INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Efficient Algorithms for the Assessment of Stochastic Multiobjective Optimizers

Andreia Próspero Guerreiro

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Doutor João Emilio Segurado Pavão Martins
Orientador:	Doutor Carlos Manuel Mira da Fonseca
Co-orientador:	Doutor Vasco Miguel Gomes Nunes Manquinho
Vogais:	Doutor Alexandre Paulo Lourenço Francisco
	Doutor Luis Manuel Silveira Russo

Novembro 2011

Abstract

In multiobjective optimization, usually due to the complex nature of the problems considered, it is sometimes not possible to find all optimal solutions in a reasonable amount of time. One alternative is to use methods to find good approximation sets, such as Stochastic Multiobjective Optimizers. To assess the outcomes produced by these multiobjective optimizers, quality indicators such as the hypervolume indicator have been proposed. The hypervolume indicator is one of the most studied unary quality indicators, and algorithms are available to compute it in any number of dimensions. It has also been included in stochastic optimizers, such as evolutionary algorithms. Therefore, the time spent computing the hypervolume indicator is an important issue. An alternative to the hypervolume indicator is the Empirical Attainment Function (EAF), which is oriented towards the study of the distribution of the output of different executions of the same optimizer, but the algorithms available are restricted to two and three dimensions. In this thesis, new efficient algorithms to compute the hypervolume indicator are proposed, and new algorithms to compute the EAF in two and four dimensions are developed. As a result, the scope of application of the EAF is extended to four objective problems, and the current upper bound on the complexity of the hypervolume indicator in four dimensions is improved to $O(n^2)$. Finally, the divide-and-conquer paradigm is shown to hold promise for the development of this type of algorithms.

Keywords

Hypervolume Indicator
Empirical Attainment Function
Stochastic Multiobjective Optimizers
Algorithms
Dimension Sweep
Multidimensional Divide and Conquer

Resumo

Em optimização multiobjectivo, geralmente devido à natureza complexa dos problemas considerados, por vezes não é possível encontrar todas as soluções óptimas em tempo razoável. Uma alternativa é usar métodos para encontrar bons conjuntos de aproximação, tais como os Optimizadores Multiobjectivo Estocásticos. Para avaliar os resultados produzidos por optimizadores estocásticos, têm sido propostos indicadores de qualidade, como o indicador de hipervolume. O indicador de hipervolume é um dos indicadores de qualidade mais estudados, havendo disponíveis vários algoritmos para o calcular em qualquer número de dimensões. Este tem também sido incluído em optimizadores estocásticos, como os algoritmos evolutivos, sendo por isso o tempo dispendido a calculá-lo um factor importante. Uma alternativa ao indicador de hipervolume é a Função de Aproveitamento Empírica (FAE), que está orientada para o estudo da distribuição dos resultados de diferentes execuções do mesmo optimizador, mas os algoritmos disponíveis estão limitados a duas e três dimensões. Nesta tese são propostos novos algoritmos eficientes alternativos aos existentes para calcular o indicador de hipervolume, e desenvolvem-se novos algoritmos para calcular a FAE em duas e quatro dimensões. Em consequência, o âmbito de aplicação da FAE é alargado a problemas envolvendo quatro objectivos e o limite superior para a complexidade do cálculo de indicador de hipervolume é melhorado para $O(n^2)$. Finalmente, mostra-se que o paradigma de divisão-e-conquista multidimensional é promissor para o desenvolvimento de algoritmos deste tipo.

Palavras Chave

Indicador de Hipervolume
Função de Aproveitamento Empírica
Optimizadores Multiobjectivo Estocásticos
Algoritmos
Varrimento Dimensional
Divisão e Conquista Multidimensional

Acknowledgements

I would like to thank my family, my parents and sister, for all the patience, understanding, for the support given and for making it possible for me to stand where I am today. I thank my friends for all the support and incentive given. To my supervisor, Carlos Fonseca, my thanks for providing other points of view when I needed them, for all the help, incentive and attention given to this thesis. To my co-supervisor Vasco Manquinho, my thanks for the helpful comments. I would like to thank Michael Emmerich for suggesting the algorithm to compute the hypervolume indicator in four dimensions. Finally, I would like to thank INESC-ID SAT group for providing me a place to work while I was studying in Lisbon and to thank the Control group of the University of Algarve for providing me a place to work close to home.

Andreia Guerreiro

Lisbon, October 2011

Contents

1 Introduction **1**

2 Related Work **3**

2.1 Notation and definitions 3

2.2 Hypervolume indicator 4

2.3 Empirical attainment function 6

2.3.1 Definition 6

2.3.2 Algorithms 10

2.4 Multidimensional divide-and-conquer 16

2.5 Concluding remarks 19

3 Empirical Attainment Function **21**

3.1 EAF in four dimensions 21

3.1.1 Base algorithm 21

3.1.2 Improved algorithm 24

3.1.3 Complexity 30

3.1.4 Experimental results 31

3.2 Output sensitive algorithm for two dimensions 33

3.2.1 Description 34

3.2.2 Complexity 43

3.2.3 Experimental results 45

3.3 Concluding remarks 49

4 Hypervolume Indicator **51**

4.1 New dimension-sweep algorithm for the four-dimensional case 51

4.1.1 General description 51

4.1.2 Individual contribution of a point in three dimensions 53

4.1.3 Data structures 55
4.1.4 Detailed description 56
4.1.5 Complexity 58
4.1.6 Experimental results 58
4.2 Divide-and-conquer algorithm for the three-dimensional case 61
4.2.1 Description 61
4.2.2 Complexity 67
4.2.3 Experimental results 68
4.3 Concluding remarks 69

5 Concluding Remarks **71**

List of Figures

2.1 A two-objective example of the hypervolume indicator. 4

2.2 Example of three runs of a multiobjective optimizer on the same two-objective problem. . 7

2.3 Representation of the superlevel sets $V_{1/3}$, $V_{2/3}$ and $V_{3/3}$ for the example in Figure 2.2 . 8

2.4 Representation of J_1, J_2 and J_3 for the example in Figure 2.2 9

2.5 A two-dimensional example and the representation of the corresponding L_1, L_2 and L_3 . . 10

2.6 Example of three runs of a multiobjective optimizer for the same three-objective problem. 11

2.7 The projection onto the (x, y)-plane in the example in Figure 2.6 for a given a value of z . 12

2.8 Example of an iteration of EAF3D 14

2.9 Example of multidimensional divide-and-conquer applied to the minima problem 17

3.1 Example of three runs of a multiobjective optimizer for the same four-objective problem. 21

3.2 The projection onto (x, y)-plane of the example of Figure 3.1 given a value of z and w . . 24

3.3 EAF4D and IEAF4D performances on spherical, random and degenerate data sets 32

3.4 Example of three runs of a multiobjective optimizer for the same two-objective problem. . 34

3.5 Two-dimensional examples where input points have equal coordinates 37

3.6 More two-dimensional examples of input sets where points have equal coordinates. 42

3.7 EAF2D and OSEAF2D performances on best case, spherical and worst case data sets . . 46

3.8 EAF2DS and OSEAF2DS performance on best-case, spherical and worst case data sets . 47

3.9 Number of cache misses of EAF algorithms for $d = 2$, in best case data set where $m = 1e^6$ 48

4.1 Three-dimensional example of the contribution of a point 54

4.2 Hypervolume indicator algorithm performance for $d = 4$ on different data sets 60

4.3 Three-dimensional example for the HVDC3D algorithm 62

4.4 Three-dimensional representation of the hypervolume indicator of Figure 4.3 62

4.5 An example of the marriage step of the HVDC3D algorithm for the example of Figure 4.3 63

4.6 Simulation of the marriage steps of the HVDC3D algorithm for the example of Figure 4.3 66

4.7 Comparison of HVDC3D and HVDS3D performance. 69

List of Tables

3.1 Example of an iteration of OSEAF2D for the example in Figure 3.6(b) 42

List of Abbreviations

CDF Cumulative Distribution Function

EAF Empirical Attainment Function

EAF2D State-of-the-art algorithm to compute the EAF in two dimensions

EAF2DS Cache-efficient version of EAF2D

EAF3D Algorithm to compute the EAF in three dimensions

EAF4D Base algorithm to compute the EAF in four dimensions

ECDF Empirical CDF

EMOAs Evolutionary Multiobjective Algorithms

FPL Algorithm to compute the d-dimensional hypervolume indicator proposed by Fonseca *et al.* [12]

HOY Algorithm to compute the d-dimensional hypervolume indicator inspired in Overmars and Yap's algorithm for Klee's measure problem

HSO Hypervolume by Slicing Objectives algorithm

HV4D $O(n^2)$ time-complexity algorithm to compute the hypervolume indicator in four dimensions

HVDC3D Asymptotically optimal $O(n \log n)$ algorithm to compute hypervolume in three dimensions using a divide-and-conquer approach

HVDS3D Asymptotically optimal $O(n \log n)$ algorithm to compute hypervolume in three dimensions using a dimension sweep approach

IEAF4D Improved version of EAF4D

IHSO Incremental HSO

IIHSO Iterated IHSO

OSEAF2D Output-sensitive algorithm to compute the EAF in two dimensions

OSEAF2DS Cache-efficient version of OSEAF2D

WFG Walking Fish Group's algorithm to compute the d-dimensional hypervolume indicator

1 Introduction

The goal of optimization is to find the best solution for a problem, where the possible solutions are evaluated according to one or more objective functions. The set of all possible solutions consists of the decision space, and the corresponding evaluations of all of these solutions according to the objective functions are elements of the objective space. Therefore, given a problem with d objectives, a possible solution is seen as a point in a decision space which is mapped onto a point in the objective space. The objective space is usually considered to be $\mathbb{R}^d$. In an optimization problem, the goal is to find the solution that has the lowest value in every objective (considering minimization, without loss of generality). When the problem involves only one objective to be minimized, the best solutions are those that have the lowest possible value for that objective. Consequently, comparing two possible solutions to know which one is the best one is a trivial problem, since the best of the two is the one with the lowest objective value. For more than one objective, the situation is not trivial, since one of the two solutions may have better values for some objectives while the other has better values for other objectives.

Usually, problems with more than one objective (multiobjective problems) do not admit a single optimal solution, but a set of (Pareto-) optimal solutions. A solution is said to be Pareto-optimal [2] if there is no other solution that is at least as good as itself according to all objectives, and is strictly better according to at least one objective. The set of all Pareto optimal solutions in decision space is called the *Pareto set*, and the correspondent set of points in the objective space is called the *Pareto front* [5].

Frequently, the Pareto set is not known, and finding it is not an easy task [22]. So, the goal becomes finding a good approximation set, i.e., a set of solutions whose images in the objective space approximate the Pareto front as well as possible. This is the goal of multiobjective optimizers. Since the Pareto front is not known, it is usually not possible to know how good an approximation set is. Therefore, when comparing optimizers in order to decide which one produces better approximations, it is necessary to compare their outcomes, by comparing the corresponding sets of points in objective space. The problem that arises is that the points of a given outcome of an optimizer might not be all better or all worse than the points of the outcome of another optimizer, or even of a different run of the same optimizer, which may happen when the optimizer is stochastic.

Many quality indicators, which map an approximation set to a real value, have been proposed to compare the quality of different approximation sets [22]. Perhaps the most popular is the *Hypervolume Indicator* [21] (also called S-metric or Lebesgue Measure). It is used in comparative studies, and it has also been integrated into multiobjective optimizers, such as Evolutionary Multiobjective Algorithms (EMOAs) [6]. Hence, it is crucial that the algorithms to compute the hypervolume indicator have good

performance.

Unary quality indicators are known to have limitations. For example, they typically consider the outcome of a single run of an optimizer. In the case of Stochastic Multiobjective Optimizers, due to their randomized nature, each execution on a particular problem instance might produce a different outcome. Therefore, using the outcome of a single execution is not enough to conclude whether or not an optimizer is better than another. In this case, the *Empirical Attainment Function* (EAF) [13] can be used to study the distribution of the outcomes of several executions of a multiobjective optimizer. Current use of the EAF is still limited, since algorithms are available to compute it only in the two and three-objective cases.

The main contribution of this thesis is the proposal of new and efficient algorithms to compute the hypervolume indicator and the EAF. In the case of hypervolume indicator, a new upper bound for the four dimensional case is established with the development of a new algorithm with $O(n^2)$ time complexity, which considerably outperforms the existing algorithms. Moreover, a new algorithm using a divide-and-conquer approach for the three dimensional case is proposed. This algorithm has asymptotically optimal time complexity $O(n \log n)$, and is the first step towards the development of an algorithm for the general case, using a multidimensional divide-and-conquer approach.

Regarding the EAF, a new algorithm for the four-dimensional case is proposed, establishing new upper bounds both on the number of output points and on the time complexity of the problem. Moreover, a new alternative algorithm to compute the two-dimensional case is proposed, which outperforms the existing one. This new algorithm has the important feature of being output-sensitive, which allows it to perform better for inputs which do not result in the worst case. This is a characteristic that none of the EAF algorithms to date have.

This document is organized as follows. Chapter 2 introduces the hypervolume indicator and the Empirical Attainment Function. Existing algorithms for both problems are reviewed. Moreover, the multidimensional divide-and-conquer approach, which has not yet been applied to either hypervolume indicator or EAF computation, is described. Chapter 3, is dedicated to the description of the two new algorithms to compute the EAF. Moreover, the performance of their implementations is analyzed experimentally, and in the two dimensional case is compared to the performance of the other algorithm available. New algorithms to compute the hypervolume indicator are described in Chapter 4, and their performance is evaluated experimentally against other, state-of-the-art, algorithms. Finally, the thesis concludes with some final remarks and ideas for future work.

2 Related Work

In Chapter 1, the importance of the hypervolume indicator and of the Empirical Attainment Function (EAF) was explained. The hypervolume indicator is a widely studied quality measure for which a considerable number of algorithms are already available, but the search for more efficient algorithms is ongoing. However, there are only two algorithms to compute the EAF, one for two dimensions and another for three dimensions, which were published only very recently [11]. Most of the existing algorithms to compute the EAF and the hypervolume indicator follow the same approach, namely, the dimension-sweep approach. Therefore, the use of alternative approaches, especially those which have proved successful on related problems, is worth researching, as it may lead to new contributions in this area.

In this Chapter, the first Section will introduce the notation used, as well as some definitions needed to understand this work. Section 2.2 will be dedicated to the hypervolume indicator, which will be formally defined, and the most relevant algorithms to compute it will be briefly described. The following Section will be concerned with the definition of the EAF. The two existing algorithms to compute it will be described, but more relevance will be given to the three dimensional case. The multidimensional divide-and-conquer approach is described in Section 2.4 as a promising alternative to dimension sweep in the context of this work, and the Chapter concludes with some final remarks.

2.1 Notation and definitions

Throughout this work, the first to fourth Cartesian coordinates will be represented by the letters x, y, z and w, respectively. When referring to a given coordinate of some point p, the corresponding letter will appear in superscript. For example, p^x represents the value of the first coordinate of point p. Alternatively, but only when considering a larger or an unspecified number of dimensions, the corresponding number may appear in superscript. For example if $p \in \mathbb{R}^d$ for some value of d, then p^i denotes the value of the i^{th} coordinate of p. The letter d will always represent the number of dimensions of the d-dimensional Euclidean space, $\mathbb{R}^d$. When enumerating points or sets, the corresponding index will appear in subscript. For example, k points may be represented as $p_1, ..., p_k$.

To introduce the hypervolume indicator and the empirical attainment function, the following definitions are needed. Minimization is assumed throughout.

Definition 1 (Weak dominance). A point $x \in \mathbb{R}^d$ is said to weakly dominate a point $y \in \mathbb{R}^d$, iff $x^i \leq y^i$ holds for all $1 \leq i \leq d$. This is represented as $x \leq y$.

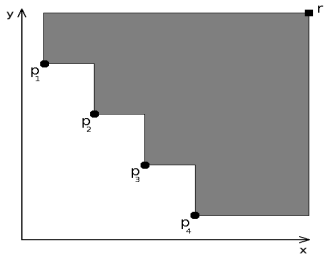


Figure 2.1: A two-objective example of the hypervolume indicator.

Definition 2 (Incomparability). Two points $x, y \in \mathbb{R}^d$ are said to be incomparable, if neither $x \leq y$ nor $y \leq x$ are true. This is represented as $x \parallel y$.

Definition 3 (Non-dominated point set). A set of points $\mathrm{X} = \{x_1, \ldots, x_m \in \mathbb{R}^d\}$ is said to be a non-dominated point set, iff $\forall_{x,y \in \mathrm{X}}, x \neq y \Rightarrow x \parallel y$

Definition 4 (Attainment). A set $\mathrm{X} = \{x_1, \ldots, x_m \in \mathbb{R}^d\}$ is said to attain a point $z \in \mathbb{R}^d$ iff $\exists x \in \mathrm{X} : x \leq z$. This is represented as $\mathrm{X} \trianglelefteq z$.

Note that the image in objective space of the set of solutions produced by a multiobjective optimizer is always a non-dominated point set. In the remainder of this work, the outcome of an optimizer will always refer to such a non-dominated point set in objective space. Moreover, the inputs used to compute the hypervolume indicator and the EAF are assumed to be non-dominated point sets.

2.2 Hypervolume indicator

The *Hypervolume Indicator* [21] is a quality indicator widely studied in the area of multiobjective optimization. It allows an easy comparison between the output of different multiobjective optimizers, by assigning a real value to a set of points. Additionally, it is a particularly interesting indicator since it reflects well a number of important quality criteria, such as the proximity of the non-dominated point set to the Pareto front, and the distribution of the non-dominated points along the Pareto front. A good multiobjective optimizer is one that produces a set of non-dominated points which are close to, and well-spread along, the Pareto front. Moreover, the hypervolume indicator has other important properties such as the ability to detect that a point set A cannot be worse than a point set B [22].

Definition 5 (Hypervolume indicator). The Hypervolume Indicator is the measure of the region dominated by a point set $\mathrm{P} \subset \mathbb{R}^d$ and bounded above by a reference point $r \in \mathbb{R}^d$ [21].

An example of the hypervolume indicator is depicted in Figure 2.1, where the region dominated by the set of points $\{p_1, ..., p_4\}$ and bounded above by the reference point r is shaded gray.

Since the computation of the hypervolume indicator has been widely studied, there is a range of algorithms proposed in the literature. There are algorithms that compute the exact hypervolume and

others, such as HypE [1], that compute an approximation. There are also algorithms to calculate the contribution of each point to the hypervolume (also called exclusive hypervolume), such as IHSO [7, 9]. The contribution, or the exclusive hypervolume, of a point p to a set of points S is the hypervolume dominated by $S \cup \{p\}$ minus the hypervolume dominated by S. Since this work is concerned only with the exact computation of the total hypervolume, the following review will be focused on algorithms for that purpose.

Currently, there are five main exact algorithms to calculate the hypervolume indicator: HSO, FPL, HOY, IIHSO and WFG. While et al. [20] proposed the Hypervolume by Slicing Objectives (HSO) algorithm, which is a dimension-sweep algorithm with time complexity $O(n^{d-1})$, where n is the number of points, and d is the number of dimensions. Assuming an input set with n input points in $\mathbb{R}^d$, the HSO algorithm starts by dividing the hypervolume of the input set into n d-dimensional slices. Considering the input points in ascending order of the first dimension, each point defines the bottom of a slice, and the next point defines the top of that slice. The hypervolume of each slice is given by multiplying the hyperarea of its $(d-1)$-dimensional base by its height.

The hyperarea of each such base is computed by sweeping along the second dimension and recursively calculating the hypervolume of the resulting $(d-1)$-dimensional slices, until a base of dimension 2 is reached. The sum of the hypervolume of all d-dimensional slices gives the exact value of the hypervolume. When $d = 2$, the hypervolume can be calculated in linear time after sorting by sweeping points in ascending order of one of the dimensions and adding the areas of the rectangles defined by each point and the next one.

Fonseca et al. [12] proposed an improved version of HSO, which has been referred to as FPL, using dedicated data structures and keeping extra information in order to avoid having to compute the whole hypervolume for k dimensions ($3 < k < d$) when possible. It also includes an $O(n \log n)$ asymptotically optimal algorithm for the three-dimensional case, which was later described in more detail by Beume et al. [5]. The FPL algorithm has a time complexity of $O(n^{d-2} \log n)$. HSO and FPL algorithms are sensitive to the order in which objectives are processed, and their performance can be improved by using heuristics to find a good objective order, as described by While et al. [19].

The HOY algorithm takes advantage of the fact that Hypervolume is a special case of Klee's Measure Problem, which is also known as the measure of the union of rectangles [14]. HOY is currently the algorithm with best time complexity for $d \geq 4$, which is $O(n^{d/2} \log n)$. It was proposed by Beume et al. [4], and is a simplification of Overmars and Yap's algorithm [16] for Klee's Measure Problem. This algorithm starts by defining the tightest axis-parallel $(d-1)$-dimensional space that includes the $(d-1)$-dimensional hyperarea of the non-dominated point set, where the last dimension is left out. Then, it recursively divides this space into smaller parts, by defining axis-parallel cut planes and keeping points associated to each subspace to which they contribute. The division stops when a portion of the initial space is reached for which the hypervolume is easy to compute. The computation is performed by sweeping points along the last dimension, d. This algorithm is not as sensitive to the order of objectives as FPL [8].

Recently, Bradstreet et al. [8] proposed a new algorithm based on IHSO [7], which is itself an algorithm based on HSO to calculate the hypervolume contribution of a point. That new algorithm is called Iterated Incremental HSO (IIHSO), and retains the time complexity of HSO, which is $O(n^{d-1})$. It iterates through each point p in the input set P, and computes its exclusive hypervolume contribution to an initially empty set S, before adding p to S. The sum of all exclusive hypervolumes is the total hypervolume of P. Also, as IHSO is sensitive to the objective order, the best order to process the objectives is determined at each step as long as five or more dimensions remain. Since this step is included in the algorithm, the final IIHSO algorithm is not sensitive to the objective order of the input set for five or more dimensions.

More recently, a new algorithm to compute the exact hypervolume indicator was proposed, known as WFG [18]. Although its worst-case complexity is stated to be exponential in the number of points ($O(2^n)$), experiments performed by While *et al.* [18] show that its runtime is good for problems with five or more dimensions. This algorithm is inspired in IIHSO in the sense that it sweeps every point p according to the last coordinate, and computes its contribution. It differs, though, in the order in which points are swept, which is now in descending order, and on how the contribuition of each point is computed. The contribution of a point p in d dimensions is computed by multiplying the absolute distance between p and r in dimension d by the exclusive contribution of p in $(d-1)$ dimensions. The contribution of p in $(d-1)$ dimensions is obtained by computing the difference between the hypervolume of the cuboid [p,r] and the hypervolume of the region dominated both by p and by any of the points not yet swept. This common hypervolume is computed recursively using WFG itself. WFG also includes the optimal $O(n \log n)$ algorithm [5] for the three dimensional base case.

While *et al.* [18] compared the performance of FPL, HOY, IIHSO and WFG. It was shown that, although HOY has the best worst-case complexity for $d \geq 4$ (when $d = 4$, FPL achieves the same time complexity as HOY), it exhibits the worst runtime performance for any number of objectives. In the four dimensional case, IIHSO obtained the best runtime. In five dimensions, IIHSO and WFG were the fastest, both with close performance. For more than five objectives, WFG had by far the best runtime, while FPL and IIHSO were slower but competitive between them, since their runtimes were close, and they outperformed one another on different data sets. To conclude, IIHSO has the best runtime for four dimensions, IIHSO and WFG are fastest for the five-dimensional case, and for more dimensions the WFG algorithm has the best performance. In spite of HOY having the best complexity, it performs worst in practice, as seen in [8] and in [18].

2.3 Empirical attainment function

2.3.1 Definition

The notion of *Attainment Function* was first formally introduced by Grunert da Fonseca et al. [13]. It studies the distribution of a random non-dominated point set by providing information about the probability of a given point being weakly dominated, i.e. *attained*, by such a random set. The *Empirical*

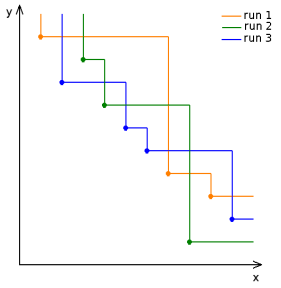


Figure 2.2: Example of three runs of a multiobjective optimizer on the same two-objective problem.

Attainment Function (EAF) estimates the *attainment function* using experimental data, obtained as the outcomes of independent runs of a multiobjective optimizer. Figure 2.2 shows an example of three different outcomes of such an optimizer on a given two-objective problem. This figure will be used as the base example for the explanations that follow.

As detailed in Fonseca et al. [11], the outcome sets of a multiobjective optimizer applied to the same problem are seen as realizations of a random non-dominated point set (RNP set). The *empirical attainment function* is defined as:

Definition 6 (Empirical attainment function). Let $\mathbf{I}\{\cdot\}\colon \mathbb{R}^d \longmapsto \{0,1\}$ denote the indicator function, and let $\mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{X}_n$ be non-dominated point set realizations drawn independently from some RNP set distribution. The empirical attainment function (EAF) is the discrete function $\alpha_n\colon \mathbb{R}^d \longmapsto [0,1]$, where

$$\alpha_n(z) = \alpha_n(\mathrm{X}_1, \ldots, \mathrm{X}_n; z) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{I}\{\mathrm{X}_i \trianglelefteq z\}$$

Recall that $\mathrm{X} \trianglelefteq z$ denotes the attainment of a point z by a set X, i.e., there is at least one $x \in \mathrm{X}$ such that $x \leq z$.

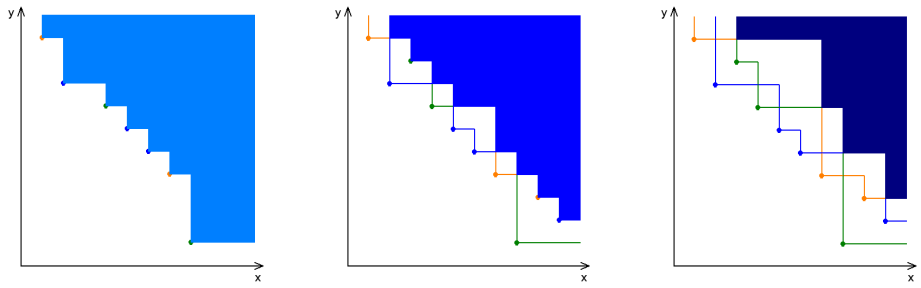
To formalize the empirical attainment function computation problem, the notion of *Minima* of a point set is useful.

Definition 7 (Minima). Given a set of points $\mathrm{X} = \{x_1, \ldots, x_m \in \mathbb{R}^d\}$, the set of minima of X under the component-wise order is the set

$$\min \mathrm{X} = \{x \in \mathrm{X} : \forall y \in \mathrm{X}, y \leq x \Rightarrow y = x\} \tag{2.1}$$

The minima of a point set is a non-dominated point set, which means that a set X is a non-dominated point set iff $\mathrm{X} = \min \mathrm{X}$.

The idea of studying the distribution of multiobjective optimizer outcomes based on the attainment function builds upon the notion of *attainment surfaces*. An (empirical) attainment surface was originally defined as the tightest set of goals attained in one optimization run [10]. Then it was extended to consider



(a) (b) (c)

Figure 2.3: The colored regions in (a), (b) and (c) represents the superlevel sets $V_{1/3}$, $V_{2/3}$ and $V_{3/3}$ for the example in Figure 2.2, respectively.

multiple runs, so that any point dominated by a given attainment surface has a higher probability of being attained than a point not dominated by that attainment surface. More accurately, attainment surfaces can be seen as the lower boundary of the EAF superlevel sets. A t/n-superlevel set of $\alpha_n(z)$ is defined as the region of the objective space where the probability of a point z being attained is at least t/n. Figure 2.3 shows an example of the t/n-superlevel sets of the EAF associated with the data represented in Figure 2.2. The t/n-superlevel set of an empirical attainment surface $\alpha_n(z)$ is formally defined as:

$$V_{t/n} = \{z \in \mathbb{R}^d : \alpha_n(z) \geq t/n\} \quad t = 1, \ldots, n \tag{2.2}$$

Considering the set of minima of $V_{t/n}$, which will be denoted as $L_t = \min V_{t/n}$, it follows that $V_{t/n}$ is also equal to the upper set of L_t, i.e.,

$$V_{t/n} \quad = \quad \{z \in \mathbb{R}^d : L_t \preceq z\} \tag{2.3}$$

because $\alpha_n(z)$ is coordinatewise monotonic.

