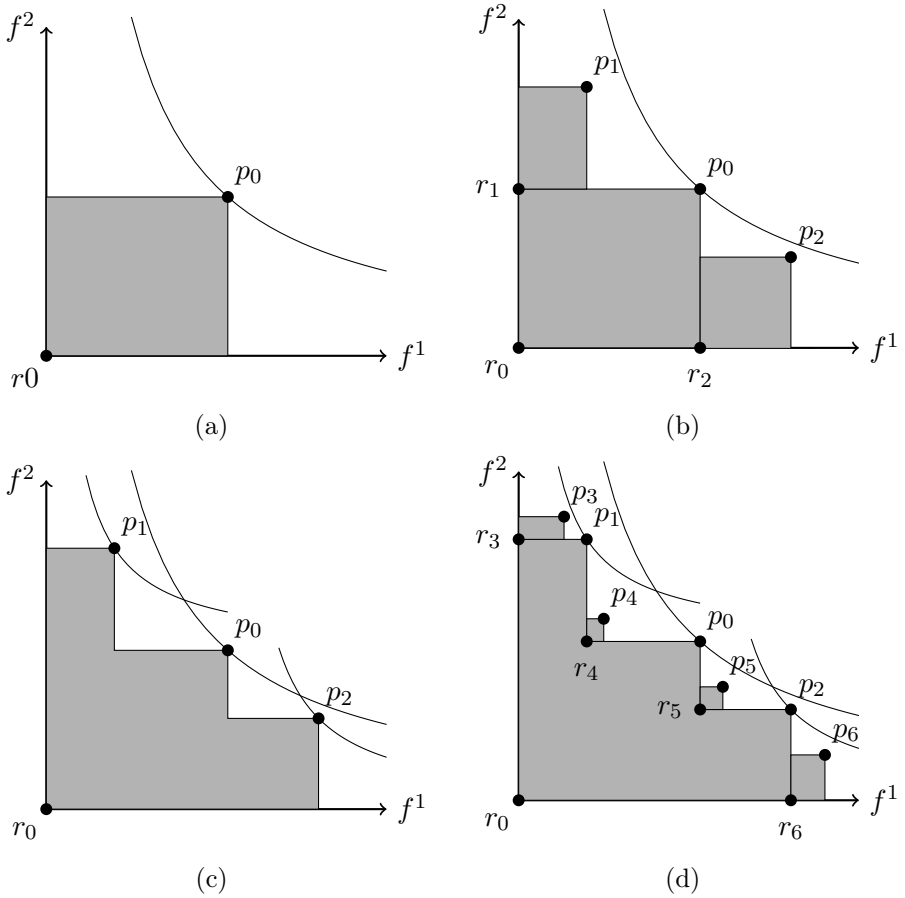


Figure 3.2: Representation of: (a) solution of the first subproblem with reference point r_0, with a curve that represents all points with hypervolume equal to p_0; (b) solution of the second and third subproblem with reference points r_1 and r_2 respectively; (c) curves that represent all points with hypervolume contribution equal to p_1 and p_2; (d) solutions of the subproblems with reference points r_3-r_6

3.4 Discussion

In this chapter we presented a dichotomic search algorithm capable of solving multiobjective problems using a sequence of hypervolume scalarizations proposed in [22]. We also presented a modification of the algorithm to allow it to obtain a specific amount of optimal points with guarantee that the calculated hypervolume is in the worst case, $(1 - 1/e)$ approximation to the optimal hypervolume of a subset of the same size, if stopped early [22]. However, the algorithm is only usable for biobjective problems and solves each subproblem sequentially. In the following chapters we will discuss the possibility of improving its efficiency by using parallelization and to generalize it for more than two objectives.

This page is intentionally left blank.

Chapter 4

Parallelization of the Dichotomic Scheme

In this chapter, we will discuss the parallelization of the dichotomic scheme presented on the previous chapter in order to speed up the overall processing time. In Section 4.1 we introduce the necessary concepts of parallelization and review the various methods available. In Section 4.2 we present the chosen parallel paradigm and the required changes to Algorithm 1 in Chapter 3 to allow for parallelization.

4.1 Basic Concepts of Parallelization

Parallel computing refers to a type of computation in which larger problems are broken down into smaller and independent subproblems allowing for many calculations or processes to be executed simultaneously by multiple processors. These processors generally communicate through either shared memory architecture, which is memory shared between all processing elements in a single address space or distributed memory architecture in which each processing element has its own local address space. Parallel computing is introduced with the goal of reducing execution time, allowing a better way of using the resources available by taking advantage of multi-core processors.

One of the earliest classification systems for parallel, and sequential, computing was proposed by Michael J. Flynn in 1972 in [8] and is still used today due to its simple and easy to understand model. His system defines four categories based upon the number of concurrent instruction streams and data streams available in the architecture, these categories are:

1. SISD (Single instruction Stream, Single Data Stream) where a single control unit performs sequential operations on a single data stream;
2. SIMD (Single Instruction Stream, Multiple Data Stream) where a single instruction is performed on different data streams, instructions can be executed sequentially or in parallel by multiple units;
3. MISD (Multiple Instruction Stream, Single Data Stream) an uncommon architecture where multiple systems operate on the same data stream and must agree on the result which is normally used for fault tolerance;

4. MIMD (Multiple Instruction Stream, Multiple Data Stream) the most common architecture currently in which multiple processors simultaneously execute different instructions on different data streams.

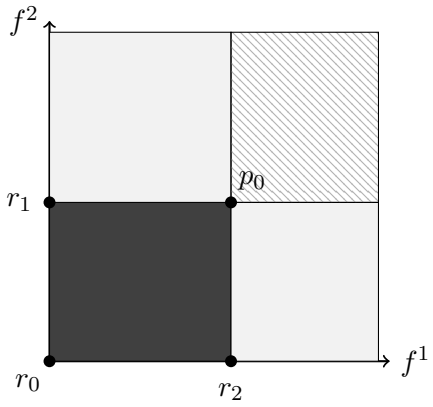
There are multiple types of parallelization, namely *bit-level*, *instruction-level*, *task/data* and *superword level* parallelism. *Bit-level* parallelization [4] consists in increasing the processor word size - the amount of information the processor can manipulate for cycle, which reduces the quantity of instructions the processor must execute in order to perform an operation on variables greater than the length of the word, i.e., a 8-bit processor that must add two 16-bit integers. In *instruction-level* parallelism [27], the processor re-orders and combines instructions into groups that can be executed in parallel without affecting the result of the program. *Task/data* parallelism [26] is the more traditional parallelism where problems are divided into smaller subproblems that are then allocated to a processor for execution. The processors then execute these sub-tasks concurrently and sometimes cooperatively. In task parallelism different calculations can be performed on the same or different sets of data while in data parallelism the same calculations are performed on the same or different sets of data. Lastly *superword level* parallelism [18] is a vectorization technique that can exploit the parallelism of inline code.

In terms of parallel programming, there are several models that are currently used. These models are abstractions and machine architecture independent. Therefore they can, theoretically, be implemented on various hardware and memory architectures. Some of the more prevalent models are: 1) Shared Memory Model, in which tasks share a common address space that they can use to read and write asynchronously. These interactions are normally moderated by a control access mechanism like locks or semaphores; 2) Threads Model in which a process can have multiple, concurrent execution paths. The main program loads and obtains all the necessary resources to activate the process. During the execution the main program creates threads to asynchronously execute subroutines that communicate with the main program or other threads through global memory, commonly using the shared memory architecture, and later are destroyed when their work is done; 3) Message Passing Model in which exists a set of tasks that use their own local memories during computation. These multiple tasks can reside on the same or different machine and exchange data by sending and receiving messages. 4) Data Parallel Model in which most of the parallel work focuses on performing operations on a data set. This data set is typically organized into a common structure like an array and the tasks work collectively, each on different portions of the data set.

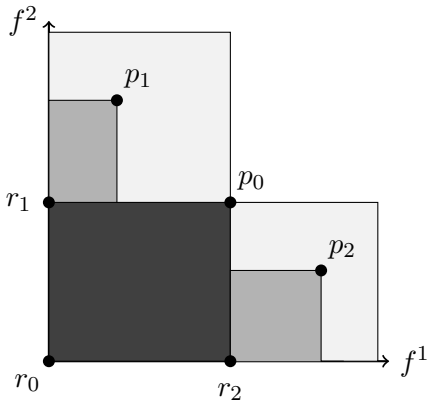
When discussing parallelism it is important to remember Amdalh's Law [23], that states that the theoretical maximum speedup possible when introducing multiple processors is the time needed by the slowest non-divisible part of the program. In other words, if a program needs 10 hours to complete and can it can be divided into a maximum of 20 independent and indivisible sub-tasks but one of these sub-tasks takes around one to complete then the minimum time that a parallelized version of this program will take is around one.

