DOCTORAL DISSERTATION (SHINSHU UNIVERSITY)

# IMPROVEMENTS IN UNDERSTANDING AND PERFORMANCE OF MULTI-OBJECTIVE DIFFERENTIAL EVOLUTION

March 2015

MARTIN DROZDÍK

SUPERVISORS:
Kiyoshi Tanaka
Hernán Aguirre
Youhei Akimoto

Dedicated to my family.

# ABSTRACT

Many optimization problems are formulated using more than one objective function. In the overwhelming majority of cases, the solution of a multi-objective problem is not a single element, but a set of solutions that can be no longer improved upon in all objectives *simultaneously*. Evolutionary algorithms are a *naturally good* tool to solve such problems, since they generate many solutions in one run. Differential evolution (DE) is one of the most simple and powerful evolutionary algorithms and its application to multi-objective optimization arises naturally. However, DE was originally developed for single-objective optimization and its generalization to the multi-objective model is not trivial.

The need to have a simple, powerful optimizer capable of solving continuous multi-objective problems has led many researchers to develop various versions of multi-objective DE. This rapid innovation happened without answering many outstanding questions, while introducing new ones. The first goal of this thesis is to answer some of these questions. In particular, we concentrate on the questions arising in *parameter setting* of multi-objective differential evolution. We investigate the relationships between the DE parameters and its performance as well as analyze the existing mechanisms to set the parameters automatically.

The next issue that arises with the transition to the multi-objective realm is the increased *computational cost*. Each solution now has a vector of objective function values and the mutual relationships of these vectors need to be tracked. This leads to several computational geometric problems. The reduction of computational cost of these problems is the second goal of this thesis.

In the first part, we concentrate on the first goal, that is improving our understanding of how DE works on multi-objective optimization problems. In single-objective DE it has been shown that the success of DE is highly affected by the right choice of its mutation and crossover parameters. Unfortunately, in multi-objective optimization the influence of these parameters on the performance of the algorithm is a poorly understood subject. Many authors use parameters which *do not* render the algorithm *invariant* with respect to *rotation* of the coordinate axes. This is a possible vulnerability, since the success of their algorithms may be caused by a hidden feature of the optimization problem and may be lost by simply rotating the coordinate axes of the problem. First, we try to see if such choice of parameters can bring consistently good performance under various *rotations* of the problem. We do this by extensive experimentation, using a large

number of parameter combinations generated on a grid, with systematically rotated benchmark problems. We find that our results are consistent with the single-objective theory, but *only for unimodal problems*. On multi-modal problems, unexpectedly, parameter settings which do *not* render the algorithm rotationally invariant have a consistently good performance for all studied rotations.

To mitigate the problem of parameter setting, methods have been developed to automatically adjust the parameters. These methods are usually presented as a part of a unified algorithm and since these algorithms vary in other aspects than the parameter control mechanism, it is difficult to compare them and to evaluate their viability in the multi-objective environment. We go through various deterministic, adaptive, and self-adaptive approaches to parameter setting, isolate the underlying parameter control mechanisms and apply them to a single simple differential evolution algorithm. We then observe its performance and behavior on a set of benchmark problems. We find that even very simple parameter control mechanisms can compete with parameter settings found by exhaustive grid search. We also notice that *self-adaptive* mechanisms seem to perform better on problems which can be optimized with a very limited set of parameters. *Adaptive* methods on the other hand encounter significant difficulties and seem to behave similarly on each benchmark problem. This is a significant vulnerability and it should be explored in more depth.

In the second part of the thesis, we address the second goal, that is the *computational cost reduction*. We are concerned with *non-dominated sorting, archiving, and diversity estimation* procedures. We propose a special data structure, called the *M-front*, to hold the best found (non-dominated) individuals. The M-front uses the geometric and algebraic properties of the Pareto dominance relation to convert *orthogonal range queries* into *interval queries* using a mechanism based on the nearest neighbor search. These interval queries are answered using dynamically sorted linked lists. The M-front can serve either to reduce the cost of non-dominated sorting, to reduce the cost of diversity estimation or as a fast archive. Experimental results show that our method can perform significantly faster than the state of the art algorithm for non-dominated sorting, with the added benefit of keeping track of *all* the non-dominated individuals all times.

We conclude our thesis with a summary of our contributions and by outlining promising directions for future work.

# PUBLICATIONS

Some ideas and figures have appeared previously in my publications [16, 17, 18, 19, 20, 21]. Source code for my programs can be found in [15].

# ACKNOWLEDGEMENTS

First of all I want to thank my parents, Mária and Vojtech, for supporting (almost) everything I do.

Next, I want to thank Professor Tanaka for his courage in putting his reputation at stake and accepting me to the doctoral programme. I want to thank all my supervisors, Kiyoshi Tanaka, Hernán Aguirre, and Youhei Akimoto for doing their work on much more than 100%, being there for me when I needed them the most, proofreading my papers late in the night and on the weekends, giving me advice and encouragement when I was lost, and most of all, for never ceasing to believe in me, even after all my failures.

Thank You, Sohyun Kim, for supporting me, and for for being with me in my darkest hours.

I want to thank my mathematical analysis Professor Zbyňek Kubáček for making me love and understand mathematics. Thanks to Erika Hönschová for her help with some difficult proofs. Thanks to Alexey Balakin for making his graphical library available free of charge and for his kind explanation and advice. Almost all graphics in this thesis are made using his library MATHGL. Thanks to the stackoverflow community for answering my programming questions and guiding me on my programming journey. Thanks to all the researchers in my field, who answered my emails and provided their papers free of charge for me.

Next, I want to thank Sébastien Verel, Bilel Derbel, Arnaud Liefooghe, and Dimo Brockhoff for their help and guidance at the Inria institute. Thanks to Sébastien Verel in particular for letting me stay in his home, until I found an appartment. Thanks to Keiko Nishizawa, Mika Asakawa and Ms Watanabe for doing the really hard work for me and for never letting me get in trouble with all the administration. Thanks to the Japanese language teachers of Matsumoto and Nagano campus for your patience and kindness in teaching me your language. Thanks to Marie-Hélène for accepting me to her home during my stay at Inria Lille.

Finally, I want to thank the Japanese taxpayers for their kindness and generosity in funding my studies for these four wonderful years.

# CONTENTS

# Part I

# INTRODUCTION TO MULTI-OBJECTIVE DIFFERENTIAL EVOLUTION

In this part we define the subject of this thesis. We start with an introduction to the broad field of computational intelligence and gradually narrow down our focus.

# 1 INTRODUCTION

## 1.1 COMPUTATIONAL INTELLIGENCE

There is relatively little consensus on what computational intelligence (CI) is. For the purposes of this thesis, we use a broad definition by Duch [22]: *"Computational intelligence is a branch of computer science studying problems for which there are no effective computational algorithms."* There are many other definitions, mostly focusing on the *tools* being used, instead of *problems* being solved.

What do we mean by *effective* computational algorithms? According to the Webster's English Dictionary, effective means: *"adequate to accomplish a purpose; producing the intended or expected result"*. That is, computational intelligence studies computational problems that we are currently simply incapable of solving. One example is to design an autopilot that can safely land a Boeing 747 aircraft. One other example is to computationally distinguish a dog and a wolf on a photograph. Even if we had all the computational power we need, there is as yet no way in which these things can be accomplished.

One major group of problems Duch's definition omits are problems for which we have effective algorithms, but these algorithms are not *efficient*. The Webster's English Dictionary defines efficient as: *"satisfactory and economical to use"*. In this category we have all NP-hard problems as well as *any problem* for which the state-of-the-art algorithms perform unsatisfactorily. Computational intelligence studies these problems as well.

Computational intelligence is heavily influenced (but not defined), by the study of living organisms. Human pilots land Boeing 747 aircraft every day and sheep can distinguish the dog from a wolf very successfully. Many methods in CI seek inspiration in nature.

Some of the main branches of CI inspired by nature are:

- *Computational swarm intelligence*, which attempts to solve complex problems using inspiration from the behavior of societies of living organisms.
- *Artificial neural networks*, which do so by constructing a simplified model of a brain.
- *Evolutionary computation*, which is inspired by the evolution of living organisms molded by natural selection, survival of the fittest, and genetics.

There are other branches, such as fuzzy systems, artificial immune systems, and many other smaller branches, some of which are covered in [25]. In this thesis we will be concerned entirely with *evolutionary computation*.

## 1.2 EVOLUTIONARY COMPUTATION

Life is hard. The environment is full of dangers and it is constantly changing. Yet it is full of living organisms which successfully inhabit this planet for billions of years. Each individual living being is a survival expert and *survival* is definitely a problem for which there are no effective algorithms. It is therefore natural that when solving difficult problems, humans look for inspiration in the science of survival, which is called evolution.

In a population of living organisms there is a never-ceasing struggle to reproduce. Individuals try to manifest their ability to survive, for example by competing in mating tournaments and rituals. Strong individual have a greater chance to succeed in these tournaments, mate and to pass their genes to the next generation. This mating pressure creates bias which favors genes that lead to stronger individuals. Genes which produce weak individuals have a smaller chance to proliferate, since their bearer has a smaller chance to survive until maturity and once mature, has a smaller chance to mate and produce offspring.

### 1.2.1 *Fundamental ideas*

The main idea of evolutionary computation is to solve complex problems by simulating evolution. We formalize the metaphor into a unified framework, described in Algorithm 1.

**Algorithm 1:** Evolutionary algorithm

```
1 generate a random population $P = \{X_1, ..., X_N\}$
2 while stopping condition not met do
3     $P_{parent}$ := select_for_mating($P$)
4     $P_{offspring}$ := recombine($P_{parent}$)
5     $P_{offspring}$ := mutate($P_{offspring}$)
6     $P := P \cup P_{offspring}$
7     $P$ := select_for_survival($P$)
8 end
9 return $P$
```

First, a set of random solutions to the problem are generated on line 1. Each such solution represents an individual organism and the set of solutions represents a population. Next, some individuals are selected to generate new individuals (line 3). Usually this selection is biased towards solutions which solve the problem better than their peers in some sense. Individuals in $P_{\text{parent}}$ are combined in a manner that imitates gene recombination, to produce new individuals $P_{\text{offspring}}$. These individuals are randomly altered in a manner which resembles gene mutation (line 5). The new individuals are added to the population. Next, the survival of the fittest is simulated by removing individuals from the polled population (line 7). Usually this selection is biased to weed out solutions which solve the problem relatively poorly compared to their peers. One such cycle is called a *generation*. After a sufficiently large number of generations, hopefully the solutions in P improve their ability to solve the problem.

### 1.2.2 *Brief history*

For an overview of the field of evolutionary computation, highlighting the landmark papers see [27]. According to this book, ideas of evolutionary computation date back as far as 1948 due to Alan Turing. Serious work in the field began in the 60's in three different places independently. Each working group solved slightly different problems and developed their algorithms to best suite these problems. Their work grew into three early paradigms of evolutionary computation:

- *Genetic algorithms* by Holland [32]
- *Evolution strategies* by Schwefel et al. [51]
- *Evolutionary programming* by Fogel et al. [28]

This division is mostly historical, since we do not view these three fields as distinct as we once did. In the 90's *genetic programming* [4] and *differential evolution* [45] joined the group. Next, we briefly introduce each field.

### 1.2.3 *Genetic Algorithms*

The invention of genetic algorithms by Holland was originally motivated by the desire to *study adaptive behavior*. However, soon they became largely employed as *function optimizers*. Genetic algorithms follow Algorithm 1.

What makes genetic algorithms distinct, is that they use primarily *binary coding*. That is, each individual in the population is represented by a string of 0s and 1s. This representation is particularly useful if the problem being solved is itself binary in nature. One such problem is the 0-1 knapsack problem whose objective is to select some items from a list, subject to constraints, maximizing a utility function. In this case, a 1 can mean that an item is selected and a 0 that it is not.

There are many ways to recombine (Algorithm 1 line 4) binary encoded individuals. One of the simplest ways is the $n$*-point crossover*, which is illustrated in Figure 1. Here we see two parent individuals,

$P_1, P_2$, which have been selected for recombination. First, the strings are cut by $n$ randomly generated lines and two child individuals, $C_1, C_2$, are generated by alternately inheriting a part of the string from either $P_1$ or $P_2$.

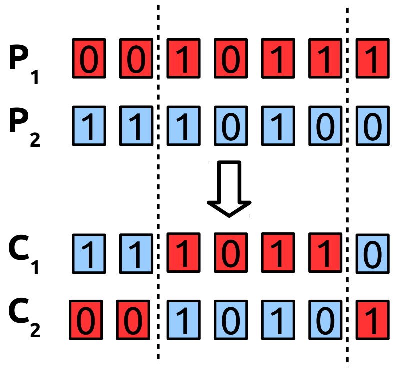


Figure 1: An illustration of two point crossover.

There are many variations to this type of recombination and all genetic algorithms are not necessary binary encoded, but binary encoding was the original idea. As was said before, the distinctions between some fields of evolutionary computation are mainly historical nowadays. Genetic algorithms were not primarily developed for optimization and the basic loop of Algorithm 1 needs to be enhanced in several ways to obtain a practical optimizer [11].

### 1.2.4 *Evolution Strategies*

The distinguishing features of evolution strategies (ESs) [5] are *real encoding* and *self-adaptation*. Real encoding means that ESs are particularly well suited for *continuous optimization*.

In an application of ES to an optimization problem:

$$f : \mathbb{R}^n \supseteq D \mapsto \mathbb{R}; \ f \text{ is to be minimized} \tag{1}$$

each individual is represented by a vector of values $X \in \mathbb{R}^n$, plus a vector of parameters $\sigma \in \mathbb{R}^m$. These parameters are used to mutate the individual (Algorithm 1line 5), usually by adding a value sampled from the multivariate normal distribution with covariance matrix obtained by some transformation of $\sigma$.

The parameters $\sigma$ undergo mutation and recombination along with the vector X. This is the main principle of *self-adaptation*. Each individual is tied to the parameters, which determine the way it is mutated. These parameters influence its chances to survive. Good parameters generate good individuals, which have a high chance to mate and to survive. These strong individuals then proliferate the good vales of their parameters. Bad parameters generate bad individuals, which do not get the opportunity to mate and have a low chance to survive. Therefore parameters are *evolved* along with the population.

### 1.2.5 *Evolutionary Programming*

The task evolutionary programming (EP) was designed to solve is to generate artificial intelligence by interpreting evolution as a form of learning. The individual in EP is a *finite state machine*, which is a program with parameters, as opposed to binary strings or real vectors of GA and ES.

### 1.2.6 *Genetic Programming*

The main difference between genetic and evolutionary programming is in the representation of the program that is being optimized. In EP, the structure is fixed and only the parameters are being evolved. Mutation plays a major part here and recombination plays a very limited role. On the other hand, in genetic programming (GP) the structure of the program is subject to evolution. The individual in GP is encoded as a *parse tree* of the program. Crossover, which exchanges subtrees between individuals plays a major role. Mutation on the other hand plays a very limited role here.

### 1.2.7 *Differential Evolution*

One of the newest fields of evolutionary computation, which was largely inspired by ES and GA is *differential evolution (DE)*. The distinguishing feature of DE is its real encoding of individuals, the same as in ES, and a particular way in which individuals are being recombined.

The fundamental principle of DE is to create new individuals by adding scaled *differences* of other individuals to each other. Let $P = \{X_1, ..., X_{NP}\}$ where $X_i = (x_{i,1}, \ldots, x_{i,n}) \in \mathbb{R}^n$, be the population. In its most basic form, DE traverses through this population, while trying to improve each individual $X_{\text{target}}$ by generating a new individual $X_{\text{trial}}$ in the following fashion. First, three distinct individuals $X_{r_1}, X_{r_2}, X_{r_3}$ are chosen from P. Then a *scaled* difference of two of these individuals is added to the third one and and intermediate individual $X_{\text{mutant}}$ is created:

$$X_{\text{mutant}} := X_{r_1} + F(X_{r_2} - X_{r_3}). \tag{2}$$

The scaling factor F is the first parameter of DE. Then the $X_{\text{trial}}$ is generated by randomly inheriting variables from either $X_{\text{mutant}}$ or from $X_{\text{target}}$. In a problem with $n$ variables, one variable called $inv$ is automatically inherited from $X_{\text{mutant}}$ to avoid generating an individual which is already in the population. This can be seen in (3), where

$\mathrm{rand}_U(0,1)$ is a generator of uniformly randomly distributed numbers in $[0;1]$.

$$x_{\mathrm{trial},i} := \begin{cases} x_{\mathrm{mutant},i} & \text{if } \mathrm{rand}_U(0,1) < \mathrm{Cr} \text{ or } i = inv \\ x_{\mathrm{target},i} & \text{else} \end{cases} \tag{3}$$

The number Cr in (3) is the second parameter of DE and it is called the *crossover probability*. Cr controls the proportion of variables which change in a single individual. When $\mathrm{Cr} = 0$, only one variable changes at a time, hence this value is particularly well suited for *separable* problems.

## 1.3 OPTIMIZATION

Next, we look at one of the main applications of differential evolution and evolutionary computation in general.

### 1.3.1 *Single-objective Model*

Optimization in general is the endeavor to find a solution that is extremal in some sense. In the simplest case, the problem is to find the global minimum or maximum of a function, often subject to constraints, such as in (1). Since minimization and maximization are mutually interchangeable by substituting $f$ for $-f$, from now on we deal only with *minimization*. The function $f$, that we are trying to optimize is called an *objective*.

If we know the algebraic definition of $f$, we can find the global extrema by exact methods, such as by finding the zeros of its gradient. Similarly, when the definition of $f$ falls into some specific class of well understood optimization problems, we may use the appropriate method. One example of well understood optimization problems are the linear or convex problems. However, sometimes the definition of $f$ is so complicated that it falls out of the realm of exact methods. This is especially the case when the evaluation of $f$ involves running a computer simulation, such as with the design of the trajectory of spacecraft around Jupiter's moons [35]. In such cases evolutionary computation is particularly useful, since it does not make any assumptions on what $f$ looks like.

More often than not, there is more than one quantity which we endeavor to optimize. For example, we may want to design an engine which has low consumption but high power. In such situations we say that we have more than one objective.

### 1.3.2 *Multi-objective Model*

In many regards, the field of multi-objective optimization is different from single-objective optimization. In single-objective optimization there is usually one single solution to the problem. But in multi-objective optimization, the solution which is optimal for one objective is usually not optimal for the other objectives. The more we improve the value of one objective, the more the values of other objectives deteriorate. This means that there are trade-offs between the various objectives.

For example, in a problem to find the optimal design of a car engine, we may consider the

- *gasoline consumption*
- *production cost* and
- *power*

to be three objectives that we are trying to optimize. It is clear that if we minimize the cost, we cannot afford to have a very powerful or a very low consumption engine. On the other hand, if we want to increase the power of the engine, we may do it by increasing its consumption or by using better materials, thus increasing the production cost. Similarly, if we want to minimize the gasoline consumption, we may do it either by decreasing the power of the engine or again by using better materials, thus increasing the production cost. The real world is full of trade-offs and therefore multi-objective optimization is a superior model to that of single-objective.