With these definitions, the EAF computation problem is formalized as [11]:

Problem 1 *(EAF computation).* Given an input sequence of non-empty non-dominated point sets:

$$S = (X_1, X_2, \ldots, X_n) \tag{2.4}$$

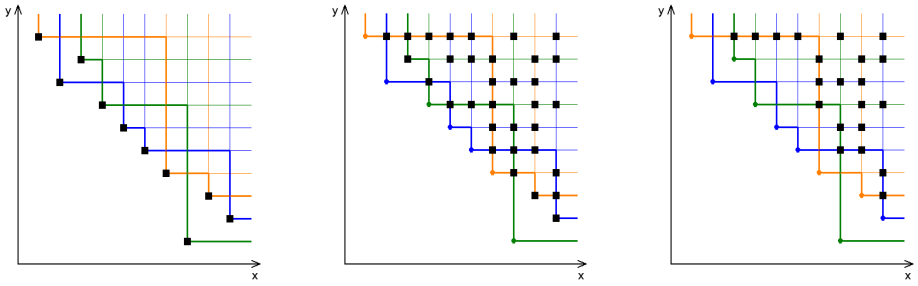
containing

$$m = \sum_{i=1}^{n} m_i, \quad m_i = |X_i| \tag{2.5}$$

input points, find the output sequence

$$R = (L_1, L_2, \ldots, L_n) \tag{2.6}$$

where L_t, $t = 1, \ldots, n$, denotes the set of minima of the t/n-superlevel set, $V_{t/n}$, of $\alpha_n(X_1, \ldots, X_n; z)$.



(a) (b) (c)

Figure 2.4: Squares in (a), (b) and (c) represent the points that define J_1, J_2 and J_3, respectively, for the example in Figure 2.2.

The total number of output points is

$$\ell = \sum_{t=1}^{n} \ell_t, \quad \ell_t = |\mathrm{L}_t| \tag{2.7}$$

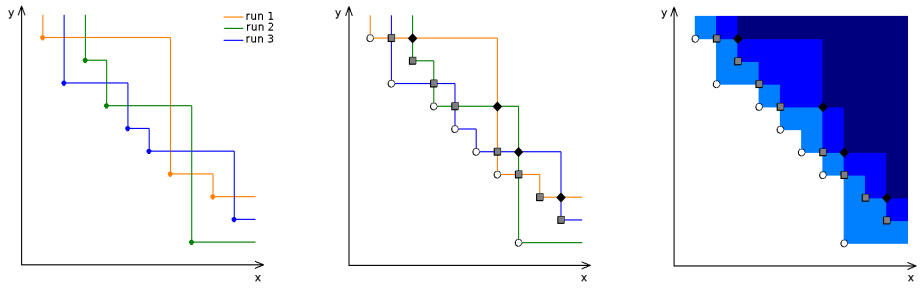
As shown by Fonseca et al. [11], the output sets L_t may be computed by finding the minima of suitable auxiliary sets J_t ($\mathrm{L}_t = \min \mathrm{J}_t$), defined as:

$$\mathrm{J}_t = \left\{ \bigvee_{i=1}^{t} z_i : (z_1, \ldots, z_t) \in \prod_{i=1}^{t} \mathrm{X}_{j_i},\ j_i \in \{1, \ldots, n\} \wedge (a < b \Leftrightarrow j_a < j_b) \right\} \tag{2.8}$$

where $\bigvee_{i=1}^{t} z_i$ denotes the *join* (component-wise maximum, or least upper bound) of points $z_1, \ldots, z_t \in \mathbb{R}^d$, $\prod_{i=1}^{t} \mathrm{X}_{j_i}$ denotes the Cartesian product of sets $\mathrm{X}_{j_1}, \ldots, \mathrm{X}_{j_t}$, and $(\mathrm{X}_{j_1}, \ldots, \mathrm{X}_{j_t})$ is any length-t subsequence of S. J_t contains all points resulting from the join of t points, each from a distinct input set, component of S. This can be visualized in Figure 2.4. It can be seen that J_1 (Figure 2.4(a)) is the union of the outcomes of all runs. J_2 (Figure 2.4(b)) and J_3 (Figure 2.4(c)) contain all points that are the join of two and three points from distinct input sets, respectively.

Figure 2.5 shows the result of computing the EAF. Figure 2.5(a) shows the initial example that was previously shown in Figure 2.2. Figures 2.5(b) and 2.5(c) show the points in L_1, L_2 and L_3. These are the points that should be returned by an algorithm that solves the EAF problem. In Figure 2.5(b), it can be seen that each set of points L_t, where $1 \leq t \leq 3$, is the set of minima of J_t, previously represented in Figure 2.4. In Figure 2.5(c), it is possible to observe that each L_t dominates the corresponding t/n-superlevel ($\mathrm{V}_{t/n}$), previously shown in Figure 2.3.

Observe that L_t can be obtained by finding the minima of J_t ($\mathrm{L}_t = \min \mathrm{J}_t$). However, the two algorithms for the EAF computation problem in two and three dimensions proposed by Fonseca et al. [11] do not enumerate the whole sets J_t in order to obtain L_t. These two algorithms are, so far, the only algorithms available to solve the EAF computation problem, with time complexities of $O(m \log m + nm)$ and $O(n^2 m \log m)$ for the two and three-dimensional cases problem, respectively. Algorithms for $d > 3$



(a) (b) (c)

Figure 2.5: (a) Example of three runs of an optimizer on the same problem. (b) Circles, squares and diamonds represent the points in L_1, L_2 and L_3, respectively. (c) The regions dominated by L_1, L_2 and L_3 are shown.

have not been reported so far. In spite of this, a lower bound on the time complexity for any algorithm that solves the d-dimensional EAF problem is known to be $\Omega\left(m \log m + m^{\lfloor d/2 \rfloor}\right)$ [11]. Upper bounds on the total number of output points are given by Fonseca et al. [11], being $\Theta(nm)$ and $O(n^2 m)$ for the two and three dimensional cases, respectively. In the d-dimensional case, the maximum total number of output points is $\Omega\left(m^{\lfloor d/2 \rfloor}\right)$.

2.3.2 Algorithms

In this Subsection, the existing algorithms to compute EAF will be explained in order to support the development and presentation of new EAF algorithms in Chapter 3.

Two dimensional case

Fonseca *et al.* [11] proposed an $O(m \log m + nm)$ time algorithm to compute the EAF in two dimensions. This algorithm is characterized by determining each L_t $(t = 1, ..., n)$ at a time. To achieve this, two queues, Q^y sorted in ascending order of coordinate y and Q^x sorted in descending order of coordinate x, are used. For simplicity, assume that all input points have distinct coordinates.

When a given L_t is to be computed, the algorithm sweeps points in ascending order of coordinate y by popping from Q^y until points from t different input sets have been swept. Denoting the last point swept by p, it is known that there is a point in L_t whose y coordinate is p^y. In order to determine the value of the x coordinate of that point, points are swept in descending order of x by popping from Q^x until a point q is found such that $q^y \leq p^y$ and that points from $t-1$ different input sets (other than the one to which q belongs to) are below p^y and strictly below q^x (according to coordinate y and x, respectively). Hence, (q^x, p^y) is a point in L_t. Finding q can be done by remembering how many points in each of the t input sets were seen up to p^y, and keeping track of how many of these points are to the left of the point from Q^x currently being swept. When all such points from one input set are swept, then q is found. Keeping track of which input set are below (q^x, p^y), the next points in Q^y are popped until a point from an input set different from the $t-1$ sets strictly below q^x is found (point p). Then, the

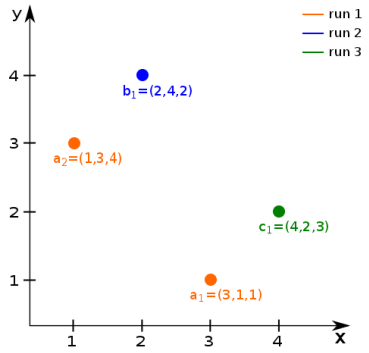


Figure 2.6: Example of three runs of a multiobjective optimizer for the same three-objective problem.

algorithm continues by popping from Q^x until it finds another point q such that (q^x, p^y) is a point of L_t, and so on. This is repeated for every L_t that is to be computed.

Three dimensional case

The algorithm proposed by Fonseca *et al.* [11] to compute the EAF in three dimensions has an important role in the algorithm for the four-dimensional case that will be presented later in this thesis. Since it is important to fully understand the algorithm for the three dimensional case, it will be described here in detail. It is a dimension sweep algorithm with $O(n^2 m \log m)$ time complexity, and will be referred to as EAF3D for short. A general idea of which points are sought will be given next, followed by a detailed description of how that task is achieved by the algorithm.

Throughout this description, it will be considered that all input points have distinct values of coordinate z. This condition will be lifted later, and an explanation of how the algorithm behaves when repeated values of coordinate z are considered will be given. Figure 2.6 is one of the figures that will be used as an example to help explain the algorithm. The expected result for this specific example is:

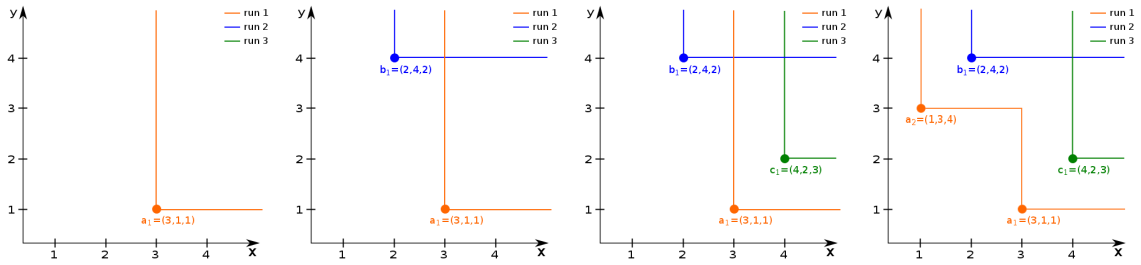
$$L_1 = \{(3,1,1), (2,4,2), (1,3,4)\}$$

$$L_2 = \{(3,4,2), (4,2,3), (2,4,4)\}$$

$$L_3 = \{(4,4,3)\}$$

The EAF3D algorithm is detailed in Algorithm 1. As it is a dimension sweep algorithm, it sweeps all points in ascending order of the third coordinate (the outer loop in line 13) and, for each point being swept (let us call it p), it computes an EAF in two dimensions, according to the projection on the (x,y)-plane of the points swept so far. Figure 2.7 depicts all iterations of EAF3D for the initial problem given in Figure 2.6.

At each iteration, Algorithm 1 determines on the (x,y)-plane the minimal points that were attained by only $t-1$ input sets in the previous iteration, not including the set to which p belongs, and which are attained also by p in the current iteration, i.e., by one more input set. These are points that belong to L_t in three dimensions and whose coordinate z is equal to p^z.



(a) $z = 1$ (b) $z = 2$ (c) $z = 3$ (d) $z = 4$

Figure 2.7: The projection of the sets of points of the example in Figure 2.6 onto the (x, y)-plane for a given value of coordinate z.

Looking at the example of Figure 2.6, since point a_1 is the first point to be swept, its projection on the (x,y)-plane is a point in L_1^* (note that $(\mathrm{L}_1^*, ..., \mathrm{L}_n^*)$ denotes the EAF in two dimensions given the projections on the (x,y)-plane of the points swept so far) in the current iteration (Figure 2.7(a)). Therefore, point a_1 itself is a point in L_1 in three dimensions. When point b_1 is swept, $(2, 4)$ and $(3, 4)$ are minimal points attained by one and two inputs sets, respectively (Figure 2.7(b)). Note that $(3, 1)$ is attained by the same number of input sets as in the previous iteration. Therefore, these points belong to L_1 and L_2 in three dimensions, respectively, with coordinate z equal to b_1^z, i.e., $(2, 4, 2) \in \mathrm{L}_1$ and $(3, 4, 2) \in \mathrm{L}_2$. When sweeping point c_1, the output points found are $(4, 2, 3) \in \mathrm{L}_2$ and $(4, 4, 3) \in \mathrm{L}_3$, and when sweeping point a_2, the points $(1, 3, 4) \in \mathrm{L}_1$ and $(2, 4, 4) \in \mathrm{L}_2$ are found.

These points can be found by determining the points in each L_t^* which, in each iteration, are not attained by L_t^* in the previous iteration. Therefore, for each new point p being swept, Algorithm 1 has stored in X_t^* and L_t^*, respectively, the minima of the projections onto the (x, y)-plane of the points swept so far (not including p) in each X_t and in each L_t determined in the previous iteration, where $t = 1, ..., n$. For example, when point c_1 is selected to be swept, $\mathrm{X}_1^* = \{(3, 1)\}$, $\mathrm{X}_2^* = \{(2, 4)\}$, $\mathrm{X}_3^* = \{\}$, $\mathrm{L}_1^* = \{(3, 1), (2, 4)\}$, $\mathrm{L}_2^* = \{(3, 4)\}$, $\mathrm{L}_3^* = \{\}$. L_t^* is updated as new points are found in the current iteration. After that, X_j^* is updated to include p, since $p \in \mathrm{X}_j$. Both X_t^* and L_t^* are maintained using height-balanced binary search trees that store only non-dominated points sets on the (x,y)-plane. Thus, it is possible to search them either by x or by y coordinate. The following operations are available for these data structures [11]:

floor$^x(p, \mathrm{X}^*)$ The point $q \in \mathrm{X}^*$ with the greatest $q^x \leq p^x$

lower$^x(p, \mathrm{X}^*)$ The point $q \in \mathrm{X}^*$ with the greatest $q^x < p^x$

ceiling$^x(p, \mathrm{X}^*)$ The point $q \in \mathrm{X}^*$ with the least $q^x \geq p^x$

higher$^x(p, \mathrm{X}^*)$ The point $q \in \mathrm{X}^*$ with the least $q^x > p^x$

The operations are performed in $O(\log m)$ time, and are available also for searching by y. Note that, since both X_t^* and L_t^* are initialized with sentinels, these operations always return a point. Note also that, in spite of considering only the projections onto the (x,y)-plane of the points in L_t^*, the points added to L_t^*

Algorithm 1 EAF computation in three dimensions

Input: $\mathrm{S} = (\mathrm{X}_1, \ldots, \mathrm{X}_n)$ // a sequence of non-dominated point sets
Output: $\mathrm{R} = (\mathrm{L}_1, \ldots, \mathrm{L}_n)$ // a sequence of non-dominated point sets

$\mathrm{X} = \biguplus_{i=1}^{n} \mathrm{X}_i$ // multiset sum, duplicate points are allowed
$m \leftarrow \sum_{i=1}^{n} |\mathrm{X}_i|$
Q is X sorted in *ascending* order of the z coordinate
$\mathrm{L}_t \leftarrow \varnothing$, $t = 1, \ldots, n$
$\mathrm{L}_t^* \leftarrow \{(-\infty, \infty, -\infty), (\infty, -\infty, -\infty)\}$, $t = 1, \ldots, n$ // Sentinels
$\mathrm{X}_i^* \leftarrow \{(-\infty, \infty, -\infty), (\infty, -\infty, -\infty)\}$, $i = 1, \ldots, n$ // Sentinels
$p \leftarrow \mathsf{pop}(\mathrm{Q})$
$j \leftarrow \mathsf{input\_set}(p)$
$\mathsf{insert}(p, \mathrm{X}_j^*)$
$\mathsf{insert}(p, \mathrm{L}_1^*)$
$\mathrm{A} \leftarrow \{j\}$
$t_{\max} \leftarrow 1$
while $\mathrm{Q} \neq \varnothing$ **do**
  $p \leftarrow \mathsf{pop}(\mathrm{Q})$
  $j \leftarrow \mathsf{input\_set}(p)$
  $q \leftarrow \mathsf{floor}^x(p, \mathrm{X}_j^*)$
  if $p^y < q^y$ **then** // always true if X_j is indeed a non-dominated point set
    $t \leftarrow t_{\max}$
    $t_{\min} \leftarrow 1$
    first intersection // part 1
    dominated points // part 2
    last intersection // part 3
    submit p to X_j^*
    submit p to $\mathrm{L}_{t_{\min}}^*$
  if $j \notin \mathrm{A}$ **then**
    $\mathrm{A} \leftarrow \mathrm{A} \cup \{j\}$
    $t_{\max} \leftarrow \min(t_{\max} + 1, n - 1)$
$\mathrm{L}_t \leftarrow \mathrm{L}_t \cup (\mathrm{L}_t^* \setminus \{(-\infty, \infty, -\infty), (\infty, -\infty, -\infty)\})$, $t = 1, \ldots, n$
return $(\mathrm{L}_1, \ldots, \mathrm{L}_n)$

Algorithm 2 first intersection

// $q = \mathsf{floor}^x(p, \mathrm{X}_j^*)$
while $t \geq t_{\min}$ **do**
  $r \leftarrow \mathsf{floor}^x(p, \mathrm{L}_t^*)$
  if $r^y \leq p^y$ **then**
    $t_{\min} \leftarrow t + 1$
  else if $r^y < q^y$ **then**
    $s_t \leftarrow (p^x, r^y, p^z)$
  else
    $s_t \leftarrow \mathsf{lower}^y(q, \mathrm{L}_t^*)$
  $t \leftarrow t - 1$

do have their third coordinate value set. This allows them to be added to the corresponding output set L_t at a later time.

To summarize, Algorithm 1 sweeps each input point p in ascending order of coordinate z (independently of the input set to which it belongs) by popping points from Q, while maintaining X_t^* and L_t^* updated according to the current cut plane defined by the z coordinate of the point being swept.

Having understood which points are sought, the issue of how to actually find them arises. The search for these points is made by sections, i.e., by non-overlapping subregions of the region attained by the

Algorithm 3 dominated points

```
1: // s_t^x ≥ p^x, q = floor^x(p, X_j^*)
2: repeat
3:    q ← higher^x(q, X_j^*)
4:    b ← max(p^y, q^y)
5:    for t = t_max down to t_min do
6:       while s_t^y ≥ b and (s_t^y > b or b > p^y) do
7:          if s_t^x ≥ q^x then
8:             s_t ← lower^y(q, L_t^*)
9:          else
10:            submit (s_t^x, s_t^y, p^z) to L_{t+1}^*
11:            s_t ← higher^x(s_t, L_t^*)
12: until q^y ≤ p^y
```

Algorithm 4 last intersection

```
1: // s_t = floor^y(p, L_t^*)
2: for t = t_max down to t_min do
3:    if s_t^x < q^x then
4:       submit (s_t^x, p^y, p^z) to L_{t+1}^*
```

Algorithm 5 submit u to L_t^*

```
1: v ← floor^x(u, L_t^*)
2: if u^y < v^y then
3:    for all w ∈ L_t^* : (u^x, u^y) ≤ (w^x, w^y) do
4:       if u^z > w^z then
5:          L_t ← L_t ∪ {w}
6:       remove(w, L_t^*)
7:    insert(u, L_t^*)
```

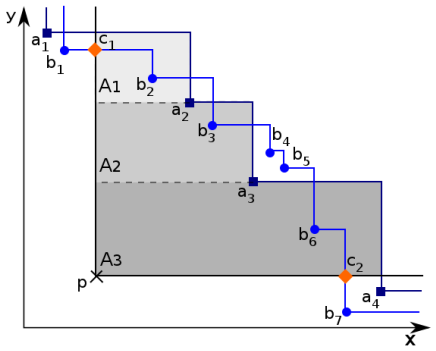


Figure 2.8: Example of a given iteration, where p is the point being swept. Points $a_1, ..., a_4$ represent the points from the same set as p. Points $b_1, ..., b_7$ are points from some L_t. $A_1 \cup A_2 \cup A_3$ represents the region dominated by p and not by any other point from its set.

current point swept and not attained by any other point already swept in its set. An example can be seen in Figure 2.8, where $A_1, ..., A_3$ represent these subregions defined by p and by the points from its set. These subregions are swept in descending order of y, and for each subregion, each L_k^* $(k = n-1, ..., 1)$ is swept in order to determine the minimal points that are attained by L_k^* and by p, which are not attained by another point from p's set. Therefore, when sweeping L_k^*, new points might be added to L_{k+1}^*.

Hence, there are four main cases to consider while sweeping p, given X_j^* ($p \in \mathrm{X}_j$ and $p \notin \mathrm{X}_j^*$) and L_t^*

$(t = n-1, ..., 1)$, at coordinate z equal to p^z:

first intersection (Algorithm 2) Assume that r is the point in L_t^* with lowest coordinate y that satisfies the following conditions: $r^x \leq p^x$, $r^y > p^y$ and $X_j^* \not\trianglelefteq (p^x, r^y)$. If such point r exists then (p^x, r^y) is attained by one more input set (p's set) starting from height p^z, and therefore (p^x, r^y, p^z) is a point of L_{t+1}. r^* corresponds to point b_1 in Figure 2.8, and (p^x, r^y) corresponds to point c_1.

dominated points (Algorithm 3) For every point $s \in L_t^*$ such that conditions $p^* < s^*$ and $X_j^* \not\trianglelefteq s^*$ are true, (s^x, s^y, p^z) is a point in L_{t+1} because at height $z = p^z$, s^* is attained by one more input set (p's set) then it was in the previous iteration. This is the case of points b_2, b_3 and b_6 in Figure 2.8.

last intersection (Algorithm 4) This case is similar to the first case (first intersection), but here x and y have their roles reversed. This is the case of point b_7, which corresponds to point r^*, and (r^x, p^y), which corresponds to c_2 in Figure 2.8.

p is itself a point in $L_{t_{min}}$, where t_{min} is the lowest index t such that $L_{t_{min}}^*$ does not attain p (line 24). Looking back at Figures 2.7(a) to 2.7(d), when a_1, b_1, c_1 and a_2 are swept, t_{min} is 1, 1, 2 and 1, respectively.

Note that the first three cases are only considered for $t \geq t_{min}$. Since the algorithm determines all points in each L_t by regions delimited in y by two consecutive points of X_j^* (as depicted in Figure 2.8), points that satisfy the first case (first intersection) are added to L_{t+1}^* later in the second phase (dominated points). This is just to avoid adding points to L_{t+1}^* that would wrongly have influence later on the dominated points phase. Algorithm 1 uses t_{max} and A to skip L_t^* which are still empty.

When the restriction of not having points with equal z coordinate is lifted, some of the assumptions made earlier are no longer true in all cases. For example, when there are no input points with equal z coordinate, it is sure that, when a point is submitted to some L_t^*, it is definitely a point belonging to L_t in three dimensions because any point added later to L_t^* that dominates it on their projection on to the (x,y)-plane, will have a higher third coordinate. But this is not guaranteed to happen when there are input points with equal z coordinates. For example, if the input points of the following example, $X_1 = \{(1,1,1)\}$, $X_2 = \{(2,2,2)\}$ and $X_3 = \{(3,3,2)\}$, are swept in the following order $(1,1,1)$, $(3,3,2)$, $(2,2,2)$, then when $(3,3,2)$ is swept, it is added to L_2^*, but it is not a minimal point of L_2 because it is dominated by $(2,2,2)$ that will be added to L_2^* later. To avoid adding wrong points to L_t, the decision that a point is definitely in L_t is delayed until it becomes clear that it is true. That decision is made only if that point is only dominated in its projection by a point with higher coordinate z (line 4 of Algorithm 5) or because there are no more points that invalidate it (line 28 of Algorithm 1). Note that points are submitted to L_t^* using Submit routine, and so are points submitted to X_j^*. However, it is considered that lines 4 and 5 of Submit do not apply to the latter case

Note that, although the problem with equal values of coordinates x and/or y was not mentioned yet, it is present, and is also solved in Submit (Algorithm 5). What can happen when there are input points with equal coordinates x and y is to have points that the algorithm finds to be points of a given

L_t, but which in fact are not because they were already attained by t input sets in previous iterations. For example, consider the initial problem $X_1 = \{1, 1, 3\}$, $X_2 = \{(2, 2, 1)\}$ and $X_3 = \{(2, 2, 2)\}$. Where sweeping point $(1, 1, 3)$, when Algorithm 3 is executed it claims that point $(2, 2)$ should be added to L_2^* and L_3^*, but point $(2, 2)$ is already in L_2^*. Therefore, in this current iteration, $(2, 2)$ is added to L_3^* and is discarded when trying to add it to L_2^* (the condition in line 2 of Algorithm 5 is not verified).

In general terms, if input points have no equal coordinates, then it is possible to admit that when sweeping two consecutive values of coordinate x, y or z (and fixing the other two coordinates), the number of input sets that attain them differs, by one at most. For example, in Figure 2.6, point $(2, 4)$ in $z = 3$ is attained by 1 input set and in $z = 4$ it is attained by one more. The problem of having input points with equal coordinates is that this is no longer true in all cases. For example, considering $X_1 = \{1, 1, 1\}$, $X_2 = \{(2, 2, 2)\}$ and $X_3 = \{(3, 3, 2)\}$, then when $z = 1$, point $(3, 3)$ is attained by only one input set and in $z = 2$ it is attained by 3 input sets. This is an issue that concerns in all EAF algorithms, the ones currently available and the new ones that will be proposed later. Because initial algorithm development often relies on this assumption, it can sometimes be difficult to adapt the initial algorithm to handle input points with repeated coordinates correctly. Note that in EAF3D, if input points do not have equal coordinates, then any point submitted to some L_t^* would always be minimal, because when compared to any point already in L_t^* it would have at least one coordinate that would be better.

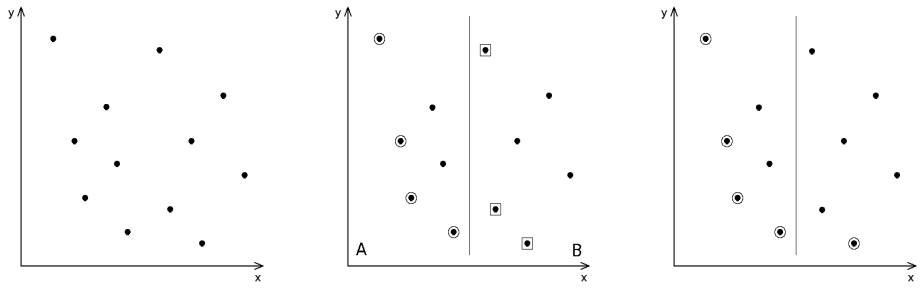
2.4 Multidimensional divide-and-conquer

Dimension Sweep is an approach that is used frequently, and that appears naturally in computational geometry [17]. As it was discussed before, it is characterized by defining a set of points to be swept in ascending, or descending, order of a specific dimension. For each point being visited, an easier subproblem is solved. Examples of the use of this approach include the computation of the hypervolume indicator and of the EAF, as seen in the previous Sections. Considering a problem with d dimensions, the input points are swept in one specific dimension, and a $(d-1)$-dimensional subproblem is solved for each point visited.

Although the dimension-sweep approach has led to the efficient solution of many computational geometry problems, this has happened mainly in low dimensions, typically $d = 2, 3$. An alternative approach which has been useful in addressing higher-dimensional problems is known as *Multidimensional Divide-and-Conquer* [3].