4.2 Parallelization of Dichotomic Scheme

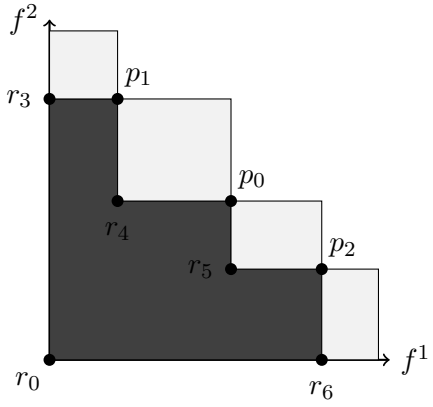
Each call of solve function in Algorithm 1 (see Lines 6-20) solves a subproblem that receives a reference point and returns either a new optimal point in case a feasible solution exists or null if no new optimal point can be found. Therefore, the most appropriate parallelization method for us is task/data parallelization with the threads model because each subproblem



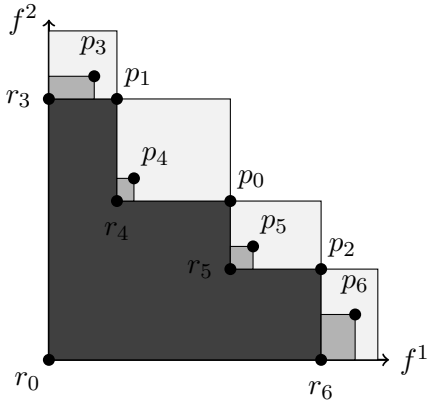
(a) Search region of r_1 and r_2 with overlap



(b) Solutions to r_1 and r_2



(c) Four new reference points and their search region with the overlap



(d) Solutions to the new reference points

Figure 4.1: Illustration of: (a) search regions of reference points r_1 and r_2 and their overlap; (b) solutions p_1 and p_2 for subproblems with reference points r_1 and r_2 respectively; (c) search areas for the new reference points without the overlap; (d) solutions to the new reference points within their search areas.

is a task that can be executed by a worker thread while the main program performs other calculations. As these subproblems are implemented by the user they are also the smallest non splittable part of our framework. Therefore, if the algorithm is able to execute multiple subproblems at the same time, the theoretical minimum execution time is reached, according to Amdalh's Law. For the case of dichotomic scheme presented in Section 3.3, Algorithm 1 (see also Algorithm 2), each two subproblems that arise in the while loop can be solved in parallel, since they are mutually independent. This fact is illustrated in Figure 4.1. Plot (a) shows two search regions, each of which is related to reference points r_1 and r_2, respectively, and where the patterned region corresponds to their overlap. A solution to each of the two subproblems cannot arise in the patterned region, since this would imply point p_0 to be dominated, leading to a contradiction, since p_0 is, by definition, a nondominated point. Therefore, both problems can be solved independently, allowing for a theoretical speed up of 2.

Note that the parallelization of each two new subproblems that arise at each iteration of the main loop can be applied to find a representation of size J as described in Algorithm 2. However, we can take further advantage of parallelization by parallelizing each

subproblem that corresponds to a reference point of every point that is in the priority queue P. This is illustrated in Figure 4.1, plot (c) which shows that the search regions of reference points r_5 and r_6, generated by p_2, and the search regions of the reference points generated by p_1, are disjoint. However, the greedy approach cannot be applied here, as we no longer choose the next point s with the highest hypervolume contribution before restarting the while loop. All available reference points are parallelized as they are generated and the solutions are obtained as the problems are solved without a specific order. Because of this, if we stop it early, the guarantee of approximation to the optimal hypervolume cannot hold.

Algorithm 3 presents a Parallelized Hypervolume Dichotomic Scheme (PHDS) that parallelizes all the available reference points in queue P. In order to parallelize the solve function we create a function *thread*, which receives the *solve* function and a new set T. This function parallelizes the solve function and queues the solution on set T. We create a thread for all reference points available (see Lines 6-20 in Algorithm 3) and, whenever no more points are available, it waits for a thread to return a new solution. Then, it calculates two new reference points (see Lines 15-19 in Algorithm 3). These two points will immediately be used for two new subproblems and it waits again for another thread to resolve. This process continues until there are no more threads running and no more reference points available.

Algorithm 3: Parallelized Hypervolume Dichotomic Scheme (PHDS)

```
   Data: (r_0^1, r_0^2)
 1 S = P = T = ∅
 2 insert((r_0^1, +∞), S)
 3 insert((+∞, r_0^2), S)
 4 s ← solve(r_0^1, r_0^2)
 5 queue(s, P)
 6 do
 7 |   s ← dequeue(P)
 8 |   insert(s, S)
 9 |   (s^1→, s^2→) ← succ(s, S)
10 |   (r^1, r^2) ← (s^1, s^2→)
11 |   thread(solve(r^1, r^2), T)
12 |   (s^1←, s^2←) ← pred(s, S)
13 |   (r^1, r^2) ← (s^1←, s^2)
14 |   thread(solve(r^1, r^2), T)
15 |   while P = ∅ and T ≠ ∅ do
16 |   |   p ← dequeue(T)
17 |   |   if H(p, S) > 0 then
18 |   |   |   queue(p, P)
19 |   |   end
20 |   end
21 while |P| > 0;
22 remove((r_0^1, +∞), S)
23 remove((+∞, r_0^2), S)
24 return S
```

4.3 Discussion

In this chapter we have described how to modify the Hypervolume Dichotomic Scheme in order to parallelize it. This parallelization strategy described can be used to find the complete efficient set and, if stopped early, it returns a subset of the latter. However, this subset no longer has the same approximation guarantee since it is not possible to implement the greedy choice. It is an open question whether it is possible to solve more than two subproblems in parallel, keeping the same approximation. Details about the implementation and an experimental analysis performed on Algorithm 4 are described in Chapter 6 and Chapter 7.

This page is intentionally left blank.

Chapter 5

Extension for larger number of objectives

In this chapter, we will discuss the generalization of the HDS presented in Algorithm 1 to more than two objectives. In Section 5.1 we will describe the required changes to our framework when dealing with more objectives. Section 5.2 introduces an extension of the framework that finds the efficient set under some assumptions.

5.1 Changes caused by more objectives

When dealing with more objectives we need to take into account that there are more nondominated points, more reference points, the subproblems cannot be solved independently and there are points with shared coordinates. Unfortunately, some of these changes make our approach harder to generalize for a larger of objectives. In the following we discuss how these changes impact our dichotomic approach described in Section 3:

More nondominated points With the increase in the number of objectives the number of nondominated points, in general, also increases. This leads to an increase on the number of subproblems to solve and, consequently, more computational time is required.

More reference points The number of reference points also increases, because each nondominated point can have as many neighboring points as those already found. In fact, an upper bound on the number of reference points, for a fixed dimension, is linear with respect to the number of points found. This bound is given in [6] which corresponds to the number of local lower bounds for a given set of nondominated points. Note that a reference point is also a local lower bound. This is illustrated in Figure 5.2, in which plot (a) shows that point p_0 generates three new reference points and p_1 generates four.

Subproblems cannot be solved independently The increase in the number of objectives also causes the subproblems to have overlapped search regions where there might exist nondominated points. Therefore, these subproblems cannot be solved independently anymore. This stems from the fact that, with $m > 2$, a nondominated point can be found by more than one reference point. This is illustrated in Figure 5.1 plots (c) and (d) (see also plot (b)), where p_1 can be found by solving the subproblems with reference points r_1