Instead of looking for one single solution to the problem, such as in single-objective optimization, the goal of multi-objective optimization is to find a large number of solutions. The set of these solutions represents the trade-offs between the various objectives that are being optimized.

It turns out that evolutionary algorithms are especially useful for multi-objective optimization, since they already contain a population of solutions, which means that they are able to generate arbitrarily many solutions to the problem in one run.

## 1.4 goals

In the previous sections it was revealed that the differential evolution algorithm is a very good choice for continuous *optimization* problems. Next, we tried to persuade the reader that the most relevant real world optimization problems are naturally formulated as *multi-objective* problems. DE, being an evolutionary algorithm is especially well suited for multi-objective problems. Problems which arise from application of DE to multi-objective optimization are the center of this thesis.

In this work we explore the challenges that arise when we try to apply *differential evolution* to solve *multi-objective* optimization problems. As the title of this thesis suggests, the first goal is to improve our understanding of multi-objective DE. In particular, we concentrate on the DE parameters $F, Cr$. First, we explain the influence of these parameters on the performance of DE. Once we have a basic understanding of this subject, we explore and analyze various mechanisms for the automatic setting of DE parameters. The second goal is to improve the performance of multi-objective differential evolution. In particular, we concentrate on the geometric computation that is specific to the multi-objective domain. Next we look at each goal separately and explain how this goal was accomplished.

### 1.4.1 *Understand the role of parameters*

The success of DE is highly affected by the right choice of parameters $F$ and $Cr$. Although significant progress has been made in the single-objective realm, the choice of competitive DE parameters for multi-objective problems is still far from being well understood. Authors of successful multi-objective DE algorithms usually use parameters which *do not* render the algorithm *invariant* with respect to *rotation* of the coordinate axes in the decision space.

In Chapter 4 we try to see if such a choice can bring consistently good performance under various *rotations* of the problem. We do this by testing a DE algorithm with many combinations of parameters on a testbed of bi-objective problems with different modality and separability characteristics. Then, we explore how the performance changes when we rotate the axes in a controlled manner. We find out that our results are consistent with the single-objective theory but *only for unimodal problems*. On multi-modal problems, unexpectedly, parameter settings which do *not* render the algorithm rotationally invariant have a consistently good performance for all studied rotations. This chapter is based on our earlier work [21].

### 1.4.2 *Understand parameter control*

As was mentioned before, parameter selection is a big issue for multi-objective differential evolution. To mitigate the problem of setting parameters, methods have been developed to automatically adjust the parameters. These methods are usually presented as a part of a unified algorithm and since these algorithms vary in other aspects than the parameter control mechanism, it is difficult to compare them.

In Chapter 5, we go through various deterministic, adaptive, and self-adaptive approaches to parameter setting, isolate the underlying parameter control mechanisms and apply them to a single simple differential evolution algorithm. We then observe its performance and behavior on a set of benchmark problems. We find that even the simplest mechanisms can compete with parameter settings found by exhaustive grid search. We also notice that *self-adaptive* mechanisms seem to perform better on problems which can be optimized with a very limited set of parameters. *Adaptive* mechanisms on the other hand exhibit significant problems on the more difficult problems. By examining the trajectory on which the evolved parameters move we reveal that the parameters in adaptive methods evolve along more or less the same path, regardless of the problem. This is a vulnerability of the methods and it should be explored in more detail.

### 1.4.3 *Improve performance*

One of the biggest advantages of evolutionary algorithms, is their ability to find many solutions at once. This is particularly useful in the multi-objective model, where we are explicitly interested in finding many distinct solutions.

However, there are drawbacks to this approach. Maintaining a big population of individuals comes with computational overhead. Evolutionary algorithms need to constantly evaluate the quality of individuals in the population in order to select individuals for survival and for recombination (Algorithm 1, lines 3 and 7). This is done using procedures such as *non-dominated sorting* [12] and diversity estimation [37]. These procedures are specific to the multi-objective model and they are relatively computationally expensive. Moreover, as we see in Algorithm 1, they are performed in each generation and their cost grows with the number of objectives and individuals.

In Chapter 6 we propose a new method to decrease the cost of these procedures, with special emphasis to *non-dominated sorting*. Our approach is to determine the non-dominated individuals at the start of the evolutionary algorithm run and to update this knowledge as the population changes. In order to do this efficiently we propose a special data structure called the *M-front*, to hold the non-dominated part of the population. The M-front uses the geometric and algebraic properties of the Pareto dominance relation to convert *orthogonal range queries* into *interval queries* using a mechanism based on the nearest neighbor search. These interval queries are answered using dynamically sorted linked lists. Experimental results show that our method can perform significantly faster than the state of the art Jensen-Fortin's algorithm [29], especially in many-objective scenarios. A significant advantage of our approach is that if we change a single individual in the population, we still know which individuals are dominated and which are not.

This approach is applicable to most multi-objective evolutionary algorithms, however some features present specifically in differential evolution can be exploited to achieve even greater reduction of computational cost.

# 2 DIFFERENTIAL EVOLUTION

## 2.1 INTRODUCTION

In this chapter we look at differential evolution in detail. We restrict ourselves to the single-objective optimization model. We leave the multi-objective model, which is the focus of this work, for the following chapter.

Kenneth Price and Rainer Storn developed DE from the so called Genetic Annealing Algorithm in 1995 [55, 53]. Their motivation was to find a solution to the Chebyshev polynomial fitting problem, but they found out that DE works remarkably well on a broad range of problems [54]. Since then, DE has been applied to many other problems with relatively great success. The main textbook on DE is [45] and a recent survey of the state of the art is [8].

## 2.2 FUNDAMENTAL IDEAS OF DE

The main idea of differential evolution is to generate mutations of individuals by adding the *difference* of existing individuals in the population. This may seem counterintuitive at first, but we explain the motivation of such design in the next section. Now we explain the canonical DE algorithm.

Let us have a continuous optimization problem:

$$f : \mathbb{R}^n \supseteq H \mapsto \mathbb{R};\ f \text{ is to be minimized}$$

where $H$ is an $n$-dimensional hyperbox:

$$H = [a_1; b_1] \times [a_2; b_2] \times \cdots \times [a_n; b_n] \tag{4}$$

We call any element of $\mathbb{R}^n$ a solution to $f$. In the context of evolutionary computation solutions are called *individuals* and from now on we use the terms *solution* and *individual* interchangeably. Algorithm 2 describes the simplest form of DE. Let $P$ be the population. First we fill $P$ with NP individuals, each drawn independently from a uniform random distribution on $H$ (line 1). We get a population $P = \{X_1, ..., X_{\text{NP}}\}$, where $X_i = (x_{i,1}, \ldots, x_{i,n}) \in \mathbb{R}^n$. This population is then evolved for $G_{max}$ generations in a so called *evolutionary loop*. The population size NP remains constant throughout the entire run of the algorithm and it is one of the three fundamental parameters of DE.

The population is traversed in a so called *generational loop* and the algorithm is attempting to improve each individual in turn. The incumbent individual being improved is called a *target* individual. For each target individual a *trial* individual, which is hopefully better than target, is generated. The generation is performed in two steps. At first an individual called *mutant* is generated by adding a scaled difference of two randomly selected individuals to a third randomly selected individual (line 5). The number F used for scaling is called the *amplification factor* and it is the second fundamental parameter of DE.

The next step is to construct the trial individual by crossover between the mutant and the target. There are various types of crossover, but in the simplest case *uniform* crossover is used. In order to avoid inheriting all variables from the target and generating a duplicate individual, a random index inv is selected and the variable with this index is inherited from the mutant. Next, the trial inherits variables either from the mutant, with probability Cr, or from the target, with probability $1 - \mathrm{Cr}$. The number Cr is called the *crossover probability* and it is the final parameter of DE.

**Algorithm 2:** Default differential evolution algorithm

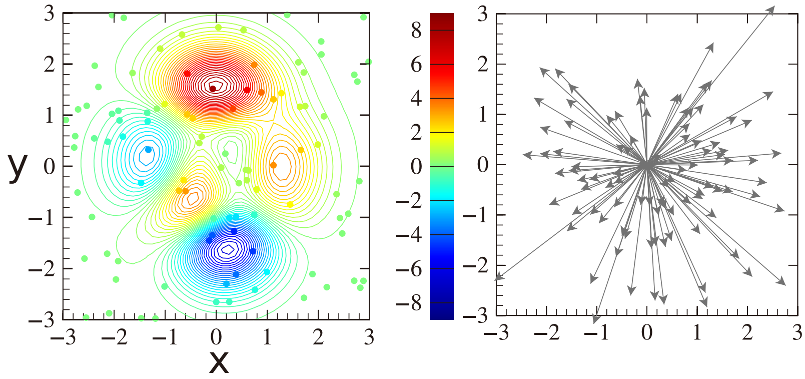
```
1  initialize $P = \{X_1, ..., X_{NP}\}$ uniformly randomly in $H$
2  for generation := 1 to $G_{max}$ do  Evolutionary loop
3      for target := 1 to NP do  Generational loop
4          randomly generate mutually distinct $r_1, r_2, r_3 \neq$ target
5          $X_{mutant} := X_{r_1} + F(X_{r_2} - X_{r_3})$
6          randomly generate inv $\in \{1, \ldots, n\}$
7          for $i := 1$ to $n$ do
8              if $\mathrm{rand}(0.0; 1.0) < \mathrm{Cr}$ or $i = \mathrm{inv}$ then
9                  $x_{trial,i} := x_{mutant,i}$
10             else
11                 $x_{trial,i} := x_{target,i}$
12             end
13         end
14         project $X_{trial}$ to $H$
15         if $f(X_{trial}) \leqslant f(X_{target})$ then
16             $X_{target} := X_{trial}$
17         end
18     end
19 end
20 report best $X$ in $P$
```

After a trial individual is generated, it is projected back to H in order to satisfy the problem constraints (4). It is then compared to the target individual on line 15. If the trial achieves a better or equal value of f, the target is replaced with the trial to simulate the survival of the fittest. The fact that an equal value of f also triggers the target to be replaced is an important part of DE, since it allows the algorithm to remain in motion even if f contains flat regions.

## 2.3 INTUITION BEHIND DE

Most people find the idea of adding random differences of vectors to other vectors baffling. Here we try to explain the motivation behind this design. We track the progress of Algorithm 2 with $F = 0.5, Cr = 0.2, NP = 100$ on the *peaks* test function: [1]

$$\begin{aligned} f(x,y) :=\ & 3(1-x)^2 * e^{-x^2-(y+1)^2} \\ & - 10(\frac{x}{5} - x^3 - y^5)e^{-x^2-y^2} \\ & - \frac{1}{3}e^{-(x+1)^2-y^2} \end{aligned}$$



(a) Population

(b) Scaled differences

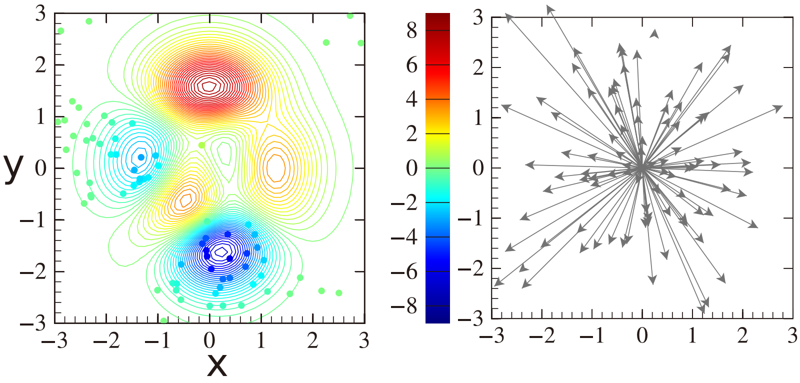
Figure 2: Generation 1

This function has several local minima and maxima. The initial population which is generated randomly in $[-3;3] \times [-3;3]$ is shown in Figure 2a. The scaled difference vectors $F(X_{r_2} - X_{r_3})$ which are generated in the first generation are plotted in Figure 2b. These vectors represent the search directions in the next generation, since they are added to existing individuals to produce new individuals. At the beginning of the algorithm, when the population is uniformly distributed, these vectors point more or less uniformly in all directions and their magnitude is relatively big. This allows the population to *explore* the search space.

The population after 5 generations is shown in Figure 3. It is starting to concentrate around the two local minima. As the population starts to concentrate more and more in more favorable regions of the search space, the scaled difference vectors decrease in size as in Figure 4b.

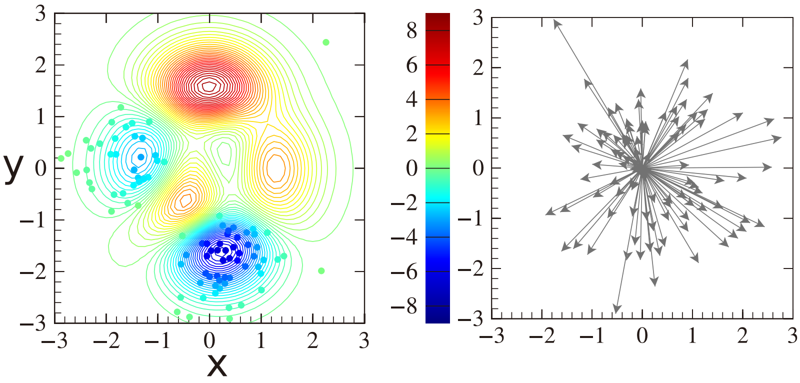
After 20 and 30 generations, as the population collapses around the local minima, the difference vector distribution also collapses. This means that the favorable area have been discovered and the al-

[1] This parameter setting is far from optimal for the given problem but it results in progress which illustrates the DE concepts very clearly.



(a) Population (b) Scaled differences

Figure 3: Generation 5



(a) Population (b) Scaled differences

Figure 4: Generation 10

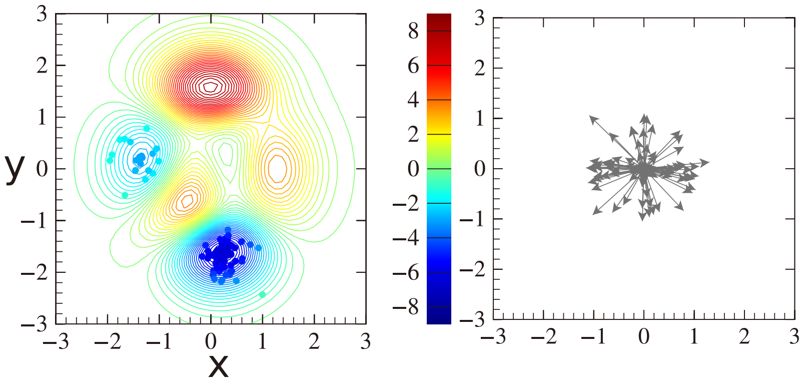
gorithm is searching in their vicinity. The search process went from being global to local. We can see this in figures 5 and 6.

There are two local minima, one of which has a lower value of f. The algorithm generates individuals in both these optima, but the individuals generated in the weaker cannot prevail against the ones in the stronger one and die out. After 50 generations the population converges in the more favorable local optimum, which is also the global optimum. We can see this in Figure 7.

The fact that the search directions are generated by sampling differences of individuals means that the algorithm does not change its behavior if the scale of the variables changes. The search directions *adapt* to the population. This behavior is called *contour matching* [45].

## 2.4 DIFFERENTIAL EVOLUTION PARAMETERS

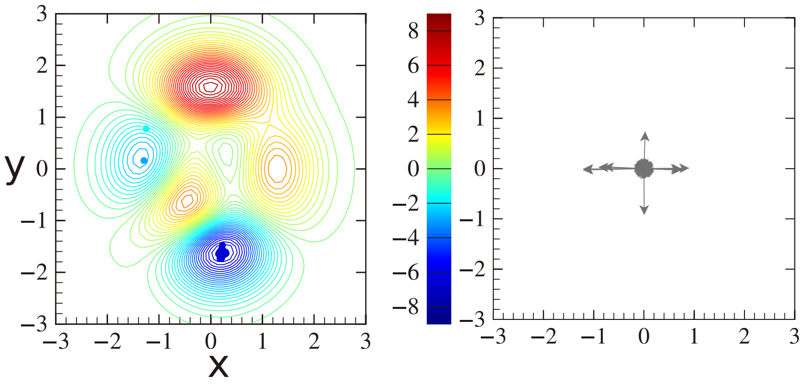
Now we review our knowledge about how DE parameters influence the search process.



(a) Population

(b) Scaled differences

Figure 5: Generation 20



(a) Population

(b) Scaled differences

Figure 6: Generation 30

### 2.4.1 *Crossover Probability and Separability*

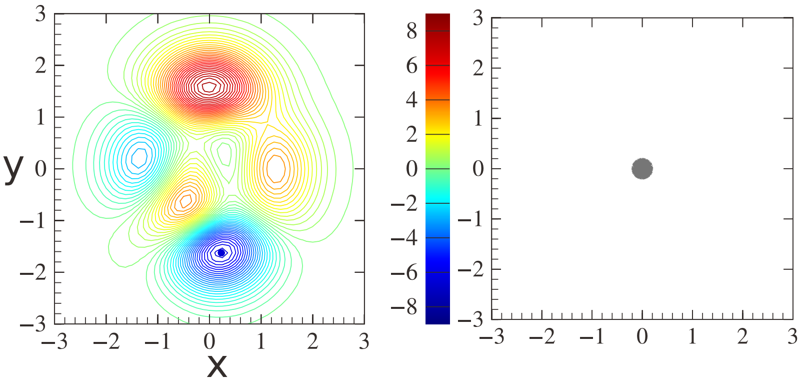
The crossover probability Cr has a very significant meaning in the context of objective function separability. In this section we summarize what we know about the relationship between the separability of the test functions and the good choice of the Cr parameter.

There are many different types of separability. One of the simplest is *additive* separability. A function $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$ is called *additively* separable if:

$$\exists f_1, \dots, f_n \text{ such that } f(x_1, \dots, x_n) = \sum_{i=1}^{n} f_i(x_i) \tag{5}$$

The most important consequence of additive separability is that the $n$-dimensional problem can be optimized sequentially *one variable at a time*. Therefore separable problems do not become much more difficult when the dimension of the search space increases [49].

Salomon [49] illustrates the problems of algorithms which vary the individuals *one variable at a time* on a quadratic function of two variables in Figure 8. The ellipses in the left part are contours of a *separable*



(a) Population

(b) Scaled differences

Figure 7: Generation 50

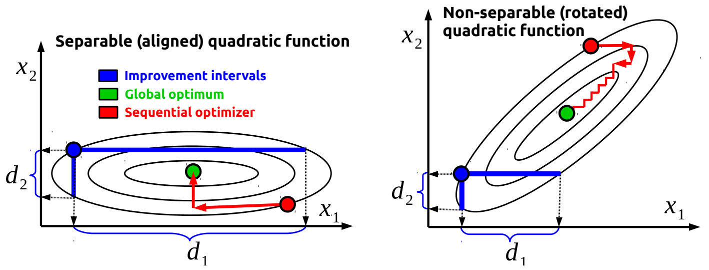


Figure 8: Illustration of variable-wise optimization on a rotated quadratic function

quadratic function. We can see two individuals on one of the contours. The blue individual represents an individual in a randomized algorithm. If we mutate one variable of this individual, the probability to get an improvement in the objective function is relatively high, since the improvement intervals $d_1, d_2$ are long. If we rotate the coordinate axes, thus rendering the function non-separable, the improvement intervals shrink.