The Multidimensional Divide-and-Conquer paradigm is briefly described by Bentley [3] as follows: *To solve a problem of N points in d-space, first recursively solve two problems each of $N/2$ points in d-space, and then recursively solve one problem of N points in $(d-1)$-dimensional space.* Therefore, the idea is to divide the main problem into several subproblems. These subproblems are then solved in three steps:

1. **Division step**, divide the problem involving N points into two problems involving $N/2$ points.
2. **Recursive step**, solve each of the $N/2$-point problems recursively in d-space.



(a) (b) (c)

Figure 2.9: (a) A set of $N = 12$ points. (b) The set of points is divided into two sets, A and B, containing $N/2 = 6$ points each. Points in the sets of minima of A and B are marked with circles and squares, respectively. (c) Points in the set of minima of the original problem are marked with circles.

3. **Marriage step**, join the solutions of the two $N/2$-point problems, by recursively solving a problem involving N points in $(d-1)$-space.

The recursive step is a recursion on the size of the set, while the marriage step is a recursion on the number of dimensions. Each of these two steps has a stopping criterion. The recursive step stops when a small number of points is reached, usually $N = 1$, while the marriage step usually stops when d equals 2 or 3, where a more or less straightforward algorithm is used to solve the problem for that number of dimensions. The marriage step is seen as a problem in $(d-1)$-space by taking advantage of the fact that the d-coordinate values of all points of one of the subproblems (let us call this set of points A) are lower than the values of all points in the other subproblem (lets call it B). In this step, the solution of one of the subsets (A or B) is correct, while the correctness of the solution for the other subset (B or A) depends on the points in the first subset (A or B). Which solution is correct (for subset A or B), depends on the problem being solved.

Multidimensional divide-and-conquer can be applied, for example, to the *Maxima* problem [3]. Given a set of points P $\subset \mathbb{R}^d$, the Maxima problem is defined, in maximization, as the problem of finding the points in P that are not weakly dominated by any other point in P, in a maximization sense. Since in this thesis minimization is assumed, the analogous problem in minimization will be explained instead, i.e., the *Minima* problem (find the minima set). The application of the multidimensional divide-and-conquer can be seen in the example which is explained next.

Consider the example in Figure 2.9. A set of $N = 12$ points represented in Figure 2.9(a), is divided into two subsets, A and B, containing $N/2 = 6$ points each, as represented in Figure 2.9(b). This corresponds to the division step. The recursive step consists of finding the set of minima of each of the subsets, A and B. In Figure 2.9(b), the set of minima of A and B are identified by circles and squares, respectively. Since all points in A have lower value of coordinate x than all points in B, the points in A found to be part of the set of minima also belong to set of minima of the original problem. Not all points found to be part of the set of minima of B are minima of the original set, since some of them are weakly

dominated by at least one point in A.

Next, the marriage step is performed. Since all points in A have lower x coordinate values than the points in B, only the points in B that have lower y coordinate values than the points in A belong to the set of minima of the original problem. Therefore, the marriage step consists of sweeping through each point of the initial problem, in ascending order of the y coordinate, which becomes a problem in $d-1=1$ dimension. All points in B that are swept before the point in A with the lowest value of the y coordinate remain marked as being part of the set of minima, while the others are unmarked. Note that each point of the initial problem is labeled as being in A or in B. Figure 2.9(c) shows the points that belong to the set of minima of the original problem. This method can also be generalized to d dimensions.

The marriage step is the most crucial step when using this approach to solve a problem, since it is necessary to figure out how to efficiently join the solutions for the two subproblems. Preparata et al. [17] apply multidimensional divide-and-conquer to the marriage step itself, dividing it into three subproblems, two problems in $(d-1)$-space with a smaller number of points and a problem in $(d-2)$-space.

The multidimensional divide-and conquer approach was already used for problems related to the ones discussed here, such as the Maxima problem (already described but from a minimization point of view) and the *ECDF* (it will be described further ahead), for which this approach reached better results than dimension-sweep. Actually, the algorithm to compute the hypervolume indicator in three dimensions [5], which was proved to be asymptotically optimal [5], is based on the dimension sweep algorithm for the Maxima problem [15]. The minimization version of the Maxima problem can also be related to the EAF problem, as the points that compose L_t are the minima of J_t, as explained in Section 2.3.1. Preparata and Shamos [17] compared the two approaches in the resolution of the Maxima problem. In four dimensions, the dimension sweep approach has time complexity $O(N^2)$, whereas the multidimensional divide-and-conquer approach achieves a time complexity of $O(N \log^2 N)$. Subsequent generalization to d dimensions led to $O(N \log^{d-2} N)$ time complexity being obtained.

The univariate CDF (Cumulative Distribution Function) problem is defined in statistics as the probability of a random variable taking a value lower than or equal to some given value. The multivariate CDF generalizes it to a vector of random variables, and is defined as the probability of each of the random variables X^i taking values less than or equal to some given value x^i, where $i \geq 1$. The ECDF (Empirical CDF) is an estimator for the CDF. Given a point $u \in \mathbb{R}^d$ and a set of points $\mathrm{P} \subset \mathbb{R}^d$, where each dimension corresponds to a random variable and each point in P corresponds to an experiment, the ECDF search problem consists of finding the probability of each random variable i, according to P, taking a value lower or equal to u^i, simultaneously [3]. Consequently, the multivariate ECDF is related to the EAF problem, since the probability of each random variable i, according to P, taking a value lower or equal to u^i can be seen as the probability of u being attained by P. However, note that the EAF computation problem is not restricted to computing the EAF at the input points only. Still, given that $|\mathrm{P}| = n$, the multivariate ECDF can be seen as a special case of the EAF [13], where $|m_i| = 1$, for $i = 1, \ldots, n$ and where n denotes the number of sets.

Bentley [3] described the Multidimensional Divide-and-Conquer approach and applied it to a range of geometry problems, where the ECDF and Maxima problems are classified as *Domination Problems*. The time complexity obtained for the Maxima problem was $O(N \log^{d-2} N)$, and for the ECDF search problem was $O(N \log^{d-1} N)$. Knowing that the multidimensional divide-and-conquer approach was successfully applied to problems related to the ones discussed here, it is worth investigating to what extent this paradigm may be applied to the computation of the hypervolume indicator and/or the EAF.

2.5 Concluding remarks

Hypervolume indicator is a widely studied quality measure, but the time complexity and the performance of existing algorithms can still be improved, as it will be shown in Section 4.1. Moreover, given the good results obtained by applying the multidimensional divide-and-conquer approach to similar problems, using this approach to compute the hypervolume would seem promising. This will also be discussed in Section 4.2.

The applicability of the Empirical Attainment Function is currently restricted, since algorithms to compute it are available only for 2 and 3 dimensions. In the next Chapter, a new algorithm for the four-dimensional case is proposed, and the computation of the EAF in two dimensions is revisited.

3 Empirical Attainment Function

This Chapter is dedicated to the development of new algorithms for the computation of the EAF in two and four dimensions. In Section 2.3 it was seen that, so far, algorithms are available for the computation of the EAF in two and three dimensions only, having $O(m \log m + mn)$ and $O(n^2 m \log m)$ time complexity, respectively. In Section 3.1, two new algorithms for the four-dimensional case are described, a simple one and an improved one, both with $O(n^2 m^2 \log m)$ time complexity. In Section 3.2 a new algorithm is proposed which improves upon the current one by being output-sensitive, i.e., its complexity depends not only on the size of the input, but also on the size of the output. The contributions presented are reviewed and discussed in the last Section.

3.1 EAF in four dimensions

This Section is dedicated to the description of new algorithms to compute the EAF in four dimensions. Subsection 3.1.1 describes a base algorithm, which is improved in Subsection 3.1.2. Their complexity will be analyzed in Subsection 3.1.3 and, finally, their performance will be experimentally assessed.

3.1.1 Base algorithm

The EAF computation problem in 4 dimensions can be approached using a dimension-sweep strategy, which leads to solving m EAF computation problems in three dimensions. Hence, the final complexity will be $O(m^2 n^2 \log m)$. In the following, it will be explained how to look at the problem in a way that leads to the dimension sweep algorithm.

Figure 3.1 will be used as an example to explain the idea. It shows the projection on the (x, y)-plane

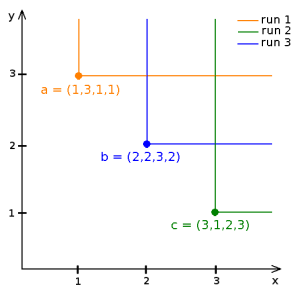


Figure 3.1: Example of three runs of a multiobjective optimizer for the same four-objective problem.

of three points, $a = (1, 3, 1, 1)$, $b = (2, 2, 3, 2)$ and $c = (3, 1, 2, 3)$, each of them belonging to a distinct outcome set. Given these sets of points, the expected output of an algorithm that computes the EAF in four dimensions is:

$$L_1 = \{(1, 3, 1, 1), (2, 2, 3, 2), (3, 1, 2, 3)\}$$

$$L_2 = \{(2, 3, 3, 2), (3, 3, 2, 3), (3, 2, 3, 3)\}$$

$$L_3 = \{(3, 3, 3, 3)\}$$

Since all input points have distinct coordinate values, the value of the forth coordinate of each point identified as belonging to L_t ($1 \leq t \leq n$) must correspond to one, and only one, point from one of the input sets. Let that input point be called p. This can be seen in the expected output for this example, since every point in L_1, L_2 and L_3 has the 4^{th} coordinate equal to either a^w, b^w or c^w. Any point q in L_t that has the 4^{th} coordinate equal to p^w is the result of the *join* between p and $t-1$ points (if $t = 1$, then $q = p$). For this to happen, those $t-1$ points must have the 4^{th} coordinate less than p^w, or q^w would not be equal to p^w. The case where t is equal to 1 (and $q = p$) can be seen in the previous example, since every point in L_1 is a point from one of the input sets. The case where q is the join between p and $t-1$ points ($t > 1$) can be seen, for example, for points in L_2, since each point in this set is the join of two points. For example, the point $(2, 3, 3, 2)$ in L_2 is the join of points a and b. Since $(2, 3, 3, 2)$ has the 4^{th} coordinate equal to b^w, then it is guaranteed that $a^w < b^w$, otherwise the 4^{th} coordinate of that point would not be the maximum of a^w and b^w, and therefore, it could not be the join of a and b. This means that those points q in L_t that have the 4^{th} coordinate equal to p^w belong to L_t only because p is present. Otherwise, they would be attained by only $t-1$ sets. Moreover, if $t = 1$, q would not be weakly dominated by any point from the input, if p was not present.

Consider a point set $T \subset \mathbb{R}^4$ and a point $p \in \mathbb{R}^4$ where $p \notin T$. Consider that each point in T belongs to one of the input sets (not necessarily the same set) and that so does p. The key idea is that if every point in T has the 4^{th} coordinate lower than the 4^{th} coordinate of point p, then the points in L_t ($t = 1, \ldots, n$) that have the 4^{th} coordinate equal to p^w would be the ones that are considered to be in L_t in the projection of $T \cup \{p\}$ on the (x, y, z)-space and that are not in L_t on the projection of T on the (x, y, z)-space.

An algorithm to compute the EAF in four dimensions can be described as follows. Each point is added, one by one, in ascending order of the 4^{th} coordinate, to some multiset, P. Every time a new point p is added to P, the projection of P on the (x, y, z)-space (represented as P^*) is used as the input to the EAF3D algorithm.[1] Note that P^* may have points dominated by other points in P^* from the same set, and even though the EAF computation problem expects points in the same input set to be non-dominated, EAF3D is prepared to deal with dominated points. Note also that it is assumed that the

[1] Strictly speaking, the input to the EAF computation problem, and to the EAF3D algorithm, is a sequence of non-dominated point sets S. However, in the preprocessing stage of the algorithm, all input sets are merged into a single multiset, X. To simplify both the notation and the explanation of the EAF4D algorithm, the input to EAF3D is abusively considered here to consist of such a multiset.

Algorithm 6 EAF4D

Input: $S = (X_1, \ldots, X_n)$ // a sequence of non-dominated point sets
Output: $R = (L_1, \ldots, L_n)$ // a sequence of non-dominated point sets

```
1: X ← ⊎_{i=1}^{n} X_i // multiset sum
2: Q^w is X sorted in ascending order of the 4^th coordinate
3: P ← {}
4: R*_1 ← ({}, ..., {}) // R*_1 = (L*_{1,1}, ..., L*_{1,n})
5: while Q^w ≠ ∅ do
6:    repeat
7:       p ← pop(Q^w)
8:       P ← P ∪ {p}
9:    until Q^w = ∅ or top(Q^w)^w ≠ p^w
10:   R*_2 ← EAF3D(P*) // R*_2 = (L*_{2,1}, ..., L*_{2,n})
11:   for all i ∈ {1, ..., n} do
12:      for all q ∈ L*_{2,i} − L*_{1,i} do
13:         L_i ← L_i ∪ {(q^x, q^y, q^z, p^w)}
14:   R*_1 ← R*_2
15: return (L_1, ..., L_n)
```

set to which each point in P^* belongs is known to the EAF3D algorithm. Every point found to be in L_t^* after the addition of p to P^* that was not in L_t^* before, will be in $L_t \subset \mathbb{R}^4$, and its fourth coordinate value will be equal to p^w. These points can be found by sweeping each L_t^* returned by EAF3D and filtering out the points that were in the corresponding L_t^* in the previous iteration. A fourth coordinate with value p^w is added to the points that remain. A base algorithm that proceeds in this way to compute the EAF in four dimensions is given as Algorithm 6. This algorithm is able to deal with points with equal w-coordinate by adding to P, at each iteration, all points with the same value of coordinate w as p.

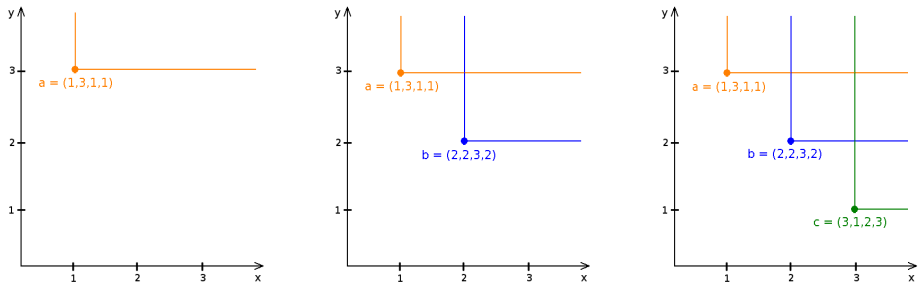
Taking Figure 3.2 as an example, assume that P is initially empty. The first point to be added is a, since it is the point with the lowest 4^{th} coordinate. Therefore, $P = \{a\}$, let us call it P_1 (see Figure 3.2(a)). Then, point b is added, since it is the point with lowest 4^{th} coordinate among the remaining points. Hence, Figure 3.2(b) represents the projection of $P = \{a, b\}$ on the (x, y)-plane, let us call it P_2. Similarly, Figure 3.2(c) represents $P = \{a, b, c\}$, let us call it P_3. Consider that the output of the EAF algorithm for three dimensions is of the form $R = (L_1, \ldots, L_n)$ and that P_1^*, P_2^* and P_3^* are the projections of P_1, P_2 and P_3 on the (x, y, z)-space, respectively. Assuming that R_k^* is the corresponding output of the EAF3D algorithm for the input P_k^*, for the current example, where $1 \leq k \leq 3$, the expected outputs are:

$$R_1^* = (\{(1,3,1)\}, \{\}, \{\})$$

$$R_2^* = (\{(1,3,1), (2,2,3)\}, \{(2,3,3)\}, \{\})$$

$$R_3^* = (\{(1,3,1), (3,1,2), (2,2,3)\}, \{(3,3,2), (2,3,3), (3,2,3)\}, \{(3,3,3)\})$$

$R = (L_1, L_2, L_3)$ is the desired output of the EAF4D algorithm on the current problem. In this example, since $P_1^* = \{\} \cup \{a^*\}$, where a^* is the projection of a on the (x, y, z)-space, all points in R_1^* are added to R, and their 4^{th} coordinate is a^w. In the next iteration, b^* is added, and $P_2^* = P_1^* \cup \{b^*\}$. Therefore, only the points in each L_t^* computed based on R_2^* that were not in the corresponding L_t^* based



(a) $z = 3$ and $w = 1$ (b) $z = 3$ and $w = 2$ (c) $z = 3$ and $w = 3$

Figure 3.2: The projection of the sets of points of the example of Figure 3.1 on the (x, y)-plane for a given value of coordinates z and w.

on R_1^*, should be added to L_t in R, with 4^{th} coordinate equal to b^w. Note that if the point $(1, 3, 1)$ in L_1^*, that is present in R_1^* and in R_2^*, were added to L_1 in R with 4^{th} coordinate equal to b^w, it would be weakly dominated by the point $(1, 3, 1, a^w)$, added previously. In the next iteration, having added c^* to obtain P_3^*, the points in R_3^* with the 4^{th} coordinate equal to c^w are obtained following the same reasoning used to add points with coordinate equal to b^w.

Deciding which points are new at each iteration of Algorithm 6 is performed in lines 11-13. In order to be able to perform this filtering step linearly in the number of output points, it is necessary to sort L_t^* from the current iteration ($L_{2,t}^*$) and the previous iteration ($L_{1,t}^*$). After having sorted each L_t^*, points from each $L_{2,i}^*$, where $i = 1, ..., n$ which are not in the corresponding $L_{1,i}^*$ are identified as final output points, their fourth coordinate is set, and they are added to L_i.

The base algorithm proposed in this Subsection can be modified in order to avoid having to sort and filter points at each iteration, in order to achieve better performance. The next Subsection describes an improved version of the base algorithm.

3.1.2 Improved algorithm

The base algorithm to compute the EAF in four dimensions can be improved by making some modifications to EAF3D that allow the sorting and filtering steps of EAF4D to be removed. These modifications should allow algorithm runtime to be reduced while keeping the same worst-case complexity.

Following the example of the previous Section, every time a point p is added to P, it is possible to verify which points are new in each L_t^* $(t = 1, \ldots, n)$ without having to sort and compare the outputs of the EAF3D algorithm for the current and the previous iteration. The idea is to modify the current EAF3D algorithm [11] to return only those points that belong to the output sequence $R = (L_1, \ldots, L_n)$ exclusively due to the presence of p. Moreover, as in the base algorithm, when considering that there are more input points with coordinate w equal to p^w, they are also added to P in the same iteration as p. In this case, EAF3D is modified to return those points that belong to the output sequence R exclusively due to the presence of one or more input point with p^w as coordinate w. Then, the expected output of

Algorithm 7 Improved EAF4D (IEAF4D)

Input: $S = (X_1, \ldots, X_n)$ // a sequence of non-dominated point sets
Output: $R = (L_1, \ldots, L_n)$ // a sequence of non-dominated point sets

```
1: R* ← ∅
2: X ← ⊎_{i=1}^{n} X_i // multiset sum
3: Q^w is X sorted in ascending order of the 4^th coordinate
4: p ← pop(Q^w)
5: P ← {}
6: while Q ≠ ∅ do
7:    N ← (∅, ..., ∅) // N = (N_1, ..., N_n)
8:    repeat
9:       p ← pop(Q^w)
10:      P ← P ∪ {p}
11:      s ← input_set(p)
12:      N_s ← N_s ∪ {p}
13:   until Q^w = ∅ or top(Q^w)^w ≠ p^w
14:   R* ← EAF3D(P*, N*)
15:   for all L_i* ∈ R* where 1 ≤ i ≤ n do
16:      for all q ∈ L_i* do
17:         L_i ← L_i ∪ {(q^x, q^y, q^z, p^w)}
18: return (L_1, ..., L_n)
```

the modified EAF3D algorithm, on the previous example, would be:

$$R_1^* = (\{(1,3,1)\}, \{\}, \{\})$$

$$R_2^* = (\{(2,2,3)\}, \{(2,3,3)\}, \{\})$$

$$R_3^* = (\{(3,1,2)\}, \{(3,3,2), (3,2,3)\}, \{(3,3,3)\})$$

Note that adding each point in R_1^*, R_2^* and R_3^* with the 4^{th} coordinate equal to a^w, b^w and c^w, respectively, to the corresponding L_t would result in the expected output of the EAF4D algorithm, described in the previous Subsection.

The pseudo code for the modified EAF algorithm for 4 dimensions (IEAF4D) is presented in Algorithm 7. The function EAF3D is the modified version of the EAF algorithm for three dimensions. In this newer version, only the new points in each set L_t^* are returned. Let us say that a point is a new point if it belongs to some L_t^* in the current iteration and did not in the previous iteration. Points are inserted into P in ascending order of the 4^{th} coordinate, and points with equal fourth coordinate are all added to P in the same iteration. These last points added to P that have equal 4^{th} coordinate are also added to their corresponding N_t. Note that, $N = (N_1, ...N_n)$ is a sequence of sets, where N_t, $t = 1, .., n$, contains only the input points of X_t that are in P in the current iteration of Algorithm 7, and were not in the previous iteration. P^* and N_t^* $(t = 1, ..., n)$ are, respectively, the projection of P and N_t on the (x, y, z)-space. These two data structures, P^* and N^*, will be passed as input to EAF3D. For simplicity, consider only input points with distinct coordinate w and that p is the last point added to P (and to the corresponding N_j, where $p \in X_j$). The original EAF3D [11] has to be modified to return only the new points that are in each L_t^* just because p^* is present. This can be done by marking the points which are

Algorithm 8 EAF computation in three dimensions

Input: $\mathrm{X} = \biguplus_{i=1}^{n} \mathrm{X}_i$ // multiset sum, duplicate points are allowed
Input: $\mathrm{N} = (\mathrm{N}_1, ..., \mathrm{N}_n)$ // N_t has the new points from input set t
Output: $\mathrm{R} = (\mathrm{L}_1, \ldots, \mathrm{L}_n)$ // a sequence of non-dominated point sets
1: $m \leftarrow \sum_{i=1}^{n} |\mathrm{X}_i|$
2: Q is X sorted in *ascending* order of the z coordinate
3: $\mathrm{D}_t \leftarrow \varnothing,\ t = 1, \ldots, n$ // D_t has the points in L_t^* that might be new
4: $\mathrm{L}_t \leftarrow \varnothing,\ t = 1, \ldots, n$
5: $\mathrm{L}_t^* \leftarrow \{(-\infty, \infty, -\infty), (\infty, -\infty, -\infty)\},\ t = 1, \ldots, n$ // Sentinels
6: $\mathrm{X}_i^* \leftarrow \{(-\infty, \infty, -\infty), (\infty, -\infty, -\infty)\},\ i = 1, \ldots, n$ // Sentinels
7: $p \leftarrow \mathsf{pop}(\mathrm{Q})$
8: $j \leftarrow \mathsf{input\_set}(p)$
9: $\mathsf{insert}(p, \mathrm{X}_j^*)$
10: $\mathsf{insert}(p, \mathrm{L}_1^*)$
11: **if** $p \in \mathrm{N}_j$ **then**
12: **if** $\mathrm{Q} = \varnothing$ **or** $\mathsf{top}(\mathrm{Q})^z \neq p^z$ **then**
13: $\mathrm{L}_1 \leftarrow \{p\}$
14: **else**
15: $\mathrm{D}_1 \leftarrow \{p\}$
16: $\mathrm{A} \leftarrow \{j\}$
17: $t_{\max} \leftarrow 1$
18: **while** $\mathrm{Q} \neq \varnothing$ **do**
19: $p \leftarrow \mathsf{pop}(\mathrm{Q})$
20: $j \leftarrow \mathsf{input\_set}(p)$
21: $q \leftarrow \mathsf{floor}^x(p, \mathrm{X}_j^*)$
22: **if** $p^y < q^y$ **then** // always true if X_j is indeed a non-dominated point set
23: $t \leftarrow t_{\max}$
24: $t_{\min} \leftarrow 1$
25: first intersection // part 1
26: dominated points // part 2
27: last intersection // part 3
28: submit p to X_j^*
29: submit p to $\mathrm{L}_{t_{\min}}^*$ given o and p
30: **if** $j \notin \mathrm{A}$ **then**
31: $\mathrm{A} \leftarrow \mathrm{A} \cup \{j\}$
32: $t_{\max} \leftarrow \min(t_{\max} + 1, n - 1)$
33: **if** $\mathrm{Q} = \varnothing$ **or** $\mathsf{top}(\mathrm{Q})^z \neq p^z$ **then**
34: add new points
35: **return** $(\mathrm{L}_1, \ldots, \mathrm{L}_n)$

"influenced" by p^*, either because they are the point p^* itself or because they are the join of p^* and other points. While marking the points, it is necessary to take into account that some of the points added to L_t^* are also weakly dominated by points in the same set as p^* besides p^*. Therefore, it is necessary to be careful to check whether a specific point is only weakly dominated by p^* among the points in its set. When considering points with equal coordinate w essentially the same thing happens, and it is necessary to mark the points in each L_t^* that are influenced by at least one input point in one of the N_t.

Algorithm 8 is the modified version of Algorithm 1. Most of the changes consists of the introduction of new statements (shaded). Moreover, a new subroutine (add new points - Algorithm 13) was introduced

Algorithm 9 first intersection

```
1: // q = floor^x(p, X*_j)
2: o ← nil
3: while t ≥ t_min do
4:    r ← floor^x(p, L*_t)
5:    if r^y ≤ p^y then
6:       o ← r
7:       t_min ← t + 1
8:    else if r^y < q^y then
9:       s_t ← (p^x, r^y, p^z)
10:      g_t ← r_t
11:   else
12:      s_t ← lower^y(q, L*_t)
13:      g_t ← s_t
14:   t ← t − 1
```

Algorithm 10 dominated points

```
1: // s_t^x ≥ p^x, q = floor^x(p, X*_j)
2: repeat
3:    q ← higher^x(q, X*_j)
4:    b ← max(p^y, q^y)
5:    for t = t_max down to t_min do
6:       while s_t^y ≥ b and (s_t^y > b or b > p^y) do
7:          if s_t^x ≥ q^x then
8:             s_t ← lower^y(q, L*_t)
9:             g_t ← s_t
10:         else
11:            submit (s_t^x, s_t^y, p^z) to L*_{t+1} given g_t and p
12:            s_t ← higher^x(s_t, L*_t)
13:            g_t ← s_t
14: until q^y ≤ p^y
```

Algorithm 11 last intersection

```
1: // s_t = floor^y(p, L*_t)
2: for t = t_max down to t_min do
3:    if s_t^x < q^x then
4:       submit (s_t^x, p^y, p^z) to L*_{t+1} given g_t and p
```

and a second version of submit (Algorithm 5) is defined as Algorithm 12. Note that, in Algorithm 8, the submit routine referred to in line 28 corresponds to the first version of submit (Algorithm 5 without lines 4 and 5) while the remaining references to submit correspond to the second version (Algorithm 12).