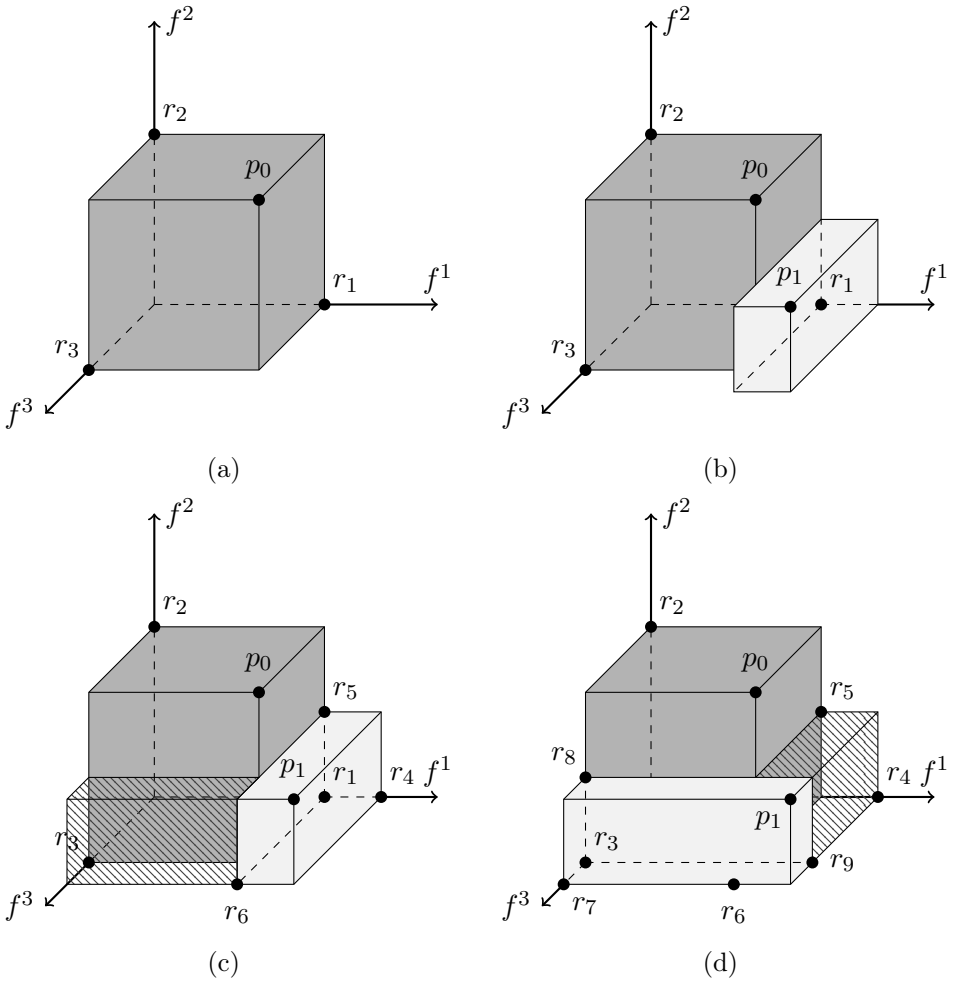


Figure 5.1: Illustration of: (a) Reference points generated by nondominated point p_0; (b) second nondominated point p_1 found by reference point r_1; (c) stripped region not calculated by the dichotomic scheme and reference points generated by p_1; (d) nondominated point p_1 found by different reference point r_3 with the reference points it generates.

and r_2. Losing subproblem independence can affect the behaviour of the greedy principle as well as it limits usage of parallelization. We recall that an important aspect to allow paralellization is to have independent subproblems.

Points with shared coordinates For $m > 2$, it is possible to have nondominated points with shared coordinates. Therefore, it is now possible for a subproblem to find a new nondominated point with null hypervolume contribution. This differs from $m = 2$, where a null hypervolume contribution means that the point had already been found (see Section 3.3). Moreover, it is also not clear if another reference point will find this new nondominated point. This is illustrated in Figure 5.2 plot (b), which shows that p_1 can be found by r_1 and also by r_3. However, p_1 would have null hypervolume contribution if found by r_3.

Whenever a nondominated point can be found by multiple reference points, the hypervolume contribution, with respect to the set of points already found, can also be miscalculated. As shown in Figure 5.1, plots (c) and (d), the region that is stripped represents a part of the hypervolume contribution of the nondominated point that is not taken into

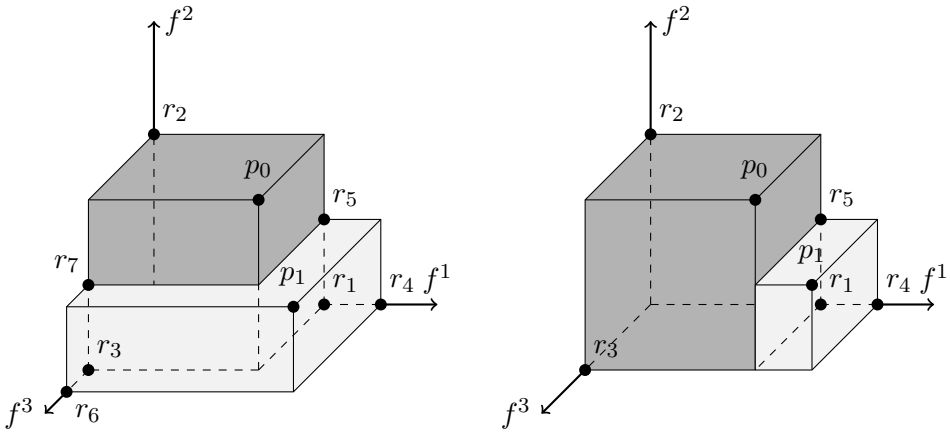


Figure 5.2: Illustration of two degenerate cases: (a) p_1 can be found by multiple reference points; (b) p_1 shares coordinates with another nondominated point (p_0)

account by the dichotomic scheme, because it only a reference point to define the search region.
Therefore, in some situations, the value of the hypervolume contribution calculated does not correspond to the actual hypervolume contribution of the point with respect to the set of nondominated points already found. Consequently, choices made related to the next point to analyse can be wrong in terms of the greedy principle.

5.2 An extension for more than two objectives

Despite the difficulties described in the previous sections, it is still possible to extend the dichomotic approach to find all nondominated points, under some assumptions about the latter. The approach described in this section does not use the greedy choice for the reasons explained in the previous section. In addition, we assume that the efficient set contains only points in the general position, because it is unclear how the algorithm would behave when faced with nondominated points with shared coordinates. Moreover, due to the fact that the an efficient point may be found more than once with positive hypervolume contribution from different reference points, it is expected that some subproblems are redundant. However, we no longer need to explicitly calculate the hypervolume contribution because the choice of the next reference point no longer needs to rely on it.

This new version, Hypervolume Dichotomic Scheme for m-Dimensions (m-HDS), is presented in Algorithm 4. Each reference point, r is implemented as a data structure that must not only keep the information of its coordinates ($r.coords$), but also, at most, m nondominated points that generated it ($r.gen$). In this version, R is a set of reference points to be used, and can be implemented as a list. The method of verifying if a nondominated point is new or feasible also changes. As we no longer calculate the hypervolume of every nondominated point we just verify whether the new point p already exists in the set of nondominated points S or not. In case it does not exist we use it to calculate the new reference points and insert it into the set S. This is executed by function $isNewPoints()$ (see Line 8 in Algorithm 4).

Algorithm 4: Hypervolume Dichotomic Scheme for m-Dimensions (m-HDS)

```
   Data: (r_0, m, n)
 1 S = R = ∅
 2 s ← solve(r_0)
 3 insert(s, S)
 4 insert(calcNewRefPoints(R, p, r_0, m, n), P)
 5 do
 6 |   r ← dequeue(R)
 7 |   p ← solve(r)
 8 |   if isNewPoint(p, S) then
 9 |   |   insert(s, S)
10 |   |   calcNewRefPoints(R, p, r, m, n)
11 |   end
12 while |R| > 0;
13 return S
```

For each new point found, there is the need to find the reference points that define the new subproblems to be solved. An algorithm explored by Lacour et al. [17] is of particular interest to us, as they present a method to calculate local lower bounds for a given set of points for a general number of objectives. This method is used in the m-HDS (see Line 4 and Line 9 in Algorithm 4) to calculate the new reference points which is presented in Algorithm 5.

Given a nondominated point p, found by reference point r, the algorithm iterates through the current set of reference points, R, and extracts the reference points that are dominated by the new nondominated point, p, creating a new set H, which also contains r (see Line 1 in Algorithm 5). For every point $h \in H$, and for every objective i, if point p dominates h, when excluding objective i, we create a new point u, which will be a new reference point (see Lines 2-4 in Algorithm 5). For all coordinates of point u, except i, we let point u share the same coordinates of point h; the coordinate i of u is equal to that of p (see Lines 5-7 of Algorithm 5). Similarly, we let the points that generated u as being the same that generated h, for all coordinates except i, and p for coordinate i (see Lines 9-12 in Algorithm 5). Lastly, the new reference point u is added to the set of reference point R (see Line 13 in Algorithm 5).

5.3 Discussion

In this chapter, we presented the changes that we need to take into account when considering more objectives and their consequences to our framework. We also introduced a new approach that does not use the greedy principle and uses a new method to calculate new reference points. As the subproblems are no longer independent, as explained in Section 5.1, it stays as an open question if it is possible to introduce parallelization to Algorithm 4. In the following chapter we will present an experimental analysis performed on Algorithm 4, as well as details about its implementation.

Algorithm 5: calcNewRefPoints

Data: (R, p, r, m, n)

$H \leftarrow r \cup extractDominatedSet(R, p)$
foreach $h \in H$ **do**
  for $i = 1, 2, ..., m$ **do**
    if $h^j < p^j$ *for all* $j = \{1, 2, ..., m\} \setminus \{i\}$ **then**
      $u.coord[i] \leftarrow p.coord[i]$
      foreach $k \in \{1, 2, ..., m\} \setminus \{i\}$ **do**
        $u.coord[k] \leftarrow h.coord[k]$
      end
      $u.gen[i] \leftarrow p$
      foreach $k \in \{1, 2, ..., m\} \setminus \{i\}$ **do**
        $u.gen[k] \leftarrow h.gen[k]$
      end
      $insert(u, R)$
    end
  end
end

This page is intentionally left blank.

Chapter 6

Experimental Analysis

In this chapter, we discuss the experimental analysis that was conducted to evaluate the performance of sequential HDS (Algorithm 1), the PHDS (Algorithm 3) and m-HDS (Algorithm 4). In Section 6.1 we introduce the problem used in the analysis and compare the CPU-time taken by HDS and PHDS. In Section 6.2 we analyze several data structures to store the set of reference points and how they impact the performance of m-HDS.
All experiments were conducted on a machine running Windows 10 Version 20H2 (OS Build 19042.985), using WSL Version 2 with Ubuntu Version 20.04.2 (64-bit), an Intel-i9 10980HK Octa-core running at 2.4 GHz (base frequency) undervolted by 0.1V with 16MB of Smart Cache and 64GB RAM.