One more illustration of problems which arise is using a sequential deterministic algorithm which finds the optimum with respect to *one variable at a time*. The red individual illustrates the path of one such an algorithm. When the function is aligned with the axes, this algorithm achieves optimum in just two iterations, while in the rotated case the algorithm not only progresses slower, but never actually reaches the optimum.

In the context of DE, Cr is the approximate number of variables that the trial inherits from the mutant. If $\mathrm{Cr} = 0$, then only one variable is inherited and the new individual is almost entirely the same as the target individual, which is already in the population. That is, small values of Cr induce the algorithm to search *along the axes*. this means that *small* values of Cr are good for separable problems, but may run into significant difficulties on non-separable problems.

### 2.4.2 *Variance as a Common Currency*

Probably the most significant work on the theoretical properties of DE has been written by Zaharie [61]. Let us collect all the trial vectors that are generated in the course of one generational loop of Algorithm 2 into a set $P_{\text{trial}}$. Then the relationship between the *variance in search space* of $P$ and $P_{\text{trial}}$ is given by the simple equation $\mathbb{E}[Var(P_{\text{trial}})] = c\mathbb{E}[Var(P)]$ where:

$$c = 2F^2Cr + \frac{Cr^2 - 2Cr}{N} + 1 \tag{6}$$

Zaharie omits the fact that in most DE variants the individuals which generate the trial individual are chosen *distinct* from the target individual (Algorithm 2 line 5). However her results hold unchanged also after adding this assumption.

The work of Zaharie is important since it transforms the two parameters into a single number $c$ (common currency) which has a very intuitive interpretation. If $c < 1$ we see that the algorithm tends to contract the population while if $c > 1$ it expands the population. Based on empirical data Kukkonen concluded in [39] [48] that a good choice of parameters is one that satisfies $c \in [1.0; 1.5]$ with the upper bound not very strict.

Having established the fundamental concepts and notation of single-objective DE, let us now look at the multi-objective realm in the following chapter.

# 3 MULTI-OBJECTIVE DIFFERENTIAL EVOLUTION

First we introduce the concept of Pareto dominance, which is essential for all multi-objective problems. Then we explain how this concept is applied to differential evolution to produce a working multi-objective optimizer and finally we go briefly through some examples of successful multi-objective differential evolution algorithms.

## 3.1 PARETO DOMINANCE

Let us have a *minimization* problem $F$ consisting of $M$ objective functions:

$$F = (f_1, f_2, \dots, f_M)\,.$$

Each function has $n$ variables:

$$f_i : D \subseteq \mathbb{R}^n \mapsto \mathbb{R} \qquad \text{for } i = 1, \dots, M.$$

The problem may contain arbitrary constraints. We say that $F$ maps the *decision* space $D$ to the *objective* space $\mathbb{R}^M$. We call the members of the *decision* space *decision vectors* and the members of objective space *objective vectors*.

Only in extremely rare occasions it holds that there is one solution, or a set of solutions, which optimizes all the objectives:

$$\operatorname*{argmin}_{f_1} = \operatorname*{argmin}_{f_2} = \dots = \operatorname*{argmin}_{f_M}.$$

Usually a solution that produces the optimal value of one objective produces sub-optimal values of the other functions. The original concept of an optimal solution needs to be revised for multi-objective problems.

A solution of a multi-objective optimization problem is said to *Pareto dominate*, or simply *dominate*, a different solution if it is *better* with respect to at least one objective while not being worse with respect to *any* objective.

More formally, we say that an individual $X_1$ *dominates* an individual $X_2$ if

$$f_i(X_1) < f_i(X_2) \text{ for } \textit{some } i \in \{1, \dots, M\}$$

and

$$f_i(X_1) \leqslant f_i(X_2) \text{ for } \textit{all } i \in \{1, \dots, M\}.$$

We call this relation *Pareto dominance* and denote it as:

$$X_1 \prec X_2 : \quad X_1 \text{ dominates } X_2.$$

If for two individuals $X_1, X_2$ neither $X_1 \prec X_2$ nor $X_2 \prec X_1$, we call $X_1$ and $X_2$ *mutually non-dominated*.

We call an individual X *Pareto optimal* if it is not dominated by any other individual $X \in D$. We call the set of all Pareto optimal individuals the *Pareto optimal set* and the image of the Pareto optimal set under F the *Pareto front*. Dominance gives us a tool to objectively compare two solutions of a multi-objective problem. It is the goal of multi-objective optimization to find the Pareto optimal set, or at least its approximation.

## 3.2 PARETO DOMINANCE AS A QUALITY CONTROL MECHANISM

The Pareto dominance relation determines which solutions of a multi-objective problem are optimal. Moreover, it can be used to determine the relative *quality* of individuals in the population at any given moment. In Figure 9 we see the population of an evolutionary algorithm on a problem with objectives $f_1, f_2$. The blue circles depict individuals which are not dominated by any other individual *in the population*. The red area depicts the part of objective space that is dominated by some individual in the population and the red circles depict individuals that are *dominated*.

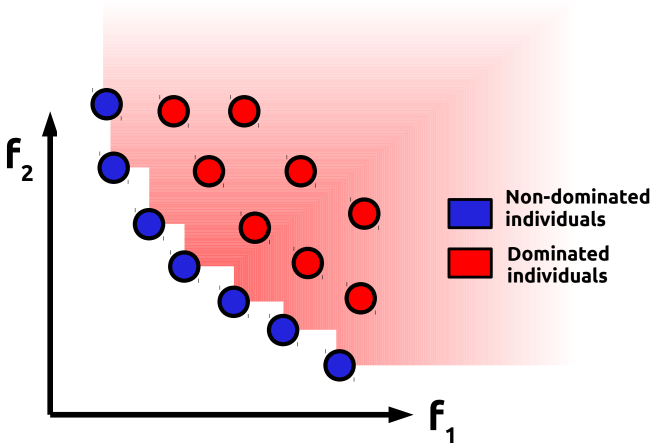


Figure 9: Non-dominated individuals in a population.

The population is now divided into dominated individuals, which are of a lesser quality, and non-dominated individuals, which are of a greater quality. This quality measure is somewhat crude and we can refine it in the following way.

The non-dominated individuals are of the best quality in the population. Let us say that they belong to the *first non-dominated front*. Now, let us *remove* the non-dominated individuals out of the population and determine the non-dominated individuals in this reduced population. We record these individuals as belonging to the *second non-dominated front*. Next we remove these individuals and repeat the

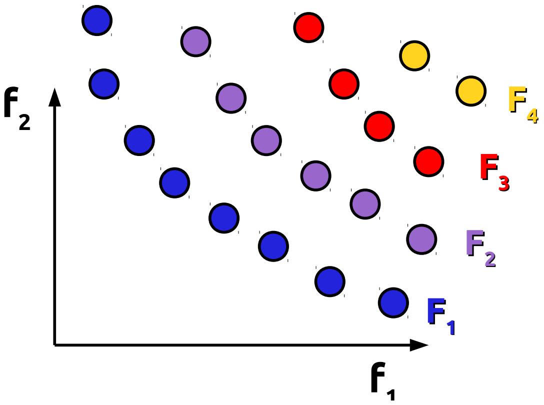


Figure 10: Non-dominated sorting of a population.

process until there are no more individuals left. We divided the population into non-dominated fronts, each defining a particular quality category. Non-dominated sorting is the process of dividing the population of a multi-objective evolutionary algorithm (MOEA) into *fronts* with respect to *Pareto dominance*. The result of non-dominated sorting, dividing the population of an evolutionary algorithm into fronts $\{F_1, F_2, F_3, F_4\}$ is shown in Figure 10.

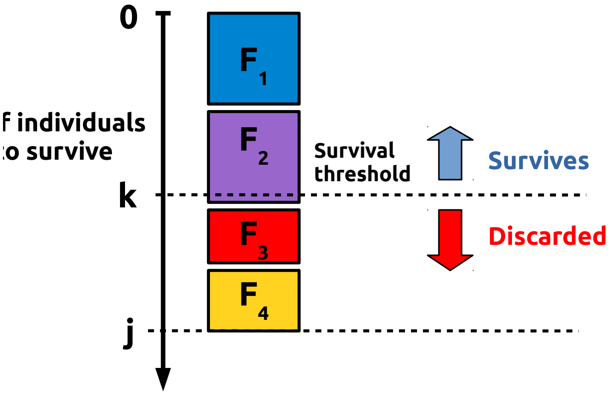


Figure 11: Selection for survival based on dominance.

Non-dominated sorting [12] is used primarily in the selection for survival (Algorithm 1 line 7). If we have a population of k individuals and we want to select the best j individuals to survive to the next generation, we can perform non-dominated sorting and select the individuals according to their non-dominated front. This is illustrated in Figure 11. We discard $F_3$ and $F_4$, but we cannot discard $F_2$, because we would lose too many individuals. Therefore survival selection based on dominance is usually supplemented by a secondary quality criterion, such as *diversity*.

## 3.3 SIMPLE EXAMPLE OF MULTI-OBJECTIVE DIFFERENTIAL EVOLUTION

By applying the concepts from the previous section to Algorithm 2, Robič and Filipič created an algorithm called DEMO [47]. This algorithm is described in Algorithm 3.

**Algorithm 3:** DEMO [47] algorithm

```
1  initialize $P = \{X_1, ..., X_{NP}\}$ uniformly randomly in the decision
   space
2  for generation := 1 to $G_{max}$ do  Evolutionary loop
3      for target := 1 to NP do  Generational loop
4          randomly generate mutually distinct $r_1, r_2, r_3 \neq$ target
5          $X_{mutant} := X_{r_1} + F(X_{r_2} - X_{r_3})$
6          randomly generate $inv \in \{1, \dots, n\}$
7          for $i := 1$ to $n$ do
8              if $rand(0.0; 1.0) < Cr$ or $i = inv$ then
9                  $x_{trial,i} := x_{mutant,i}$
10             else
11                 $x_{trial,i} := x_{target,i}$
12             end
13         end
14         project $X_{trial}$ to decision space
15         if $X_{target}$ dominates $X_{trial}$ then
16             discard $X_{trial}$
17         else if $X_{trial}$ dominates $X_{target}$ then
18             replace $X_{target}$ with $X_{trial}$
19         else if $X_{target}$ and $X_{trial}$ are mutually non-dominated then
20             add $X_{trial}$ to the end of the population
21         end
22     end
23     Trim the P to size N using non-dominated sorting[47] and
       MNN diversity[37]
24 end
```

The part from line 4 to line 14 is identical to that of Algorithm 2. The one-to-one survival selection of original DE (Algorithm 2 line 15) cannot be straightforwardly generalized to multi-objective problems for reasons we mentioned in the previous section. What happens instead is that the $X_{trial}$ and $X_{target}$ are compared with respect to *Pareto dominance*. If one of them dominates the latter, only one of them survives. The other one is discarded. If the two are mutually non-dominated (line 19), $X_{trial}$ is added into the population, increasing the population size. Therefore, at the end of each generational loop we end up with a population whose size is in the interval [NP; 2NP]. In order to maintain the same population size, the number of individuals needs to be reduced to NP. This is performed using *non-dominated sorting* and *diversity estimation* (line 23). In the original paper, Robič and Filipič use the *crowding distance* [12] procedure to estimate diversity, but in this work we shall use a more modern *product of M nearest neighbor distances* procedure [37].

Next we illustrate the working of this algorithm on a simple bi-objective problem:

$$f(x, y) = (x - 1)^2 + 3y^2$$
$$g(x, y) = (y - 1)^2 + 3x^2$$

The contour lines of these functions are depicted in Figure 12. The contours of f are drawn with solid lines and the contour lines of g are drawn with dashed lines. Some Pareto optimal points are plotted with black dots. Note that these points are either the optima of f or g itself or points where the contour lines of f and g touch.

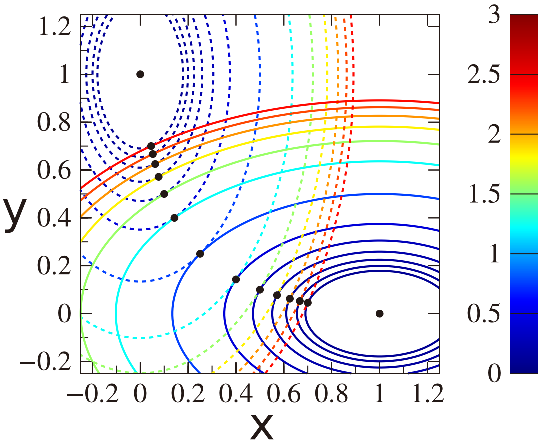
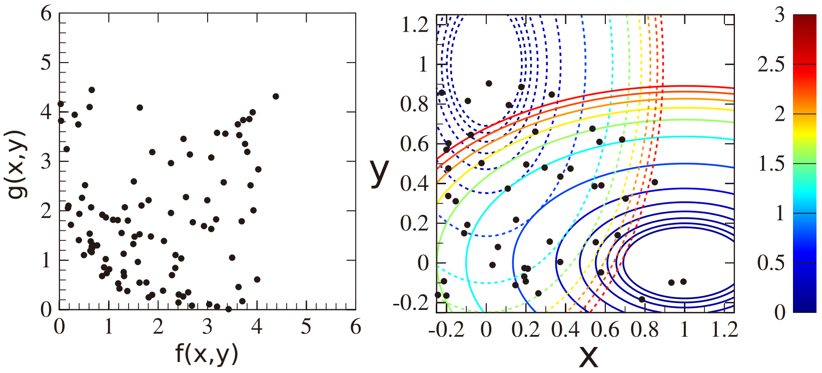


Figure 12: Decision space.

We initialize the population randomly and plot both the decision space and objective space in Figure 13. The situation after 5 generations is plotted in Figure 14. Already after 5 generations we see that the population starts to shift towards the places where the two sets of contour lines touch. In the objective space on the other hand, we see that the population begins to concentrate closer and closer to the origin, signifying that the value of both functions is decreasing.

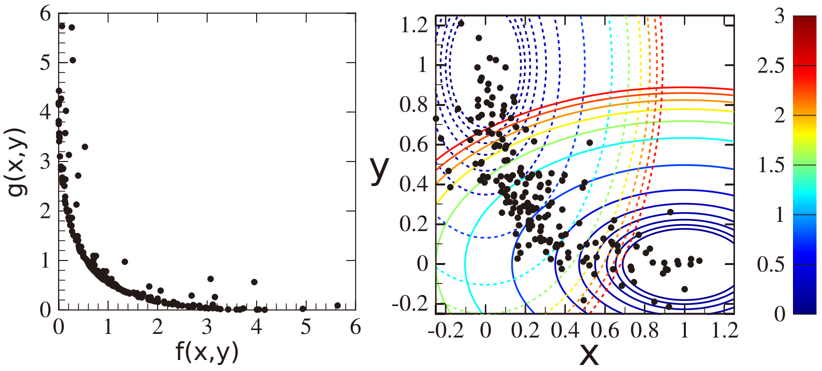
The situation after 10, 30 and 100 generations is depicted in Figures 15, 16 and 17 respectively. We see a sharp contrast with the single-objective differential evolution, where the population collapsed to a single point. In the multi-objective case, the population converges to the Pareto optimal set, which is a curve connecting the global minima of f and g.



(a) Objective space

(b) Decision space

Figure 13: Generation 1



(a) Objective space

(b) Decision space

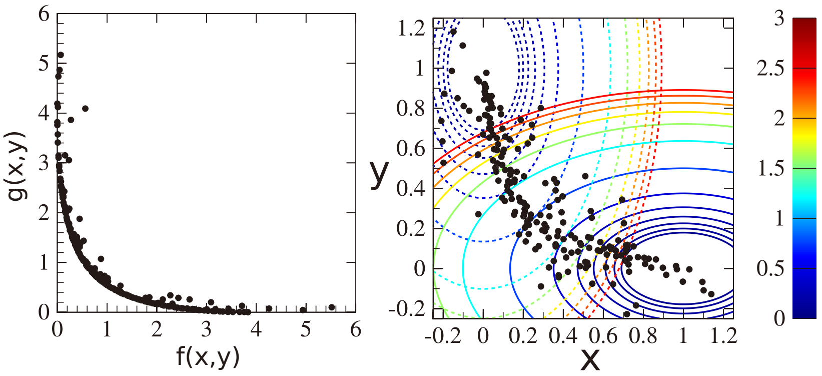
Figure 14: Generation 5

Since the population never collapses, the difference vector size remains big, which prevents the population from exploring the area near the Pareto optimal set in more detail. This causes the population in generation 100 look not much different than the population in generation 10. This testifies that the generalization of differential evolution to the multi-objective model is not trivial and many aspects need to be considered.

Next, we briefly go through the most notable work in multi-objective differential evolution.

## 3.4 NOTABLE ALGORITHMS

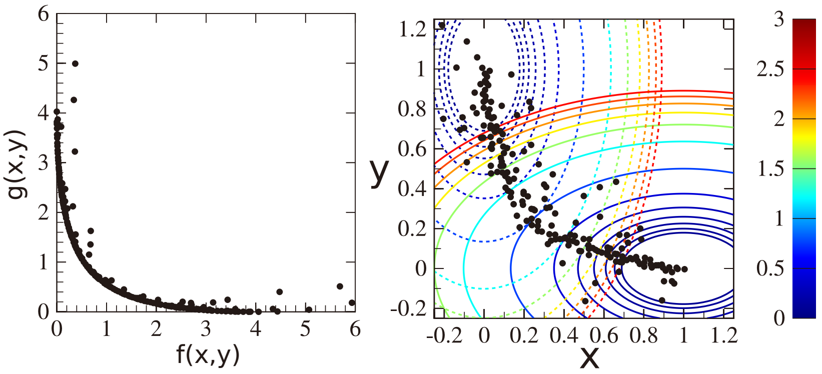
[43] contains an overview of multi-objective differential evolution algorithms until the year 2008. One of the first attempts to generalize DE to the multi-objective realm was by Abbass [2], who proposed the PDE (Pareto-frontier differential evolution). Shortly after, Abbass improved his algorithm by introducing *self-adaptation* to automatically



(a) Objective space

(b) Decision space

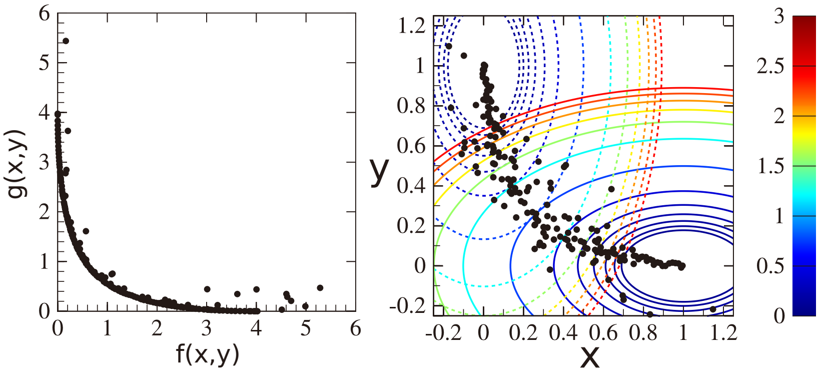
Figure 15: Generation 10



(a) Objective space

(b) Decision space

Figure 16: Generation 30



(a) Objective space

(b) Decision space

Figure 17: Generation 100

set the parameters F and Cr. Algorithms designed to deal with constraints have been developed by Kukkonen [37]. Much effort has been invested into automatic parameter setting, known as parameter control. We will investigate these algorithms in Chapter 5. Recently algorithms specifically designed for problems containing a great number of objectives have been proposed [56, 14].