Previously, in the explanation of Algorithm 1, it was seen that the decision that a point u belongs to L_t, $t \in \{1, ..., n\}$, was delayed until it was sure that u was definitely a minimum of J_t. Therefore, that decision was made only when u was removed from tree L_t^* (in submit) or when the algorithm terminated. In Algorithm 8, this decision had to be delayed even further because it is necessary to know also whether u is new with respect to the L_t of EAF3D of the previous iteration of EAF4D algorithm. This can only be guaranteed after processing all points with the same coordinate z as u (in Algorithm 8). Therefore, when a point might be a new point of L_t, it is added to D_t. Whether points in D_t are really new or not is decided after processing all points with coordinate z equal to u^z, so that all points that could invalidate

Algorithm 12 submit u to L_t^* given h and p

1: $v \leftarrow \mathsf{floor}^x(u, \mathrm{L}_t^*)$
2: **if** $u^y < v^y$ **then**
3: **for all** $w \in \mathrm{L}_{t+1}^* : (u^x, u^y) \leq (w^x, w^y)$ **do**
4: **if** $u^z > w^z$ **then**
5: **if** $p \in \mathrm{N}_j$ **or** $(t > 1$ **and** $(h \in \mathrm{L}_{t-1}$ **or** $h \in \mathrm{D}_{t-1}))$ **then**
6: $\mathrm{D}_t \leftarrow \mathrm{D}_t \cup \{u\}$
7: $\mathsf{remove}(w, \mathrm{L}_t^*)$
8: $\mathrm{D}_t \leftarrow \mathrm{D}_t - \{w\}$
9: $\mathsf{insert}(u, \mathrm{L}_t^*)$

Algorithm 13 add new points

1: **for** $t = 1$ **to** $t = n$ **do**
2: **for all** $u \in \mathrm{D}_t$ **do**
3: $e \leftarrow \sum_{k=t}^{n} \mathrm{I}(u \in \mathrm{D}_k)$
4: **if** $e = 1$ **then**
5: **if** $\exists_{k \in \{1,..,n\}}.(\mathrm{X}_k^* \trianglelefteq u$ **and** $(\mathrm{X}_k^* - \mathrm{N}_k^*) \not\trianglelefteq u)$ **then**
6: $\mathrm{L}_t \leftarrow \mathrm{L}_t \cup \{u\}$
7: **else**
8: $m \leftarrow \sum_{k=1}^{n} (\mathrm{I}(\mathrm{X}_k^* \trianglelefteq u)$ **and** $\mathrm{I}((\mathrm{X}_k^* - \mathrm{N}_k^*) \not\trianglelefteq u))$
9: **for** $i = 0$ **to** $e - \min(e, m) - 1$ **do**
10: $\mathrm{D}_{t+i} \leftarrow \mathrm{D}_{t+i} - \{u\}$
11: **for** $i = e - \min(e, m)$ **to** $e - 1$ **do**
12: $\mathrm{D}_{t+i} \leftarrow \mathrm{D}_{t+i} - \{u\}$
13: $\mathrm{L}_{t+i} \leftarrow \mathrm{L}_{t+i} \cup \{u\}$
14: $\mathrm{D}_t \leftarrow \varnothing$, $t = 1, \ldots, n$

u are processed. The ones that are indeed new are added to the corresponding L_t (line 34).

To know whether a point might be new in L_t (to add it to D_t), it is necessary to know if it is in L_t^* possibly due to a new input point. Note that each point that is submitted to some L_t^* is submitted because of the influence of one or two points. It is always influenced by the point being swept (p) and sometimes, by a point of L_{t-1}^* (whenever $t > 1$ is true). Therefore, a point submitted to some L_t^* might be new if at least one of the points that are responsible for its submission is a new input point or it is influenced by at least one of the new input points (lines 5 and 6 of Algorithm 12).

After all points with equal coordinate z are swept, it is verified which points in each D_t are really new. This is the goal of Algorithm 13. If all input points were guaranteed to have distinct coordinates, then lines 1, 2, 5, 6 and 14 of this algorithm, would suffice. What these lines do is to add to L_t only the points u in D_t that are dominated by at least one new input point from some set N_k (note that $\mathrm{N}_k \subseteq \mathrm{X}_k$) and whose points already swept from set X_k which are not new ($\mathrm{X}_k - \mathrm{N}_k$) do not dominate u. The rest of the lines of Algorithm 13 are there because of the issues that arises when dealing with repeated coordinates of input points.

As seen earlier, the problem of computing the EAF becomes more difficult when considering that input points might have equal coordinates, as there are assumptions that are no longer true. Therefore, the problems that arise in the algorithm to compute the EAF in three dimensions, also become a problem in the computation of the EAF in four dimensions. The solutions introduced in the algorithm to compute

the EAF in 3 dimensions solve most of the problems related to equal coordinates in the four-dimensional problem, but not all of them. The remaining problems are solved by delaying the decision of stating that a point submitted to some L_t^* and marked as new, is really new. As in the original version of EAF3D, if it is determined that some point was wrongly submitted to an L_t^* it is removed from L_t^*. In the modified version, if it might be a new point then it is also unmarked as a possible new point of L_t^* by removing it from D_t.

When the same point u is in more than one D_t, then it is necessary to verify if that specific point is new in all of those D_t, and for that it is not enough to know if the condition in line 5 (Algorithm 13) is true. If this specific point is in e different D_t, and m is the number of new input sets that satisfy simultaneously conditions $I(X_k^* \trianglelefteq u)$ and $I((X_k^* - N_k^*) \not\trianglelefteq u)$, i.e., the number of different input sets that dominate u in the current iteration and that did not in the previous iteration of EAF4D. Then, the identification of u as a new point is only correct in the $\min(e, m)$ of the highest indices of those D_t. For example, assume the following initial problem:

1. $X_1 = \{(1, 1, 1, 2)\}$
2. $X_2 = \{(2, 2, 2, 1)\}$
3. $X_3 = \{(2, 2, 2, 1)\}$

The expected result is:

1. $L_1 = (\{(2, 2, 2, 1), (1, 1, 1, 2)\})$
2. $L_2 = (\{(2, 2, 2, 1)\})$
3. $L_3 = (\{(2, 2, 2, 2)\})$

and the computation of the two EAFs in three dimensions results in:

1. $R_1 = (\{(2, 2, 2)\}, \{(2, 2, 2)\}, \{\})$
2. $R_2 = (\{(1, 1, 1)\}, \{\}, \{(2, 2, 2)\})$

So, looking specifically at the second EAF in three dimensions, where the point from the first input set is the new one ($(1, 1, 1, 2) \in X_1$), when $(1, 1, 1)$ is swept, L_1^*, L_2^* and L_3^* will be {}, {} and {}, respectively and so $(1, 1, 1)$ is added to L_1^* (and also to L_1). Processing the first and the second $(2, 2, 2)$ would cause $(2, 2, 2)$ to be added to D_2 and D_3, respectively. However, $(2, 2, 2)$ in L_2 in three dimensions is not new, as it can be seen in the result returned in the previous iteration of EAF4D. Therefore, $(2, 2, 2)$ is only new in D_3 (note that $e = 2$ and $m = 1$).

Both algorithm proposed here have two improvements not described in the pseudocodes. One is that if a point is detected to be dominated by a point from its set when projected on (x,y,z)-space, then it

is skipped in the following computations of EAF in three dimensions. The other improvement is that in line 24 of Algorithm 8, instead of initializing t_{min} to 1 when the point being swept is not new, it is initialized to k where L_k^* is the set to which the input point was added (the value of t_{min} after that point was swept in EAF3D of the previous iteration of EAF4D).

Having a description of a base and an improved algorithm to compute the EAF in four dimensions, their performances can now be analyzed. This is done in the next Subsection.

3.1.3 Complexity

The main loop of the base algorithm to compute the EAF in four dimensions (Algorithm 6) consists of computing m EAF in three dimensions involving up to m points each. Since EAF3D has a cost of $O(n^2 m \log m)$, running it m times has a total cost of $O(n^2 m^2 \log m)$. In addition, Algorithm 6 also sorts each EAF3D output, and compares two EAF3D outputs (filtering step), m times. Sorting m EAF3D outputs also has a cost of $O(n^2 m^2 \log m)$, since the maximum number of output points of the EAF3D in three dimensions is known to be $O(n^2 m)$ [11] and $n \leq m$. In pratice, the output points of an EAF3D are not sorted all together, but in disjoint subsets (L_1, ..., L_n), which is generally faster. The filtering step sweeps through each L_t ($t = 1, ..., n$) of two EAF3D outputs, therefore sweeping $O(n^2 m)$ points per EAF3D output. Since the points of each L_t in each EAF3D output are sorted, the filtering step is performed in linear time for each L_t. Therefore, the total cost of filtering points is $O(n^2 m^2)$ which is also a new bound on the number of output points. Hence, the base algorithm for the computation of the EAF in four dimensions has a time complexity of $O(n^2 m^2 \log m)$.

The spacial complexity of the algorithm is dominated by the storing of the output points, and is therefore $O(n^2 m^2)$.

The improved algorithm to compute the EAF in four dimensions has the same time and space complexities as the base version. All modifications introduced in EAF3D are performed in constant time and need constant extra space, except for the modifications of adding the subroutine detailed in Algorithm 13. Note that verifying if a point is possibly new in Algorithm 12 has constant cost since it is known in constant time whether a point in each L_t^* is in D_t or in L_t and whether an input point is in the N_j corresponding to its set.

When considering that all points have distinct coordinate w, then it is possible to see that Algorithm 13 adds a cost of $\log(m)$ per point added to some L_t^*, since it verifies if each of those points is dominated by the only non-empty N_k and not by $(X_k^* - N_k)$, $k \in \{1, ..., n\}$. Therefore, it adds a cost of $O(n^2 m \log m)$ per point swept in four dimensions. Note that if c points with equal coordinate w where processed one at a time, processing these c points would cause Algorithm 13 to have a total extra cost of $O(cn^2 m \log m)$. In the improved algorithm, all points with equal coordinate w are considered together. Then, if these c points all belong to distinct input sets, then Algorithm 13 verifies if each possible new point in each L_t^* is dominated by one of these c input points (let us call it q) but not by the other points from q's sets. Therefore, having the same $O(cn^2 m \log m)$ time complexity as it would have if points were processed

separately. Note, that if some of the c input points belonged to the same input set, then the time complexity associated to those c input points would be $O(kn^2m\log m)$, where k is the number of distinct inputs sets to which the c input points belong to. Hence, the total cost added by Algorithm 13 is $O(n^2m^2\log m)$.

Looking in an abstract way to the two algorithms proposed for the computation of EAF in four dimensions and the existing algorithm for three dimensions, the former are an extension of the later. All are dimension sweep algorithms to compute EAF in d dimensions ($d = 4$ in the first case and $d = 3$ in the second) that compute m EAF in $(d-1)$ dimensions. For each swept point in dimension d the points in each L_t in their projection in $(d-1)$ dimensions that were not present in the corresponding L_t of the previous sweep are determined. Then, those points are added to the correspondent L_t in d dimensions with the d^{th} coordinate equal to the d^{th} coordinate of the point being swept in dimension d. The algorithms for the four dimensional case differ from the three dimensional case in the sense that the former need to compute the whole EAF in $(d-1)$ dimensions, i.e., from scratch, in each iteration, while the latter computes the EAF in $(d-1)$ dimensions on the top of the EAF in $(d-1)$ of the previous iteration, i.e., in a constructive way.

In principle, this method of computing the EAF in d dimensions by computing m EAF in $(d-1)$ dimensions could also be used to construct an algorithm for any number of dimensions, obtaining an algorithm with a time complexity of $O(m^{d-2}n^2\log m)$. However, this complexity might be rather excessive, since the available lower bound on the maximum number of output points in d dimensions is $\Omega\left(m^{\lfloor d/2\rfloor}\right)$. Moreover, the number of output points with more than four dimensions may also become very high, which may impair the use of EAF in higher numbers of dimensions.

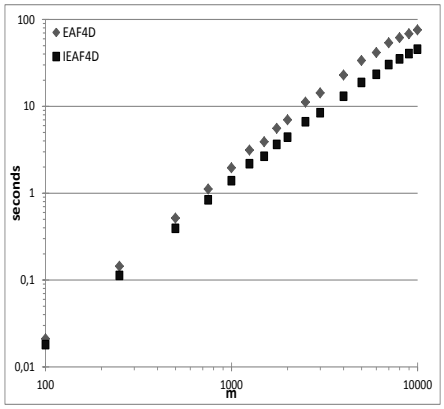
3.1.4 Experimental results

In this Section, it will be explained how the base EAF4D (described in Section 3.1.1) and the improved version (IEAF4D, described in Section 3.1.2), were tested, how their performance were compared, and in which conditions.

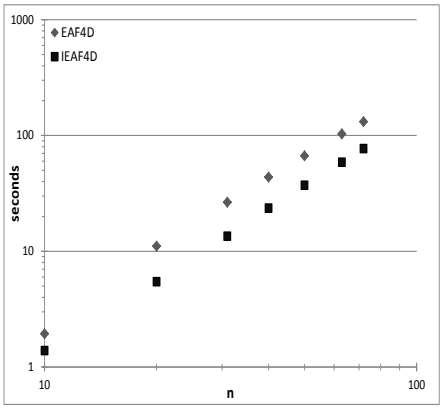
Both algorithms were implemented in C and were compiled using gcc version 4.6.0. To check that the implementations were correct, some tests were created, namely, tests to exercise specific parts of the algorithm. These tests included limit situations, such as input points with the same coordinate values as other points in the same input set or other input sets. Besides these tests, the data sets made available by the Walking Fish Group were used [8][1]. The results were verified using the invariance of the EAF with respect to permutations of the objectives, and the property of idempotence, i.e., EAF(EAF(S))=EAF(S). Although a formal proof of this property is not available yet and will be the subject of further work, the EAF computation problem was formulated with idempotence in mind, and this property could be verified experimentally in all cases.

To compare algorithm performance, three different data sets from the Walking Fish Group were used,

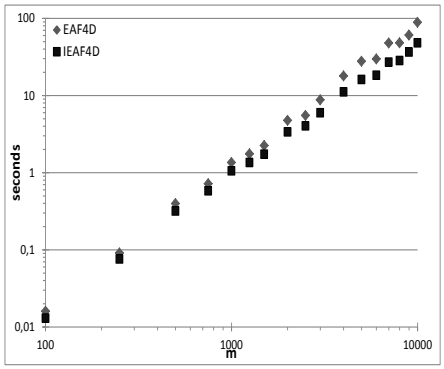
[1] http://www.wfg.csse.uwa.edu.au/hypervolume/index.html#data



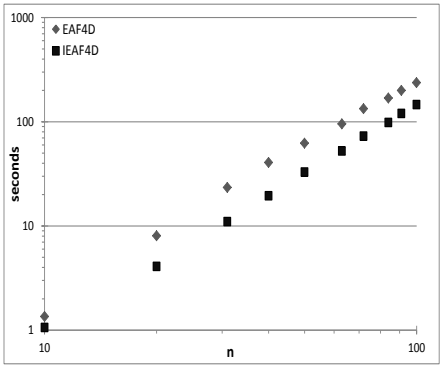
(a) n=10



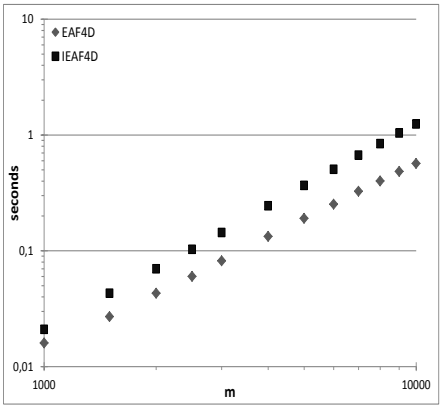
(b) m=1000



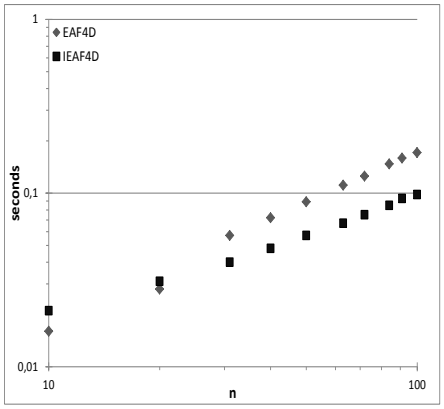
(c) n=10



(d) m=1000



(e) n=10



(f) m=1000

Figure 3.3: Results for the spherical data set (a and b), for the random data set (c and d) and for the degenerate data set (e and f).

namely the spherical data set, the random data set and the degenerate data set. The spherical data set was obtained by generating a non-dominated point set with 10000 points, where points are spread along a spherical front. k points were then randomly selected ($k = 10, 50, ..., 1000$), 10 times (per each different value of k). Therefore, they can be seen as 10 different outcomes for the same problem. The disadvantage of these data sets is that, since the original set of points was a non-dominated point set, the 10 subsets

will not have dominated points among them, unless there are repeated points in different subsets. The random data set was constructed in the same way, but the original set of points was generated by generating randomly non-dominated points. Finally, the same applies also to the degenerate data set, but in this case, the points have the characteristic that when sorted in ascending order of coordinate w, all points will also be sorted in descending order of the remaining coordinates.

Tests were performed on an Intel Core 2 Duo P8600, with 2.40GHz and 3MB cache size. Figure 3.3 shows the runtimes of an execution of EAF4D and IEAF4D for each test. It can be observed that in the spherical and random data set, IEAF4D always outperforms the EAF4D, the former being up to twice as fast as the latter. On the degenerate data set, when n is constant and small, EAF4D outperforms IEAF4D. This is not a surprise, since in this particular data set, given its structure, it is known that in each EAF4D iteration the output of EAF3D consists of one point per output set L_t, which is always new compared to the previous iteration of EAF4D. Therefore, when m grows under constant n, the effort spent in the sorting and filtering steps is always the same for each point swept in four dimensions. On the other hand, in this data set, when m is constant and n grows, the effort spent in sorting and filtering steps by EAF4D increases. In this case, IEAF4D starts by performing worse than EAF4D, but its performance improves as n grows.

It can also be seen that, on the spherical and random data sets, the runtime of both algorithms grows approximately quadratically as expected, both for constant n and for constant m. This is also observed to the degenerate data set when n is constant, but the runtime grows linearly as n increases when m is constant, showing that this is a favorable case for both algorithms, specially for IEAF4D. This is justified by the fact that each point in the degenerate data set either dominates or is dominated by every other point with respect to dimensions (x, y, z). Therefore, the maximum total number of output points in this case is $O(n)$ for the three-dimensional subproblems. In this particular case, EAF3D performs in $O((n + m) \log m)$ and EAF4D and IEAF4D will perform in $O((m + n)m \log m)$, which explains the different slopes, for constant m and constant n.

Having these results shows that, in spite of IEAF4D adding a little extra work to the EAF3D algorithm to avoid having to sort and filter points, it is worth it in general.

3.2 Output sensitive algorithm for two dimensions

It is known that the worst-case time complexity of the algorithms to compute the EAF is directly related to the maximum number of output points. It is also known that not all inputs result in the maximum number of points, thereby raising the possibility of developing output-sensitive algorithms in order to obtain better performance. In Subsection 3.2.1, a new algorithm for the two dimensional case having this characteristic is described, and its complexity is analyzed in Subsection 3.2.2. Later, in Subsection 3.2.3, the new algorithm is compared against the existing algorithm for the two dimensional case.

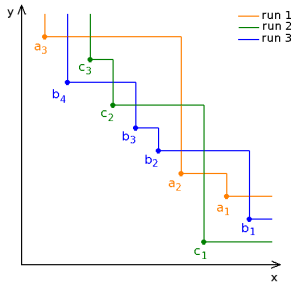


Figure 3.4: Example of three runs of a multiobjective optimizer for the same two-objective problem.

3.2.1 Description

An algorithm to solve the EAF computation problem in 2 dimensions is currently available [11], with a complexity of $O(m \log m + mn)$. This algorithm exhibits the same complexity independently of the number of output points, since the number of operations performed after the sorting step is always $\Theta(mn)$. This result can be improved by developing an algorithm that is output sensitive. Such an algorithm will still perform $O(mn)$ steps in the worst case, but will be able to achieve m steps in the best scenario, which is the one where each input set either totally dominates, or is totally dominated by, each of the remaining input sets. The final complexity will still be $O(m \log m + mn)$ due to the sorting at the beginning of the algorithm.

The description of the output-sensitive algorithm will follow next, but since considering points with equal coordinates makes the task of developing an output-sensitive algorithm harder and less clear, this description will be divided into four parts. The first part describes a first version of the output-sensitive algorithm where only input points with distinct coordinates are allowed. In the second part, the problem of having equal coordinates is explained in detail. A second version of the output-sensitive algorithm where input points with equal x-coordinate values are allowed is described in the third part. Finally, a third (and final) version of the algorithm is detailed in the fourth part, where no restrictions on the input points are imposed.

All input points have distinct coordinates

Algorithm 14 presents the pseudocode for the output-sensitive algorithm to solve the EAF computation problem in two dimensions. Figure 3.4 illustrates the input sets that will be used as an example to explain the main ideas. It is assumed that all points have distinct coordinate values.

This algorithm follows the idea of sweeping through every point in ascending order of the y-coordinate and keeping track of the last point visited for every set. For example, in Figure 3.4, after processing point b_3, the last visited points of each set are a_2, b_3 and c_1. These points are kept sorted with respect to the x-coordinate. Following the same example, the order in which they are kept, is b_3, a_2 and c_1. Each time a new point is selected (the next one is c_2), it is checked which of the points kept have the x-coordinate between the x-coordinate of the new point (c_2^x) and the last point visited in the same set (c_1^x), which

Algorithm 14 EAF 2D

Input: $S = (X_1, ..., X_n)$
Output: $R = (L_1, ..., L_n)$

```
 1: X ← ⊎_{i=1}^{n} X_i // multiset sum
 2: X^y is X sorted in ascending order of y
 3: L_t ← ∅, t = 1, ..., n
 4: v_t ← nil, t = 1, ..., n
 5: a_t ← 0, t = 1, ..., n
 6: c ← 1
 7: while X^y ≠ ∅ do
 8:    p ← pop(X^y)
 9:    s ← input_set(p)
10:    if a_s = 0 then
11:       a_s ← c
12:       c ← c + 1
13:    v_{a_s} ← p
14:    k ← a_s − 1
15:    while k ≥ 1 and p^x < v_k^x do
16:       L_{a_s} ← L_{a_s} ∪ {(v_k^x, p^y)}
17:       switch(v, a, k, k + 1)
18:       k ← k − 1
19:    L_{a_s} ← L_{a_s} ∪ {p}
20: return (L_1, ..., L_n)
```

are a_2 and b_3 in the given example. These points in between are swept in descending order of their x-coordinate. The join of the new point and each of these points are points to be added to some L_t (where t is the number of points kept that are to the left of each of the points in between, plus 1). In the given example, the join of c_2 and a_2 results in the point (a_2^x, c_2^y), and, since a_2 is the second point of the points kept, it is added to L_3. Following the same reasoning, the join of c_2 and b_3 results in the point (b_3^x, c_2^y), and since b_3 is the first point of the points kept, it is added to L_2. Finally, the new point is added to the L_t set to which it belongs, where t is the number of points kept to its left plus one. In this case, since c_2 does not have any points to its left, it is added to L_1.

In Algorithm 14, the last visited points in every set are kept in v (they are kept sorted in ascending order of their x-coordinate), where v maps each position (order number) to one of these points. For example, after processing b_3, v would be (b_3, a_2, c_1). Vector a is used to keep track of the position of the point of each set that is in v, mapping the index of each set to the position of its last visited point in v. Hence, in this example, assuming that indices 1, 2 and 3 correspond to sets a, b and c, respectively, vector a would be $(2, 1, 3)$. To maintain the points in v sorted while points in v are swept, the new point $p \in X$ being processed starts from the last position of the point from its set in v (or from position t if only points from $t - 1$ sets were processed so far and p's set is not one of them), and switches positions in v with each point of v swept (line 17).

Vector v is responsible for indicating from which value of coordinate x, at the current height, the region is dominated by each L_t ($t = 1, ..., n$). For example, if p is the point being processed, (v_t^x, p^y) indicates that the region attained by at least t input sets above p^y in y-coordinate is the one dominated by (v_t^x, p^y). Therefore, assume that p is indeed the point being processed, and that p^x dominates a value

Algorithm 15 EAF 2D - switch(v, a, i, j)

Input: v, a, i, j

1: $p \leftarrow \mathrm{v}_j$
2: $s \leftarrow \mathsf{input\_set}(p)$
3: $\mathrm{v}_j \leftarrow \mathrm{v}_i$
4: $\mathrm{a}_{\mathsf{input\_set}(\mathrm{v}_j)} \leftarrow j$
5: $\mathrm{v}_i \leftarrow p$
6: $\mathrm{a}_s \leftarrow i$

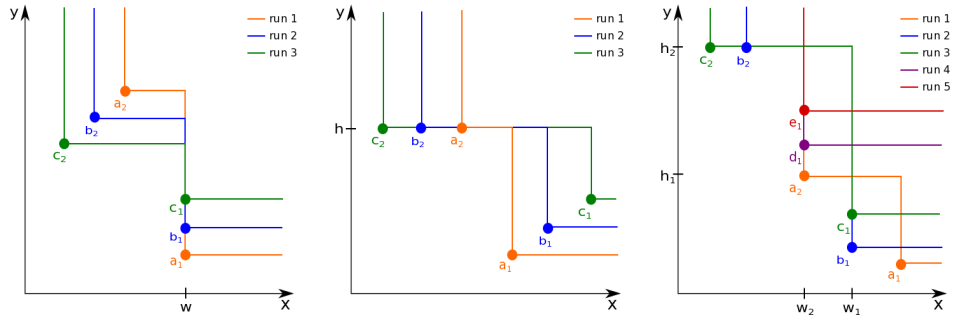
of x where some L_t begins (v_t^x). If (v_t^x, p^y) is not dominated by any other previously processed point from p's set, it indicates that the region dominated from that value of x, and from the current height (p^y) is now dominated by one more input set (p's set), i.e., it is dominated by at least $t+1$ sets. Hence, (v_t^x, p^y) is definitely a point in L_{t+1}, since the region immediately below this point and the region immediately to its left is dominated by only t points (lines 15-18). This is true because it is assured that the region immediately below p^y is dominated by a maximum of t points (if only points from t sets where processed so far) or because the region immediately below p^y that is dominated by at least $t+1$ sets starts from a value of coordinate x higher than v_t^x. Moreover, if the first t points of v are all the points in v that are lower than p according to the x-coordinate, then, starting from height p^y, the region at p's right is dominated by at least $t+1$ input sets. Therefore, p is itself a point of L_{t+1} (line 19). In this case, it is said that the final position of p in v is $t+1$.

If v contains a point from p's set, then let such a point be called q, otherwise let q be (∞, p^y). To ensure that all output points are found, it is enough to verify the points in v between p and q according to coordinate x. This is true because at p's height (p^y), the points in v to the right of q ($q \leq (v_t^x, p^y)$) and the points in v to the left of p, are dominated by the same number of input sets as they were immediately below height p^y. Therefore, p has no influence on the region to its left or on the region at q's right. Hence, given an input where all points have distinct coordinate values, Algorithm 14 is able to produce all points in each L_t, $t = 1, ..., n$, and all points produced are correct.

The problem of having input points with equal coordinates

When considering that the input points can have equal coordinates, the input points are swept in ascending order of coordinate y and points with equal coordinate y are swept in ascending order of coordinate x. Considering this scenario, the algorithm as it is will still produce all points in each L_t, $t = 1, ..., n$, but some of the output points produced might be incorrect. Figure 3.5 shows some examples where the algorithm would produce incorrect output points. Another problem that arises when using Algorithm 14 with input points with equal coordinates is that it is no longer possible to guarantee the output-sensitivity of the algorithm.

Looking at Figure 3.5(a), it is possible to observe these problems. In this case, the first points of each input set have equal coordinate x. After processing points b_1, a_1 and c_1 (all having coordinate x equal to w), v would be (a_1, b_1, c_1) and L_1, L_2 and L_3 would include points (w, a_1^y), (w, b_1^y) and (w, c_1^y), respectively. The problem arises when processing c_2, because since c_1 in v is to the right of a_1 and b_1, and c_2 should



(a) (b) (c)

Figure 3.5: Two-dimensional examples of input sets where points with equal coordinate can be problematic.

be to the left of a_1 and b_1 in v, then c_2 will switch positions in v with these two points. Therefore, points c_2, (w, c_2^y) and (w, c_2^y) would be added to L_1, L_2 and L_3, respectively. The last two points are not minimal points because (w, c_2^y) and (w, c_2^y) are dominated by (w, b_1^y) (point already in L_2) and (w, c_1^y) (point already in L_3), respectively. This happens only because it is not possible to guarantee that the new points added to each L_t $(t = 2, ..., n)$ have lower x coordinates than the points already there, as it would happen if none of the input points had equal coordinates. Following the same reasoning, processing b_2 would incorrectly add point (w, b_2^y) to L_3. In this example it is possible to see that the algorithm is no longer output-sensitive, even if the incorrect points were not added to the corresponding L_t by verifying whether their x-coordinate is lower than the x-coordinate of the last point added to that L_t. For example, when processing point c_2 only one output point should be produce (add c_2 to L_1). Therefore, only one operation should be executed but, instead, processing c_2 has a cost of $O(n)$ since it switches positions with one point in each of the other input sets, and produces a point in each swap.