6.1 Two-dimensional case

In order to successfully test the dichotomic search scheme, we reproduced the work in [22] for the case of a biobjective knapsack problem with a cardinality constraint. This problem consists of maximizing two linear sum objective functions, with a cardinality constraint that allows only k items to be selected, which is formalized as follows.

$$\begin{aligned} \max \quad f(x) := \quad & \left(\sum_{i=1}^{n} a_i x_i, \sum_{i=1}^{n} b_i x_i \right) \\ s.t. \quad & \sum_{i=1}^{n} x_i = k \\ & x_i \in \{0,1\} \quad \forall i \in \{1,\dots,n\}, \end{aligned}$$

where x_i is a binary variable that takes value 1 if element i is chosen and 0 otherwise, a, b represent the value of element i for the first and the second objective, respectively and k represents the maximum number of elements that can be chosen.

The authors in [22] show that the hypervolume scalarization of this problem leads to the following optimization problem with $r = (r^1, r^2)^T$ as the reference point.

$$
\begin{aligned}
\max \quad f(x) := \quad & \left(\sum_{i=1}^{n} a_i x_i - r^1\right) \cdot \left(\sum_{i=1}^{n} b_i x_i - r^2\right) \\
& = \sum_{i=1}^{n}\sum_{j=1}^{n} a_i b_j x_i x_j - \sum_{i=1}^{n}(r^2 a_i + r^1 b_i)x_i + r^1 r^2 \\
s.t. \quad & \sum_{i=1}^{n} a_i x_i \geq r^1 \\
& \sum_{i=1}^{n} b_i x_i \geq r^2 \\
& \sum_{i=1}^{n} = k \\
& x_i \in \{0,1\} \quad \forall i \in \{1,\ldots,n\}, i
\end{aligned}
$$

Since this is a quadratic formulation, the authors linearized it into an ILP formulation in order to use a MILP solver. We used the following ILP linearization of the hypervolume scalarized biobjective knapsack with cardinality constraints. We define $Q_{ij} = a_i \cdot b_i$ for all $i, j \in \{1,\ldots,n\}$ and $y_{ij}, i, j \in \{1,\ldots,n\}$, that will get value 1 if and only if $x_i = 1$ and $x_j = 1$. The formulation is as follows:

$$
\begin{aligned}
\max & \sum_{i=1}^{n}\sum_{j=1}^{n} Q_{ij} y_{ij} - \sum_{i=1}^{n}(r^2 a_i + r^1 b_i)y_{ii} + r^1 r^2 \\
s.t. & \sum_{i=1}^{n} a_i y_{ii} \geq r^1 \\
& \sum_{i=1}^{n} b_i y_{ii} \geq r^2 \\
& \sum_{i=1}^{n} y_{ii} = k \\
& \sum_{\substack{i=1 \\ i\neq j}}^{n} y_{ij} \leq (k-1) \cdot y_{jj} \qquad \forall j \in \{1,\ldots,n\} \\
& y_{ij} = y_{ji} \qquad \forall i, j \in \{1,\ldots,n\}, i < j \\
& y_{ij} \leq y_{ii} \qquad \forall i, j \in \{1,\ldots,n\}, i \neq j \\
& y_{ij} \geq y_{ii} + y_{jj} - 1 \qquad \forall i, j \in \{1,\ldots,n\}, i < j \\
& y_{ij} \in \{0,1\}
\end{aligned}
$$

This formulation was implemented in SCIP version 7.0.1.0 in `C++` version 14 and compiled with g++ version 9.3.0, which corresponds to the method *solve*() in HDS and PHDS. In both cases the framework was set such that the algorithm only stopped when all non-dominated points were found. Details about the implementation are provided in Chapter 7.

A series of tests were performed on HDS (Algorithm 1) and PHDS (Algorithm 3) of the dichotomic scheme in order to compare the results. We followed the same generation of problem instances as used in [22], that is, a_i and b_i follow a uniform discrete distribution within

n	k	HDS	PHDS	Speedup
10	$n/4$	1.00	0.53	1.89
	$n/2$	1.43	0.78	1.83
	$3n/4$	0.28	0.20	1.40
20	$n/4$	31.64	13.96	2.27
	$n/2$	62.99	21.84	2.88
	$3n/4$	15.51	7.97	1.95
30	$n/4$	367.09	82.31	4.46
	$n/2$	975.19	190.52	5.12
	$3n/4$	277.19	65.80	4.21
40	$n/4$	1389.58	384.70	3.61
	$n/2$	4499.89	1558.89	2.89
	$3n/4$	1535.07	483.32	3.18
50	$n/4$	3863.71	1030.37	3.75
	$n/2$	28621.70	10224.93	2.80
	$3n/4$	6272.20	1795.22	3.49

Table 6.1: Average execution time, in seconds, for dichotomic scheme solved sequentially (HDS) and with parallelization (PHDS) for instances of size n and cardinality constraint k and speedup of the parallel implementation

[1,250] and were generated using the function *random.randint*() available in *Python*, version 3.8.5. A total of 25 instances were generated for each size $n = \{10, 20, 30, 40, 50\}$ and cardinality values $k = \{n/4, n/2, 3n/4\}$. The running time is measured only with respect to Algorithm 1 and Algorithm 3 using function $std :: chrono :: high\_resolution\_clock :: now()$ in `C++`. The results obtained by the dichotomic scheme were validated against the output of a brute force algorithm implemented in *Python*.

Table 6.1 presents the average running time of the 25 instances for each n and k for both Algorithm 1 referred as HDS and Algorithm 3 referred as PHDS. We also present a speedup of the parallel implementation. We can observe that the parallel algorithm has an overall lower execution time than sequential, with a higher speedup on higher values of n. The smaller speedup value for size $n = 10$ and 20 is expected because there are less nondominated points to be found and, consequently, less subproblems to solve in parallel, leading to a smaller benefit with parallel. For larger n, the benefits of parallel execution become more evident, with speedup values of up to fives times. This is very significant as, with this size, problems can take upwards of hours to solve.

We can also observe that, for the hypervolume scalarized biobjective knapsack problem, a cardinality value of $k = n/2$ seems to be worst case scenario, having the highest execution time when compared with the other cardinality values tested, across all different sizes of n.

6.2 Multidimensional case

In this section, we describe the experimental analysis of dichotomic search scheme for $m > 2$ (Algorithm 4) for different methods of selecting the subproblem to solve. Since we did not have the ILP linearized formulation of the knapsack problem for $m > 2$ objectives, we decided to consider a set of nondominated points as input data. Therefore, rather than calling *solve* (see Line 7 in Algorithm 4) to solve an ILP formulation, the algorithm returns

Dims	Number Points	Subproblems Generated				Subproblems Solved			
		Selection Method			Total	Selection Method			Total
		Queue	Random	Stack		Queue	Random	Stack	
3D	10	27.00	27.00	27.00	30.00	**21.24**	22.16	25.08	30.00
	20	57.00	57.00	57.00	60.00	**47.28**	48.88	54.28	60.00
	30	87.00	87.00	87.00	90.00	**74.72**	76.64	83.64	90.00
	40	117.00	117.00	117.00	120.00	**106.68**	108.28	113.92	120.00
	50	147.00	147.00	147.00	150.00	**135.64**	137.88	141.8	150.00
4D	10	44.96	45.12	45.28	50.04	**26.68**	31.00	40.88	50.04
	20	108.20	108.64	108.40	113.52	79.84	**73.56**	94.40	113.52
	30	174.00	174.44	174.64	179.76	143.76	**122.24**	153.52	179.76
	40	240.88	241.52	241.48	246.48	211.04	**183.44**	217.48	246.48
	50	311.56	311.56	311.92	316.96	272.48	**233.40**	285.92	316.96
5D	10	84.04	84.84	85.88	94.64	42.00	**39.72**	64.68	94.64
	20	237.40	238.28	237.64	248.40	131.60	**103.32**	175.40	248.40
	30	403.24	404.68	406.44	416.40	246.64	**186.56**	294.64	416.40
	40	586.04	588.44	586.60	597.52	464.92	**324.52**	456.64	597.52
	50	792.20	794.44	792.44	803.44	630.48	**454.88**	619.12	803.44

Table 6.2: Total number of subproblems generated and solved by the algorithm and average number of subproblems generated and solved by each of the selection methods for each number of objectives and number of nondominated points

the point, from the input data, that maximizes the hypervolume indicator with respect to the reference point chosen. For this reason, we did not record the overall CPU-time, but the number of subproblems generated and solved.

We consider the following three selection methods to determine the next subproblem to solve: Queue, which is a First In, First Out (FIFO) data structure and corresponds to a Breadth-first search; Stack, which is a First In, Last Out (FILO), corresponding to a Depth-first search; and Random, in which a reference point from in R (see Line 6 in Algorithm 4) is chosen at random. The random element of R is obtained by using `std::uniform_int_distribution` and `std::mt19937`, available in `C++` version 14. The function *extractDominatedSet*() (see Line 1 in Algorithm 5) used is part of a library for filtering and maintaining a set of nondominated points implemented in `C++` and is presented in [5].