## 3.5 COMPARISON WITH OTHER METHODS

To illustrate the power of DE, we present a comparison of the simple DEMO algorithm [47], presented in Algorithm 3, with the very well known NSGAII algorithm [12]. We have chosen NSGAII in particular, because it is very similar to the DEMO algorithm. In fact, the pruning procedure, based on dominance depth and crowding distance is the same. This means that the observed difference in performance can be attributed to the difference in the algorithm fundamentals, instead of auxiliary mechanism, such as pruning. The results are taken from [47].

Two versions of NSGAII have been compared. One with binary encoding of the individuals and one with real encoding. The parameters of DE are $F = 0.5$, $Cr = 0.3$, and $NP = 100$. The presented results are averages and standard deviations for 10 independent 250-generation runs. For each problem, the best result is marked in bold.

Table 1: Comparison of DEMO and NSGAII on the ZDT problems

| | Average Euclidean distance from the Pareto front | | |
|---|---|---|---|
| | NSGAII binary | NSGAII real | DEMO |
| ZDT1 | **0.000894 (0.00000)** | 0.033482 (0.00475) | 0.001083 (0.00011) |
| ZDT2 | 0.000824 (0.00000) | 0.072391 (0.03168) | **0.000755 (0.00004)** |
| ZDT3 | 0.043411 (0.00004) | 0.114500 (0.00794) | **0.001178 (0.00005)** |
| ZDT4 | 3.227636 (7.30763) | 0.513053 (0.11846) | **0.001037 (0.00013)** |
| ZDT6 | 7.806798 (0.00166) | 0.296564 (0.01313) | **0.000629 (0.00004)** |

We can see that DEMO outperformed both versions of NSGAII on all but one problem. These experiment are only illustrative, since the scope of this experiment is very limited. Nevertheless, DE is much more simple to understand and implement than the SBX operators and polynomial mutation present in NSGAII. Therefore DE should be considered as a promising direction of further research.

Part II

# IMPROVING OUR UNDERSTANDING OF MULTI-OBJECTIVE DE

In this part we will analyze existing multi-objective differential evolution algorithms and try to understand their behavior.

# 4 ROLE OF PARAMETERS IN MULTI-OBJECTIVE DIFFERENTIAL EVOLUTION

## 4.1 INTRODUCTION

The need for a versatile multi-objective optimizer has motivated researchers to generalize the basic DE for multi-objective problems. Now we have a great number of multi-objective DE variants. Many of them use the mechanism in Algorithm 2 to generate new individuals. In a problem with $n$ variables a new individual is created using a crossover variation operator which randomly selects $k; k \leqslant n$ variables which are perturbed. The magnitude of the mutation is generated by scaling a *difference* of randomly chosen individuals.

Many research papers on DE such as [14] or [47] provide little insight into how the authors chose the parameters for their benchmarking. We find this striking, since many authors choose their parameters such that the crossover operator perturbs only a small number of variables in an existing individual. In other words, the search for the Pareto optimal set proceeds *along the coordinate axes*. Since these algorithms perform very well [14, 47], we have a suspicion that this may be due to some characteristic of the problem, such as separability (section 2.4.1), that makes it easy to optimize along the axes. This would mean that if the axes are transformed, the algorithm should lose some performance.

Very strict warning against the practice of perturbing a small number of variables at a time has been raised as soon as 1996 by Salomon [49]. Salomon empirically demonstrated that the stellar performance of many popular single-objective genetic algorithms owes to the fact that most of the benchmark functions were *separable* and that the low mutation rate caused them to be optimized *one component at a time*. Once Salomon stripped the separability by *rotating* the principal axes of the benchmark functions, many algorithms were significantly slowed down, while some failed to converge completely. Salomon's theoretical results state that, in some cases, the probability of finding the global optimum can drop below that of random search. We are concerned that the same is true for the multi-objective realm since many authors perform their experiments with separable test functions.

In DE the number of variables that are perturbed is controlled by the Cr parameter. If all variables are perturbed, the algorithm has the same performance *regardless of rotation*. Let us have a parameter setting, that perturbs only a *small proportion* of the variables, which outperforms a setting that perturbs *all* variables. In this chapter we attempt to answer this question: Is this *exceptionally good* performance on a problem with a particular alignment of the coordinate axes balanced by *exceptionally bad* performance on a different alignment?

We do this *empirically* by observing the performance of a simple multi-objective algorithm DEMO (Differential evolution for multi-objective optimization (Algorithm 3)) [47] on a bi-objective subset of the WFG (Walking Fish Group) test suite [34]. We run all our experiments with a fixed population size and a fixed number of variables, while varying the *parameters*. Then we gradually *rotate* the problems in a controlled manner and observe the new behavior.

The answer to our question is, unexpectedly, *negative*. We find a statistically significant difference between the performance on the rotated problems and the original ones. Closer inspection reveals that a systematic performance *loss* happens when we rotate the *separable* problems, but the performance is still *significantly better* than for a rotationally invariant algorithm. We find that this happens for *multi-modal* problems, while *single-modal* problems exhibit the behavior we would expect from the work of Salomon.

In the following section we provide some background information on separability in the multi-objective realm. Next, we introduce the experimental design, where we explain which problems are used and why were they chosen. In addition we introduce a new performance metric called the *relative hypervolume*, and explain the controlled manner in which the rotations are generated. Finally, we present our data along with a discussion.

## 4.2 SEPARABILITY OF MULTI-OBJECTIVE PROBLEMS

Huband et al. from the Walking Fish Group (WFG) define separability from the optimizational standpoint [34]. A variable $x_i$ is separable if the set of global optima of a problem:

$$\underset{x_i}{\operatorname{argmin}}\, f(x_1, \dots, x_n)$$

is the same for any choice of the other variables $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$. For example, an additively separable function is WFG-separable, hence WFG-separability is a *generalization* of additive separability. The authors define a separable multi-objective problem as one where *each* objective is separable. The majority of the frequently used DTLZ and ZDT problems [13] are WFG-separable [34], while their objective functions are *not* additively separable as in (5).

We mentioned in Chapter 2 that in the single-objective realm separable problems are easily solvable using small values of Cr, while non-separable ones may produce significant difficulties. The multi-objective model is fundamentally different from the single-objective model because all objectives are being optimized *simultaneously*. The global optima of each optimized function constitute only a relatively small subset of the Pareto optimal set. Therefore, it is appropriate to ask if the problems of sequential algorithms which are illustrated in Figure 8 persist in multi-objective optimization. Also, while additively separable *unimodal* functions are inherently similar to the quadratic function in Figure 8, it is not clear if the intuition holds for *multi-modal* functions or for functions which are WFG-separable but not additively separable.

Note that $\mathrm{Cr} = 1$ is the *only* value of *Cr* for which the DE algorithm is rotationally invariant with probability 1. Rotational invariance does not by itself imply good performance. Its merit is that it allows us to *generalize* a single observation to an entire invariance class [31].

## 4.3 EXPERIMENTAL DESIGN

In this section, we describe which test problems we chose and why. We explain what we mean by *rotating the problem* and we propose a new performance metric which we use. We perform all our experiments using Algorithm 3.

### 4.3.1 *WFG Problems*

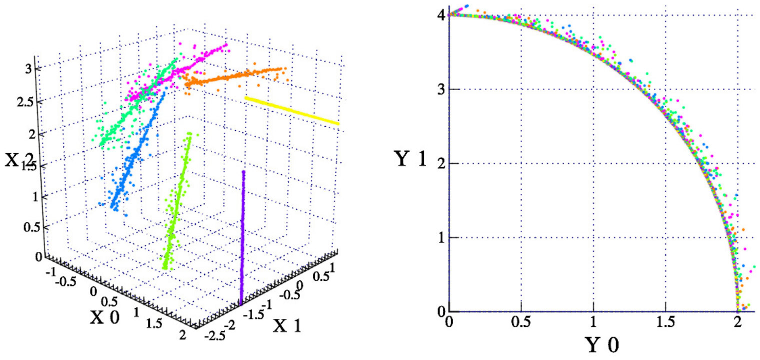
In order to explore the relationship between the control parameters of DE and the characteristics of the problem, we chose 4 problems from the WFG test suite [34]. These problems have been chosen since they have the same Pareto front and contain all possible combinations of the *WFG-separability* and *modality* characteristics. They are summarized in Table 2. We chose the number of variables to be 10 of which one is a positional variable.

Table 2: Characteristics of the selected WFG problems

| | WFG4 | WFG7 | WFG6 | WFG9 |
|---|---|---|---|---|
| separable | yes | yes | no | no |
| unimodal | no | yes | yes | no |

### 4.3.2 *Rotations in* $\mathbb{R}^n$

As humans we have a very good intuitive understanding of rotation in 2 or 3 dimensional space. However in higher dimensions things



(a) Decision space

(b) Objective space

Figure 18: Illustration of DE population on problems with various rotations

are not as intuitive as they might seem. An elementary rotation by the angle $\phi$ is characterized by the matrix:

$$R^e = \begin{pmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{pmatrix}$$

We can generalize this rotation to $n$-dimensional space by taking an $n$-dimensional identity matrix $I$ and replacing $I_{i,i}, I_{i,j}, I_{j,i}, I_{j,j}$ by $R^e_{1,1}, R^e_{1,2}, R^e_{2,1}, R^e_{2,2}$ respectively. We can see that the rotation is not executed *around an axis* as we might intuitively feel, but *around an* $n-2$ *dimensional subspace* which is coincidentally a 1-dimensional axis in the intuitive 3-dimensional case. For our experiments, we generate the rotation matrix $R$ by applying a rotation to *each* $n-2$ *dimensional subspace* in sequence, one rotation after the other.

We rotate the *entire decision space* (DS). This way the entire Pareto optimal set is always attainable since the entire decision space rotates along. In the case of WFG problems this means rotating a $n$-*dimensional hyper-box*. For example, in order to initialize the population in Algorithm 3 in the rotated DS (line 1), we first initialize the population in the original DS and then multiply by $R^{-1}$. Similar process is used to project the individual to the rotated DS on line 14. To evaluate the objective value of an individual we first multiply the decision vector by $R$ and evaluate the original objective functions.

In Figure 18 we can see an illustration of the DEMO algorithm population after 250 generations, with a small value of $Cr = 0.2$ on the WFG7 problem with 2 objectives and 3 variables, which is rotated by 0, 15, 30, 45, 60, 75, and 90 degrees. Various colors show the population on various rotations of the same problem. We see that the Parto front (in objective space) remains the same, while the Pareto optimal set (in decision space) rotates.

### 4.3.3 *Relative Hypervolume*

In our experiments, we use only one performance metric, the hypervolume (HV) [65], since it includes information on both convergence and spread of the individuals. With WFG problems, it is not easy to choose the reference point for the HV. Even if we choose the point as tight as possible, there are some individuals after the initialization of DE which dominate the reference point. Therefore the HV at the start is not zero and it is hard to say if a certain attained HV is good or bad. Moreover, it is hard to make quantitative comparisons based on HV. If some algorithm achieves HV of 100 and another one achieves a HV of 99.98, it may seem that the difference is not very big, but it all depends on the HV *at initialization*. If the algorithms started with $\text{HV} = 0$, the interpretation of the results would be quite different from one where $\text{HV} = 99.99$ at the start.

We attempt to mitigate this problem by subtracting the HV at initialization ($\text{HV}_{\text{init}}$) and normalizing the result using the *maximal attainable hypervolume* ($\text{HV}_{\text{max}}$). We define the relative hypervolume (RHV) in the following equation:

$$\text{RHV} := \frac{\text{HV} - \text{HV}_{\text{init}}}{\text{HV}_{\text{max}} - \text{HV}_{\text{init}}}. \tag{7}$$

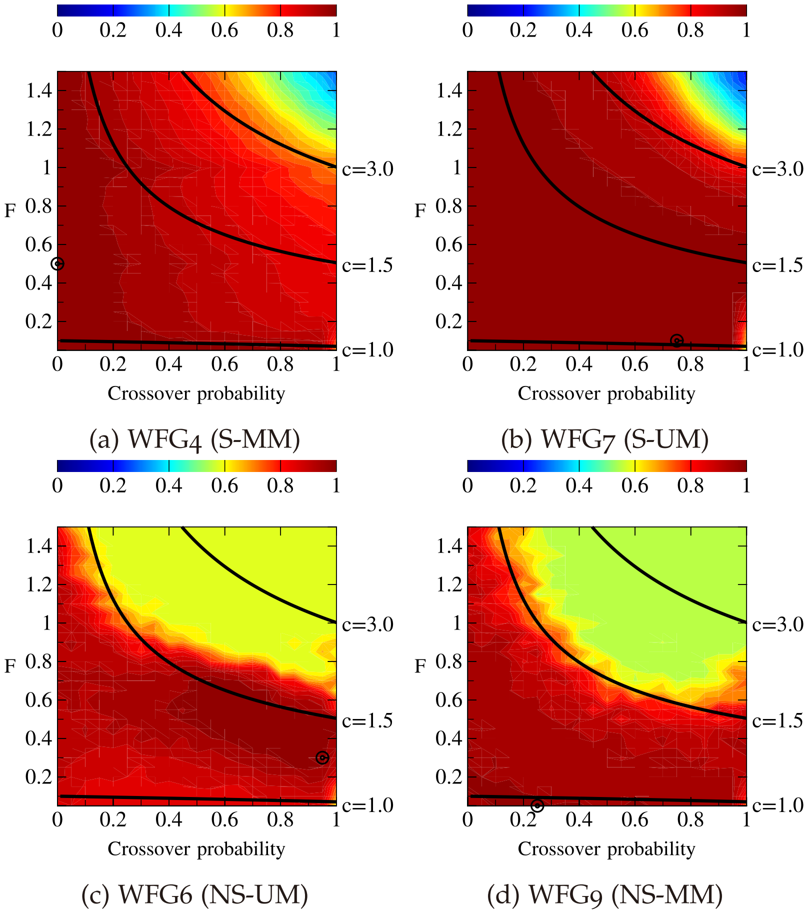
We compute $\text{HV}_{\text{max}}$ deterministically by integrating the space between the true Pareto front (PF) and the reference point. From (7), we have $\text{RHV} \in [1; -\infty)$. $\text{RHV} = 1$ implies convergence, RHV at initialization is 0 and $\text{RHV} < 0$ indicates an algorithm which is receding from the Pareto front.

We use RHV since its normalized nature is more intuitive and it is more robust with respect to the selection of the reference point. It may be more meaningful to compare two algorithm runs in terms of RHV. If we have two algorithm runs starting from the same randomly initialized population then the ratio of their RHVs is independent of the choice of the reference point. [1] On the other hand, two independent runs which produce the same final population may yield different relative hypervolume.

## 4.4 RESULTS AND DISCUSSION

In our experiments we varied the parameters $F \in [0.05; 1.5]$, $Cr \in [0; 1]$ equidistantly with a resolution of 0.05. For each combination we performed 10 runs of Algorithm 3. To simplify the setup, the population size was kept constant at 100 individuals and the length of each run was 250 generations. We explored the rotations from 0 to 90 degrees with a resolution of 5 degrees. In the following we discuss our results

[1] Given that the reference point is dominated by all individuals in the population.



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

(c) WFG6 (NS-UM)

(d) WFG9 (NS-MM)

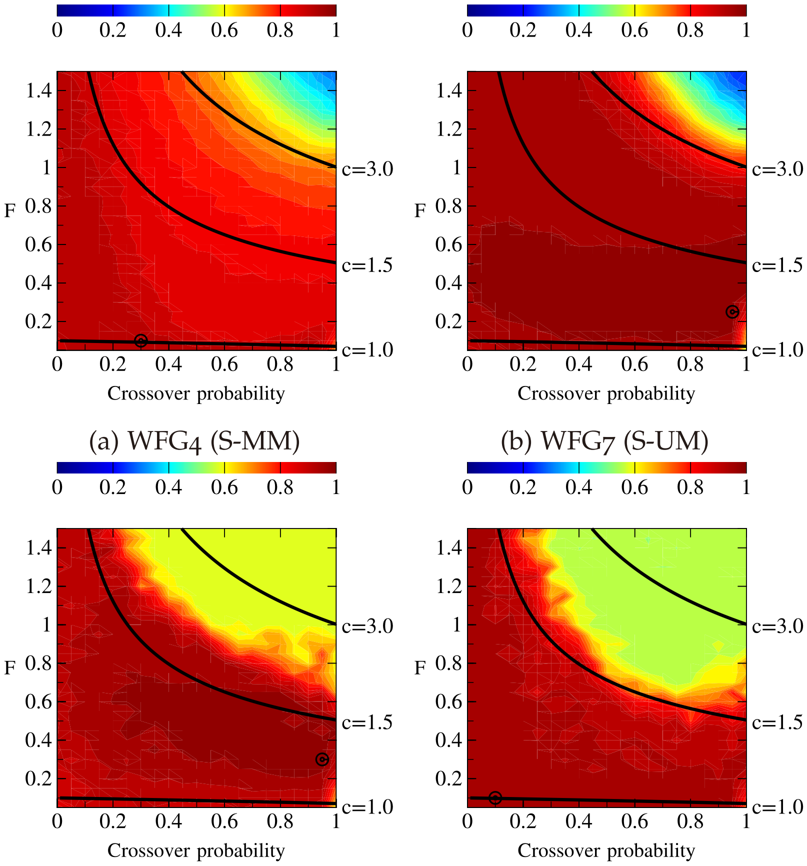
Figure 19: Average RHV without rotation

on a subset of the experimental data. To simplify the analysis, in each section we keep either F, Cr or the rotation angle fixed.

### 4.4.1 *Fixed Rotation Angle*

Figure 19 shows the average RHV on *non-rotated* problems. For illustration, we plot the combinations of F and Cr which result in $c = 1.0, 1.5$ and $3.0$ according to (6). The circle marks the combination of parameters with the best RHV. For each problem, an L-shaped favorable region containing RHV of 0.8 and higher, roughly corresponds to $c \in [1; 1.5]$. Low value of Cr is more robust, since it allows for a wider interval of F values. Unexpectedly, this holds also for non-separable problems WFG6 and WFG9.