Figure 3.5(b) shows another example where the problems of Algorithm 14 for the example of Figure 3.5(a) also occur, but in this case the problem arises because some points have equal y coordinates. Similarly, as in the example of Figure 3.5(a), when processing point c_2, it would switch positions with b_1 and a_1, would correctly add a point to L_1 (point c_2), and would incorrectly add a point to L_2 and a point to L_3, which would be points (a_1^x, h) and (b_1^x, h), respectively. The difference relatively to the previous example is that, at the time c_2 is processed, the points incorrectly produced are not dominated by any point already added to the corresponding L_t. The point added to L_2 (L_3) when processing c_2 is dominated by a point added after processing b_2 (a_2), at the time the point b_2 (a_2) itself is added to L_2 (L_3). Again, this only happens because some input points have equal coordinates, in this case y, otherwise it would be guaranteed that any point added to some L_t would have coordinate x better than the points previously added, but coordinate y would be worse. As in the previous example, it would be possible to guarantee that incorrect points would not be in L_t at the end of the algorithm. This could be achieved by verifying, every time a new point is added to L_t, if it dominates the last point added to that L_t, and removing that last point if it is dominated. Still, this does not suffice to assure the output-sensitivity of the algorithm.

Input points can have equal coordinate x

Due to the problems described earlier, it is necessary to use a more elaborate method to ensure the correctness of the results and the output-sensitivity of the algorithm, independently of input points having equal coordinates or not. Regarding this, there are a few details that are important to note. One is that when there are points with equal coordinate x (this only happens between points of distinct sets, otherwise one or more input sets would not be non-dominated point sets), the order among these points in v is irrelevant. Consequently, there is a freedom to choose the order among them in v. Another detail is that if some point p from some input set X_s is processed, and q is the point of X_s in v, then p only contributes to the region strictly to the left of q, i.e., if q has equal coordinate x as other points in v, then while sweeping points in v between p^x and q^x, the points with coordinate x equal to q^x should not be swept at all. In the example of Figure 3.5(a) this was seen when point c_2 is processed since it does not contribute to the region starting at value w of x-coordinate it should not sweep points a_1 and b_1.

The two details explained above lead to the solution to solve the problem of having points with equal x-coordinate values. The solution, given the previous example, is to move q's position in v to the leftmost position in v among points with coordinate x equal to q^x. This would avoid producing incorrect points and extra operations as it was seen in the example of Figure 3.5(a). Hence, in this example, as soon as point c_2 is chosen to be processed, $\mathrm{v} = (a_1, b_1, c_1)$ should be modified to (c_1, a_1, b_1). This is what is done in Algorithm 16, which is a second version of Algorithm 14 which fixes some of its problems, specifically the ones related with points with equal values of coordinate x. In this algorithm, it is necessary to keep track of the positions of the points in v where a specific value of x begins and ends, i.e., to know where the consecutive points with equal coordinate x are placed in v. This allows one to know if a certain point in v has more points in v with equal coordinate x to its left, or right, and where they are, without having to search for them in v.

Using the example of Figure 3.5(c), before processing points b_2 and c_2, the following information should be available: the points in v that have coordinate x equal to w_2 are between positions 1 and 3; the points with x equal to w_1 are between positions 4 and 5; and the points with x equal to a_1^x are between positions 6 and 6. This information is obtained through functions $\mathsf{begin}(x)$ and $\mathsf{end}(x)$ which can be executed in constant time. It is assumed that this information is updated every time v is modified. Looking back to the example of Figure 3.5(a), when c_2 is chosen to be processed, $\mathrm{v} = (a_1, b_1, c_1)$ is modified to (c_1, b_1, a_1) by switching the position of the point from c_2's set in v (c_1) with the leftmost point in v with coordinate x equal to c_1^x (line 9). Therefore, since it is known that c_1^x begins and ends in position 1 and 3 respectively, c_1 moves to position 1. When c_2 replaces c_1 in v (line 10), the information that c_2^x begins and ends in position 1, and that c_1^x begins and ends in position 2 and 3, respectively, is updated. Note that by doing this switch early when a point is chosen to be processed, it allows v_t to be initialize to $(\infty, -\infty)$, where $t = 1, ..., n$, and initialize $\mathrm{a}_t = t$, i.e., initialize with sentinels for each input set, therefore removing the need to verify whether any point from the same input set as the current one was already processed (lines 10-12 of Algorithm 14).

Algorithm 16 EAF 2D

Input: $S = (X_1, ..., X_n)$
Output: $R = (L_1, ..., L_n)$
1: $X \leftarrow \biguplus_{i=1}^{n} X_i$ // multiset sum
2: X^y is X sorted in *ascending* order of y
3: $L_t \leftarrow \varnothing, t = 1, \ldots, n$
4: $v_t \leftarrow (\infty, -\infty), t = 1, \ldots, n$
5: $a_t \leftarrow t, t = 1, \ldots, n$
6: **while** $X^y \neq \varnothing$ **do**
7: $\quad p \leftarrow \mathsf{pop}(X^y)$
8: $\quad s \leftarrow \mathsf{input\_set}(p)$
9: $\quad \mathsf{switch}(v, a, \mathsf{begin}(v^x_{a_s}), a_s)$
10: $\quad v_{a_s} \leftarrow p$
11: $\quad k \leftarrow a_s - 1$
12: $\quad$ **while** $k \geq 1$ **and** $p^x < v^x_k$ **do**
13: $\quad\quad L_k \leftarrow L_k \cup \{(v^x_k, p^y)\}$
14: $\quad\quad i \leftarrow \mathsf{begin}(v^x_k)$
15: $\quad\quad \mathsf{switch}(v, a, i, k+1)$
16: $\quad\quad k \leftarrow i - 1$
17: $\quad L_{a_s} \leftarrow L_{a_s} \cup \{p\}$
18: **return** $(L_1, \ldots, L_n)$

There is one more important detail to notice when dealing with equal coordinates among input points when a point p is being processed. If while sweeping through points in v, a set of points that have some value w higher than p^x as their x-coordinate is found, then p contributes with the addition of only one output point with w as coordinate x. It means that, at height p^y, if the region attained by L_i,...,L_j $(1 \leq i \leq j \leq n)$ starts exactly at value w of coordinate x, then the point (w, p^y) is dominated by j inputs sets plus one (p's set), so the only output point with w as coordinate x added when processing p is the point (w, p^y) added to L_{j+1} (note that in Algorithm 14, (w, p^y) would be correctly added to L_{j+1}, but incorrectly added to $L_{i+1}, ..., L_j$). All of this can be observed in the example depicted in Figure 3.5(c), where a_2 is point p in the explanation above, the point being processed that, when switching positions with points with coordinate x equal to w_1 (points b_1 and c_1), should produce only one output point, the point (w_1, h_1) added to L_3. This point should be produced when sweeping b_1 in v and point c_1 should be skipped and not even processed (which does not happen in Algorithm 14, where it would process c_1 and add point (w_1, h_1) to L_2, which is dominated by the point (w_1, c_1^y) already in L_2).

The solution given above, is what is done in Algorithm 16 in line 13, where (v^x_k, p^y) is added to L_{k+1} (where k corresponds to j from the explanation above). To avoid producing incorrect output points, the algorithm takes advantage of the freedom of choosing the order of points in v with equal coordinate x, and switches the positions in v of p and of the leftmost point in v with coordinate x equal to v^x_k (line 15). Executing this step, and updating k to skip the positions of all points in v with coordinate x equal to v^x_k (line 16), plus the solution explained earlier of line 9, allows to avoid the problems of producing incorrect points and of executing excessive operations of Algorithm 14 in problems such as the one of Figure 3.5(a).

Input points can have equal coordinate x and/or y

The last detail explained above, of switching the point of X^y being processed with the leftmost point in v that has equal coordinate x as the point in v being verified, can be generalized. This is needed when considering points with equal y-coordinate values. Consider a set of points A in v whose coordinate x is w and whose points fill positions i to j in v ($1 \leq i \leq j \leq n$). Also consider a set of points B whose coordinate y is h and which are the next points to be processed by the algorithm after the points in v. Moreover, consider that expressions $\forall p \in \mathrm{B} \Rightarrow p^x < w$ and $\forall p \in \mathrm{B} \wedge p \in \mathrm{X}_s \Rightarrow \mathrm{v}^x_{\mathrm{a}_s} > w$ are true. In this scenario, the point (w, h) should be added to L_t, were $t = i + \mathsf{max}(|\mathrm{A}|, |\mathrm{B}|), ..., i + \mathsf{max}(|\mathrm{A}|, |\mathrm{B}|) + \mathsf{min}(|\mathrm{A}|, |\mathrm{B}|) - 1$. This can also be observed in Figure 3.5(c), considering $\mathrm{A} = \{a_2, d_1, e_1\}$ (in this case $i = 1$ and $j = 3$) and $\mathrm{B} = \{c_2, b_2\}$, point (w_2, h_2) should be added to L_4 and L_5. Note that if points d_1 and e_1 were not input points, therefore having $\mathrm{A} = \{a_2\}$ ($i = j = 1$), the only output point that should be produced would be point (w_2, h_2) added to L_3.

The previous considerations indicate that processing all points with equal y coordinates together and swapping, when necessary, a point's position in v with another point with equal coordinate x, might be a good way to get around all the problems seen earlier related to input points with equal coordinates. Algorithm 16 gets around of some of these problems, but only the problems related to equal values of coordinate x and not of coordinate y. Algorithm 17 is the final version of an algorithm to compute the EAF in 2D dimensions which fixes the flaws of Algorithm 16 by processing the points in X with equal coordinate y at the same time, as suggested before.

Algorithm 17 uses a stack (T) to keep the next point to be swept and all the following points that have equal coordinate y. This stack is created in the loop between lines 9 and 17. To manipulate the stack, a few functions are used. Function $\mathsf{push}(p, \mathrm{T})$ allows to insert p at the top of the stack T, $\mathsf{pop}(\mathrm{T})$ removes and retrieves the point at the top of the stack T, and $\mathsf{peek}(\mathrm{T})$ only retrieves the point at the top without modifying the stack T. Note that, similarly to what was done in Algorithm 16, the points q from the same set as the points in T are all moved to the leftmost position in v of all points with coordinate x equal p^x (line 13), and are then replaced by the new points from its set in T (line 14). Since points in X that have equal y-coordinates are sorted in ascending order of coordinate x, points are inserted in T also in ascending order of x. k is initialized as the rightmost position in v of the points in T, which indicates the last position in v which is influenced by, at least, one point in T. Note that b_i, $i = 1, ..., |\mathrm{T}|$, indicates which is initially, the rightmost position in v of the first i points in T.

After the initialization of T and k, the algorithm starts sweeping the points in v (lines 19-39), looking for the final positions in v of the points in T and adding the output points found. Since points in v are swept in descending order of coordinate x, the goal while sweeping is to find the final position in v of the point at the top of the stack because, since it is the point in the stack with higher value of coordinate x its final position in v will be found first. Therefore, after finding the final position of the point at the top of the stack, it is removed from the stack, and the algorithm starts looking for the final position of the next point at the top of the stack. The algorithm stops when there are no more points left in the stack.

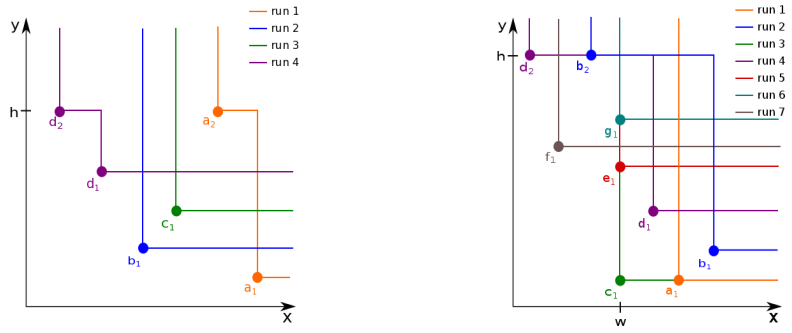
Algorithm 17 EAF 2D

Input: $\mathrm{S} = (\mathrm{X}_1, ..., \mathrm{X}_n)$
Output: $\mathrm{R} = (\mathrm{L}_1, ..., \mathrm{L}_n)$

1: $\mathrm{X} \leftarrow \biguplus_{i=1}^{n} \mathrm{X}_i$ // multiset sum
2: X^y is X sorted in *ascending* order of y
3: $\mathrm{T} \leftarrow \varnothing$ // a stack
4: $\mathrm{L}_t \leftarrow \varnothing, t = 1, \ldots, n$
5: $\mathrm{v}_t \leftarrow (\infty, -\infty), t = 1, \ldots, n$
6: $\mathrm{a}_t \leftarrow t, t = 1, \ldots, n$
7: **while** $\mathrm{X}^y \neq \varnothing$ **do**
8: $\quad k \leftarrow 0$
9: $\quad$ **repeat**
10: $\quad\quad p \leftarrow \mathsf{pop}(\mathrm{X}^y)$
11: $\quad\quad s \leftarrow \mathsf{input\_set}(p)$
12: $\quad\quad \mathsf{push}(p, \mathrm{T})$
13: $\quad\quad \mathsf{switch}(\mathrm{v}, \mathrm{a}, \mathsf{begin}(\mathrm{v}^x_{\mathrm{a}_s}), \mathrm{a}_s)$
14: $\quad\quad \mathrm{v}_{\mathrm{a}_s} \leftarrow p$
15: $\quad\quad b_{|\mathrm{T}|} \leftarrow \mathsf{max}(k, \mathrm{a}_s)$
16: $\quad\quad k = \mathrm{b}_{|\mathrm{T}|}$
17: $\quad$ **until** $\mathrm{X}^y = \varnothing$ **or** $p^y \neq \mathsf{top}(\mathrm{X}^y)$
18: $\quad l \leftarrow 0$ // number of points of T seen in v so far
19: $\quad$ **while** $\mathrm{T} \neq \varnothing$ **do**
20: $\quad\quad$ **if** $k \geq 1$ **and** $\mathrm{v}_k \in \mathrm{T}$ **then**
21: $\quad\quad\quad l \leftarrow l + 1$
22: $\quad\quad\quad k \leftarrow k - 1$
23: $\quad\quad$ **else**
24: $\quad\quad\quad$ **if** $k < 1$ **or** $\mathsf{peek}(\mathrm{T})^x \geq \mathrm{v}^x_k$ **then**
25: $\quad\quad\quad\quad q \leftarrow \mathsf{pop}(\mathrm{T})$
26: $\quad\quad\quad\quad s \leftarrow \mathsf{input\_set}(q)$
27: $\quad\quad\quad\quad \mathsf{switch}(\mathrm{v}, \mathrm{a}, \mathrm{a}_s, k + l)$
28: $\quad\quad\quad\quad \mathrm{L}_{k+l} \leftarrow \mathrm{L}_{k+l} \cup \{q\}$
29: $\quad\quad\quad\quad l \leftarrow l - 1$
30: $\quad\quad\quad\quad$ **if** $l = 0$ **and** $\mathrm{T} \neq \varnothing$ **then**
31: $\quad\quad\quad\quad\quad k \leftarrow \mathrm{b}_{|\mathrm{T}|}$
32: $\quad\quad\quad$ **else**
33: $\quad\quad\quad\quad m \leftarrow \mathsf{end}(\mathrm{v}^x_k) - \mathsf{begin}(\mathrm{v}^x_k)$
34: $\quad\quad\quad\quad i \leftarrow \mathsf{begin}(\mathrm{v}^x_k)$
35: $\quad\quad\quad\quad$ **for** $j \leftarrow 0$ **up to** $\mathsf{min}(m, l) - 1$ **do**
36: $\quad\quad\quad\quad\quad t = i + \mathsf{max}(m, l) + j$
37: $\quad\quad\quad\quad\quad \mathrm{L}_t \leftarrow \mathrm{L}_t \cup \{(\mathrm{v}^x_k, p^y)\}$
38: $\quad\quad\quad\quad\quad \mathsf{switch}(\mathrm{v}, \mathrm{a}, i + j, t)$
39: $\quad\quad\quad\quad k \leftarrow i - 1$
40: **return** $(\mathrm{L}_1, \ldots, \mathrm{L}_n)$

When sweeping points in v, given a stack T, every time a point in T is found in v, the information of how many points of T were found so far, l, is updated. This is performed in lines 20-23. These l points will be kept in consecutive positions in v, where the order among them is irrelevant. This is because when a point in v_k is found not to be in T and has v^x_k higher than $\mathsf{peek}(\mathrm{T})^x$, then to know which are the new output points, it is enough to know how many are the input sets that dominate the point (v^x_k, p^y).

Let q be the point at the top of the stack (the rightmost point in stack). If the final position of q in v is found, then q is added to L_t, where $t = k + l$, and then q is removed from the stack, and l is decremented (lines 24-29). If the stack is still not empty, but for now there is no point from it to place



(a) (b)

Figure 3.6: More two-dimensional examples of input sets where points have equal coordinates.

Table 3.1: Execution of Algorithm 17 for the example in Figure 3.6(b) when d_2 and b_2 are processed, after adding them to stack T (lines 19-39).

k	T	l	v	$v_k \in T$	$\mathsf{peek}(T)^x \geq v_k^x$	output points
7	(d_2, b_2)	0	$(f_1, e_1, c_1, g_1, d_2, a_1, b_2)$	T	-	-
6	(d_2, b_2)	1	$(f_1, e_1, c_1, g_1, d_2, a_1, b_2)$	F	F	$L_7 \leftarrow L_7 \cup \{(a_1^x, h)\}$
5	(d_2, b_2)	1	$(f_1, e_1, c_1, g_1, d_2, b_2, a_1)$	T	-	-
4	(d_2, b_2)	2	$(f_1, e_1, c_1, g_1, d_2, b_2, a_1)$	F	F	$L_6 \leftarrow L_6 \cup \{(w, h)\}$ $L_5 \leftarrow L_5 \cup \{(w, h)\}$
1	(d_2, b_2)	2	$(f_1, b_2, d_2, g_1, c_1, e_1, a_1)$	F	T	$L_3 \leftarrow L_3 \cup \{(b_2^x, h)\}$
1	(d_2)	1	$(f_1, d_2, b_2, g_1, c_1, e_1, a_1)$	F	F	$L_2 \leftarrow L_2 \cup \{(f_1^x, h)\}$
0	(d_2)	1	$(d_2, f_1, b_2, g_1, c_1, e_1, a_1)$	F	T	$L_1 \leftarrow L_1 \cup \{(d_2^x, h)\}$
0	$()$	0	$(d_2, f_1, b_2, g_1, c_1, e_1, a_1)$	-	-	-

($l = 0$), then the points in v between positions k and the current position of the current rightmost point in T ($b_{|T|}$) are not influenced by points in T, i.e., these points v_k are dominated in coordinate x by the same number of input sets as they were in the region immediately below p^y (line 30). Therefore, k is updated to $b_{|T|}$ which is the current position in v of the rightmost point in T, which allows to skip points in v with no interest without having to go through them. Figure 3.6(a) is an example where this happens and where it is necessary to avoid unnecessary operations. When d_2 and a_2 are being processed, after defining 4 as the final position in v of a_2, points b_1 and c_1 should not be swept since the remaining point in T (d_2) only has influence in values of coordinate x lower than d_1^x. Therefore, after placing a_1 and removing it from T, since $l = 0$ and T has d_2, k is updated to 1, which is the rightmost position of the remaining points in T.

If the current point being seen in v (v_k) is not in T but there are output points to be produced (lines 35-38), considering m as being the number of points with coordinate x equal to v_k^x, the first $\mathsf{min}(m, l)$ points with coordinate x equal to v_k^x in v, will switch position with the last $\mathsf{min}(m, l)$ points of T in v seen so far. This allows to add point (v_k^x, p^y) to L_t, where $t = i + \mathsf{max}(m, l), ..., i + \mathsf{max}(m, l) + \mathsf{min}(m, l) - 1$ (p^y is the value of coordinate y of the points in T).

To help understand how the algorithm works when the stack has more than one point, Table 3.1 shows the steps performed by Algorithm 17 in lines 19-39, when processing points b_2 and d_2 of the example of

Figure 3.6(b). For example, line 1 of Table 3.1 indicates that when the algorithm enters the first iteration of the while loop in line 19, k is equal to 7 and l is zero. The stack T contains (d_2, b_2) and v contains $(f_1, \underline{e_1, c_1, g_1}, d_2, a_1, b_2)$. Note that the consecutive points in v with equal x coordinates are underlined. Given this, it is explicit whether v_7 is in T, which is true ($b_2 \in$ T). It is also explicit whether the point at the top of the stack is to the left of the current point being seen in v (v_7), but since v_7 is in T, the answer to this question is irrelevant. Also in row 1, it is seen that no output points are produced in this iteration.

3.2.2 Complexity

The algorithm that computes the EAF in 2D sweeps each input point once (let us call it p) and for each point swept it might sweep n other points, the points that p switches places with, in v. Therefore, the time complexity of processing a point of X is directly related to the number of points in v to which p switches positions with. In Algorithm 14 it was shown that for problems where all points have distinct coordinates, the algorithm is output-sensitive. This is true because when a new point from X is processed, it is inserted in v and then only switches positions with another point in v if it generates a new output point (and the points generated are all correct). Moreover, each exchange between two points is performed in constant time.

Algorithm 17 is an algorithm based on Algorithm 14 but where input points with equal coordinates are admitted. In order to preserve the output-sensitivity property of Algorithm 14, Algorithm 17 has to maintain some properties from Algorithm 14. These algorithms differ in that Algorithm 14 sweeps points in X, while Algorithm 17 sweeps values of coordinate y in X, i.e., in each sweep, it considers all points with some specific value of coordinate y. Therefore, to preserve the output-sensitivity, Algorithm 17 has to meet 4 requirements as it sweeps through points with a value of coordinate y equal to h (let us call A the set of points with coordinate y equal to h):

1. The switch between two points in v has to be performed in constant time.
2. There can only be a constant number of switches between a point in A and a point in v where no output points are produced.
3. Every value w of coordinate x of points in v should only be swept if there is, at least, one output point equal to (w, h) ((w, h) belongs to, at least, one L_t, $t = 1, ..., n$). The exception is only allowed when the previous case is fulfilled.
4. For each value w of coordinate x of swept points in v, the number of points switched with x equal to w is equal to the number of output points equal to (w, h), i.e., the number of L_t in which (w, h) appears ($t = 1, ..., n$).

To deal with equal x coordinates, Algorithm 17 needs a way of keeping information about the positions in which the points with a specific value of x coordinate in v begin and end. This information can be

accessed and updated in constant time, which is required by requirement 1. Hence, it is only necessary to relate each point in v with some structure that indicates in which position the corresponding value of x begins and ends. Points with equal x coordinates are all related to the same structure. As a result, adding or removing a point from a set of points with equal coordinate x can be done in constant time, by modifying the starting or the end position in v of the structure related to that value of x. These structures (only the ones affected) are updated when a new point replaces an older one in v, and when a point is moved to another position, affecting only the structure related to the points to its left, to its right and/or its own. Therefore, updating this information is performed in a constant number of constant time operations, per switch.

Algorithm 16 sweeps points in X. Let us call the point currently being swept p. For each point p, considering that q is the point in v from the same set as p, a switch is made once between q and the leftmost point in v with value q^x as coordinate x (line 9) and then p substitutes q in v. Therefore, as soon as a point is selected to be processed, there is maximum of one exchange of two points, in order to guarantee that no wrong output points are produced and that no unnecessary exchanges between points are performed, as seen earlier for the example of Figure 3.5(a). This is true even in the case of Algorithm 17 which processes a set of points in X altogether (points in T with coordinate y equal to h). This agrees to requirement 2 because, for each point $p \in$ A, this is the only case where p switches with a point where no output point is produced.

When there is only one point with coordinate y equal to h, and this point is being swept, Algorithm 17 meets requirement 3, since, in this sense, it works in the same way as Algorithm 14 because it only sweeps a point v_k $(k = 1, ..., n)$ if it has a value of x of interest, i.e., if (v_k^x, h) is an output point. It differs from Algorithm 14 when there is more than one point that has coordinate y equal to h (let us consider the set of points in A), since all of points are processed together and points of v are swept until all points in A are in their final positions in v. In that case, there might be some position of v (v_k) where all points in A that are to its right are in their final position and the points in A to its left have other points from the same set that dominate (v_k^x, h), as it was seen in Figure 3.6(a). In this case, requirement 3 is guaranteed to be met by updating k when this situation is detected (line 30) to a value that is kept in b that indicates a position of v of interest.

Finally, requirement 4 is also assured by Algorithm 17. This is true because, given a set of points in v with coordinate x equal to w, there are as many switches of points as there are output points (w, h) produced (lines 35-38). Therefore, Algorithm 17 is an output-sensitive algorithm with $O(m \log m + mn)$ worst-case time complexity, and a $O(m \log m + m)$ time complexity in the best case.

The space complexity of the algorithm to compute the EAF in 2 dimensions is also output-sensitive, having $O(mn)$ in the worst case and $O(m)$ in the best case. This space complexity is achieved because, during the execution of the algorithm, it is only necessary to keep the m points in memory once in a queue (X). It is also necessary to keep a fix number of arrays of size n, for example v and other arrays that keep the information needed to know in which position of v, a value of x begins and ends. This space is always needed, independently of the resulting number of output points. Since m is always greater or

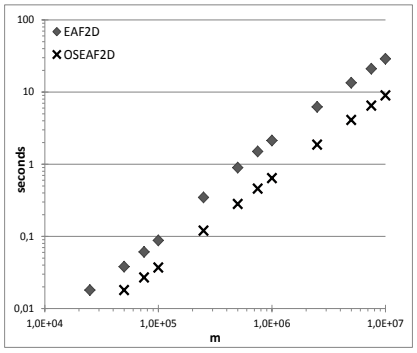
equal to n, the final space complexity needed by the algorithm is $O(m)$ plus the number of output points which are kept in memory until the algorithm terminates.

3.2.3 Experimental results

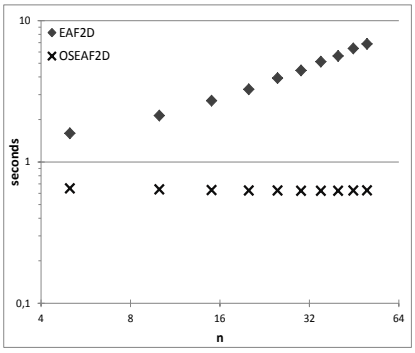
Algorithm 17 was implemented in C. In order to test if the implementation was correct, a few data sets were created. Among these data sets, tests were created to test specific parts, as, for example, limit situations as when points with equal coordinate are present in the input. These tests include, for example, the problems depicted in Figure 3.5 and 3.6, to check whether the program behaves as expected. Other data sets used to test the performance of the algorithm were also used to verify if the results were correct. To test if the outputs were correct, three methods were used. One method consisted of comparing the results of the implementation of Algorithm 17 against those produced by the implementation of the other algorithm available [11]. The other two methods exploited the properties of EAF, the idempotence, and the invariance with respect to the switch of dimensions.

To test the performance of the algorithm, three types of data sets were created, named worst-case data set, spherical data set and best-case data set. The worst case was constructed using the formula described in [11] where, given a number of input sets n and a number of input points m, a data set is produced whose corresponding output of the EAF has the maximum number of output points possible for those values of n and m ($l = (2m - n + 1)/2$). The best case was constructed using the output of the EAF for the worst case data set. This data corresponds to a data set where the output is equal to the input. Finally, the spherical data set was constructed using the following formula to generate the points of each input set: $p^i = 1 - |X^i|/||X||$, $X^i \sim \text{Normal}(0, 1)$, $i = 1, 2$. The number of output points for this type of set is approximately half of the maximum number of output points possible.