A total of 25 instances were generated for each number of nondominated points considered (10, 20, 30, 40 and 50) and for each number of objectives $m = 3$, 4 and 5. The nondominated points were generated using Mullers method to obtain points uniformly distributed on the surface of a hyperdimensional sphere [19].

For each instance, we recorded the total number of subproblems generated and solved by the algorithm (see Line 10 and Line 7 of Algorithm 4, respectively). We decided to record these values because, as explained in Section 5.1, the number of reference points generated by each nondominated point, for $m > 2$, is not always the same, as opposed to $m = 2$, in which two reference points are always generated by each nondominated points. Note that a reference point originates a subproblem and, as such, both of these can be seen as the same.

The preliminary result we obtained were all equal between the different selection methods, which means that all selection methods generated and solved the same number subprob-

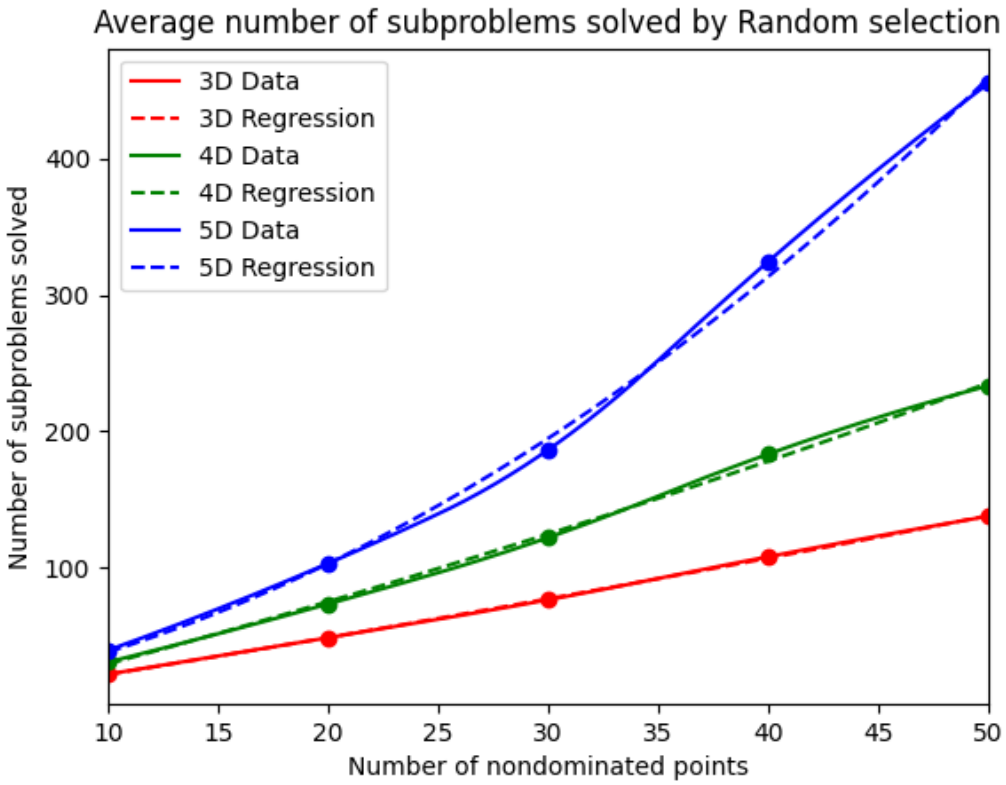


Figure 6.1: Plot of the average number of subproblems solved by random selection method for each number of objectives m and for each number of nondominated points

lems in each scenario by the end of the algorithms execution. However, we noticed that some selection methods arrived at the complete set of nondominated points faster than others. As such we also recorded the number of subproblems at the moment that the nondominated set was found.

Table 6.2 presents the average number of subproblems generated and solved for each number of nondominated points, in column *Number Points*, and number of objectives, in column *Dims*. The columns *Total* represent the number of subproblems generated and solved by the algorithm at the end of its execution. The columns under Selection Method represent the average number of subproblems generated and solved by each method.

As we can observe, for 3 objectives, the queue selection method solves, on average, slightly fewer subproblems when compared to the other two. However for 4 and 5 objectives, the random selection method solves less subproblems overall, by a significant margin. As random selection is the most beneficial for higher number of objectives, we decided to analyze how the number of subproblems solved grows with the number of nondominated points used as input. We performed a linear, quadratic and cubic regression for each number of objectives of the random selection method. In Figure 6.1 is illustrated a line plot of the average values, represented by the solid lines, alongside the regression with better R^2 value for each number of objectives, represented by the dashed lines. The best regression for all cases was quadratic, meaning that, the number of subproblems may grow quadratically with the number of nondominated points.

The expressions that we obtained for the quadratic regressions, corresponding to the dashed lines in Figure 6.1, are as follows:

- 3 objectives: $y = 0.01x^2 + 2.50x - 3.66$ with a R^2 of 0.97;
- 4 objectives: $y = 0.02x^2 + 3.98x - 12.02$ with a R^2 of 0.89;
- 5 objectives: $y = 0.13x^2 + 2.45x + 0.46$ with a R^2 of 0.72;

6.3 Discussion

In this chapter we presented the experimental analysis performed on the HDS, the PHDS and m-HDS algorithms. The speedup obtained by the PHDS is substantial, when compared to the sequential version, specially for larger sizes of input. In the case of the m-HDS algorithm, we were able to conclude that a random selection method for the next subproblem to solve has the best performance. This selection method could be useful for a version of m-HDS capable of using the greedy principle, presented in Section 3.3. It is left as an open question if the results obtained pertaining the selection method are due to the method used for generating the inputs or if there is no correlation and random selection is the best in all scenarios.

Chapter 7

Framework User Manual

One of the objectives of this dissertation is to build a framework in `C++` that implements the dichotomic search algorithm for $m = 2$, Hypervolume Dichotomic Scheme (HDS), described in Chapter 3. This framework will serve as a base that can be expanded to any biobjective problem as long as an Hypervolume Scalarization can be derived. The authors in [22] have developed a simple prototype for performing experiments using python and using SCIP as an ILP solver. However, the solver was solving hypervolume scalarizations from data files that were generated for each new subproblem. This has a strong overhead on the overall running time. Moreover, no attempt to make a code that could be reused for other problems was considered. For this reason, we developed a framework that integrates SCIP `C++` libraries with the dichotomic approach and allows to be easily adapted to other problems. The framework is available on github [1] and the current version has implemented the HDS, allowing to find for the complete nondominated set (see Algorithm 1) or to find a subset of the nondominated set that has an approximation guarantee (see Algorithm 2), as discussed in Section 3.3.

In Section 7.1 we will identify the requirements that ensure that the framework functions as expected. In Section 7.2 we give an overview of the functions available to the user and an explanation of their functionality. In Section 7.3 we present a step-by-step guide on how to use the framework, and in Section 7.4 we present examples on how to implement the problem described in Section 6.1 with SCIP in `C++`.

7.1 Requirements Elicitation

To ensure that the framework has all the expected functionalities and works as intended, we gathered a set of the requirements. Each requirement includes an unique ID, a small description and the priority. The priority is a value from 1-5, with 5 being the highest and 1 the lowest.

7.1.1 Functional Requirements

In this subsection we will present a list of functional requirements. These requirements define what the program must be able to do and its behavior, and are essential for the

[1] Frameworks code and user guide available at https://github.com/AlexdBrito/HVScalar

correct functioning of the framework. These requirements are presented in Table 7.1.

ID	Description	Priority
FR-01	The user can implement a data structure and set it on the framework	3
FR-02	The user can implement an input function and set it on the framework	3
FR-03	The user can implement a function to initialize the problem and set it on the framework	5
FR-04	The user can implement a function that solves an instance of the problem and returns a point	5
FR-05	The user can implement a function that frees the problems resources and set it on the framework	5
FR-06	The user can choose to calculate all optimal points or a subset of them	2
FR-07	The user can specify the number of points that belong to the Pareto Front (J) to be calculated	4
FR-08	If the problem does not have J optimal points the framework must return all that were calculated	5
FR-09	If the problem does not have J optimal points the framework must warn the user	1
FR-10	The user can opt for a verbose version of the framework, giving more information on each step	2
FR-11	The user can indicate a folder where all the frameworks verbosity output will be written to	2
FR-12	The results must be displayed on the console and/or sent to a file indicated by the user	3
FR-13	The framework must accept problems that are not implemented with SCIP	4
FR-14	In the case of error the framework must correctly free all resources and warn the user	4

Table 7.1: Functional Requirements

7.1.2 Non-Functional Requirements

In this subsection we will present a list of non-functional requirements. These requirements are not essential and failing to implement them does not hinder the basic functionalities of the framework. They should not be ignored however, as they are fundamental for a good user experience and overall usability. These requirements are presented in Table 7.2.

ID	Description	Priority
NFR-01	All functions must have explicit and easy to understand names	4
NFR-02	All functions must be documented with description of the functionality, input and output	3
NFR-03	All loose values and variables must be global defined with an easy to understand name	3
NFR-04	All resources must be correctly freed before the end	5

Table 7.2: Non-Functional Requirements

7.2 Overview of available functions

In this section we give an overview of the variables and functions, implemented in the framework, that the user interacts with. We will go over their inputs and outputs as well as an explanation of what each variable or function does and their purpose.