The effect of introducing a rotation by 5 degrees is shown in Figure 20. The two figures seem identical, but the *ratio* of these averages in Figure 21 reveals a difference. A value of less than 1 indicates that the rotation caused the performance to decrease. We highlighted the contour at level 1 and marked the maximal and minimal value by circles. In order to make the results most readable we chose a color scale of



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

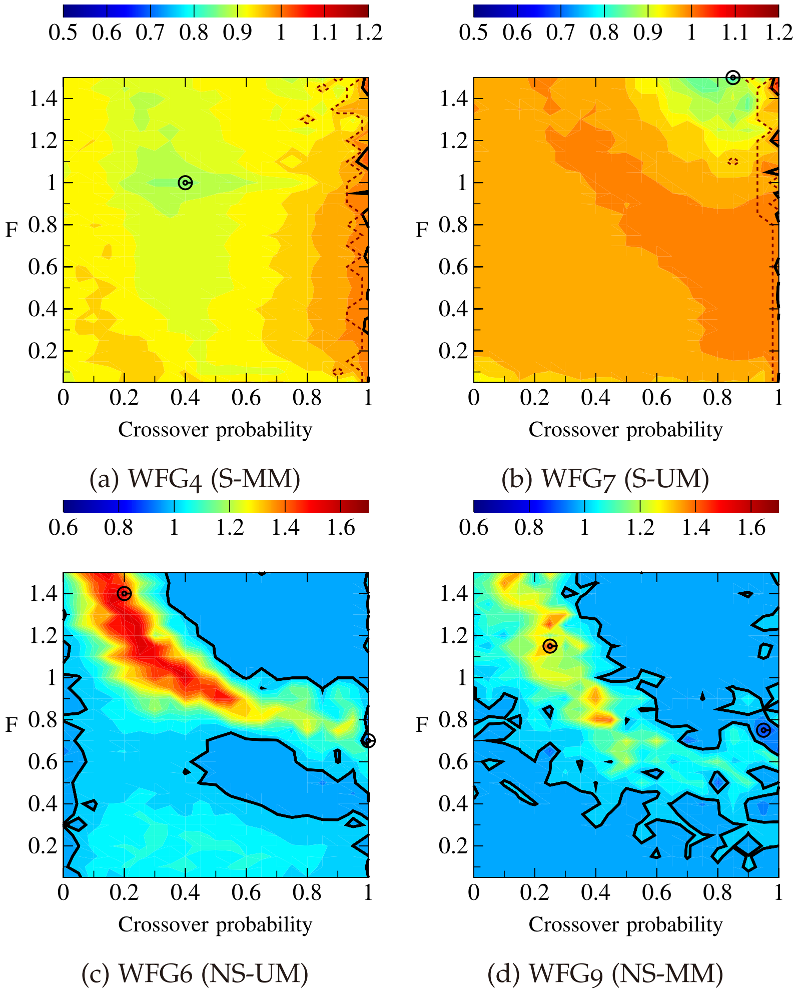
(c) WFG6 (NS-UM)

(d) WFG9 (NS-MM)

Figure 20: Average RHV with rotation angle of 5 degrees

[0.5; 1.2] for separable problems and [0.6; 1.7] for non-separable problems. The separable problems on the left half exhibit a performance loss consistent with Salomon's single-objective results. Performance dropped for almost all Cr smaller than 1. Non-separable WFG6 and WFG9 do not show such a systematic decrease. In some areas we even see an *increase* of performance.

It seems that there is relatively little difference between the rotated and non-rotated data. These result may seem not as significant as Salomon's. However, there is an important methodological difference. When he mentions that the performance on the rotated benchmark is *six orders of magnitude* worse than the performance on the non-rotated benchmark ([49, p.273]), he means that *the minimal attained value* $2.65 \cdot 10^5$ is six orders of magnitude worse in absolute numbers. But the *value at initialization* was three orders of magnitude greater yet. This means that both algorithms started somewhere near $2.65 \cdot 10^8$ and the non-rotated one progressed to $2.65 \cdot 10^{-1}$ while the rotated one progressed to $2.65 \cdot 10^5$. In terms of relative hypervolume,



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

(c) WFG6 (NS-UM)

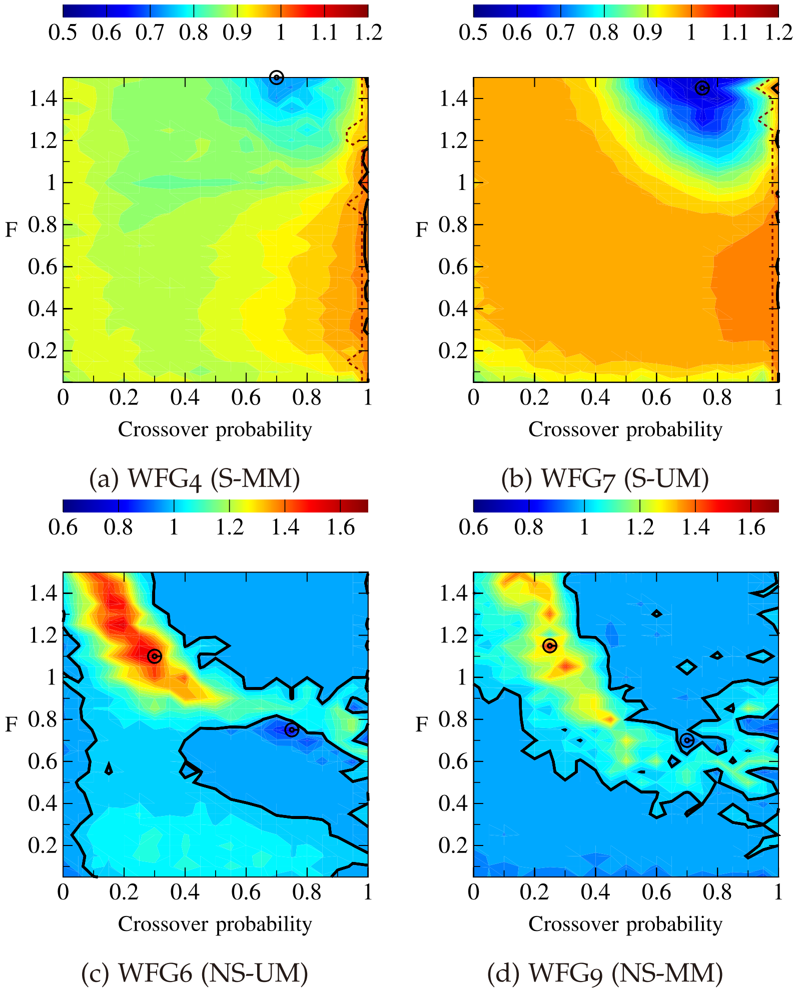
(d) WFG9 (NS-MM)

Figure 21: $\frac{\text{Average RHV with a rotation of 5 degrees}}{\text{Average RHV without a rotation}}$

this would be a very small difference [2]. In order to provide a scale-independent comparison, we compared all data using a two-tailed Wilcoxon signed rank test at a significance level of 0.05. For separable problems in Figures 21 and 22 we separate the parameter space with a dashed line into two areas. The area on the right is such that the rotated and non-rotated data is not significantly different, while on the left there is a *significant decrease in performance*. The data for non-separable problems contains areas of both significant decrease and significant decrease, as well as areas with no significant difference so in this case the separation cannot be plotted so comprendiously.

The effects are more visible with 45 degree rotation in Figure 22. Again, there is a *systematic* decrease in performance for the separable problems for $\text{Cr} < 1$. However, this decrease does not imply that $\text{Cr} = 1$ is a good choice. Looking at Figures 19 and 20, we see that $\text{Cr} = 1$

[2] Assuming that the minimum of the given function is 0, the difference would be on the order of $10^{-3}$.



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

(c) WFG6 (NS-UM)

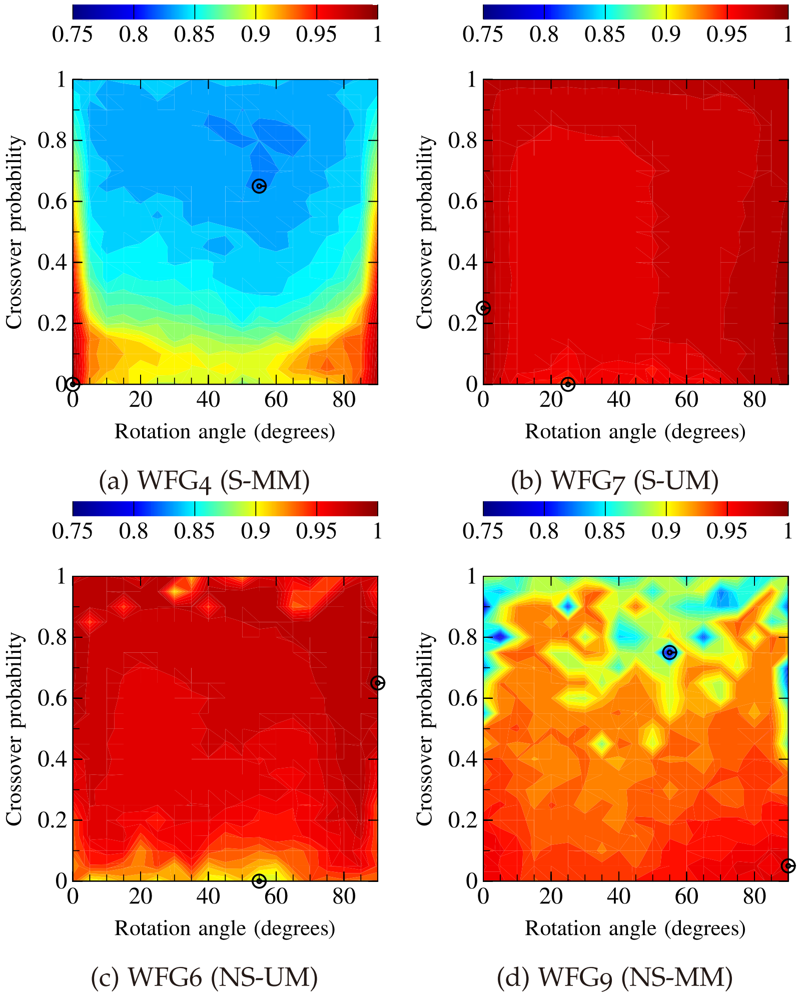
(d) WFG9 (NS-MM)

Figure 22: $\frac{\text{Average RHV with a rotation of 45 degrees}}{\text{Average RHV without a rotation}}$

is a consistently bad choice for the *multi-modal* problems WFG4 and WFG9.

### 4.4.2 *Fixed* F

In Figures 19 and 20 we see that F = 0.5 is compatible with many different values of Cr and achieves consistently good performance. The average RHV for F = 0.5 is shown in Figure 23. For *multi-modal* problems WFG4 and WFG9, *very low* values of Cr are *consistently* good for all studied rotations, while for *uni-modal* problems WFG6 and WFG7 *big* values of Cr yield a *consistently* good performance. On the other hand, poor performance is achieved with *big* values of Cr for multi-modal problems and *small* values for uni-modal problems. The data for WFG4 and WFG9 suggests that the exceptionally good performance of a small Cr setting does *not* have to be balanced by an exceptionally bad performance after the problem is rotated. Based on the observation from Figure 23 we see that for each problem either



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

(c) WFG6 (NS-UM)

(d) WFG9 (NS-MM)

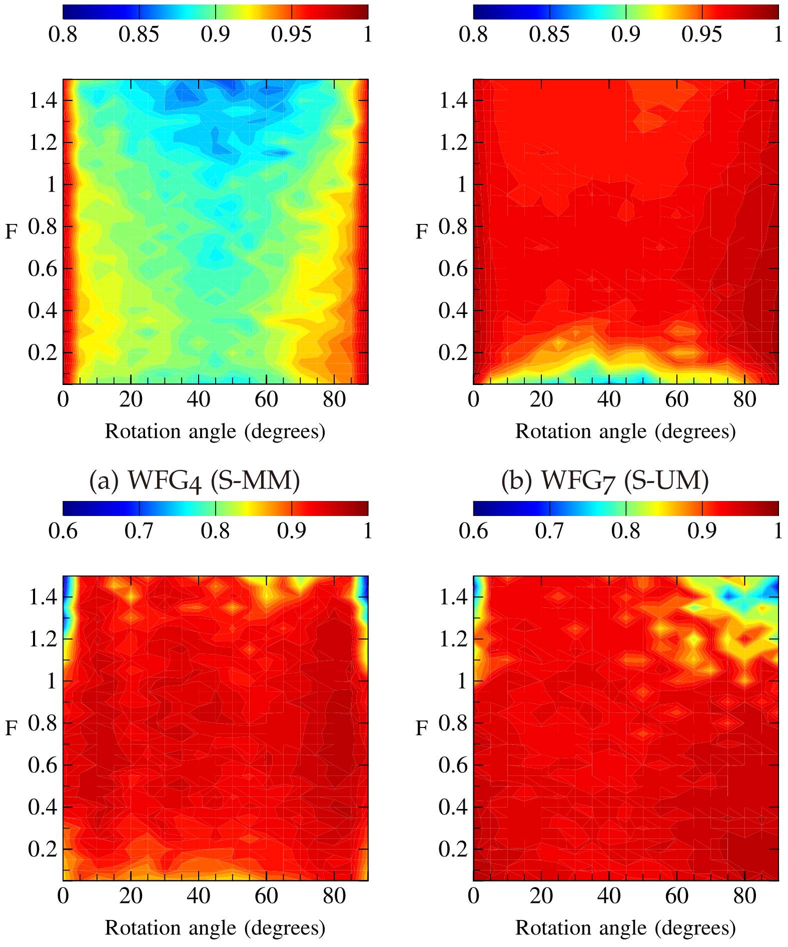
Figure 23: Average RHV for $F = 0.5$

$Cr = 0.1$ or $Cr = 0.9$ perform well through the observed spectrum of rotations.

### 4.4.3 *Fixed* Cr

In Figures 24 and 25 we see data with a fixed value of $Cr = 0.1$ and $Cr = 0.9$ respectively. For $Cr = 0.1$ the regions with the best performance are for rotations which are either close to 0 or 90 degrees. This is true also for non-separable problems, but it is more visible for separable problems. The data for $Cr = 0.9$ seems different. The choice of Cr close to 1 means that the algorithm is nearly rotationally invariant. The gained robustness with respect to coordinate rotation is balanced by lost robustness in the choice of F. Almost in all cases the interval with favorable values of F became shorter.

We see that for values of $Cr < 1$ there is a performance loss when the coordinate axes are rotated, but does the performance drop below that of a rotationally invariant choice of $Cr = 1$? The data sup-



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

(c) WFG6 (NS-UM)

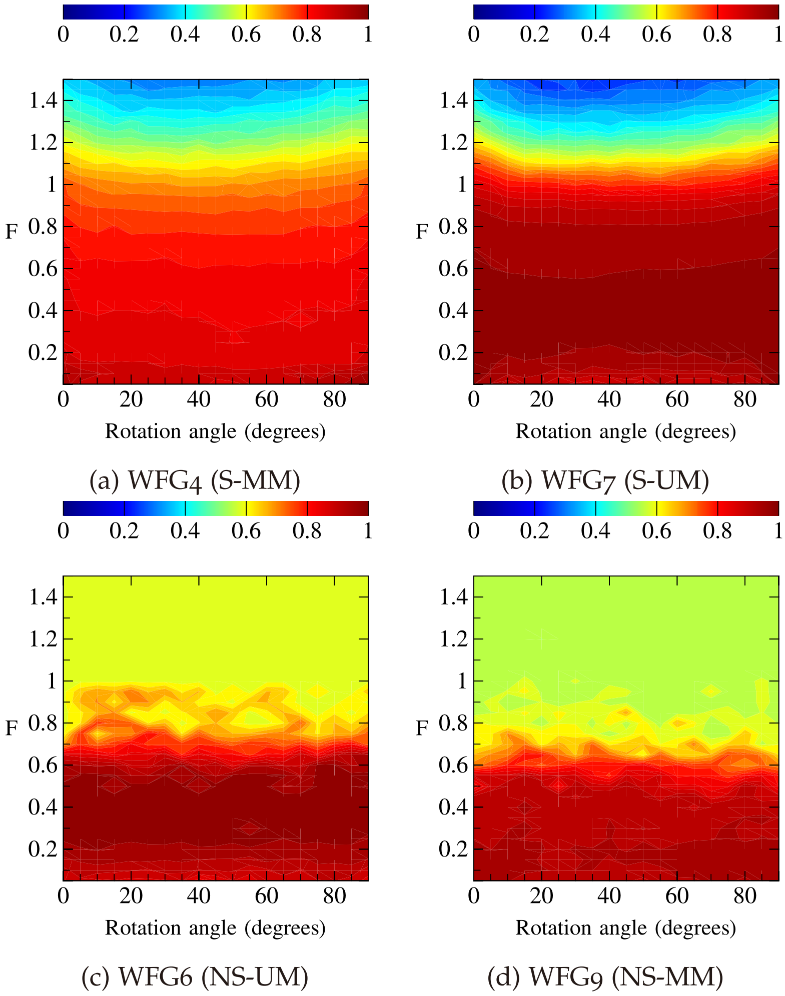
(d) WFG9 (NS-MM)

Figure 24: Average RHV for Cr = 0.1

porting a *negative* answer is presented in Figure 26. Here we divided the average RHV with Cr = 0.1 by the average RHV attained with a rotationally invariant Cr = 1. The interpretation of the dashed and full contour lines is the same as for Figures 21 and 22. For WFG4, the setting of Cr = 0.1 statistically significantly outperformed Cr = 1 for *all rotations* and *all* values of F. This means a definitive *negative* answer to our main question. The results are similar for the second multi-modal problem WFG9. Here we see a small region in which the data for Cr = 0.1 and Cr = 1 are not significantly different and Cr = 1 is significantly better in a few isolated cases. The unimodal problems on the other hand show that Cr = 1 is significantly better for most rotations and for the best performing values of F.

### 4.4.4 *Comparison in terms of generational distance*

Until now, we were concerned with a single performance metric, the dominated hypervolume. In order to give a more complete image, we



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

(c) WFG6 (NS-UM)

(d) WFG9 (NS-MM)

Figure 25: Average RHV for $Cr = 0.9$

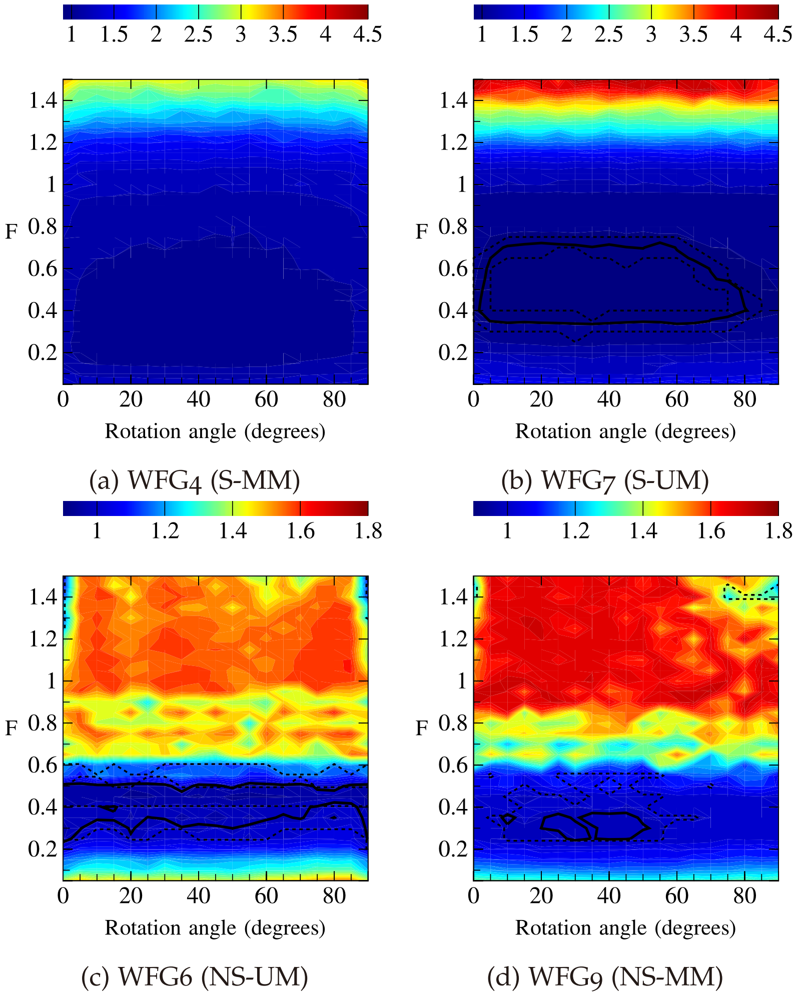
give the results in terms of a different metric, the so called *generational distance*. The generational distance GD of a Pareto front approximation $A = \{Y_1, \dots, Y_N\}$ is given by:

$$GD(A) = \frac{\sum_{i=1}^{N} d(Y_i)}{N}$$

where $d(Y_i)$ is the squared Euclidean distance of $Y_i$ from the closest point on the true Pareto front. GD is only concerned with convergence and is indifferent to other quality criteria. The difference with hypervolume is that GD is to be minimized instead of maximized.