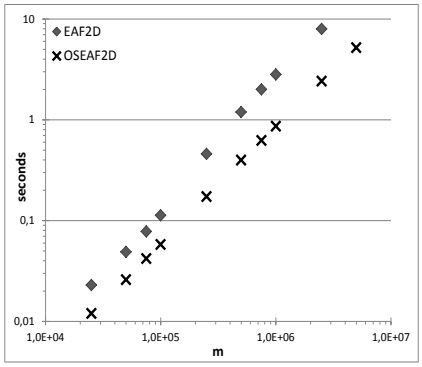
For each data set type, two sets of tests were constructed, one having n constant and varying m and the other maintaining m constant and varying n. The implementations of both algorithms were compiled using gcc version 4.5.1. Tests were performed on an Intel Core 2 Duo P8600, with 2.40GHz and 3MB cache size. Figure 3.7 shows the performance of the output-sensitive algorithm presented here (named OSEAF2D) against the other algorithm known to compute the EAF in two dimensions [11] (named EAF2D). For each test, an execution of each algorithm was considered. The results show that the output-sensitive algorithm is in general faster than the other algorithm. In general, the output-sensitive algorithm, becomes even more efficient compared to the other algorithm as n grows when m is constant. One important fact that is clear in Figure 3.7(b) is the output-sensitivity of the algorithm proposed here, since in the best case data set where the number of input sets (n) has no influence on the number of output points, when fixing the number of input points and varying n, the run time does not vary, as expected. Moreover, this fact results also as a way of validating the output-sensitivity of the algorithm in the presence of input points with equal coordinates. This is because the best-case data set was constructed as the output of the EAF for the worst-case data set, therefore corresponding to a data set where each distinct value of coordinate x and y in each test appears once in each input set, i.e., each value is repeated n times.



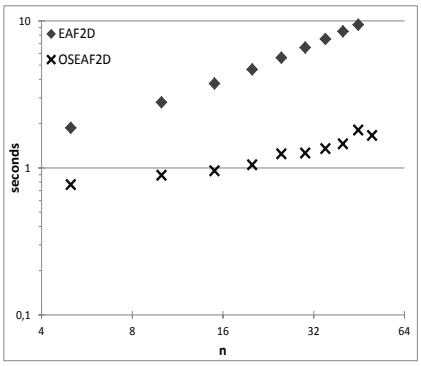
(a) n=10



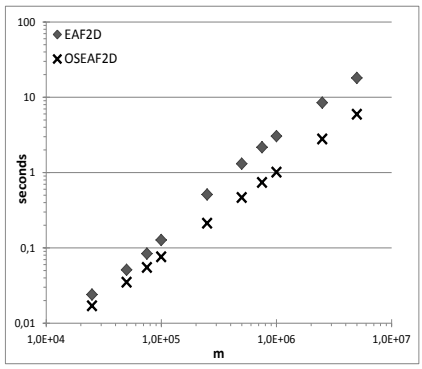
(b) m=10^6



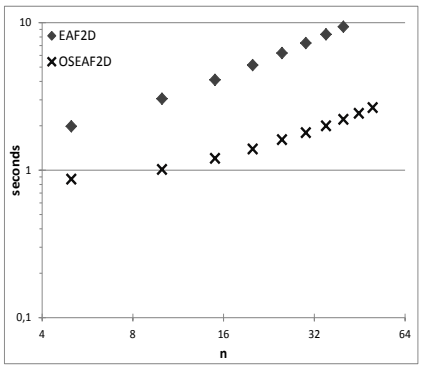
(c) n=10



(d) m=10^6



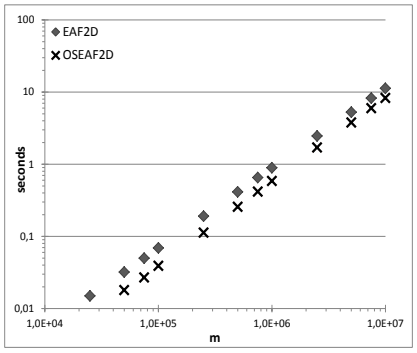
(e) n=10



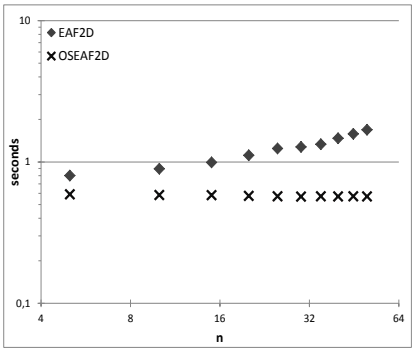
(f) m=10^6

Figure 3.7: Results for the best-dcase data set (a and b), for the spherical data set (c and d) and for the worst-case data set (e and f).

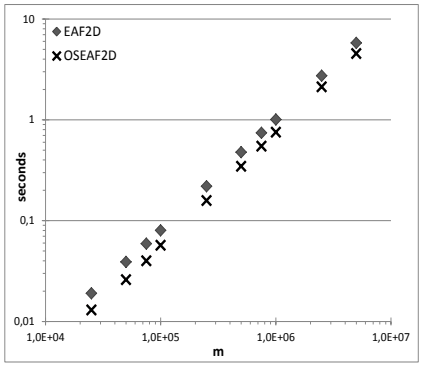
Further experiments were made to try to improve algorithm performance. As it will be seen here and for the algorithms in the following Chapter, the way in which points are kept in memory has influence on the algorithm's performance. Since algorithms to compute the EAF (and also the hypervolume indicator) usually sweep points according to, at least, one dimension, keeping points in memory sorted in the same way as they will be swept (and in consecutive memory positions), instead of just sorting references to the points (which probably will not be in consecutive memory position and will not be in any particular order), might be enough to improve significantly the algorithm's performance. This was implemented in



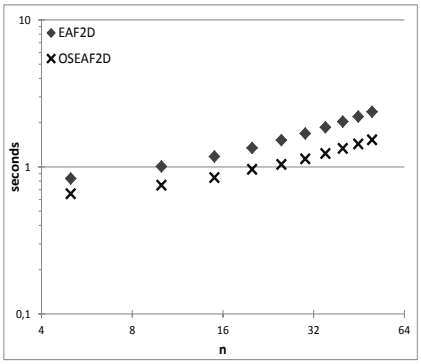
(a) n=10



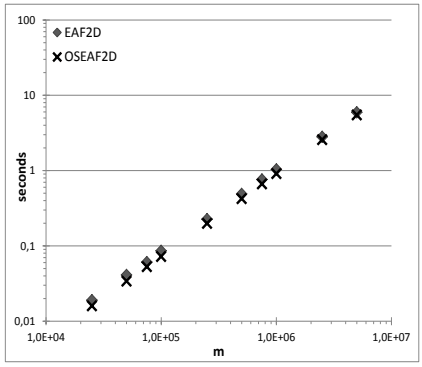
(b) m=10^6



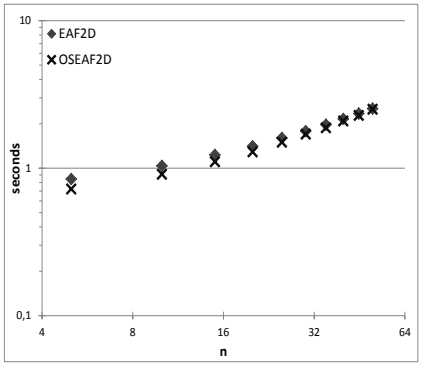
(c) n=10



(d) m=10^6



(e) n=10



(f) m=10^6

Figure 3.8: Performance of the improved version of the algorithms. Results for the best-case data set (a and b), for the spherical data set (c and d) and for the worst-case data set (e and f).

both algorithms, the output-sensitive algorithm described here (OSEAF2D) and the previously published algorithm [11] (EAF2D), by ordering in memory the input points in ascending order of y-coordinate.

Figure 3.8 shows the performance of the improved algorithms. It is possible to observe that in the case of the output-sensitive algorithm, this modification has little effect. On the opposite, the performance of the other algorithm (EAF2D) improved significantly. It became up to 10 times faster than the version without this improvement. It is possible to see that EAF2D became faster and in many cases is almost as fast as OSEAF2D, as it is the case for the spherical and worst case data sets. It also obtained good

results in the best case data set. The explanation for the fact that this improvement had a big influence in EAF2D but not in the OSEAF2D is that, in the output-sensitive algorithm, points are swept according to coordinate y only once, while EAF2D sweeps points according to coordinate y, exactly n times. Therefore, this improvement has greater impact on the latter.

Figure 3.9 shows this improvements in terms of cache-misses (the Linux tool perf was used in order to gather this information), where EAF2DS and OSEAF2DS correspond, respectively, to the versions of EAF2D and OSEAF2D, where points are previously sorted in memory. This figure shows that, with this improvement, EAF2DS uses the cache much more efficiently than the previous version (EAF2D), i.e., when EAF2DS algorithm sweeps to the next point, since it is next to the current point in memory, it is probably already loaded in cache, therefore, not occurring a cache-miss and not having to fetch that missed point. On the other hand, Figure 3.9 shows that OSEAF2D already uses cache efficiently, since

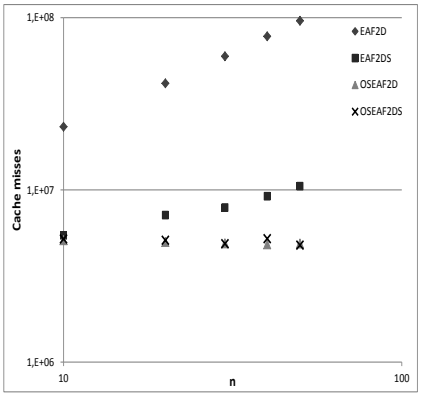


Figure 3.9: Number of cache misses of EAF algorithms for $d = 2$, in best case data set where $m = 1e^6$

the number of cache-misses does not vary much in comparison to OSEAF2DS. This fact is one more advantage of the output-sensitive algorithm.

Comparing both algorithms for the EAF in two dimensions, the one proposed here and the one published in [11], the latter has the advantage of allowing each front to be computed separately. On the other hand, in spite of both algorithms sharing the same worst-case time complexity, $O(m \log m + mn)$, the the new algorithm being output-sensitive allows it to adapt to the type of the input problem and to be more efficient on simpler problems. The experiments presented here show that output-sensitivity can be very important in problems such as the EAF, where algorithm complexity depends especially on the number of output points, allowing the algorithms to perform faster when the input does not result in the worst case. For problems in two dimensions, this property already has a positive and significant impact in algorithm performance. Furthermore, it was shown that this algorithm also has the advantage of using cache efficiently. Therefore, algorithms of this type for more dimensions would probably have an even greater impact. Moreover, the ideas used in this new algorithm can also be useful in the development of other algorithms to compute the EAF in more than 2 dimensions.

3.3 Concluding remarks

In this Chapter, two contributions were made. Firstly, two $O(n^2m^2 \log m)$ time algorithms were proposed to compute the EAF in four dimensions, a problem for which no algorithm was available to date. Moreover, a new output-sensitive algorithm to compute the EAF in two dimensions (OSEAF2D) is proposed.

The algorithms to compute the EAF in four dimensions, besides being the first algorithms for the four dimensional case, also establish $O(n^2m^2)$ as an upper bound on the number of output points, and $O(n^2m^2 \log m)$ as an upper bound on the time complexity of the four-dimensional case. Between the two algorithms proposed, it was seen that IEAF4D outperforms the base version (EAF4D) most of the times, being up to twice as fast.

The proposal of the OSEAF2D, besides improving upon the current state-of-the-art algorithm (EAF2D), highlights the importance of developing output-sensitive algorithms to compute the EAF as a way of achieving performance which depends on actual input sets and side-stepping the fast growth of the number of output points in the worst case. Another feature of the output-sensitive algorithm is that it is already cache efficient, drawing attention to the fact that algorithms to compute the EAF might benefit from having points previously sorted in memory in order to become more cache efficient.

Comparing the OSEAF2D with EAF2D, the former has the advantage of being more efficient when computing all output sets L_t ($t \in \{1, ..., n\}$), especially when the input does not approach the worst case, while the latter has the advantage of allowing just a few output sets to be computed without having to compute all of them, as OSEAF2D does. Moreover, OSEAF2D may turn out to be helpful when developing algorithms for higher numbers of dimensions, whether output-sensitive or not.

All of the available algorithms to compute the EAF, except for the state-of-the-art algorithm for two dimensions, follow the same abstract idea. They sweep points according to the last dimension d and compute m EAF in $(d-1)$ dimensions. In the case of the algorithms for four dimensions, they have to compute the whole EAF in $(d-1)$ dimensions from scratch, whereas the OSEAF2D and the algorithm to compute the EAF for three dimension use an incremental method, i.e., compute the EAF in $(d-1)$ dimensions starting from the EAF in $(d-1)$ dimensions from the previous iteration. Although this method can be generalized to more than four dimensions, it will probability have a complexity worse than required, which may compromise its usefulness in practice.

4 Hypervolume Indicator

Two new algorithms to compute the hypervolume indicator are proposed in this Chapter. The first one, in Section 4.1, is a dimension-sweep algorithm which establishes a new, quadratic, upper bound on the time complexity of the four-dimensional case, and is also very fast in practice. The second algorithm implements a divide-and-conquer approach to the computation of the hypervolume indicator in the three-dimensional case, and is described in Section 4.2. This algorithm is proposed as a first step towards the development of a full multi-dimensional divide-and-conquer algorithm for any number of dimensions, and is shown to be asymptotically optimal like the existing algorithm of Beume *et al.*. [5]. The Chapter concludes with a brief discussion of the implications of the work presented for the development of new, efficient algorithms for the hypervolume indicator.

4.1 New dimension-sweep algorithm for the four-dimensional case

In this Section, a quadratic-time algorithm to compute the hypervolume indicator in four dimensions is proposed. Like the WFG algorithm [18], the new algorithm follows a dimension-sweep approach, and implements an iterated incremental computation of the hypervolume indicator. However, the resulting time-complexity upper bound is better than the currently established upper bound of $O(n^2 \log n)$ for the four-dimensional case [4, 12].

The next Subsection presents the main ideas behind the proposed algorithm. An algorithm to compute the individual contribution of a point in three dimensions, on which the main algorithm relies, is described in Subsection 4.1.2. A description of the data structures used by the algorithms and a detailed description of their operation are given in Subsections 4.1.3 and 4.1.4. Subsection 4.1.5 discusses in detail how the $O(n^2)$ time complexity is achieved and, finally, experimental results are presented in Subsection 4.1.6.

4.1.1 General description

The proposed algorithm operates by sweeping the input points in ascending order of their fourth coordinate values. For each point swept, the hypervolume of a slice bounded below by the current point and bounded above by the next point to be swept is computed. The hypervolume of a slice can be determined by multiplying its height (the absolute difference between the fourth coordinate values of the next point and of the current point) by the volume produced by the projection of the points swept so far, including

Algorithm 18 HV4D - Main algorithm

Input: X // a set of n points in $\mathbb{R}^4$
Input: r // $r \in \mathbb{R}^4$ is the reference point
Output: h // Total hypervolume

```
1: Q is a queue containing X ∪ {(−∞, −∞, −∞, r^w)} sorted in ascending order of coordinate w
2: S* ← ∅
3: v ← 0
4: h ← 0
5: p ← pop(Q)
6: while Q ≠ ∅ do
7:     v ← v + contribution(p*, S*, r*)
8:     S* ← S* ∪ {p*}
9:     q ← pop(Q)
10:    h ← h + (q^w − p^w) · v
11:    p ← q
12: return h
```

the current point, onto the (x,y,z)-space. This volume can be calculated by adding the individual contribution of the point being swept to the volume associated with the previous slice. Therefore, the problem of computing the hypervolume of a slice reduces to the simpler problem of computing the volume of the region exclusively dominated by a point in three dimensions. Algorithm 18 details this procedure, where $\mathsf{contribution}(p^*, \mathrm{S}^*, r^*)$ denotes the contribution of a point p^* to a non-dominated point set S^*, given a reference point r^*, all in three-dimensional space. As before, the * is used here to denote the projection onto $(d-1)$-dimensional space of (sets of) points in d-dimensional space ($d = 4$ in this case).

Algorithm 18 can be related to the WFG algorithm [18], which has a similar structure and works for any number of dimensions. The main difference between the two algorithms consists of the way in which individual contributions are defined and computed, which leads to different time complexities for each one.

In Algorithm 18, it is possible to observe that sweeping through every point is performed in $O(n)$ steps, and that, in order to obtain $O(n^2)$ time complexity for this algorithm, the contribution of each point must be computed in (amortized) $O(n)$ time. Although it is not clear how to achieve this when S^* includes points dominated by p^*, Algorithm 18 may be modifed to guarantee that individual contributions are computed while involving only non-dominated points, as detailed in Algorithm 19.

Denoting by T^* the set of all points in S^* which are dominated by p^*, Algorithm 19 computes the contribution of p^* to the set S^* by determining the contribution of the set T^* to the set $\mathrm{S}^* - \mathrm{T}^*$ first, and then subtracting it from the volume dominated by S^* before computing the contribution of p^* to the set of points $\mathrm{S}^* - \mathrm{T}^*$. On the example in Figure 4.1(a), Algorithm 19 proceeds by first removing point q_1^* from S^*, then calculating the contribution of q_1^* to $\mathrm{S}^* - \{q_1^*\}$, and finally subtracting it from volume v before computing the contribution of p^* to $\mathrm{S}^* - \{q_1^*\}$, which includes the volume previously dominated exclusively by q_1^*. Note that, since each point is added to and removed from S^* at most once, the total number of calls to $\mathsf{contribution}()$ remains $O(n)$. Algorithm 19 can also be seen as a generalization of Beume *et al.*'s algorithm [5] to four dimensions.

Algorithm 19 HV4D - Main algorithm – second version

Input: X // a set of n points in $\mathbb{R}^4$
Input: r // $r \in \mathbb{R}^4$ is the reference point
Output: h // Total hypervolume

```
1: Q is a queue containing X ∪ {(−∞, −∞, −∞, r^w)} sorted in ascending order of coordinate w
2: S* ← ∅
3: v ← 0
4: h ← 0
5: p ← pop(Q)
6: while Q ≠ ∅ do
7:    for all s* ∈ S* : p* ≤ s* do
8:       S* ← S* − {s*}
9:       v ← v − contribution(s*, S*, r*)
10:   v ← v + contribution(p*, S*, r*)
11:   S* ← S* ∪ {p*}
12:   q ← pop(Q)
13:   h ← h + (q^w − p^w) · v
14:   p ← q
15: return h
```

4.1.2 Individual contribution of a point in three dimensions

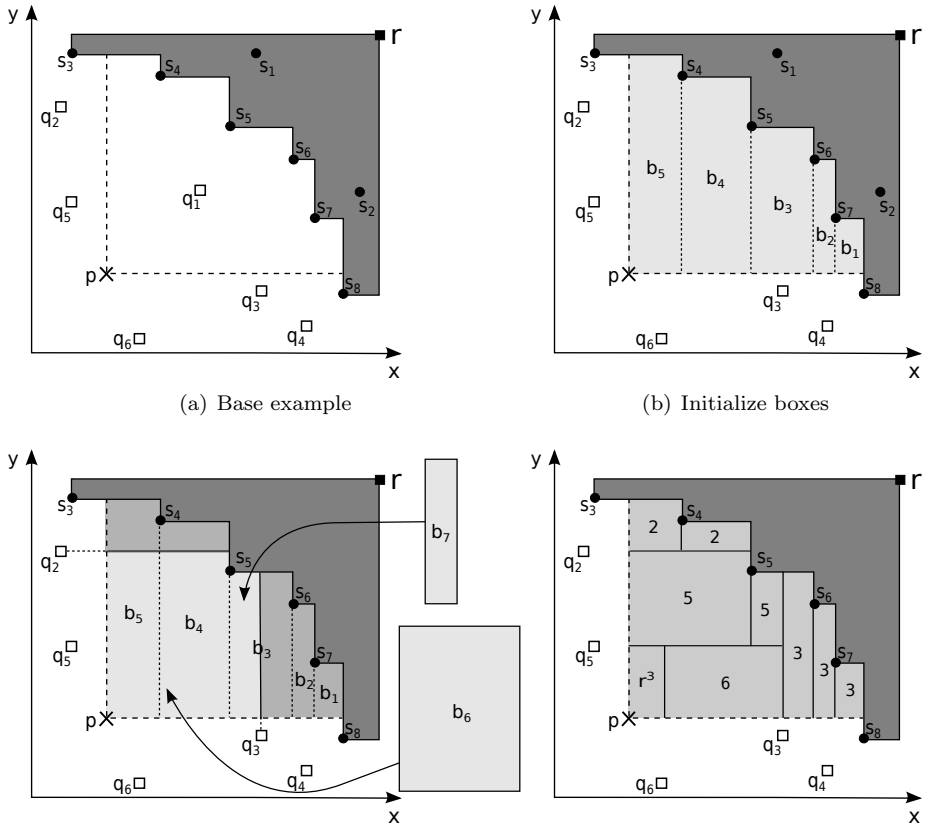
In order to achieve $O(n)$ time complexity in the computation of the contribution of a single point to a non-dominated point set S^* in the conditions stated above (i.e., $p^* \in \mathbb{R}^3$, $\mathrm{S}^* \subset \mathbb{R}^3$, and $\nexists q^* \in \mathrm{S}^* : p^* \leq q^*$), a method inspired in Emmerich and Fonseca's algorithm [9] to compute all individual contributions given a set of non-dominated points in three dimensions is proposed.

Given a non-dominated point set $\mathrm{S} \subset \mathbb{R}^3$ and a reference point $r \in \mathbb{R}^3$, the method proposed by Emmerich and Fonseca [9] computes the contribution of each point p in S to the set $\mathrm{S} - \{p\}$. The volume dominated exclusively by each point is divided into cuboids (or boxes), and the sum of their volumes is computed. This is done by sweeping through every point p in S in ascending order of the third coordinate.[1] For each of these points, the region of the (x,y)-plane at $z = p^z$ which is exclusively dominated by p is partitioned into smaller non-overlapping rectangular areas. This partitioning can be obtained by sweeping those points that have coordinate z lower than p^z along one of the dimensions x or y.

The example in Figure 4.1 will be used throughout the remainder of this Section to illustrate the computation of single-point contributions. The base problem is depicted in Figure 4.1(a), and the corresponding division into rectangles is shown in Figure 4.1(b). After this division, the height of each box is determined by sweeping the points with z coordinate higher than p^z in ascending order of that coordinate. In that process, some boxes might be closed and new boxes might be added. One important aspect is that the boxes that need to be updated can be efficiently accessed, in constant time.

In contrast with the algorithm of Emmerich and Fonseca [9], in the present algorithm it is sufficient to update the height of the boxes corresponding to the single point whose contribution is to be determined. Ignoring the presence of q_1 in the example of Figure 4.1(a), as it would have been removed in a previous

[1] Note that, in [9], maximization is assumed. For clarity and consistency, the description here considers minimization problems instead.



(a) Base example

(b) Initialize boxes

(c) Simulate closeBoxesLeft and closeBoxesRight

(d) Expected result

Figure 4.1: Example of a problem in 3 dimensions, where the goal is to determine the contribution of p to S ($\mathrm{S} = \{q_1, ..., q_6\} \cup \{s_1, ..., s_8\}$), given the reference point r. In this problem, $s_t^z \leq p^z$ $(t = 1, ..., 8)$ and $q_i^z > p^z$ $(i = 1, ..., 6)$. It is assumed that $p^z = 0$ and $q_i^z = i$.

Algorithm 20 HV4D - contribution

Input: $p \in \mathbb{R}^3$
Input: $\mathrm{S} \subset \mathbb{R}^3$
Input: $r \in \mathbb{R}^3$ // The reference point
Output: c // contibution
1: $\mathrm{S}_1, \mathrm{S}_2 \leftarrow \mathsf{split}(\mathrm{S}, p^z)$ // $\mathrm{S}_1 = \{q \mid q \in \mathrm{S} : q^z \leq p^z\}$, $\mathrm{S}_2 = \{q \mid q \in \mathrm{S} : q^z > p^z\}$
2: $\mathrm{B} \leftarrow \mathsf{createBoxesBase}(p, \mathrm{S}_1)$
3: $c \leftarrow \mathsf{determineBoxesHeight}(p, \mathrm{S}_2, \mathrm{B}, r)$
4: **return** c

step, the contribution of p would be computed as the sum of the volumes of the boxes depicted in Figure 4.1(d), where the numbers indicate their heights.

The main steps of the computation of the contribution of a point $p \in \mathbb{R}^3$ to a set $\mathrm{S} \subset \mathbb{R}^3$, as described above, are detailed in Algorithm 20. All of them can be implemented in $O(n)$ time, as long as S is a non-dominated point set and there are no points in S which are dominated by p, which is guaranteed to happen by Algorithm 19. Furthermore, points must be kept sorted with respect to dimensions two, in order to delimit the base of the boxes (see Figure 4.1(b)), and dimension three, to allow their heights to be determined.

4.1.3 Data structures

Algorithm 19 receives a non-dominated point set $\mathrm{X} \subset \mathbb{R}^4$ as input, and sets up a queue Q containing all points in X in ascending order of the fourth coordinate, w. A sentinel is added to Q in order to ensure that point q in line 12, which is used to determine the height of the slice, always exists. During the execution of Algorithm 19, S^* is updated. S^* is stored in a data structure that maintains all points sorted in ascending order of coordinates y and z, using two doubly linked lists. These sorted lists are maintained also for the two subsets S_1 and S_2 of S in Algorithm 20, and support the following operations:

next$^y(p, \mathrm{S})$ The point following p in S with respect to coordinate y

next$^z(p, \mathrm{S})$ The point following p in S with respect to coordinate z

higher$^y(p, \mathrm{S})$ The point $q \in \mathrm{S}$ with the least $q^y > p^y$

higher$^z(p, \mathrm{S})$ The point $q \in \mathrm{S}$ with the least $q^z > p^z$

getXRightBelow(p, S) The point $q \in \mathrm{S}$ with the least $q^x \geq p^x$ such that $q^y \leq p^y$

Operation next is performed in constant time as long as p is itself in S, while the remaining ones are performed in linear time.

In Algorithm 20, the volume exclusively dominated by a point is partitioned into cuboids, here referred to as boxes. Each box b is defined by its lower corner (l^x, l^y, l^z) and its upper corner (u^x, u^y, u^z). Boxes are kept in a doubly-linked list (B) in order to make it possible to easily access those that need to be updated or removed. Since there is no overlap between boxes in the list, it is possible to keep the list of boxes sorted in ascending order of coordinate x. When a box is created, only (l^x, l^y, l^z) and (u^x, u^y) are known. Boxes are kept in the list as long as their upper limit u^z is not known. When this value is determined, the box is closed, i.e., its hypervolume is calculated, and the box is removed from the list. In order to manage the list of boxes, the following operations are implemented:

pushLeft(B, b) Add box b to the left of the list of boxes B

closeAllBoxes(B, z) Close all boxes in list B, setting the corresponding value of u^z to z and returning the sum of the volumes of those boxes.

closeBoxesLeft(B, y, z) From left to right, close all boxes in list B for which $u^y > y$, setting the corresponding value of u^z to z and l^y to y. After closing those boxes, push to the left of B a new box whose lower corner coincides with p, and has $u^y = y$ and u^x equal to the u^x of the last box removed. Finally, return the total volume of the closed boxes.

closeBoxesRight(B, x, z) From right to left, close all boxes in list B for which $u^x > x$, setting their u^z to z. If the last removed box is such that $l^x < x$, l^x is updated to x before closing it, and a new box is pushed to the right of B with the same corners, but with u^x set to x. Return the total volume of the closed boxes.