The framework uses only one structure, `HVS` to store all the necessary data. This structure holds information about all the user implemented functions and basic information about the problem such as the number of nondominated points to find, if set, the objective direction, either maximization or minimization, the initial reference point and the data structure created by the user. It is required as input by all the frameworks functions.
We define a point, `HVS_Point`, as a vector of doubles. We also define a set `HVS_Set`, as a set of pointers to points (`HVS_Point`). The set also has a comparator, which keeps the points ordered in relation to the first objective. Lastly we define two enumerates, `HVS_ObjectiveDir` and `HVS_SolveType`. The former defines the objective direction of the problem (`HVS_MIN` and `HVS_MAX`) and the latter defines the whether the goal is to find all nondominated points (`HVS_2D_ALL`) or to find a subset of all nondominated points (`HVS_2D_NUM`).
The variable `data` and functions `input`, `init`, `solve` and `close` refer to user implemented data structure and functions, respectively. The variable `numNdPnts` refers to an integer and variables `verbose` refers to a boolean. More details are given below.

The available functions are the following:

- `HVSstart(HVS)`: Receives a pointer to the framework data structure, initialized it and sets default values.
- `HVSfree(HVS)`: Receives a pointer to the framework data structure and frees all associated data.
- `HVSsetProblemDS(HVS, data)`: Receives a user implemented data structure that contains the problems variables, saving a reference to it on the framework data structure.
- `HVSsetInput(HVS, input)`: Receives a user implemented function that reads the problems input, saving a reference to it on the framework data structure.
- `HVSsetInit(HVS, init)`: Receives a user implemented function that initializes the problem and all the necessary variables, saving a reference to it on the framework data structure.

- `HVSsetSolve(HVS, solve)`: Receives a user implemented function that solves a version of the problem and returns the best point calculated, saving a reference to it on the framework data structure.
- `HVSsetClose(HVS, close)`: Receives a user implemented function that frees all variables associated with the problem, saving a reference to it on the framework data structure.
- `HVSsetObjectiveDir(HVS, HVS_ObjectiveDir)`: Receives the value representing the objective of the function, `HVS_MAX` for maximization and `HVS_MIN` for minimization, setting it on the frameworks data structure.
- `HVSsetNumNdPoints(HVS, numNdPnts)`: Receives an integer value indicating how many nondominated points the framework needs to find before stopping, only used when `HVS_2D_NUM` is selected as the solving method.
- `HVSsetIRefPoint(HVS, HVS_Point)`: Receives a point that is set as the initial reference point for the frameworks solving process.
- `HVSsetVerbose(HVS, verbose)`: Receives a boolean. If set to *True*, the framework will output additional information during the solving process.
- `HVSsolve(HVS, HVS_SolveType)`: Receives and executes the solving method desired. This function will execute all the user implemented functions in the following order: `input`, `init`, `solve` and `close`. If called with `HVS_2D_ALL`, it will calculate all nondominated points, while with `HVS_2D_NUM`, calculates the number of nondominated points indicated by `HVSsetNumNdPoints()`, defined above. If the number of solutions desired is not set, `HVS_2D_NUM` will calculate all nondominated points.
- `HVSgetSols(HVS)`: Returns the set of nondominated points found.
- `HVSprintSet(HVS_Set)`: Receives a `HVS_Set` of points and print them to the console.

7.3 User Guide

In this section we present a step-by-step guide on how a user can use the framework to implement a problem, run the desired algorithm and print the resulting nondominated set on the console. The flow of the framework can be seen in Figure 7.1, in which the colored words represent what the user must implement, and is presented in the following steps:

1. Implement functions `input`, `init`, `solve` and `close`. These functions are responsible for reading all the inputs necessary, initializing the problem, solving an instance of the problem and freeing all the resources allocated to the problem, respectively.
2. If, in order to solve a subproblem additional variables are needed, excluding the reference point, create a data structure that holds all the required variables associated to the implemented problem.
3. On the `main()` function, declare the main data structure `HVS` and initialize it by calling `HVSstart(HVS)`.

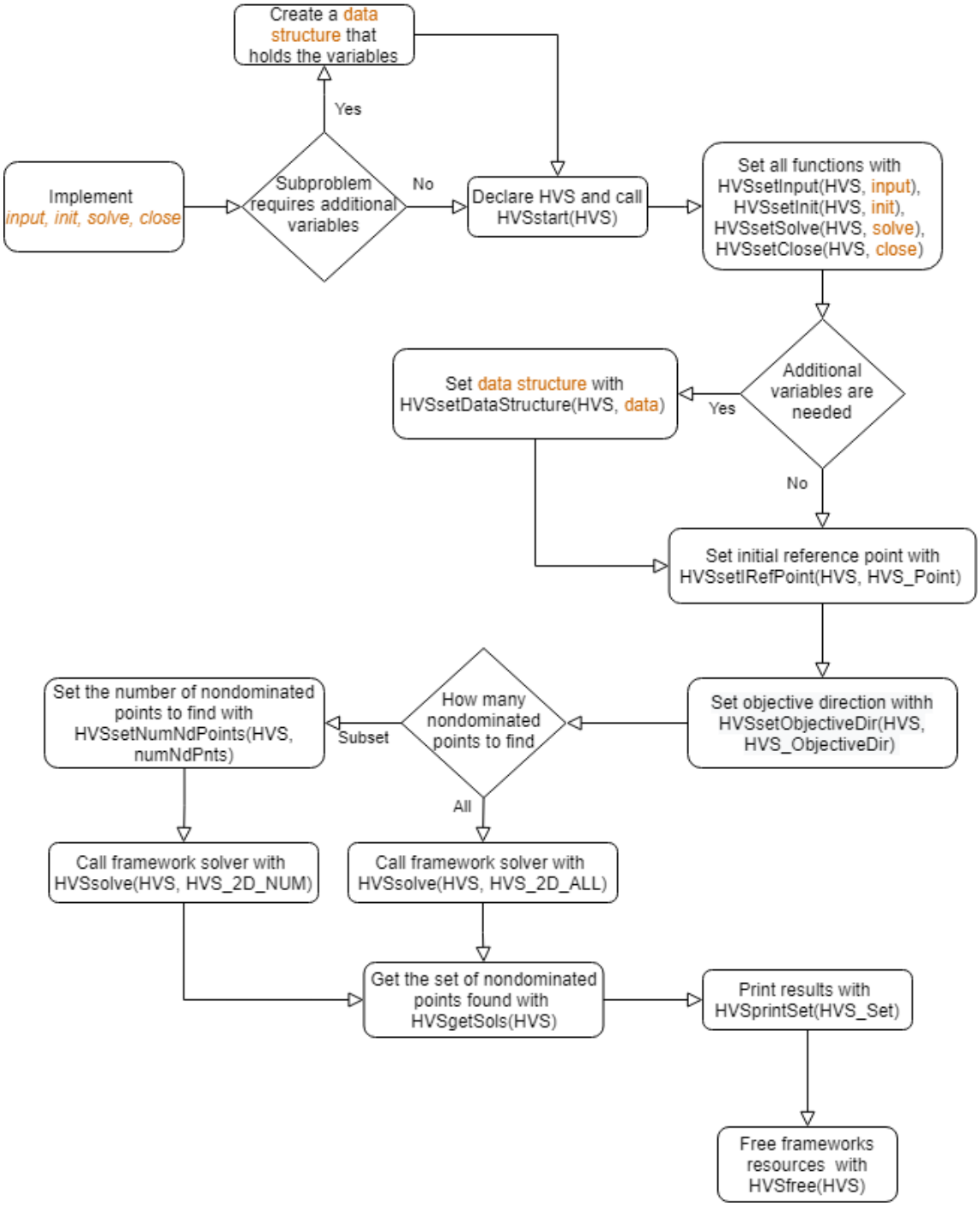


Figure 7.1: Flow diagram of the steps required to use the framework. The highlighted words represent functions or a data structure that the user must implement

4. Set all the user implemented functions on the framework by calling, in any order, `HVSsetInput(HVS, input)`, `HVSsetInit(HVS, init)`, `HVSsetSolve(HVS, solve)`, `HVSsetClose(HVS, close)`, `HVSsetDataStructure(HVS, data)`.

5. In order to set the initial reference point, the function `HVSsetIRefPoint(HVS, HVS_Point)` should be called. The objective of the problem should also be set using function `HVSsetObjectiveDir(HVS, HVS_ObjectiveDir)`.

6. If the user wants to calculate a specific number of nondominated points, they must

call function `HVSsetNumNdPoints(HVS, numNdPnts)`, with `numNdPnts` being an integer of how many points to calculate.

7. With all necessary variables set, function `HVSsolve(HVS, HVS_SolveType)` should be called. This will run the algorithm chosen.

8. The results can be obtained by using function `HVSgetSols(HVS)`, which returns a `HVS_Set`, and displayed using function `HVSprintSet(HVS_Set)`.