In Figure 27 we see the results of the same experiment as for Figure 26, that is, the comparison of a small and big value of Cr. Again, the hypothetical isocurve, where the two average performances are the same is marked by a bold line.

When we take into account that GD is to be minimized, the pattern is strikingly similar. Again, for multi-modal problems the small value of $Cr = 0.1$ surprisingly outperforms the rotationally invariant big value $Cr = 1.0$ for almost the entire spectrum of rotations. On the



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

(c) WFG6 (NS-UM)

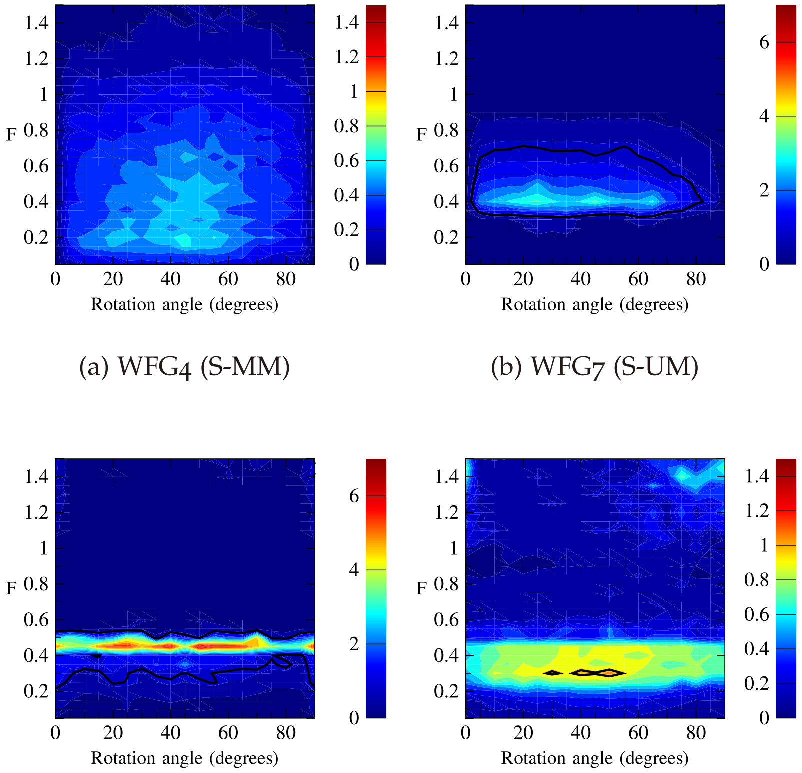
(d) WFG9 (NS-MM)

Figure 26: $\frac{\text{Average RHV with Cr} = 0.1}{\text{Average RHV with Cr} = 1}$

other hand, for unimodal problems, we can see a performance loss. This performance loss is more systematic for the separable WFG7 problem, which is consistent with the single-objective theory.

## 4.5 CONCLUSION

In this chapter we showed how the behavior of the differential evolution algorithm on bi-objective problems changes when the coordinate axes of the decision space are rotated. Our findings show that the change is significant even for small rotations. There is a consistent *drop* in performance on *separable* problems while the qualitative properties of the change for *non-separable* problems are much less predictable. Unexpectedly, for *multi-modal* problems, *low* values of crossover probability perform better through the observed spectrum of rotations. As a future work we propose to see if this holds for problems other than the ones we studied and if this is the case, to find the cause of this behavior.



(a) WFG4 (S-MM)

(b) WFG7 (S-UM)

(c) WFG6 (NS-UM)

(d) WFG9 (NS-MM)

Figure 27: $\frac{\text{Average GD with Cr} = 0.1}{\text{Average GD with Cr} = 1}$

# 5 COMPARISON OF PARAMETER CONTROL MECHANISMS

> *It seems that perfection is attained,*
> *not when there is nothing more to add,*
> *but when there is nothing more to take away.*
>
> — Antoine de Saint Exupéry

## 5.1 INTRODUCTION

Differential evolution (DE) [45] is a simple to understand, but nevertheless powerful optimizer. However, as we showed in the previous chapter, its performance of is highly sensitive to the choice of parameters. Moreover, this dependency changes from problem to problem. Selection of well performing fixed parameters for a particular optimization problem is a relatively little understood subject, especially in the multi-objective realm. In order to solve this problem and to bring the differential evolution algorithms closer to perfection by simplifying the parameter selection process, many researchers proposed methods to set the parameters automatically.

According to the taxonomy proposed in [23], parameter setting techniques are divided into *parameter tuning*, which happens before the run, and *parameter control*, which happens during the run. Parameter control techniques are further divided into *deterministic, adaptive,* and *self-adaptive*. *Deterministic* techniques apply the parameters according to a given rule, while ignoring any feedback from the search process. *Adaptive* techniques continually update their parameters using feedback from the population. *Self-adaptive* techniques, which originate in evolution strategies, attach different parameters to each individual. These parameters undergo mutation and recombination along with the individuals. Better parameter values lead to individuals with a higher chance to survive and therefore have a higher chance to propagate to the next generation.

Each mentioned paradigm of parameter control is represented by numerous algorithms in the literature. One of the first attempts to control parameters in DE is the (multi-objective) SPDE algorithm [1] belonging to the self-adaptive category. An adaptive mechanism based on population diversity for both single- and multi-objective DE has been proposed by Zaharie in [60]. The use of fuzzy controllers to adapt the parameters has been proposed by Liu et al. [41] The SaDE algorithm [46], originally proposed for single-objective DE, adapts the used DE strategies as well as the parameters. SaDE, which is an adaptive algorithm according to our classification, has been generalized to multi-objective realm and subsequently improved to OW-MOSaDE [33]. A comparison of *single-objective* adaptive and self-adaptive methods is presented in [7] and in [6].

A typical modern multi-objective algorithm is in fact an orchestra of sub-algorithms, each playing its own instrument. There is a sub-algorithm to initialize the population, a sub-algorithm to select individuals for reproduction, a sub-algorithm to maintain diversity and so on. Various techniques for parameter control are usually published as a part of a unified production-ready algorithm. Apart from the parameter control mechanism, this algorithm usually has its own sub-algorithms to perform tasks not related to parameter control. These sub-algorithms usually vary from algorithm to algorithm and make the comparison of algorithms difficult, since it is not clear if the difference in performance should be attributed to the parameter control mechanism itself or to the difference in sub-algorithms. For example, to estimate diversity of an individual, the OW-MOSaDE algorithm [33] uses the harmonic average distance measure, while the JADE2 algorithm [63] uses the product of distances. In order to isolate these effects, we implement all the parameter control methods within a simple multi-objective DE algorithm DEMO [47] with the product of $M$ nearest neighbors pruning procedure [37].

In this work we want to find out if some parameter control paradigm is inherently better in terms of performance and whether the parameter control mechanisms can find favorable parameters in problems which can be successfully optimized only with a limited set of parameters. We are also interested in finding an explanation of the observed performance. We do this by observing the evolution of parameters used by the parameter control methods throughout the optimization process. For this paper, we tried to choose representative examples from each group. We compare one deterministic, three adaptive and four self-adaptive methods. Some of the methods we present here are originally used only for single-objective optimization, but they can be easily adopted to multi-objective optimization.

We conclude, based on our limited results, that *self-adaptive* methods are the most robust methods, while performing on par with the best fixed parameter settings. We found out that adaptive methods may have significant problems to find favorable values of parameters. Moreover, they seem to adapt their parameters in patterns *regardless* of the problem.

The population size NP is also considered a parameter of DE, and there have been attempts to adapt the population size as well [57], but in this paper we restrict ourselves to parameters F and Cr. Moreover, strategies to generate $X_{\text{trial}}$, different than the one in (2) and (3) have been proposed, but in this work we shall consider *only* the default strategy. Next, we present all the parameter control mechanisms that we consider in this study. Then we explain the details of our experimental setup. Finally, we discuss the results obtained.

## 5.2 APPROACHES TO PARAMETER CONTROL IN DE

Now we introduce the mechanisms that we use in this chapter, using the classification introduced in the previous section.

### 5.2.1 *Deterministic Mechanism for Parameter Control*

The parameters in the MDDE algorithm [64] are initialized as relatively big values to prevent premature convergence, and then monotonically decreased in a geometric sequence according to:

$$F_g := F_0 \exp(-a_0 \frac{g}{g_{max}})$$
$$Cr_g := Cr_0 \exp(-a_1 \frac{g}{g_{max}}),$$

where $g$ is the current generation and $g_{max}$ is the maximum number of generations.

### 5.2.2 *Adaptive Mechanisms for Parameter Control*

#### 5.2.2.1 *JADE2*

The adaptive mechanism in the JADE2 algorithm generates new values of F and Cr each time a new $X_{trial}$ is to be generated. If a particular $X_{trial}$ Pareto dominates the $X_{target}$, the combination of F and Cr which generated the $X_{trial}$ is recorded as a *successful* one. The values of F are generated from a Cauchy distribution with median $\mu_F$ and scale $\gamma = 0.1$, while the values of Cr from a Normal distribution with mean $\mu_{Cr}$ and $\sigma = 0.1$. At the end of each generation the parameters of these distributions are updated by:

$$\mu_F := (1-c)\mu_F + c.\text{avg}_L(F_s)$$
$$\mu_{Cr} := (1-c)\mu_{Cr} + c.\text{avg}_A(Cr_s),$$

where $c \in [0;1]$ is a learning factor, $\text{avg}_L(F_s)$ is the Lehmer mean of all successful F's and $\text{avg}_A$ is the arithmetic mean of successful Cr's in the previous generation. In our experiments we held $c = 0.1$ in accordance with the advice of authors.

#### 5.2.2.2 *OW-MOSaDE*

Objective-wise MOSaDE [33] attempts to learn which value of Cr is good for a particular objective. For each objective $f_i \in (f_1, \ldots, f_m)$ OW-MOSaDE holds one value of $\mu_{i,Cr}$. These values are updated at the end of each generation if the $X_{trial}$ generated by a particular Cr improves objective $f_i$. In addition, a master $\mu_{Cr}$ is updated if *all* objectives are improved simultaneously. At each generation, one of these $m+1$ values is randomly chosen to serve as the mean of a normal random distribution with $\sigma = 0.1$, which is sampled to generate the values of Cr. That is, each generation the algorithm concentrates on either one randomly chosen objective or attempts to improve all objectives at once. As opposed to JADE2, there is no learning factor, but the successful values of Cr are retained for $lp$ generations, where $lp = 50$ is a learning period. The value F is not adapted, but generated randomly from a fixed set of normal distributions for each individual.

#### 5.2.2.3 *Control of Diversity Adaptation Algorithm (PDCaDE)*

Zaharie discovered a simple algebraic relationship between the expected variance of the DE population before and after the generation of new individuals [61]. Based on her theoretical results, Zaharie developed an algorithm which monitors the variance of the population in the decision space and alters the parameters according to this relationship, so that the variance of the population decreases in a specified, evenly manner throughout the entire run. The motivation behind this algorithm is to prevent premature convergence and to use the allocated budget of generations evenly.

The algorithm does not have a specific name, so we call it *Population Diversity Control Adaptive DE (PDCaDE)* in the rest of this article. PDCaDE introduces a new parameter $\gamma$, which we held constant at $\gamma = 1.25$ for all our experiments. This value was determined by some limited tuning, since the author does not provide a fixed value which should be used for multi-objective problems.

### 5.2.3 *Self-adaptive Mechanisms for Parameter Control*

The main idea behind self adaptive mechanisms is that each individual remembers the set of parameters *by which it was created*. This way if an individual is created by a good set of parameters and survives into the next generation, the parameters it carries survive too. On the other hand bad parameter combinations get pruned away.

In all self-adaptive DE mechanisms considered in this paper, the principle is the same. New individuals $X_{trial}$ are generated using (2) and (3) with the exception that the F and Cr are not fixed, but replaced by $F_{trial}$ and $Cr_{trial}$, which are created on the spot and then attached to the newly generated $X_{trial}$. If we denote by $F_i$ and $Cr_i$ the parameter values attached to individual $X_i$, then the methods to create $F_{trial}$ and $Cr_{trial}$ can be summarized using simple equations which we summarize in Table 3. These approaches introduce new parameters, but some authors claim that these parameters should be fixed at values in Table 4. We too use the parameters in Table 4 in our experiments.

Table 3: Summary of used *self-adaptive* mechanisms

| Name | Main formula |
|---|---|
| SPDE [1] | $F_{\text{trial}} := \text{rand}_N(0,1)$<br>$Cr_{\text{trial}} := Cr_{r_1} + \text{rand}_N(0,1)(Cr_{r_2} - Cr_{r_3})$ |
| jDE [6] | $F_{\text{trial}} := \begin{cases} \text{rand}_U(0.1, 1.0) & \text{if } \text{rand}_U(0,1) < \tau_1 \\ F_{\text{target}} & \text{else} \end{cases}$<br>$Cr_{\text{trial}} := \begin{cases} \text{rand}_U(0.1, 1.0) & \text{if } \text{rand}_U(0,1) < \tau_2 \\ Cr_{\text{target}} & \text{else} \end{cases}$ |
| DEMOwSA [62] | $F_{\text{trial}} = \frac{F_i + F_{r_1} + F_{r_2} + F_{r_3}}{4} e^{\tau \text{rand}_N(0,1)}$<br>$Cr_{\text{trial}} = \frac{Cr_i + Cr_{r_1} + Cr_{r_2} + Cr_{r_3}}{4} e^{\tau \text{rand}_N(0,1)}$ |
| SAMDE [44] | $F_{\text{trial}} = F_{r_1} + F'(F_{r_2} - F_{r_3})$<br>$Cr_{\text{trial}} = Cr_{r_1} + F'(Cr_{r_2} - Cr_{r_3})$ |
| $\text{rand}_N(\mu, \sigma)$ | - generator of normal random numbers |
| $\text{rand}_U(a, b)$ | - generator of uniform random numbers |

## 5.3 experimental design

### 5.3.1 *Algorithmic Framework*

Algorithm 4 shows the unified framework which we use for comparing the selected parameter control mechanisms. The lines that apply *only* to self-adaptive mechanisms are highlighted in *yellow*, while the ones that apply *only* to adaptive mechanisms are highlighted in *blue*. All our methods with the exception of PDCaDE follow this model. The implementation of PDCaDE has its specifics, which are limited to line 8, since each *variable* has its own set of parameters.

Some methods have their own parameters, which we held constant at the values recommended by their authors. Some methods also use several strategies to generate new individuals, but in this work we limited ourselves to the default strategy described in Equations (2) and (3).

### 5.3.2 *Problems*

#### 5.3.2.1 *WFG problems*

To test the mechanisms in various conditions, we chose a subset of the WFG [34] test suite with the same concave Pareto front and all

Table 4: Additional parameters of *self-adaptive* mechanisms

| Name | additional parameters |
|---|---|
| SPDE [1] | $Cr_{init} := rand_N(0.5, 0.15)$ |
| jDE [6] | $\tau_1 = 0.1, \tau_2 = 0.1$ |
| DEMOwSA [62] | $\tau = \frac{1}{\sqrt{2n}}$ |
| SAMDE [44] | $F' := rand_U(0.7, 1.0)$ |
| $rand_N(\mu, \sigma)$ | - generator of normal random numbers |
| $rand_U(a, b)$ | - generator of uniform random numbers |

**Algorithm 4:** Adaptive and self-adaptive DEMO [47] algorithm

```
initialize $P = \{X_1, ..., X_{NP}\}$ uniformly randomly in the decision space
initialize F and Cr generators
initialize values of $F_i$ and $Cr_i$ for $i = 1, \ldots, NP$
for generation := 1 to $G_{max}$ do  Evolutionary loop
    for target := 1 to NP do  Generational loop
        generate $F_{trial}$ and $Cr_{trial}$
        compute $F_{trial}$ and $Cr_{trial}$ using Table 3
        generate $X_{trial}$ using $F_{trial}$ and $Cr_{trial}$ from (2) and (3)
        attach $F_{trial}$ and $Cr_{trial}$ to $X_{trial}$
        project $X_{trial}$ to decision space
        if $X_{target}$ dominates $X_{trial}$ then
            discard $X_{trial}$
        else if $X_{trial}$ dominates $X_{target}$ then
            replace $X_{target}$ with $X_{trial}$
        else if $X_{target}$ and $X_{trial}$ are mutually non-dominated then
            add $X_{trial}$ to the end of the population
        end
        update success memories
    end
    update parameter generators
    Trim P to size NP using non-dominated sorting [47] and MNN diversity [37]
end
```

possible combinations of separability and modality characteristics as in the previous chapter. These problems are summarized in Table 2. We held the number of variables fixed at 10 and performed tests for 2 and 3 objectives and 10 variables.

#### 5.3.2.2 *Quadratic problems*

As we shall later see, even the non-separable multi-modal WFG problems can be successfully optimized using many combinations of fixed parameters. To test the ability of parameter control mechanisms to solve challenging problems we developed a scalable problem, that can be solved by relatively few combinations of F and Cr, called Q. The problem Q consists of $m$ functions: $Q = (q_1, \dots, q_m)$. Each function is a quadratic form $q_m(X) = (X - c_m)D_m(X - c_m)^T$ where

$$\begin{aligned} D_1 &= \mathrm{diag}(1, 2, 4, \dots, 2^{n-1}), \\ D_2 &= \mathrm{diag}(2^{n-1}, 1, 2, \dots, 2^{n-2}), \\ &\dots \end{aligned}$$

and the vectors $c_i$ are generated uniformly randomly in a unit sphere. The resulting problem is then rotated in the decision space around all $n-2$ rotation subspaces by 45 degrees. [1] Moreover, the population for this problem is generated randomly uniformly in a sphere of radius $2^{10}$ which is shifted from the origin in a random direction by $2^{14}$.