Algorithm 21 HV4D - contribution - createBoxesBase

Input: $p \in \mathbb{R}^3$
Input: $\mathrm{S}_1 \subset \mathbb{R}^3$ // $\forall q \in \mathrm{S} \Rightarrow q^z \leq p^z$
Input: $r \in \mathbb{R}^3$ // The reference point
Output: B // boxes list
1: $\mathrm{S}_1 \leftarrow \mathrm{S}_1 \cup \{(r^x, -\infty, -\infty), (-\infty, r^y, -\infty)\}$
2: $\mathrm{B} \leftarrow \varnothing$
3: $q \leftarrow \mathsf{higher}^y(p, \mathrm{S}_1)$
4: $m \leftarrow \mathsf{getXRightBelow}(p, \mathrm{S}_1)$
5: **while** $q^x > p^x$ **do**
6: **if** $q^x < m^x$ **then**
7: $b \leftarrow ((q^x, p^y, p^z), (m^x, q^y, p^z))$
8: $\mathsf{pushLeft}(\mathrm{B}, b)$
9: $m \leftarrow q$
10: $q \leftarrow \mathsf{next}^y(q, \mathrm{S}_1)$
11: $b \leftarrow ((p^x, p^y, p^z), (m^x, q^y, p^z))$
12: $\mathsf{pushLeft}(\mathrm{B}, b)$
13: **return** B

Operation pushLeft is performed in constant time. Operation closeAllBoxes is performed in k steps, and the remaining operations in $k + 1$ steps. Therefore, all have a cost of $O(k)$, where $k \leq n$ represents the number of boxes removed, and n represents the total number of points.

4.1.4 Detailed description

Algorithm 19 sweeps every point p in Q and determines the contribution of its projection on (x,y,z)-space, p^*, to the volume dominated by S*. This may cause the removal of points in S* that are dominated by p^*. Point removal can be performed in constant time, but requires the computation of the corresponding contributions, as well. After computing the contribution of p^*, it is added to S* while keeping the lists used to maintain S* sorted in ascending order of both y and z coordinates, which can be implemented in linear time. Furthermore, Algorithm 19 guarantees that, when calculating the contribution of any point p^*, all points in S* are kept sorted in ascending order of coordinates y and z, and no point in S* is dominated by any other point in S* or by p^* itself. As described before, these constraints allow the contribution of a point to be computed in linear time.

Algorithm 20 computes the 3-dimensional contribution of p to the set of points in S. The computation consists of two parts: the bases of an initial set of boxes are determined first (Algorithm 21) and then box heights are found (Algorithm 22).

To determine the bases of the boxes, the points in S whose z coordinate is lower than or equal to p^z (S$_1$) and which are dominated by p with respect to the x and y coordinates, but not by any other point in S$_1$, are swept (points $s_4, ..., s_7$ in Figure 4.1(a)). Boxes are created from right to left by sweeping points in S$_1$ in ascending order of coordinate y, starting from point $\mathsf{higher}^y(p, \mathrm{S}_1)$ (s_7), which is the lowest point in S$_1$ higher than p^y, and stopping when a point to the left of p is found (s_3). Note that such points always exist because of the presence of the sentinels $(r^x, -\infty, -\infty)$ and $(-\infty, r^y, -\infty)$, although this is not represented in Figure 4.1. All points between the starting (s_7) and the end (s_3) points that do not

Algorithm 22 HV4D - contribution - determineBoxesHeight

Input: $p \in \mathbb{R}^3$
Input: $\mathrm{S}_2 \subset \mathbb{R}^3$ // $\forall q \in \mathrm{S} \Rightarrow q^z > p^z$
Input: B is a list of boxes
Output: c // contribution

```
1: S_2 ← S_2 ∪ {(−∞, −∞, r^z)}
2: q ← higher^z(p, S_2)
3: while not empty(B) do
4:    if q^x ≤ p^x then
5:       if q^y ≤ p^y then
6:          c ← c + closeAllBoxes(B, q^z)
7:       else
8:          c ← c + closeBoxesLeft(B, q^y, q^z)
9:    else
10:      c ← c + closeBoxesRight(B, q^x, q^z)
11:   q ← next^z(q, S_2)
12: return c
```

fulfill all the above conditions are skipped (s_2). Each of the points that satisfy the above conditions (s_7, s_6, s_5, s_4) defines l^x and u^y of a box as well as u^x of the next box. For example, in Figure 4.1(b), point s_5 defines l^x and u^y of box b_3 and u^x of box b_4. The value of u^x for the first and rightmost box created is determined by getXRightBelow(p) (s_8). Finally, the end point (s_3) only defines u^y of the last and leftmost box. In the example of Figure 4.1(a), after executing the first part of the algorithm, the list of boxes contains $b_1, ..., b_5$ as depicted in Figure 4.1(b).

The next step consists of determining the height of the boxes, updating and closing them (Algorithm 22), which may imply shrinking the base of one or more boxes and closing them, and then adding another box. For this purpose, only points with a value of coordinate z higher than p^z (S_2) are needed ($q^2, ..., q^6$). Therefore, points in S_2 are swept in ascending order of coordinate z as long as there are still boxes to be closed. While processing each point q, there are three cases to consider, depending on its projection on the (x, y)-plane:

1) q is to the left of and above p (e.g. q_2, q_5)

2) q is to the right of and below p (e.g. q_3, q_4, q_6)

3) q dominates p (e.g. the sentinel $(-\infty, -\infty, r^z)$ which is not represented in Figure 4.1)

Note that q is never dominated by p on the (x,y)-plane, because it would also be dominated in (x, y, z)-space in that case, but those points were previously removed in Algorithm 19.

Case 1 causes the algorithm to call function closeBoxesLeft. Looking at q_2 in Figure 4.1(c), it is possible to observe that boxes are popped from the left until a box whose upper corner is not dominated by q_2 is found, and are closed only partially by updating their lower corner and determining the z coordinate value of the upper corner which is q_2^z in this case (darker area of boxes b_4 and b_5). The sum of the volumes of the closed boxes is added to c. In order to account for the area which becomes uncovered due to the shrinking of the boxes before they are closed (lighter area), a new box is inserted to the left of the list of boxes (b_6) covering that area.

Case 2 causes the algorithm to call function closeBoxesRight, which is what happens while processing point q_3 in the example. This function closes all boxes that are to the right of q_3, popping them from the right, and setting the z coordinate of their upper corner to q_3^z. If there is any box whose upper corner is dominated by p on the (x, y)-plane, but whose lower corner is not (b_3), the box is updated by setting coordinate x of its lower corner to q_3^x, and then closed. A new box is added to the right of the box list (b_7), covering the area left unaccounted for by the last box closed. The sum of the volumes of all boxes closed is added to c.

Finally, the last case causes the algorithm to call closeAllBoxes to close all remaining open boxes. The z coordinate of their upper corners is set to q^z (in the example, this is equal to r^z, due to the presence of the sentinel). The volume of all of the closed boxes is added to c and the algorithm terminates returning the total volume of the region exclusively dominated by p. Figure 4.1(d) shows the expected result, i.e., all the boxes closed and their corresponding heights.

4.1.5 Complexity

Starting with Algorithm 20, it is not difficult to show that its complexity is $O(n)$. The reasoning is that splitting S in two subsets (S_1 and S_2) and each of the two main stages of the algorithm can be performed in $O(n)$ time. Regarding the box initialization stage, note that for each point in S_1 with coordinate y higher than p^y, of which there are at most n points, at most one box is created (in constant time). Therefore, $O(n)$ complexity is achieved. The second stage processes all points with coordinate z higher than p^z, which are also at most n points. For each of these points, k boxes are closed, $k \in [0, n]$. Moreover, at most one box is created, which can happen only if at least one box is closed. Note that if there are t points with the third coordinate lower or equal to p^z, then, the first stage can create up to t boxes, while the second step can create at most $n - t$ boxes, which gives a total of up to n boxes created. Therefore, the maximum number of closed boxes is also n. Independently of which function is used to close boxes (closeBoxesRight, closeBoxesLeft or closeAllBoxes) k steps are performed if boxes are only closed, or $k + 1$ steps, if is also created a box, leading to $O(k)$ cost either way. Therefore, the total cost of the operations of Algorithm 20 amortizes to $O(n)$.

Algorithm 19 sweeps through n points and, for each one, it determines the highest points in S lower than p according to coordinates y and z, in order to keep the lists of points associated to S^* sorted, at a cost of $O(n)$. Moreover, for each point swept, it computes the contribution of k dominated points and the contribution of the point being swept. Note that each point of the initial set X is added to S^* and removed from it at most once. Therefore, the algorithm computes a maximum of $2n$ contributions, each at a cost of $O(n)$, which results in $O(n^2)$ amortized time complexity.

4.1.6 Experimental results

In this Section, the performance of the algorithm to compute the hypervolume indicator in four dimensions, described earlier (here referred to as HV4D) is compared against the state-of-the-art algorithms.

IIHSO is the fastest algorithm known for the four-dimensional case, but unfortunately it was not possible to compare HV4D directly against IIHSO, for the lack of an available implementation. Therefore, in order to infer how HV4D compares to IIHSO, the same experiments performed by While *et al.* [18] were reproduced on a similar machine. This allows HV4D to be compared directly against the algorithms tested by While *et al.* [18], which then allows the performance of HV4D and IIHSO to be compared indirectly, based on the referred observation that IIHSO is up to 2.5 times faster than WFG, on the data sets used.

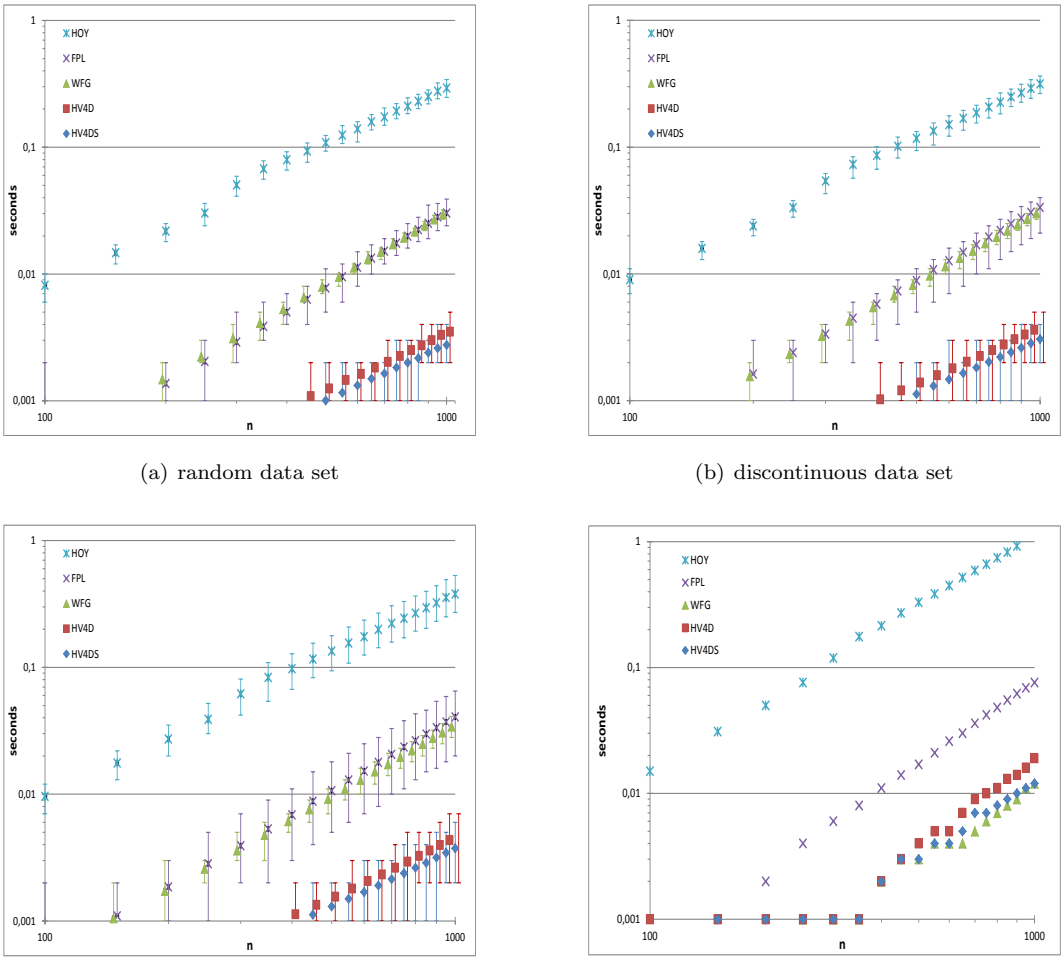
Previous experiments have shown that presorting points in memory may have a good impact on dimension sweep algorithm performance. HV4D seems to be sensitive to this feature and, therefore, two C implementations were tested: HV4D and HV4DS, where the latter includes the presorting of points in memory. Previous testing showed that, for four dimensions, the other algorithms considered did not seem to be sensitive to this feature, or at least, any gain in runtime did not seem to compensate the amount of extra work needed to sort points in memory. Therefore, the original versions of HOY, FPL (version 1.3) and WFG were used. The implementations of all algorithms were compiled using gcc version 4.6.0.

The experiments were performed on an Intel Core 2 Duo P8600, with 2.40GHz and 3MB cache size. The data sets used were the spherical, random and discontinuous data sets made available by the Walking Fish Group[1]. Each data set contains several fronts with sizes ranging from 100 to 1000 points. There are 20 fronts of each size in each data set. Each point in the plots depicted in Figure 4.2 represents the average runtime on a front of a specific size, taking into account all objective permutations and all 20 fronts. The error bars represented in those plots show the minimum and maximum observed runtime in each case. In order to avoid overlapping error bars and to make them more visible, point markers were shifted slightly to the left or to the right in each case.

A fourth data set is considered, which will be referred to as the "hard" data set. This is a data set designed to make S^* grow as much as possible. In this case, every time a point's contribution is computed, HV4D has to sweep all points in memory. This is a set of points that, when sorted in ascending order of the fourth coordinate, is also sorted in ascending order of the first coordinate and in descending order of the second and third coordinates. In the "hard" case data set only one front of each size, and the original objective order are considered.

Figure 4.2 shows that comparing HV4D and HV4DS, the latter can be up to two times faster. This improvement is more evident in the "hard" case data set. Regarding the remaining algorithms, HOY is the slowest, as expected. FPL and WFG have similiar behavior in the random, discontinuous and spherical data sets, where WFG was a little faster than FPL. Comparing the performance of HV4DS against WFG on the three WFG data sets, it is observed that HV4DS can be up to 10 times faster than WFG. It is also possible to observe that the observed maximum runtime for HV4DS is still up to 2.5 times lower than the minimum runtime achieved by WFG. Knowing that the fastest algorithm so far for the four-dimensional case (IIHSO) is up to 2.5 times faster than WFG in these data sets, it is possible to

[1] http://www.wfg.csse.uwa.edu.au/hypervolume/index.html#data



(a) random data set

(b) discontinuous data set

(c) spherical data set

(d) "hard" case data set

Figure 4.2: Comparison of the performance of HV4D, HV4DS, WFG, FPL and HOY. For each algorithm the minimum, average and maximum runtime for a given front size in each data set is depicted: random (a), discontinuous (b), spherical (c), and the "hard" case (d).

conclude that HV4DS is at least as fast as IIHSO. HV4DS is expected to be 4 times faster than IIHSO on these data sets.

Looking at the results on the "hard" data set, as expected, HV4D and HV4DS run more slowly than on the other data sets of similar sizes. Looking at the performance of WFG, it can be seen that the "hard" case data set is actually an easy data set for WFG. In this case, WFG achieves a performance similar to HV4DS, and is faster than HV4D. In this case, nothing can be concluded with respect to IIHSO performance, since there is no information about how IIHSO behaves on this specific data set.

It is also possible to observe that the runtime of all algorithms varies with different fronts of the same size and/or different objective order, and that WFG seems to be the one whose runtime varies less. Not much can be said with respect to the runtime variation of HV4D and HV4DS based on these experiments because the algorithms are so fast that their runtime is close to clock accuracy, which is 1 ms on the machine used to perform the experiments.

Based on the experimental data presented, it should be reasonable to claim that HV4DS is now the fastest algorithm for computing the hypervolume indicator in four dimensions. HV4DS can be up to 10 times faster than FPL and WFG on the WFG data sets, and it is expected to be up to 4 times faster than IIHSO on average on those data. The "hard" case data set indicates that it might be reasonable to expect HV4DS to perform faster on any other data set than on the "hard" data set for a front of the same size. Moreover, it is necessary to test HV4DS on larger data sets in order to be able to draw conclusions relatively to its runtime variability when using different objective orderings. To conclude, HV4DS establishes a new, tighter upper bound ($O(n^2)$) on the complexity of the hypervolume indicator. In addition, the plots suggest that the advantage of HV4DS over the other algorithms will be even larger for larger fronts.

4.2 Divide-and-conquer algorithm for the three-dimensional case

4.2.1 Description

As a first step towards the development of a multidimensional divide-and-conquer algorithm to compute the hypervolume indicator in any number of dimensions, the three-dimensional case is considered here. Given a non-dominated point set $\mathrm{X} \subset \mathbb{R}^3$ of size $|\mathrm{X}| = n$ and a reference point $r \in \mathbb{R}^3$ as input, the proposed algorithm operates as follows:

Division step If $n = 1$, let x denote the single element of X. Return the volume of the cuboid $[x, r]$ if $x \leq r$, and zero otherwise.

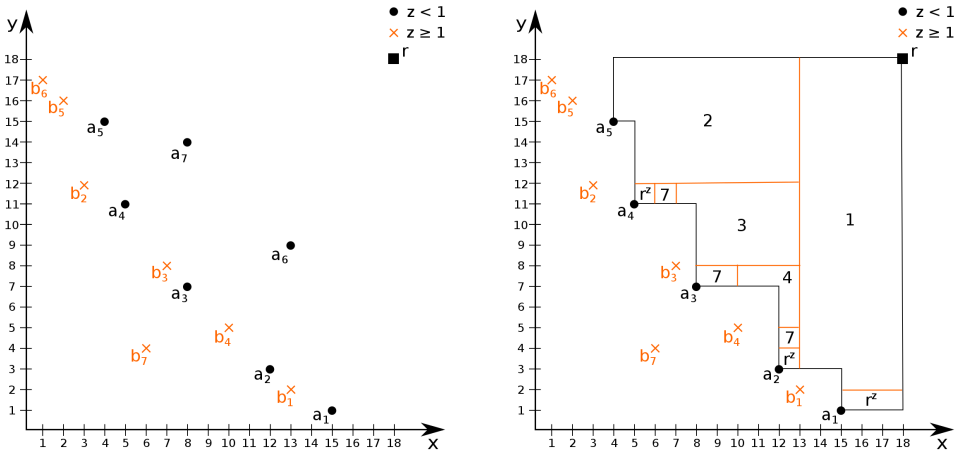
If $n > 1$, partition X into two subsets, A and B, with $n/2$ points each, in such a way that $\forall a \in \mathrm{A}, \forall b \in \mathrm{B}, a^z \leq b^z$.

Recursive step Let s^z denote the lowest z-coordinate value of the points in B. Recursively compute the hypervolume indicator of A given the reference point $s = (r^x, r^y, s^z)$ and that of B given r.

Marriage step Compute the volume not yet accounted, i.e., the volume of the region above the cut plane $z = s^z$ which is dominated by A but not by B, given the reference point r.

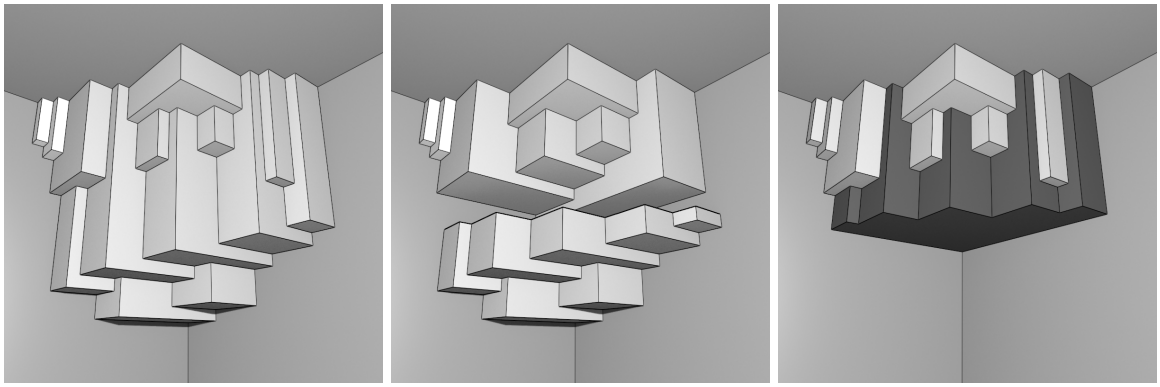
This procedure is performed after having sorted all points according to coordinates y and z. Since points are kept sorted according to coordinate z, the division step can be easily performed in linear time, while keeping points in the two subsets A and B sorted by y and z coordinates. Keeping points sorted according to coordinate y is important for the marriage step.

Figure 4.3(a) illustrates a problem with 14 points divided into two sets of 7 points each, sets $A = \{a_1, ..., a_7\}$ and $B = \{b_1, ..., b_7\}$. Figure 4.4(a) depicts the 3-dimensional volume dominated by the 14 points represented in Figure 4.3(a). Figure 4.4(b) represents the volume dominated by A and B returned by the recursive step, given their respective reference points. Note that for visualization purposes, the cut plane defined by s^z was considered to be below 1. Finally, figure 4.4(c) represents the volume above



(a) (b)

Figure 4.3: (a) An example where $r = (18, 18, 10)$. The initial input set is divided into two subsets $\mathrm{A} = \{a_1, ..., a_7\}$ and $\mathrm{B} = \{b_1, ..., b_7\}$, by the cut plane set at $z = 1$. Assume that $a_i^z = -i$ and $b_i^z = i$ for $i = 1, ..., 7$. (b) The expected result of the marriage step applied to A and B given the reference point r.



(a) (b) (c)

Figure 4.4: (a) The total volume dominated by the points represented in Figure 4.3(a). (b) The volume of A given the reference point $s = (r^x, r^y, 1)$ and B volume given the reference point r. (c) The darker volume represent the volume above the cut plane $z = s^z$ dominated by A but not by B.

the cut plane dominated by A but not by B in a darker color, i.e., the volume computed in the merge step.

The most important step of the algorithm is the marriage step. Since the recursion tree will have $O(\log n)$ depth, in order to achieve the goal of $O(n \log n)$ time complexity established by the optimal algorithm of Beume *et al.* [4], the marriage step must have $O(n)$ time complexity. However, it is not trivial to achieve this goal.

Computing the desired volume can be achieved by partitioning the region dominated by A on the (x,y)-plane into non-overlapping sub-regions of constant height. For example, given the problem of Figure 4.3(a), the expected result is represented in Figure 4.3(b). These sub-regions will be divided further into rectangles (or boxes), similarly to what was done in the four-dimensional hypervolume indicator algorithm when computing a contribution of a point in three dimensions, as explained in Section 4.1.

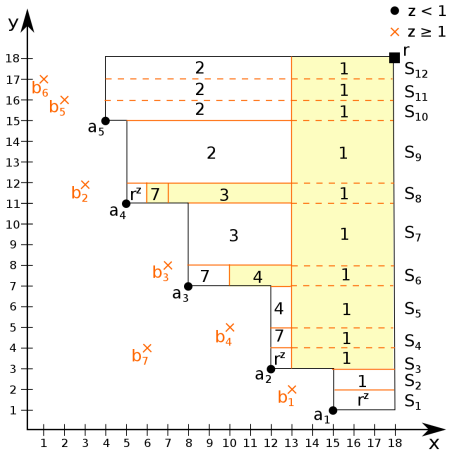


Figure 4.5: $S_1, ..., S_{12}$ represent all slices of the problem represented in Figure 4.3. Boxes separated by dashed lines indicate that bottom one is created and computed only once, and then are kept unchanged. Boxes with yellow background are boxes which will not be stretched along coordinate x anymore.

However, these boxes will have a different representation/data structure in the algorithm of this Section. Note that in Figure 4.3(b), the base of all sub-regions starts at coordinate z equal to 1 (s^z) and their top ends at $z = k$, where k is the number depicted in the center of each sub-region (or box). This value of coordinate z that defines the top of each sub-region (or box) will be referred to its height, while the absolute difference between the value of coordinate z of the top and the bottom of a sub-region (or box) will be referred as the absolute height. Moreover, since points in A which are dominated by other points in A in their projection onto the (x, y)-plane do not contribute to the volume to be computed in the marriage step, these points are ignored (a_6 and a_7).

Points in A and in B are merged and swept in ascending order of coordinate y (independently of the set to which each point belongs) and, for each one, the volume of the corresponding slice is computed, i.e., the volume between two consecutive points in y, or between the last point and the reference point. Figure 4.5 shows all slices ($S_1, ..., S_{12}$) for the problem depicted in Figure 4.3. To compute the volume of a slice, it must be divided into non-overlapping boxes along coordinate x, where each box has a different height (in z coordinate) (see Figure 4.5). While sweeping points, if the point being swept is from A it will cause the stretching along coordinate x of the leftmost box and, eventually the addition of new boxes. If the point being swept is from B, it may cause the ceiling of some boxes to be lowered and the union between boxes with the same height, which can also be seen as the removal of old boxes and the introduction of a new box.

Algorithm 23 details the computation of the marriage step. As input, the split sets A and B, the reference point r and the value of the z coordinate of the cut plane are received. The algorithm sweeps every point q_i, $i = 1, ..., |q|$ and, while doing this, it keeps track of the height of boxes in the current slice, which are defined by points from B which have been swept so far that are not dominated on the (x,z)-plane by any other swept point in B. These points are kept in a list H which is kept sorted in ascending order of the z coordinate (and consequently, sorted in descending order of x). As the algorithm runs, it is necessary to keep track of point l and values t and a. t indicates up to which value of y the

volume has been computed. l is the last visited point from A with lowest x coordinate. l is conveniently set to $(r^x, -\infty, -\infty)$ until the first point from A is swept. a is the lateral area, i.e., the area on the (x,z)-plane of the current slice, which is delimited between l^x and r^x in coordinate x. The boxes and their heights are defined by point c (c is in H) and all points in H before c (with lowest z coordinate and therefore, with highest coordinate x).

The height of the lateral area between l^x and the point before c in H ($\mathsf{prev}(c, \mathrm{H})$) is c^z. Considering $k = \mathsf{prev}(c, \mathrm{H})$, if k is not the first point of H, the region between k^x and $\mathsf{prev}(k, \mathrm{H})^x$ in coordinate x, has height k^z, and so on. For example, when point a_4 is chosen to be swept, the information about slice S_7 (see Figure 4.5) it is available, and H would be $((r^x, -\infty, -\infty)$, b_1, b_3, b_7, $(-\infty, -\infty, r^z))$. c, l and a would be b_3, a_3 and 10, respectively. Therefore, $a = 10$ is the lateral area which is between a_3^x and r^x, where the height between a_3^x and b_1^x is b_3^z and between b_1^x and r^x is b_1^z. In this example, t indicates that the volume has been computed up to b_3^y in coordinate y. Therefore, the volume of slice S_7 is $a \cdot (a_4^y - t)$ which is 30.

There is one more data structure that is fundamental to achieve the desired time complexity, which is v. v_i is the point after which q_i should be inserted in H, where $i = 1, ..., n$ and where $|\mathrm{q}| = n$. Before starting the divide-and-conquer process and after the presorting step, each v_i ($i = 1, ..., n$) is initialized as $(r^x, -\infty, -\infty)$. This value is then updated while moving up the recursion tree. Later on it will be explained how to update v_i, but for now it is only important to know what it represents.