9. Lastly, the function `HVSfree(HVS)` must be called to release all the frameworks variables.

7.4 Implementation example

In this section we will follow the steps given in Section 7.3 and give example of an implementation of the ILP linearization of the hypervolume scalarized bi-objective knapsack with cardinality constraints, presented in Section 6.1 in `C++` using SCIP. We will give examples of the implementation of functions `init, solve` and `close` and an example of a data structure that holds all the variables needed by the problem. The examples presented are simplified. Throughout the examples `SCIP_CALL()` is used. This is a built in define of SCIP and ensures that, in case of error, it is caught and the user is warned of the function that gave an error.

Figures 7.2, 7.3, 7.4 and 7.5 are parts of function `init`. Figures 7.6 and 7.7 refer to function `solve`. Figure 7.8 presents a snippet of the implementation of function `close`. Figure 7.10 refers to the main function of the program.

As shown in Section 7.3 the first steps consists of implementing the functions input, init, solve and close, and a data structure if necessary. We will not shown an example of function input.

In order to create a problem with SCIP, we first need to start the SCIP environment, add the plugins and set the problems objective. This is illustrated in Figure 7.2. Variable `scip` will be used throughout the examples and refers to the one initialized in this figure.

```
/* init SCIP */
SCIPcreate(scip);

/* include default plugins */
SCIP_CALL(SCIPincludeDefaultPlugins(scip));

/* define problem */
SCIP_CALL(SCIPcreateProbBasic(scip, "Bi-Objective Knapsack"));

/* set objective function direction */
SCIP_CALL(SCIPsetObjsense(scip, SCIP_OBJSENSE_MAXIMIZE));
```

Figure 7.2: Declaration of the SCIP environment, add addition of the default plugins, definition of a problem and setting of the objective direction

```
std::vector<std::vector<SCIP_Real>> matrixQ;
std::vector<SCIP_VAR *> matrixY;

/* calculate matrixQ[0][1] and add it to SCIP as a variable */
matrixQ[0][1] = arrayA[0] * arrayB[1];
SCIP_CALL(SCIPcreateVarBasic(scip, &(matrixY.at(1)), ("y01"), 0.0,
↪  1.0, matrixQ[0][1], SCIP_VARTYPE_BINARY));
SCIP_CALL(SCIPaddVar(scip, matrixY.at(1)));

/* calculate r1r2 and add it to SCIP as a variable */
SCIP_CALL(SCIPcreateVarBasic(scip, &(matrixY.back()), ("r1r2"), 1.0,
↪  1.0, iRefPoint[0] * iRefPoint[1], SCIP_VARTYPE_BINARY));
SCIP_CALL(SCIPaddVar(scip, matrixY.back()));
```

Figure 7.3: Calculation of two binary constraints, creation of the corresponding SCIP variables and addition the variable to SCIP

```
SCIP_CONS *cons1 = nullptr;
std::vector<SCIP_CONS *> cons4;
/* init constraints */
/* cons1 -> sum{i=1,..,n} (a{i}y{ii}) >= r1 */
/* cons4 -> sum{i=1,..,n; i!=j} (y{ij}) <= (k-1)y{jj} for all j in
↪  {1,..,n} */
SCIP_CALL(SCIPcreateConsBasicLinear(scip, &cons1, "cons1", 0, nullptr,
↪  nullptr, iRefPoint.first, SCIPinfinity(scip)));

for (int i = 0; i < numElems; i++) {
    SCIP_CALL(SCIPaddCoefLinear(scip, cons1, matrixY.at(numElems * i +
    ↪  i), arrayA[i]));
}

for (int j = 0; j < numElems; j++) {
    SCIP_CALL(SCIPcreateConsBasicLinear(scip, &cons4.at(j), ("cons4_"
    ↪  + std::to_string(j + 1)).c_str(), 0, nullptr, nullptr,
    ↪  -SCIPinfinity(scip), 0));
    for (int i = 0; i < numElems; i++) {
        if (i != j) {
            SCIP_CALL(SCIPaddCoefLinear(scip, cons4.at(j),
            ↪  matrixY.at(numElems * i + j), 1));
        }
    }
    SCIP_CALL(SCIPaddCoefLinear(scip, cons4.at(j), matrixY.at(numElems
    ↪  * j + j), - (coef - 1)));
}
```

Figure 7.4: Creation of a simple constraint and a vector of constraints, setting of their coefficients

The next step consists of adding the binary variables to the problem. Figure 7.3 shows the calculation of two coefficients, `Q[0][1]` and r^1r^2.Matrix `MatrixQ` holds the coefficients and `MatrixY` hold the associated SCIP variables. `iRefPoint` is the initial reference point. As coefficient `Q[0][1]` is associated to a binary variable, the lower bound is set to 0.0 and the upper bound is set to 1.0. In contrast, as r^1r^2 must always be considered, both lower and upper bounds are set to 1.0.

```
/* add constraints to scip */
SCIP_CALL(SCIPaddCons(scip, cons1));

for (int i = 0; i < cons4.size(); i++) {
    SCIP_CALL(SCIPaddCons(scip, cons4.at(i)));
}
```

Figure 7.5: Addition of the constraints to SCIP

```
SCIP *scipCp = nullptr;

/* creating the SCIP environment that will be used to solve the
↪  problem  */
SCIPcreate(&scipCp);

/* updating matrixQ in position (i,i) due to new reference point  */
for(int i = 0; i < numElems; i++) {
    matrixQ[i][i] = arrayA[i] * arrayB[i] - (refPoint[HVS_Y] *
    ↪  arrayA[i] + refPoint[HVS_X] * arrayB[i]);
    SCIPchgVarObj(scip, matrixY.at(numElems * i + i), matrixQ[i][i]);
}
/* updating r1r2 due to new reference point */
SCIPchgVarObj(scip, matrixY.back(), refPoint[HVS_X] *
↪  refPoint[HVS_Y]);

/* updating constraints due to new reference point */
SCIPchgLhsLinear(scip, cons1, refPoint[HVS_X]);
SCIPchgLhsLinear(scip, cons2, refPoint[HVS_Y]);

/* copying the problem to the new SCIP env */
SCIPcopy(scip, scipCp, nullptr, nullptr,
↪  (std::to_string(refPoint[HVS_X]) + "," +
↪  std::to_string(refPoint[HVS_Y])).c_str(), 1, 1, 0, 1, nullptr);
```

Figure 7.6: Modification of the necessary problem variables and constraints due to new reference point and creation of a copy of the SCIP environment in other to be solved

After that, we need to create the constraints as well as to set their coefficient. If a constraint is an equality, the upper bound and the lower bound must be set to the same value. If a constraint is an inequality with one bound being infinite, `SCIPinfinity(scip)` must be used. Figure 7.4 shows examples of how to create basic constraints in SCIP and how to

add coefficients to each constraint. In this figure, `numElems` refers to the number of values for each objective and `coef` refers to the coefficient constraint of the problem.

The last step necessary in order to initialize a SCIP problem is to add the created constraints to the environment. This is illustrated in Figure 7.5, in which the two constraints created previously are added.

In order to solve a subproblem, we need to take in consideration that, because each subproblems has a different reference point, all the problems variables and constraints that have reference to the reference point must be updated. This is illustrated in Figure 7.6. In this example we also create a copy of the SCIP environment after all the modifications are done. This is done, because in order to modify the problems variables and constraints it is required to have a reference of the container that holds them. Therefore, as we have the information of the variables and constraints of the problem created previously we do the necessary alterations on the main SCIP environment. We also create a copy of the problem because it makes it easier to solve multiple subproblems in a row, as we do not need to free the solved and transformed state in order to go back to the problem specification, as shown in Section 2.4.

```
/* solving the problem */
SCIPsolve(scipCp);

/* getting the best solution  */
SCIP_SOL *sol = SCIPgetBestSol(scipCp);

int numVars = SCIPgetNOrigVars(scipCp);
SCIP_VAR **vars = SCIPgetOrigVars(scipCp);
double *results = (double *)malloc(sizeof(double)*numVars);
SCIPgetSolVals(scipCp, sol, numVars, vars, results);

/* calculating the hypervolume coordinates of the solution  */
SCIP_Real hvX = 0;
SCIP_Real hvY = 0;
for(int i = 0; i < numElems; i++) {
    hvX += arrayA[i] * results[i * numElems + i];
    hvY += arrayB[i] * results[i * numElems + i];
}
free(results);
/* freeing the SCIP environment */
SCIPfree(&scipCp);
```

Figure 7.7: Solving of a SCIP subproblem, extraction of the results and calculation of the new nondominated point

With the copy of the modified SCIP environment we can call the function to solve the subproblem, extract the results and free the copied SCIP. This is shown in Figure 7.7, in which the copied environment is solved. We use `SCIPgetBestSol`, `SCIPgetNOrigVars`, `SCIPgetOrigVars` and `SCIPgetSolVals` to extract the value of the binary variables of the problem. From their output, it is possible to calculate the resulting nondominated point.

```
/* freeing the variables */
for (int i = 0; i < matrixY.size(); i++) {
    SCIP_CALL(SCIPreleaseVar(scip, &matrixY.at(i)));
}

/* freeing the constraints */
SCIP_CALL(SCIPreleaseCons(scip, &cons1));

for (int i = 0; i < cons4.size(); i++) {
    SCIP_CALL(SCIPreleaseCons(scip, &cons4.at(i)));
}

/* freeing the SCIP environment */
SCIP_CALL(SCIPfree(&scip));
```

Figure 7.8: Freeing the variables, constraints and the SCIP environment.