In this work, we explore the Q problem for 2, 3, and 4 objectives, while the number of variables remains fixed at 10.

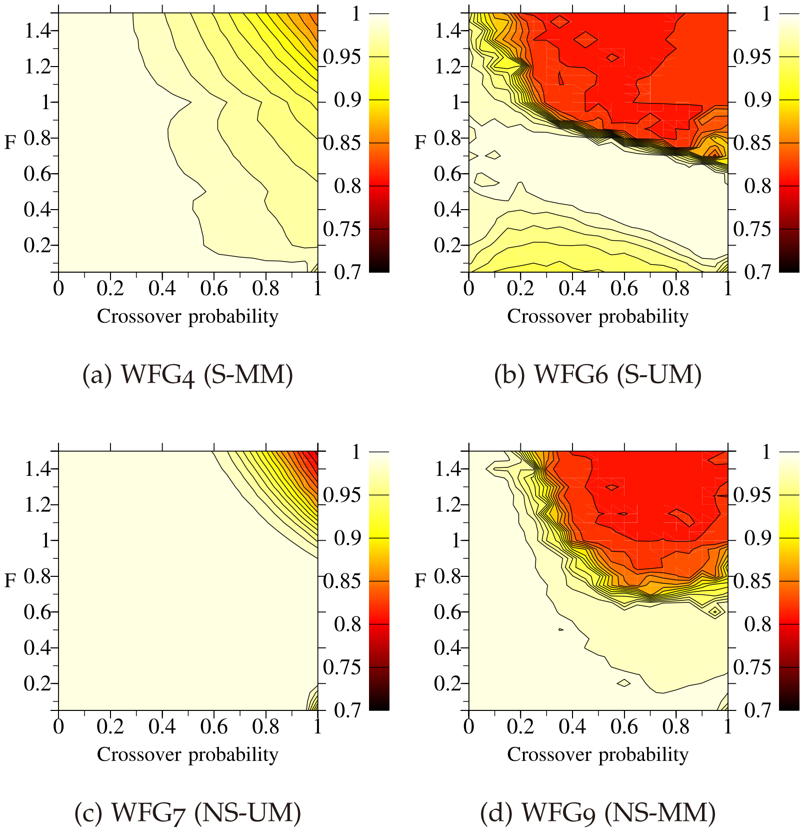
### 5.3.3 *Observed Statistics*

In this work we are interested in the *performance* of the various methods as well as in observing their *behavior*. To measure the performance, we use the hypervolume [65] metric, since it measures both convergence and diversity of the resulting Pareto front approximation. As a reference point for both types of problems we first construct the hyperbox which contains the entire true Pareto front and add a unit vector to its upper corner.

In order to simplify the interpretation of the hypervolume, we normalize it by dividing it by the maximal attainable hypervolume in the case of WFG problems, and by the volume of the hyperbox between the origin and the reference point for the Q problem. This way we know that the maximal attainable normalized hypervolume, corresponding to complete convergence is 1.

In order to observe the *behavior* of the mechanisms, we log each combination of F and Cr that the algorithm uses in one generation.

[1] Details on this methodology can be found in [21].



(a) WFG4 (S-MM)

(b) WFG6 (S-UM)

(c) WFG7 (NS-UM)

(d) WFG9 (NS-MM)

Figure 28: Average normalized hypervolume for 2 objectives

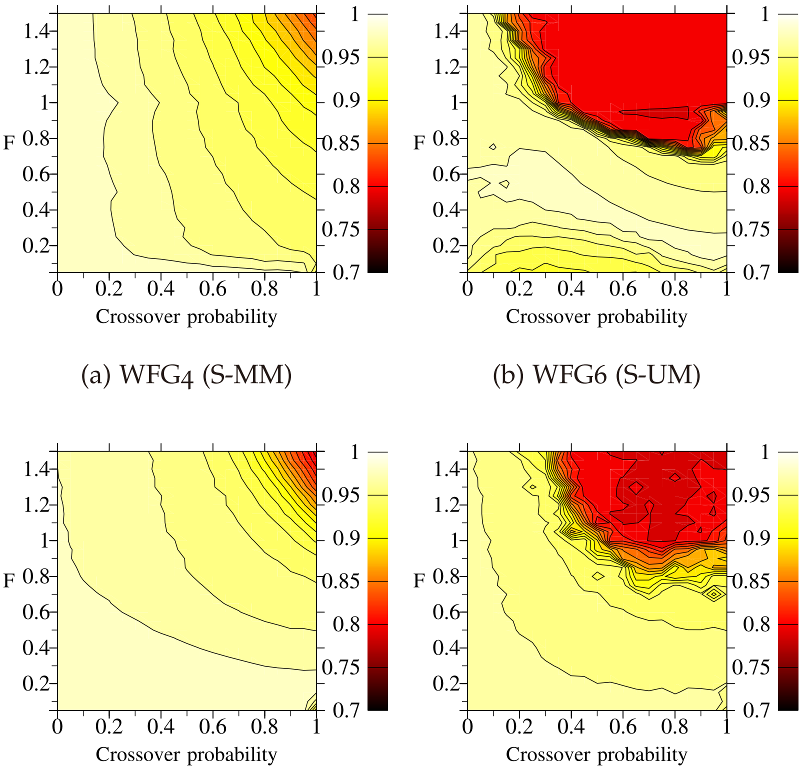
## 5.4 RESULTS AND DISCUSSION

### 5.4.1 *Parameter Tuning*

For each problem we performed a preliminary tuning of the F and Cr parameters by grid search. We explored the ranges $F \in [0.05; 1.5]$ and $Cr \in [0; 1]$ with a resolution of 0.05. For each combination of parameters, we ran 10 independent runs of the Algorithm 4 with fixed parameters. The average normalized hypervolume from this tuning is presented in the form of heat-maps, with hot colors meaning good performance. The tuning results for the WFG problems are in Figure 28 and 29. In each figure we see a red, L-shaped region of favorable values. We provided theoretical explanation of this shape in section 4.4.1.

### 5.4.2 *Parameter Control on WFG problems*

For each of the studied methods we ran 50 independent runs with a fixed population size $NP = 500$ individuals. Each run was limited by 500 generations. The average normal hypervolume along with the



(a) WFG4 (S-MM)

(b) WFG6 (S-UM)

(c) WFG7 (NS-UM)

(d) WFG9 (NS-MM)

Figure 29: Average normalized hypervolume for 3 objectives

standard deviation across the 50 runs is presented in Tables 5 and 6. The value of normalized hypervolume at the start of the run is denoted as *start*. For each problem, based on the initial tuning, we constructed an *ideal* set of fixed parameters and ran the algorithm for 50 independent runs with these settings. Within the group of adaptive methods and the group of self-adaptive methods we marked the highest value in bold.

We can see that both adaptive and self-adaptive methods are performing quite good in comparison with the ideal parameter set. The only exception is the deterministic MDDE algorithm, which shows significant problems for the non-separable WFG6 and WFG9 problems.

For each method we plot the path of the average used F and Cr with respect to the generation. We call this plot the *trajectory* of that method. The trajectories of adaptive methods along with the deterministic MDDE method are plotted in Figure 30 and the trajectories for the self-adaptive methods are in Figure 31. The small crosses are plotted for each 10 generations and the *final* reached value is marked by a large *square*. The optimal value of F and Cr is marked by a *black*

Table 5: Average normalized hypervolume for 2-objective WFG problems

| | | 2 objectives | | | |
|---|---|---|---|---|---|
| | | WFG4 | WFG6 | WFG7 | WFG9 |
| | start | 0.774<br>(1.2e-02) | 0.618<br>(1.5e-02) | 0.706<br>(8.6e-03) | 0.666<br>(2.2e-02) |
| | ideal | 0.999<br>(6.9e-05) | 0.999<br>(5.9e-04) | 0.999<br>(3.5e-04) | 0.996<br>(1.4e-03) |
| | MDDE | 0.998<br>(3.4e-04) | 0.820<br>(8.6e-04) | 0.999<br>(7.5e-06) | 0.905<br>(8.3e-02) |
| adaptive | JADE2 | **0.999**<br>(7.2e-06) | **0.992**<br>(4.4e-03) | 0.999<br>(1.0e-05) | **0.996**<br>(8.7e-04) |
| | OW-MOSaDE | 0.997<br>(6.6e-04) | 0.975<br>(6.4e-03) | **0.999**<br>(9.1e-06) | 0.993<br>(2.0e-03) |
| | PDCaDE | 0.998<br>(1.7e-03) | 0.980<br>(7.1e-03) | 0.999<br>(4.9e-05) | 0.993<br>(2.2e-03) |
| self-adaptive | DEMOwSA | 0.998<br>(2.9e-04) | **0.991**<br>(3.8e-03) | 0.999<br>(2.1e-05) | 0.989<br>(3.5e-03) |
| | jDE | **0.999**<br>(7.5e-06) | 0.985<br>(1.7e-02) | **0.999**<br>(1.4e-05) | **0.996**<br>(1.0e-03) |
| | SAMDE | 0.999<br>(8.0e-06) | 0.980<br>(1.4e-02) | 0.999<br>(1.5e-05) | 0.995<br>(9.5e-04) |
| | SPDE | 0.999<br>(1.1e-05) | 0.970<br>(1.1e-02) | 0.999<br>(8.8e-06) | 0.996<br>(4.7e-04) |

*circle* in each graph. Moreover, all graphs contain the contour lines of the average normalized hypervolume obtained by parameter tuning.

We can see that the adaptive methods behave consistently and their trajectories look alike across the various problems. On the other hand their self-adaptive counterparts seem a lot less consistent.

The situation is very similar for 3 objectives. The trajectories for the adaptive and deterministic methods in Figure 32 seem to behave indifferently with respect to the problem and to the number of objectives. On the other hand, self-adaptive methods in Figure 33 again perform much less predictably. Looking at the results of parameter tuning in Figures 28 and 29 we see a possible explanation. The heatmaps of normalized hypervolume for problems WFG4 and WFG6 have more structure than those of WFG7 and WFG9. The contour lines are more evenly distributed, which may help the algorithms

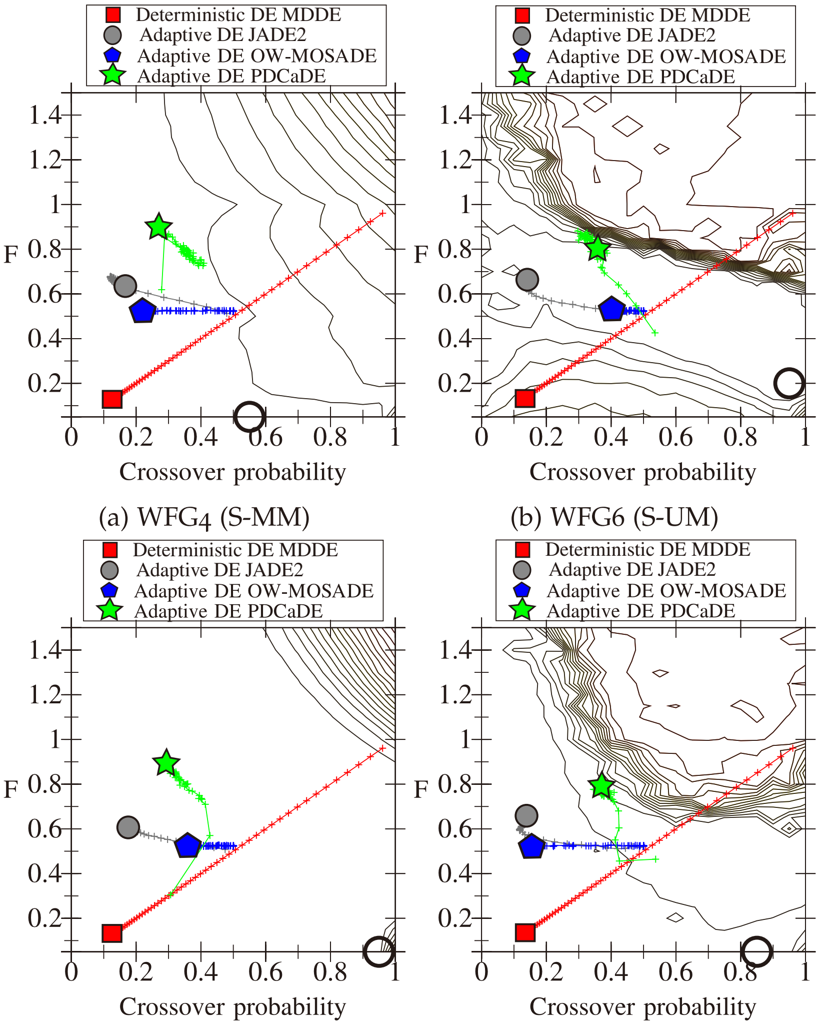
Table 6: Average normalized hypervolume for 3-objective WFG problems

| | | 3 objectives | | | |
|---|---|---|---|---|---|
| | | WFG4 | WFG6 | WFG7 | WFG9 |
| | start | 0.669<br>(1.9e-02) | 0.563<br>(8.9e-03) | 0.646<br>(1.0e-02) | 0.575<br>(2.0e-02) |
| | ideal | 0.987<br>(2.2e-04) | 0.983<br>(1.6e-03) | 0.988<br>(1.1e-04) | 0.976<br>(2.9e-03) |
| | MDDE | 0.978<br>(6.7e-04) | 0.807<br>(3.6e-02) | 0.986<br>(1.4e-04) | 0.892<br>(8.4e-02) |
| adaptive | JADE2 | **0.982**<br>(4.3e-04) | **0.977**<br>(3.5e-03) | 0.978<br>(4.5e-04) | **0.965**<br>(1.9e-03) |
| | OW-MOSaDE | 0.968<br>(1.3e-03) | 0.971<br>(8.3e-03) | 0.977<br>(5.2e-04) | 0.961<br>(1.6e-03) |
| | PDCaDE | 0.974<br>(2.6e-03) | 0.966<br>(8.4e-03) | **0.979**<br>(9.6e-04) | 0.962<br>(2.2e-03) |
| self-adaptive | DEMOwSA | 0.972<br>(1.0e-03) | **0.979**<br>(1.6e-03) | 0.975<br>(9.3e-04) | 0.959<br>(1.8e-03) |
| | jDE | 0.982<br>(7.0e-04) | 0.967<br>(1.3e-02) | **0.981**<br>(5.3e-04) | **0.968**<br>(2.7e-03) |
| | SAMDE | 0.983<br>(5.8e-04) | 0.968<br>(8.8e-03) | 0.977<br>(5.4e-04) | 0.963<br>(1.5e-03) |
| | SPDE | **0.985**<br>(6.6e-04) | 0.964<br>(1.1e-02) | 0.980<br>(3.8e-04) | 0.965<br>(1.6e-03) |

find favorable parameter values. On the other hand, the heat-maps for WFG7 and WFG9 have large plateaus associated with favorable parameters, separated by steep cliffs from plateaus with unfavorable parameters. Consequently we see that on WFG4 and WFG6 problems, the trajectories of self-adaptive methods aim correctly for the more favorable regions, while on the WFG7 and WFG9 problems, the behavior seems more random.

### 5.4.3 Q *problems*

In the same way as for the WFG problems, we performed a limited parameter tuning on the Q problems. The corresponding heat-maps are in Figure 34.



(a) WFG4 (S-MM)

(b) WFG6 (S-UM)

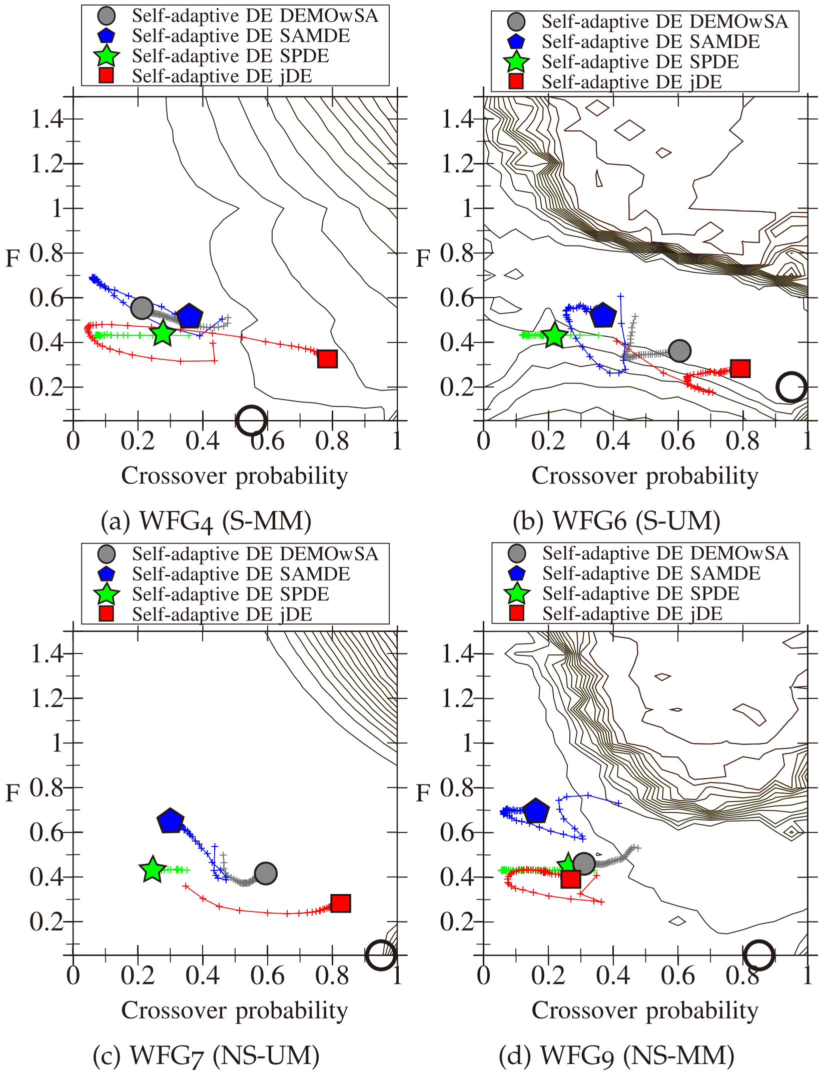
(c) WFG7 (NS-UM)

(d) WFG9 (NS-MM)

Figure 30: Parameter trajectories of adaptive and deterministic methods for 2 objectives

The contrast with the data for WFG is immediately visible. Especially for 2 objectives, the area of favorable parameter combinations is relatively small. Moreover the favorable area is surrounded by steep cliffs. Even a small change in one parameter may mean the difference between a successful convergence and total failure. On such hard problems, the difference in performance of parameter control methods becomes apparent. The averages and standard deviations of 50 independent runs for 500 generations with a fixed population size of 500 individuals are in Table 7.