Line 8 of Algorithm 23 computes the volume of the current slice whose lateral area was computed in the previous iteration and is zero in the first iteration. Lines 10-19 deal with the cases where q_i is a point from A and lines 20-35 deal with the cases where q_i is a point from B. In the case where $\mathrm{q}_i \in$ A, the algorithm updates the lateral area to account for the area between q_i^x and l^x. For example, when $\mathrm{q}_i = a_4$, the lateral area to be added in lines 10-19 is the area between a_4^x and a_3^x. To do this, c and the following points in H which contribute to this lateral area are swept. Note that if q_i is dominated on the (x,y)-plane by another point in A, there is no need to update the lateral area and, therefore, q_i is skipped (fails check of line 11). After updating the lateral area, l is updated.

When $\mathrm{q}_i \in$ B, then H has to be updated, and so does the lateral area. Therefore, it is necessary to add q_i to H, in line 21, (for now ignore the subsequent lines 22 and 23) and to remove the points from H which are dominated by q_i on the (x,z)-plane. When a point is removed from H, the box defined by that removed point (if any) has to be lowered to q_i^z (lines 26-31). If only points in H to the right of l^x (that have higher coordinate x) are removed, then the leftmost box height has to be lowered and therefore, the lateral area is updated. If q_i defines the height of the left most box in the lateral area, c is updated (line 34). Note that $\mathrm{q}_i^x < l^x$ is always true, otherwise q_i would be dominated.

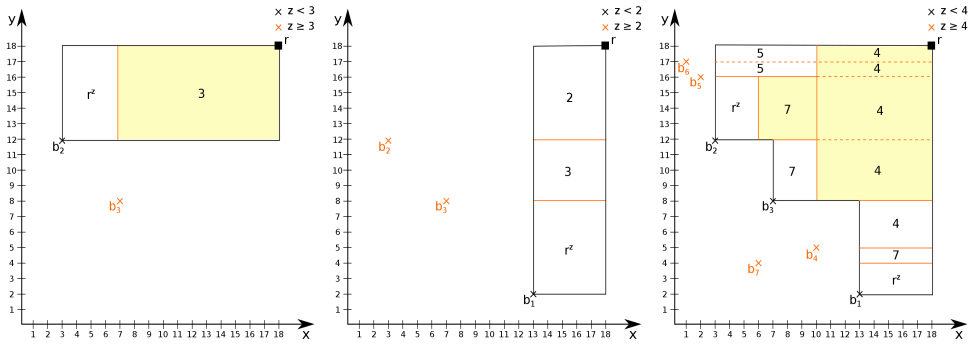
The last aspect that needs to be explained is how to know where q_i should be inserted in H when q_i is a point from B, i.e., how to update v_i ($i = 1, ..., n$) in order to know where to add q_i in H in constant time. For example, in Figure 4.5, the problem is how to know that when $i = 7$ ($\mathrm{q}_7 = b_3$) then v_7 should be b_1. Remember that after the initial presorting before starting the divide-and-conquer process,

Algorithm 23 marriage(A,B,r,s^z)

Input: $\mathrm{A} \subset \mathbb{R}^3$
Input: $\mathrm{B} \subset \mathbb{R}^3$
Input: r // $r \in \mathbb{R}^3$ is the reference point
Input: s^z // $\forall a \in \mathrm{A} \Rightarrow a^z \leq s^z$ and $\forall b \in \mathrm{B} \Rightarrow a^z \geq s^z$ ($z = s^z$ is the cut plane)
Output: h // Volume above s^z dominated by A but not by B

1: $(\mathrm{q}_1, ..., \mathrm{q}_n)$ is $\mathrm{A} \cup \mathrm{B}$ sorted in ascending order of the y coordinate
2: $\mathrm{H} \leftarrow [(r^x, -\infty, -\infty), (-\infty, -\infty, r^z)]$
3: $l \leftarrow (r^x, -\infty, -\infty)$ // l is the point of set A visited so far with lowest x
4: $a \leftarrow 0$ // Lateral area
5: $t \leftarrow -\infty$ // up to which height in y the hypervolume is computed
6: $c \leftarrow \mathsf{last}(\mathrm{H})$ // current slice boxes are defined by points in H up to c
7: **for** $i = 1$ **to** $i = |\mathrm{q}|$ **do**
8: $\quad h \leftarrow h + a \cdot (\mathrm{q}_i^y - t)$
9: $\quad t \leftarrow \mathrm{q}_i^y$
10: $\quad$ **if** $\mathrm{q}_i \in \mathrm{A}$ **then**
11: $\quad\quad$ **if** $\mathrm{q}_i^x < l^x$ **then**
12: $\quad\quad\quad k = l^x$ // the area accounted in a is the area in the (x,z)-plane between k and r^x of coordinate x
13: $\quad\quad\quad$ **while** $\mathrm{q}_i^x \leq c^x$ **do**
14: $\quad\quad\quad\quad a \leftarrow a + (c^z - s^z) \cdot (k - c^x)$
15: $\quad\quad\quad\quad k \leftarrow c^x$
16: $\quad\quad\quad\quad c \leftarrow \mathsf{next}(c, \mathrm{H})$
17: $\quad\quad\quad a \leftarrow a + (c^z - s^z) \cdot (k - \mathrm{q}_i^x)$
18: $\quad\quad$ **if** $\mathrm{q}_i^x \leq l^x$ **then**
19: $\quad\quad\quad l \leftarrow \mathrm{q}_i$
20: $\quad$ **else**
21: $\quad\quad \mathsf{insert}(\mathrm{q}_i, \mathrm{v}_i, \mathrm{H})$ // insert q_i after v_i in H
22: $\quad\quad$ **if** $l^x \geq \mathrm{q}_i^x$ **and** $l^x \leq \mathrm{v}_i^x$ **then**
23: $\quad\quad\quad \mathrm{v}_i \leftarrow l$
24: $\quad\quad k \leftarrow \mathsf{prev}(\mathrm{q}_i, \mathrm{H})^x$ // the area on the (x,z)-plane between k and r^x in coordinate x, is correctly computed
25: $\quad\quad w \leftarrow \mathsf{next}(\mathrm{q}_i, \mathrm{H})$
26: $\quad\quad$ **while** $(\mathrm{q}_i^x, \mathrm{q}_i^z) \leq (w^x, w^z)$ **do**
27: $\quad\quad\quad$ **if** $l^x < k$ **then**
28: $\quad\quad\quad\quad a \leftarrow a - (w^z - \mathrm{q}_i^z) \cdot (k - \max(w^x, l^x))$
29: $\quad\quad\quad k \leftarrow w^x$
30: $\quad\quad\quad \mathsf{remove}(w, \mathrm{H})$
31: $\quad\quad\quad w \leftarrow \mathsf{next}(\mathrm{q}_i, \mathrm{H})$
32: $\quad\quad$ **if** $l^x < k$ **then**
33: $\quad\quad\quad a \leftarrow a - (w^z - \mathrm{q}_i^z) \cdot (k - l^x)$
34: $\quad\quad$ **if** $c^z \geq \mathrm{q}_i^z$ **then**
35: $\quad\quad\quad c \leftarrow \mathrm{q}_i$
36: $h \leftarrow h + a \cdot (r^y - t)$
37: **return** h

v_i ($i = 1, ..., n$) was initialized to $(r^x, -\infty, -\infty)$, which is the first point in H. This value is then updated when l is between q_i and v_i in coordinate x, which is not needed in the current execution of the marriage step, but may be useful in the marriage step at higher levels of the recursion tree. What this means is that the information about v_i comes from the recursion on set B. For example, let us see, in the example of Figure 4.5, that when $\mathrm{v}_7 = b_1$ when $i = 7$ ($\mathrm{q}_7 = b_3$). For that, let us simulate the recursion on the sets where b_3 is present (the number indicates the recursion level):



(a) $r = (18, 18, 4)$, $s^z = 3$ (b) $r = (18, 18, 4)$, $s^z = 2$ (c) $r = (18, 18, 10)$,$s^z = 4$

Figure 4.6: The marriage steps where point b_3 (of Figure 4.3) is present, moving towards the top of the recursion tree. s^z indicates at which value of coordinate z was set the cut plane to obtain the two sets of each figure, i.e., from which value of z, the hypervolume of A has to be accounted in the marriage step.

1. A = $\{a_1, ..., a_7\}$ and B = $\{b_1, ...b_7\}$ (Figure 4.3 where $b_3 = \mathrm{q}_7$)
2. A = $\{b_1, ..., b_3\}$ and B = $\{b_4, ..., b_7\}$ (Figure 4.6(c) where $b_3 = \mathrm{q}_4$)
3. A = $\{b_1\}$ and B = $\{b_2, b_3\}$ (Figure 4.6(b) where $b_3 = \mathrm{q}_2$)
4. A = $\{b_2\}$ and B = $\{b_3\}$ (Figure 4.6(a) where $b_3 = \mathrm{q}_1$)

Let us now see how the point in v corresponding to b_3 is updated through all of the marriage steps at these levels of recursion:

1. **level 4** When b_3 (q_1) is swept, v_1 is $(r^x, -\infty, -\infty)$ and l is $(r^x, -\infty, -\infty)$. b_3 is then correctly added to H after $(r^x, -\infty, -\infty)$. Since no point from A was swept and l is $(r^x, -\infty, -\infty)$, v_1 remains unchanged.
2. **level 3** When b_3 (q_2) is swept, v_2 is still $(r^x, -\infty, -\infty)$ and l is b_1. b_3 is again correctly added to H after $(r^x, -\infty, -\infty)$. Since l is b_1 and $b_3^x \geq b_1^x \geq r^x$, then v_2 is updated to b_1.
3. **level 2** When b_3 (q_4) is swept, since it is a point from A, it is not added to H and v_4 is kept unchanged (b_1).
4. **level 1** Finally, when reaching the first level, when b_3 is swept it is added to H after $\mathrm{q}_7 = b_1$, as desired. Since in this case $l = a_3$ and $b_3^x \geq a_3^x \geq b_1^x$, v_7 is updated to a_3.

Note than when level 1 is reached, v_7 is updated to a_3. This will only be useful if b_3 is again part of set B in even higher levels of the recursion tree, which is not the case in this example.

It is guaranteed that v_i is always correct and therefore, that q_i is inserted in H in the correct place, because to correctly add q_i to H it has to be ensured that v_i is in H, at the time q_i is swept, and it is the closest point to q_i according to coordinate x with higher coordinate x (which is also the closest point to q_i according to coordinate z with lower coordinate z). This is guaranteed because, in the bottom

marriage steps, by having v_i set to $(r^x, -\infty, -\infty)$, the above conditions are fulfilled. In the following recursion levels, by updating each v_i to the leftmost point seen so far in set A (whose projection onto the (x, z)-plane will always be non-dominated according to $\mathrm{A} \cup \mathrm{B}$), it is assured that v_i will be in H at the time q_i is inserted in H. Moreover, v_i is only updated if this point from A, according to coordinate x, is closer or is as close to q_i as the current v_i. Therefore, v_i is always correct.

4.2.2 Complexity

The complexity of the algorithm depends mostly on the marriage step, which must be performed in linear time in order for the algorithm to achieve asymptotically optimal $O(n \log n)$ time complexity. The division step poses no difficulty, as it can clearly be implemented in linear time after sorting. To achieve $O(n)$ time complexity in the marriage step, the box height updates have to be performed by sweeping only the points that determine the height of the boxes that need to be updated. These box height updates can happen when only updating the lateral area, either by sweeping points from A or by sweeping points from B and eventually removing points from H.

Each point q_i in B, is associated with the height of at most one box. This box can be a partial box or a complete box. Denoting by k the bottom left corner of some slice where $k^y \geq \mathrm{q}_i^y$ and where q_i is in H when $y = k^y$, it is said that a box associated to q_i is complete if $\mathrm{q}_i^x \geq k^x$. It is said that is a partial box if $\mathrm{q}_i^x < k^x$ and $\mathsf{prev}(p, \mathrm{H})^x \geq k^x$ (see Figure 4.3, where partial boxes are represented with a white background, while complete boxes have a yellow background). For example, in slice S_7, there is one complete box associated to points b_1, a partial box associated to b_3, and b_7 has no box associated with it. Note that there is at most one partial box per slice, because there is no overlap between boxes. Therefore, updating the lateral area can be seen as creating partial boxes, completing boxes or removing boxes (removing points from H).

After completing a box associated to a point q_i, q_i is no longer processed unless it is removed from H. Each point is swept once, it can be added to and removed from H once and it can only complete a box once. Since all of these steps are performed in constant time per point, they lead to an overall time complexity of $O(n)$. Finally, there can only be a total of n partial boxes, one per each slice, in each marriage step. Hence, the time complexity of the marriage step amortizes to $O(n)$ and consequently a $O(n \log n)$ time complexity is obtained for the whole algorithm. The main features responsible for the achievement of this time complexity are having points already sorted in ascending order of coordinate y before executing the marriage step, and the maintenance of v.

The space complexity of the algorithm is $O(n)$. In the implementation of this algorithm, each point was stored as a node of a multiple doubly-linked list containing besides the point's coordinates, the information of the next and the previous points according to coordinate y and z. Moreover, such a node also keeps the information about which set, A or B, the point belongs to, and the information about where in H it should be inserted, i.e., v_i.

4.2.3 Experimental results

To test whether the algorithm was returning correct results, its output was compared against FPL (version 1.3) [12] using the data sets made available online by the Walking Fish Group [18]. A small tolerance, of about 10^{-15} was considered. Some specific tests were constructed by hand, including tests with repeated coordinates. For some of those tests, each step of the algorithm was observed in order to guarantee that the algorithm was behaving correctly.

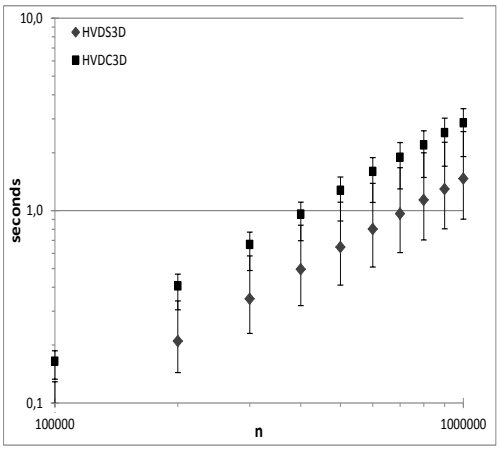
To test the performance of the divide-and-conquer algorithm, three sets of tests were constructed, which were the spherical data set, the cliff data set and the uxuydz data set. The spherical data set was generated using the following formula: $p^i = |X^i|/||X||$, $X^i \sim \text{Normal}(0, 1)$, $i = 1, 2, 3$. The cliff data set was generated using the formula: $p^i = 1 - |X^i|/||X||$, $X^i \sim \text{Normal}(0, 1)$, $i = 1, 2$, $p^3 \sim \text{Uniform}(0, 1)$. In the uxuydz data set, if points are sorted in ascending order of x coordinate, then they will be also sorted in ascending order of y and in descending order of z coordinate. For each type of data, different size data sets were constructed (1×10^6,2×10^6,...,10×10^6).

The divide-and-conquer algorithm proposed here, referred to as HVDC3D, was compared against the asymptotically optimal algorithm proposed by Beume *et al.* [5] for three dimensions. The WFG implementation of this algorithm was used, and will be referred to as HVDS3D (DS from Dimension Sweep). Since it was observed in the previous algorithms proposed in this thesis that presorting the input points in memory can help to speed up the algorithm by enabling a more efficient use of the cache, this characteristic was included from the start in this HVDC3D implementation.

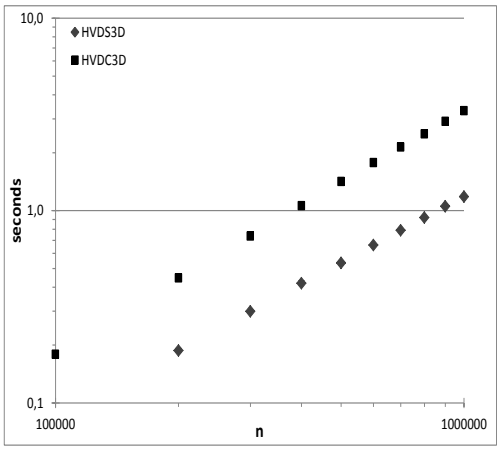
Each algorithm was run on all six permutations of the objectives for each data set. Average runtimes are represented on the plots of Figure 4.7. On those plots, error bars are used to represent the minimum and maximum runtime observed for each set. Note that in some cases the bars are not visible, because the algorithm runtime did not vary much with objective ordering.

In Figure 4.7 it is possible to observe that the divide-and-conquer approach did not outperform the dimension sweep approach, but the most important fact to notice is that the runtime grows similarly in both cases. Since HVDC3D is recursive, some overhead was expected in comparison to an iterative implementation. Therefore, a full multidimensional divide-and-conquer algorithm for the hypervolume indicator may still be competitive with the current ones.

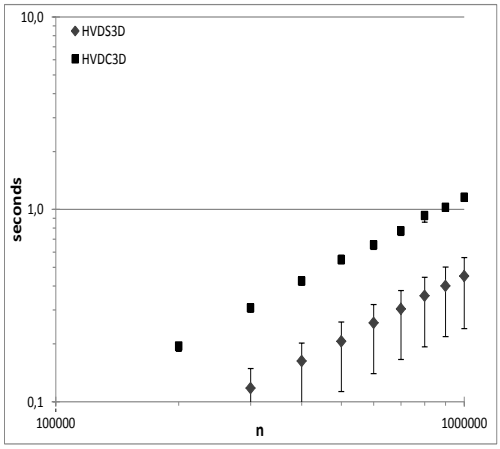
It could be argued that HOY already uses a divide-and-conquer approach, but not multidimensional divide-and-conquer, which is the final goal of what was started in this Section. The divide-and-conquer approach applied to HVDC3D is still a new way to approach the problem, since it differs from HOY in the sense that HVDC3D divides the problem according to the number of points while HOY divides the problem according to the space that includes the area dominated by the initial set of points. Therefore, HVDC3D introduces a new perspective on how to approach hypervolume indicator computation.



(a) cliff



(b) spherical



(c) uxuydz

Figure 4.7: Comparison of HVDC3D and HVDS3D performance.

4.3 Concluding remarks

The work presented in this Chapter makes two major contributions to the current state-of-the-art of the hypervolume indicator. The first main contribution is the proposal of a new algorithm (HV4DS) to compute the hypervolume indicator in four dimensions that establishes $O(n^2)$ as a new tighter upper bound on the time complexity for $d = 4$. The second main contribution is the introduction of a new divide-and-conquer approach to the computation of the hypervolume indicator. This is achieved by proposing an algorithm (HVDC3D) for the base case in three dimensions, whose time complexity matches the optimal $O(n \log n)$ time complexity of $d = 3$.

It was also shown that, in addition to having the lowest complexity, HV4DS was faster in practice than the other algorithms tested, up to 10 times in most cases. In most of the tests performed, it improved

upon WFG performance, at least, up to 10 times. Therefore, it is expected to improve upon IIHSO (which was, to date, the fastest algorithm for $d = 4$) performance by about 4 four times. HV4DS is also characterized by its efficient use of cache in comparison to HV4D.

Although, it does not outperform the current fastest algorithm for the three dimensional case, HVDC3D shows how divide-and-conquer can be applied to the computation of the hypervolume indicator, and that it may be possible to develop efficient algorithms using the multidimensional divide-and-conquer approach. Matching the optimal time complexity, this algorithm is just the starting point for the development of a multidimensional divide-and-conquer algorithm for the hypervolume indicator.

5 Concluding Remarks

As discussed in Chapter 1, quality indicators such as the hypervolume indicator are important methods for the assessment of optimizer performance. In particular, the hypervolume indicator has been widely studied, and has even been integrated into some evolutionary multiobjective optimizers [6]. Therefore, it is very important that algorithms to compute it have a good performance. On the other hand, the Empirical Attainment Function, which provides a deeper look into optimizer performance by allowing certain aspects of the distribution of the different outcomes of stochastic multiobjective optimizers to be studied, is still limited by the unavailability of algorithms to compute it in more than three dimensions.

The aim of this thesis was to contribute to the state-of-the-art in algorithms for the assessment of stochastic multiobjective optimizers, such as those to compute the hypervolume indicator and the EAF. This goal was achieved through three different types of contributions: new algorithms that outperform existing ones were developed; new algorithms were developed for a case for which no algorithm was available; and an asymptotically optimal algorithm based on different paradigm was developed.

Specifically, in the hypervolume indicator case, this work contributes with two new algorithms: an algorithm for the four-dimensional case and another for the three dimensional case. The algorithm for $d = 4$ (HV4D), proposed in Section 4.1, is an important contribution since it establishes a new, tighter upper bound of $O(n^2)$ for $d = 4$. It was also shown to outperform the existing algorithms in practice. The algorithm for $d = 3$ (HVDC3D), proposed in Section 4.2, contributes to the introduction of a new approach to the computation of hypervolume indicator, the multidimensional divide-and-conquer approach, by defining its base case. This algorithm achieves the optimal $O(n \log n)$ time complexity of the three dimensional case.

For the EAF, two new algorithms were presented: an algorithm for the four dimensional case and another for the two-dimensional case. The former was proposed in Section 3.1 and is the first algorithm available for the four-dimensional case. In Section 3.2, a new algorithm for $d = 2$ (OSEAF2D) which outperforms the existing one was proposed. This new algorithm has the interesting characteristic of being output-sensitive, a desirable feature which has not been investigated in connection with the EAF, so far.

Especially in the case of HV4D and EAF2D, the issue of cache efficiency was raised. Experiments indicate that algorithms to compute the hypervolume indicator and the EAF, which usually follow a dimension sweep approach, may benefit considerably from the presorting of points in memory in order to reduce the number of cache misses and improving their runtime.

To conclude, it was shown that existing algorithms could still be improved upon. Issues such as cache efficiency and output-sensitivity were approached in order to arrive at more efficient algorithms. A new

point of view on the computation of hypervolume indicator was introduced by implementing a divide-and-conquer approach. Overall, new and efficient algorithms for the assessment of stochastic multiobjective optimization were developed.

The work developed in this thesis opens up some new directions for future work. In the case of the hypervolume indicator, it should be possible to extend the approach used by HV4D to at least five dimensions, although it is not yet clear how this may be achieved. HVDC3D should be generalized to any number of dimensions by implementing the remaining aspects of a multidimensional divide-and-conquer algorithm, which is also not an easy task because of the $O(n^2)$ and $O(n^{d/2} \log n)$ time complexity bounds for $d = 4$ and $d > 4$, respectively, already imposed by the existing algorithms. Moreover, the applicability of multidimensional divide-and-conquer to hypervolume indicator computation also encourages the application of this paradigm to the computation of the EAF. Finally, a new direction for the computation of EAF was pointed out, i.e., pursuing output-sensitive algorithms. The good results obtained by sorting points in memory, applied to EAF2D and hypervolume indicator algorithms, also highlight this as an issue worth considering in other algorithms, such as those to compute EAF in three and four dimensions.

Bibliography

[1] J. Bader and E. Zitzler. HypE: An Algorithm for Fast Hypervolume-Based Many-Objective Optimization. Technical report, Computer Engineering and Networks Laboratory (TIK), ETH Zurich, November 2008.

[2] A. Ben-Tal. Characterization of Pareto and lexicographic optimal solutions. In G. Fandel and T. Gal, editors, *Multiple Criteria Decision Making Theory and Application*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, page 1–11. Berlin, Germany: Springer-Verlag, 1980.

[3] Jon Louis Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23:214–229, April 1980.

[4] Nicola Beume. S-metric calculation by considering dominated hypervolume as klee's measure problem. *Evol. Comput.*, 17:477–492, December 2009. Code available at: http://ls11-www.cs.tu-dortmund.de/people/beume/#software.

[5] Nicola Beume, Carlos M. Fonseca, Manuel López-Ibáñez, Luís Paquete, and Jan Vahrenhold. On the complexity of computing the hypervolume indicator. *IEEE Transactions on Evolutionary Computation*, 13(5):1075–1082, 2009.

[6] Nicola Beume, Boris Naujoks, and Michael Emmerich. SMS-EMOA: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653–1669, 2007.

[7] L. Bradstreet, L. While, and L. Barone. A fast incremental hypervolume algorithm. *Evolutionary Computation, IEEE Transactions on*, 12(6):714 –723, December 2008.

[8] L. Bradstreet, L. While, and L. Barone. A fast many-objective hypervolume algorithm using iterated incremental calculations. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1 –8, July 2010.

[9] Michael Emmerich and Carlos M. Fonseca. Computing hypervolume contributions in low dimensions: Asymptotically optimal algorithm and complexity results. In Ricardo Takahashi, Kalyanmoy Deb, Elizabeth Wanner, and Salvatore Greco, editors, *Evolutionary Multi-Criterion Optimization*, volume 6576 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin / Heidelberg, 2011.

[10] Carlos M. Fonseca and Peter J. Fleming. On the performance assessment and comparison of stochastic multiobjective optimizers. In H.-M. Voigt et al., editors, *Parallel Problem Solving from Nature, PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 584–593. Springer, Heidelberg, Germany, 1996.

[11] Carlos M. Fonseca, Andreia P. Guerreiro, Manuel López-Ibáñez, and Luís Paquete. On the computation of the empirical attainment function. In Ricardo Takahashi, Kalyanmoy Deb, Elizabeth Wanner, and Salvatore Greco, editors, *Evolutionary Multi-Criterion Optimization*, volume 6576 of *Lecture Notes in Computer Science*, pages 106–120. Springer Berlin / Heidelberg, 2011. Code available at: http://eden.dei.uc.pt/∼cmfonsec/software.html.

[12] Carlos M. Fonseca, Luís Paquete, and Manuel López-Ibáñez. An improved dimension-sweep algorithm for the hypervolume indicator. In *IEEE Congress on Evolutionary Computation*, pages 1157–1163, Piscataway, NJ, July 2006. IEEE Press. Code available at: http://iridia.ulb.ac.be/∼manuel/hypervolume.

[13] Viviane Grunert da Fonseca, Carlos M. Fonseca, and Andreia O. Hall. Inferential performance assessment of stochastic optimisers and the attainment function. In Eckart Zitzler, Kalyanmoy Deb, Lothar Thiele, Carlos A. Coello, and David Corne, editors, *Evolutionary Multi-criterion Optimization (EMO 2001)*, volume 1993 of *Lecture Notes in Computer Science*, pages 213–225. Springer, Heidelberg, Germany, 2001.

[14] Victor Klee. Can the measure of $\cup[a_i, b_i]$ be computed in less than $O(n \log n)$ steps? *American Mathematical Monthly*, 84:284–285, 1977.

[15] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[16] Mark H. Overmars and Chee-Keng Yap. New upper bounds in klee's measure problem. *SIAM J. Comput.*, 20:1034–1045, December 1991.

[17] F. P. Preparata and M. I. Shamos. *Computational Geometry. An Introduction.* Springer Verlag, Berlin, Germany, 2 edition, 1988.

[18] L. While, L. Bradstreet, and L. Barone. A fast way of calculating exact hypervolumes. *Evolutionary Computation, IEEE Transactions on*, PP(99):1, 2011. Code available at: http://www.wfg.csse.uwa.edu.au/publications.html#hypervolume.

[19] L. While, L. Bradstreet, L. Barone, and P. Hingston. Heuristics for optimizing the calculation of hypervolume for multi-objective optimization problems. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2225 – 2232 Vol. 3, September 2005.

[20] L. While, P. Hingston, L. Barone, and S. Huband. A faster algorithm for calculating hypervolume. *Evolutionary Computation, IEEE Transactions on*, 10(1):29 – 38, February 2006.

[21] E. Zitzler and L. Thiele. Multiobjective optimization using evolutionary algorithms - A comparative case study. In A. E. Eiben et al., editors, *Parallel Problem Solving from Nature, PPSN V*, volume 1498 of *Lecture Notes in Computer Science*, pages 292–301. Springer, Heidelberg, Germany, 1998.

[22] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M. Fonseca, and Viviane Grunert da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.