SCIP requires all variables and constraints to be freed. Figure 7.8 shows how to free these variables and constraints and finally how to free SCIP itself. If some variable of constraint is not freed before calling `SCIPfree`, an error is given.

As the problem presented requires additional variables, like for example `matrixQ`, we need to create a data structure that stores them. This is illustrated in Figure 7.9. The data structure shown only has the variables that were used in these examples. Therefore, other necessary variables, like for the other constraints, are omitted.

```
struct problemVariables {
    int numElems; /**< Number of elements for each objective */
    int maxElems; /**< Max number of elements in each solution */
    std::vector<SCIP_Real> arrayA; /**< Gain for objective A */
    std::vector<SCIP_Real> arrayB; /**< Gain for objective B */
    SCIP *scip; /**< Pointer for SCIP*/
    std::vector<std::vector<SCIP_Real>> matrixQ;
    std::vector<SCIP_VAR *> matrixY;
    SCIP_CONS *cons1 = nullptr;
    std::vector<SCIP_CONS *> cons4;
};
```

Figure 7.9: Demonstration of the data structure required for the biobjective knapsack problem

With all required functions implemented, we can call all the required functions on `main()`, solve the problem and print the resulting set of nondominated points found, which corresponds to steps 3 to 9 presented in Section 7.3. Figure 7.10 contains an example of how the `main()` function could be implemented. Although we call `setInitialRefPoint()` on the main function, it can be called inside the user defined function `input`, allowing the reference point to be passed as input alongside the problems data. In this example we use `HVS_2D_NUM` in order to find 5 nondominated points, as set by `HVSsetNumNdPoints()`.

```
HVS *hvs;
HVSstart(&hvs);

HVSsetInput(hvs, readInput);
HVSsetInit(hvs, initProblem);
HVSsetSolve(hvs, solveProblem);
HVSsetClose(hvs, closeProblem);
HVSsetDataStructure(hvs, new problemVariables());
HVSsetNumNdPoints(hvs, 5);
setInitialRefPoint(data, {0.0,0.0});
HVSsetVerbose(hvs, true);
HVSsolve(HVS_2D_NUM, hvs);
HVS_Set results = HVSgetSols(hvs);
HVSprintSet(results);

HVSfree(&hvs);
```

Figure 7.10: Example of the `main()` function

This page is intentionally left blank.

Chapter 8

Conclusions and Future Work

In this dissertation we proposed a version of the Hypervolume Dichotomic Scheme (HDS) algorithm, proposed in [22], that is capable of solving multiple subproblems in parallel, namely Parallelized Hypervolume Dichotomic Scheme (PHDS). This version can be stopped early in order to obtain a subset of nondominated set. But, unlike the sequential version, the subset found does not have the same approximation guarantees, as it no longer uses the greedy principle. We performed an experimental analysis in order to compare the execution time of HDS and PHDS. We concluded that the parallel version is significantly faster, specially for larger input sizes.

In addition we also proposed an extension of HDS, namely, m-HDS, for any number of objectives, but under the assumption that the nondominated points are in general position, i.e. no two nondominated points share a coordinate. This extension also does not use the greedy principle and it uses a different method of calculating the reference points based on the work of Lacour et al. in [17]. We also have tested m-HDS for 3, 4 and 5 objectives and analyzed the effect of different choices for the next subproblem to be solved. We have found that a random choice allows to find the complete nondominated set by generating and solving less number of subproblems.

Finally, we implemented HDS in a way that could be used by a user that has knowledge of `C++` and SCIP to solve any biobjective combinatorial optimization problem, given that is possible to derive the corresponding hypervolume scalarization. The implementation is available in a public repository for the scientific community and practitioners to use it. In addition, we provided the required documentation to help the user to implement it, with an application example for a biobjective knapsack problem.

Future Work

A future work opportunity is to further improve the proposed m-HDS. This version works under the assumption that the nondominated points are in the general position. However, this is impractical for real world usage, as it requires knowledge of the nondominated set a priori. Therefore, further work would be to investigate how to overcome the problems that arise when relaxing this assumption. The version described in this work computes only an m-dimensional box that is bounded from below by the reference point and from above by the optimal point. We showed that the hypervolume contribution of this point is not only a m-dimensional box, but a m-dimensional orthogonal polyhedron that is bounded

from below by a set of reference points and other nondominated points found. In order to compute this hypervolume contribution, more sophisticated methods are required, as those described in [11]. Other interesting aspect to consider is the parallel implementation of m-HDS, which raises difficulties since the subproblems to be solved are not independent, in general. Some preliminary experiments indicated that is not possible to simply extend PHDS for more than two objectives, given that some nondominated points would not be found. We suspect that this issue is related to the orderding of the problems to be solved. Lastly, is it of particular interest to expand the framework presented here to other classes of problem.

This page is intentionally left blank.

References

[1] Tobias Achterberg. *Constraint Integer Programming.* Doctoral thesis, Technische Universität Berlin, Fakultät II - Mathematik und Naturwissenschaften, Berlin, 2007.

[2] V. Joseph Bowman Jr. On the relationship of the Tchebycheff norm and the efficient frontier of multiple-criteria objectives. In Hervé Thiriez and Stanley Zionts, editors, *Multiple Criteria Decision Making*, pages 76–86. Springer Berlin Heidelberg, 1976.

[3] Vira Chankong and Yacov Y Haimes. *Multiobjective decision making: theory and methodology.* Dover Publications, 2008.

[4] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach.* Gulf Professional Publishing, 1999.

[5] Duarte Dias, Alexandre Jesus, and Luís Paquete. A software library for archiving non-dominated points. GECCO '21: Genetic and Evolutionary Computation Conference, Lille, France, 2021.

[6] Kerstin Dächert, Kathrin Klamroth, Renaud Lacour, and Daniel Vanderpooten. Efficient computation of the search region in multi-objective optimization. *European Journal of Operational Research*, 260(3):841–855, 2017.

[7] Matthias Ehrgott. *Multicriteria optimization*, volume 491. Fifth edition, 2005.

[8] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9):948–960, 1972.

[9] Saul Gass and Thomas Saaty. The computational algorithm for the parametric objective function. *Naval Research Logistics Quarterly*, 2(1-2):39–45, 1955.

[10] Arthur M Geoffrion. Proper efficiency and the theory of vector maximization. *Journal of Mathematical Analysis and Applications*, 22(3):618–630, 1968.

[11] Andreia P. Guerreiro and Carlos M. Fonseca. Computing and updating hypervolume contributions in up to four dimensions. *IEEE Transactions on Evolutionary Computation*, 22(3):449–463, 2018.

[12] Andreia P Guerreiro, Carlos M Fonseca, and Luís Paquete. The hypervolume indicator: Problems and algorithms. *CoRR*, abs/2005.00515, 2020.

[13] Osman Güler. *Linear Programming*, pages 195–205. Springer, New York, 2010.

[14] Y.Y. Haimes, L. S. Lasdon, and D. A. Wismer. On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-1(3):296–297, 1971.

[15] Ignacy Kaliszewski. A modified weighted tchebycheff metric for multiple objective programming. *Computers & Operations Research*, 14(4):315–323, 1987.

[16] Kathrin Klamroth and Tind Jørgen. Constrained optimization using multiple objective programming. *Journal of Global Optimization*, 37(3):325–355, 2007.

[17] Renaud Lacour, Kathrin Klamroth, and Carlos M. Fonseca. A box decomposition algorithm to compute the hypervolume indicator. *Computers & Operations Research*, 79:347–360, 2017.

[18] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.*, 35(5):145–156, May 2000.

[19] Mervin E. Muller. A note on a method for generating points uniformly on n-dimensional spheres. *Communications of ACM*, 2(4):19–20, April 1959.

[20] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, New York, 2006.

[21] L. Paquete. *Stochastic Local Search Algorithms for Multiobjective Combinatorial Optimization*. PhD thesis, Technische Universtät Darmstadt, 2006.

[22] Luís Paquete, Britta Schulze, Michael Stiglmayr, and Ana C. Lourenço. Computing representations using hypervolume scalarizations. *Computers & Operations Research*, 2021 (in press).

[23] David P. Rodgers. Improvements in multiprocessor system design. *SIGARCH Computer Architecture News*, 13(3):225–231, June 1985.

[24] Meinolf Sellmann and Serdar Kadioglu. Dichotomic search protocols for constrained optimization. In *International Conference on Principles and Practice of Constraint Programming*, pages 251–265. Springer, 2008.

[25] Ralph E Steuer and Eng-Ung Choo. An interactive weighted Tchebycheff procedure for multiple objective programming. *Mathematical programming*, 26(3):326–344, 1983.

[26] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. *SIGPLAN Not.*, 28(7):13–22, July 1993.

[27] David W Wall. Limits of instruction-level parallelism. pages 176–188, 1991.

[28] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms - A comparative case study. In A. E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN V, 5th International Conference, Amsterdam, The Netherlands, September 27-30, 1998, Proceedings*, volume 1498 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 1998.