On the 2-objective Q problem *all* the adaptive methods fail completely. Out of 50 runs, not one of them approached close enough to the Pareto front. Some minor success has been achieved by the deterministic MDDE method, but the best performers are the *self-adaptive* methods. On the 3-objective problem, OW-MOSaDE catches up, while



(a) WFG4 (S-MM)

(b) WFG6 (S-UM)

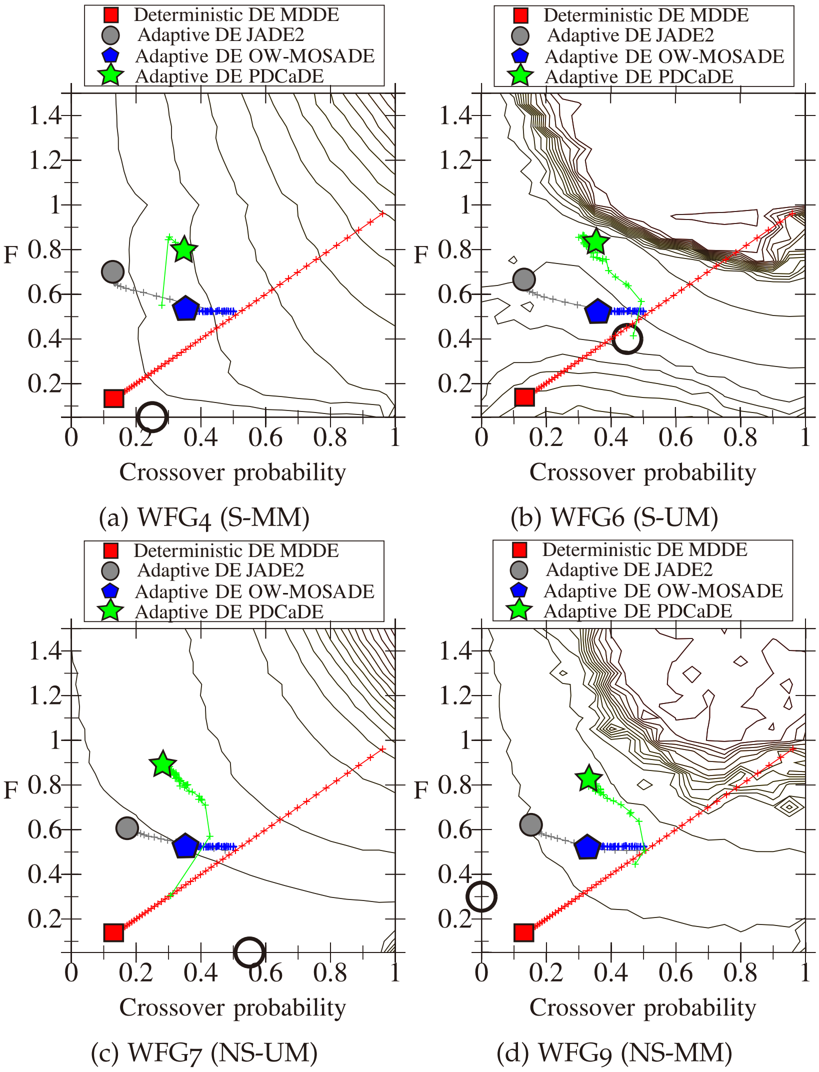
(c) WFG7 (NS-UM)

(d) WFG9 (NS-MM)

Figure 31: Parameter trajectories of self-adaptive methods for 2 objectives

the other adaptive methods are lagging. For the 4-objective problem, the performances even out. We see a steady decrease in hypervolume as we increase the number of objectives. But this decrease applies also to the algorithm with ideal parameters and may be attributed either to the fact that the number of individuals needed to represent the Pareto front rises exponential with the dimension of the problem [3], or that the problems with more objectives are simply more difficult.

Let us examine this behavior in more detail. The trajectories for adaptive methods are in Figure 35, and for self-adaptive methods in Figure 36. The adaptive trajectories for the 2, 3 and 4 objective Q problems are very similar. Also they resemble those of the WFG problems. The PDCaDE algorithm seems to always converge to $\mathrm{Cr} = 0.4$ and $\mathrm{F} = 0.8$. The OW-MOSaDE cannot adapt the distribution of the F parameter and invariably pushes the value of Cr down. This makes sense for the WFG problems, but it is counterproductive for the 2-



(a) WFG4 (S-MM)

(b) WFG6 (S-UM)

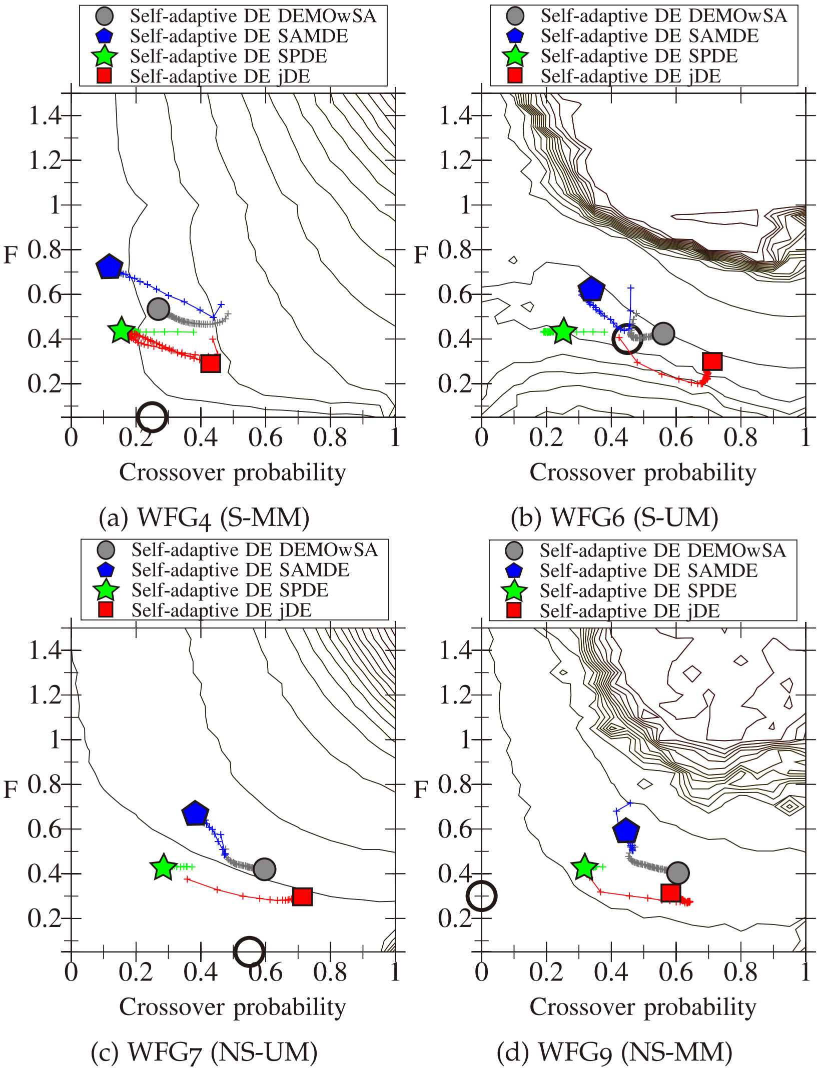
(c) WFG7 (NS-UM)

(d) WFG9 (NS-MM)

Figure 32: Parameter trajectories of adaptive methods for 3 objectives

objective Q problem. The JADE2 mechanism seems to be lured towards small values of Cr even more. This suggests that small values of Cr tend to produce individuals which have a relatively high probability to dominate other individuals.

The behavior of self-adaptive mechanisms is completely different. On the 2-objective problem all self-adaptive mechanisms achieve at least half of the possible hypervolume. This is even true for the SPDE mechanism which does *not* find the area of favorable parameter combinations. It seems that the fact that the parameters of SPDE are generated randomly helps it generate favorable parameter combinations often enough to converge partially. The adaptive algorithms also generate their parameters randomly, but the centers of the random distributions from which these parameters are generated are shifting in the wrong direction.



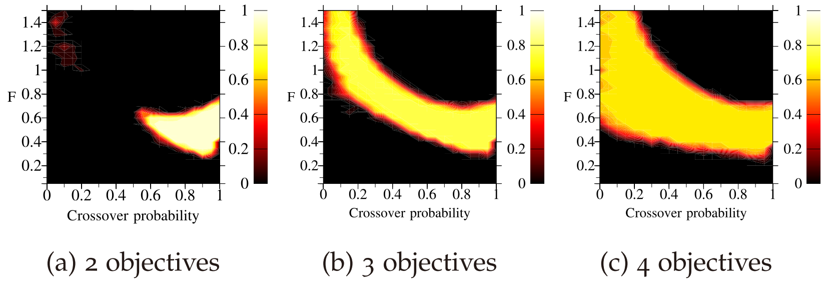
(a) WFG4 (S-MM)

(b) WFG6 (S-UM)

(c) WFG7 (NS-UM)

(d) WFG9 (NS-MM)

Figure 33: Parameter trajectories of self-adaptive methods for 3 objectives



(a) 2 objectives

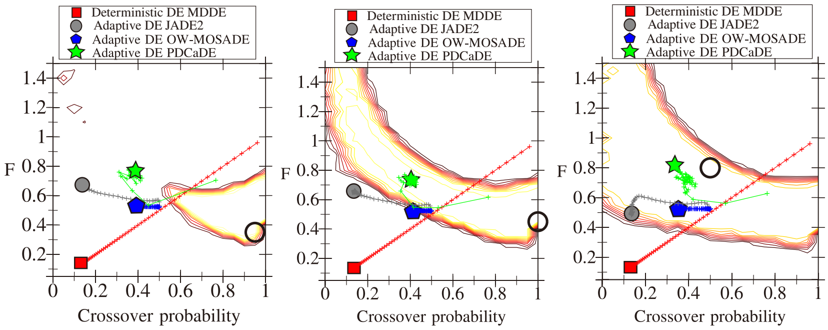
(b) 3 objectives

(c) 4 objectives

Figure 34: Average normalized hypervolume for the Q problem

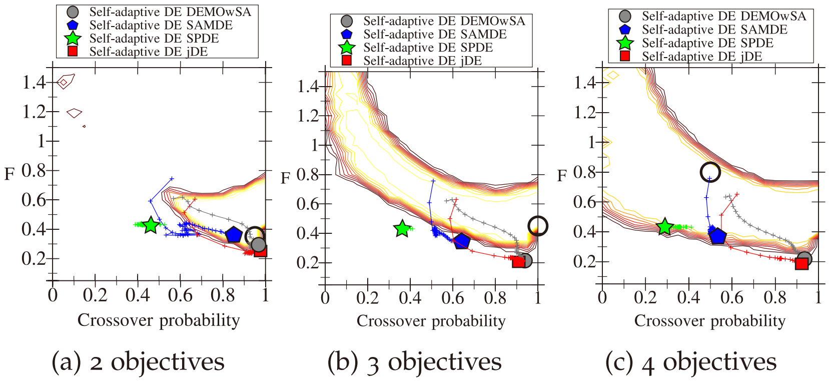
Table 7: Average normalized hypervolume for the Q problem

| | | 2 objectives | 3 objectives | 4 objectives |
|---|---|---|---|---|
| | start | 0.000<br>(0.0e+00) | 0.000<br>(0.0e+00) | 0.000<br>(0.0e+00) |
| | ideal | 0.999<br>(2.1e-05) | 0.783<br>(6.7e-04) | 0.673<br>(2.4e-03) |
| | MDDE | 0.128<br>(2.8e-01) | 0.732<br>(1.2e-01) | 0.653<br>(5.7e-03) |
| adaptive | JADE2 | 0.000<br>(0.0e+00) | 0.175<br>(2.8e-01) | 0.648<br>(5.8e-03) |
| | OW-MOSaDE | 0.000<br>(0.0e+00) | **0.668**<br>(1.1e-01) | 0.654<br>(4.3e-03) |
| | PDCaDE | 0.000<br>(0.0e+00) | 0.000<br>(0.0e+00) | **0.659**<br>(8.7e-03) |
| self-adaptive | DEMOwSA | **0.999**<br>(2.1e-05) | **0.783**<br>(6.8e-04) | **0.652**<br>(5.8e-03) |
| | jDE | 0.745<br>(4.0e-01) | 0.433<br>(3.7e-01) | 0.651<br>(6.1e-03) |
| | SAMDE | 0.994<br>(1.5e-02) | 0.778<br>(2.0e-03) | 0.638<br>(7.6e-03) |
| | SPDE | 0.548<br>(4.6e-01) | 0.640<br>(2.4e-01) | 0.643<br>(6.7e-03) |



(a) 2 objectives (b) 3 objectives (c) 4 objectives

Figure 35: Parameter trajectories of deterministic and adaptive methods for the Q problem.



(a) 2 objectives (b) 3 objectives (c) 4 objectives

Figure 36: Parameter trajectories of self-adaptive methods for the Q problem.

The relatively good performance of the deterministic MDDE algorithm can be attributed to luck, since the trajectory of this algorithm briefly crosses the region containing favorable parameter values.

## 5.5 CONCLUSION

In this chapter we compared various deterministic, adaptive, and self-adaptive mechanisms of parameter control in multi-objective differential evolution. We did this by isolating the mechanisms and applying them to a single multi-objective algorithm. We then tested this algorithm on a set of known benchmark problems as well as one new problem. We measured the performance of these methods in terms of hypervolume, as well as their behavior in measuring *which parameters* they found.

We found out that on the usual benchmark problems even the simple mechanisms can lead to results comparable with parameter tuning. On the new problem, which we proposed exactly because it can be optimized only by a small set of parameters, the *self-adaptive* methods were the only ones that managed to find a satisfactory Pareto front for all objective dimensionalities. The deterministic method also achieved some limited success, but it is hard to determine if we can attribute this to luck or to the underlying quality of the method. After examining the progress of the parameters used by the adaptive methods we found out that each algorithm evolves its parameters in a more or less problem independent way. This effect should be examined in more detail as a part of future work.

Part III

# IMPROVING THE PERFORMANCE OF MULTI-OBJECTIVE DE

Here we introduce some innovations which reduce the computational cost of mutli-objective evolutionary algorithms in general, but which are especially well suited for differential evolution algorithms.

# 6 REDUCING THE COMPUTATIONAL COST

## 6.1 INTRODUCTION

As we saw in section 3.1, belonging to a specific non-dominated front gives us a measure of relative quality of an individual with respect to the rest of the population. Many MOEAs such as NSGA-II [12], GDE3 [38] or DEMO [47] require that non-dominated sorting is performed at each generation to determine the individuals that *survive into the next generation*. This procedure often becomes time-consuming compared to the rest of the algorithm. This is especially the case with large populations and/or high number of objectives. One can parallelize the objective function evaluations for the population into available processors, however the sorting has to be performed in serial and costs more computational time as the number of objectives grows due to the nature of Pareto dominance. As an example we profiled a run of a MOEA algorithm called GDE3 [38] implemented on a single processor on a WFG9 [34] problem with 4 objectives and an initial population size of 1000 individuals for 500 generations. We found out that approximately 82% of computer time and 79% of computer instructions[1] were used on non-dominated sorting.

In computationally expensive problems this ratio will change. Nevertheless, there are many problems that are impossible to solve with exact methods, whose objective functions are extremely cheap to evaluate. For example, the objective value of a solution in the traveling salesman problem with tens of thousands of nodes can be evaluated on the order of milliseconds. The same goes for the vehicle routing problem [24]. There are many instances of real world problems, where reducing the cost of non-dominated sorting brings very significant improvement in the entire calculation.

The first to recognize and address this problem were Deb et al. in [12]. Their method called the *fast non-dominated sorting* compares each individual with each other and caches the result of these comparisons in order to avoid comparing the same two individuals twice.

For further improvement, several researchers have proposed different approaches, which are mainly categorized into the following three.

*Divide and conquer:* These methods are based on an article by Kung et al. [40]. They divide the problem with respect to both *dimension* (number of objectives) and *population size*. The first attempt by Jensen

[1] We used a computer profiler Callgrind, which runs the program on a virtual machine and counts the instructions executed.

[36] in 2003 achieved significant speedup and a reduction in computational complexity, but it failed to deal with the case where two individuals have a same value for a certain objective. Treating this case turned out to be more difficult than it seemed. Luckily, this problem was recently solved by Fortin et al. [29] while preserving both speed and computational complexity. These algorithms achieve astonishing improvement in both theory (computational complexity) as well as in practice (computational wall clock time). Unfortunately the performance declines to the level of fast non-dominated sorting for a high number of objectives. Moreover these algorithms are rather complicated and *static* in the sense that when one individual is changed there is no easy way to determine the non-dominated fronts of the modified population other than to run the algorithm again.

*Reducing the number of dominance comparisons:* These methods try to *infer* domination relationships using the transitivity property of Pareto dominance. Notable recent algorithms are the *climbing sort* and the *deductive sort* by McClymont and Keedwell [42]. These algorithms achieve very significant speedup, but they are specifically designed for populations where the domination relationship between individuals is *relatively common*. Unfortunately this assumption does not hold for problems with a large number of objectives. The fewer domination relationships there are, the fewer such relationships can be inferred and the performance suffers. Even so, these methods constitute a significant innovation since they are *dynamic* in the sense that when one individual changes, the information about the non-dominated fronts can be efficiently updated.

*Archiving the non-dominated individuals:* The problem these methods try to solve is different from the original non-dominated sorting. Schütze [52] calls this problem the *dynamic non-dominance problem*. Instead of starting from scratch and computing the non-dominated fronts for a certain population, these methods concentrate on *keeping* and *updating* a single non-dominated front. These methods are of course *dynamic* as in the previous paragraph. Notable research has been done by Fieldsend et al. [26] and by Schütze [52]. Both studies propose original data structures to hold and maintain the set of non-dominated individuals. Although the speedup achieved by these methods over the brute force method is significant, it is not competitive with the divide and conquer methods.

In this chapter we propose a new method to reduce the cost of non-dominated sorting. This method is closely linked with the dynamic non-dominance problem.

The main idea is to compute which individuals are non-dominated *at the beginning* of the MOEA and then *update* this knowledge *each time an individual changes*. This way the non-dominated individuals are known at *all times*. Thus we do not need to call non-dominated sorting to compute the first front. In the case there is more than one front to compute then we apply non-dominated sorting just on the subset of dominated individuals. When the number of objectives grows, there are fewer and fewer dominated individuals which reduces the need to call non-dominated sorting. Therefore our method thrives on problems in which the number of non-dominated solutions is large.

We keep track of the non-dominated individuals by storing them in a special data structure which we call an *archive*. We update this archive whenever an individual in the population changes. The computational cost of updating the archive is critical, therefore a major part of our work is dedicated to an efficient implementation of a fast archive which we call an M-front.

The M-front keeps track of all the non-dominated individuals in the population. This means that when a new individual is generated, the M-front needs to determine *if* this individual is dominated by any individual from the M-front and if it is not, to determine *which* individuals are dominated by the new individual. The M-front uses the geometric and algebraic properties of the Pareto dominance relation to reduce the number of individuals which need to be compared. It converts the *orthogonal range* queries related to Pareto dominance to *interval* queries. In order to answer these interval queries *efficiently*, the M-front keeps all its individuals *sorted* in linked lists. There is one linked list for each objective and this list keeps all the individuals sorted with respect to that objective. In order to convert an orthogonal range query into interval queries, an *auxiliary individual* needs to be chosen from the M-front. The role of this individual is just to perform the conversion. We found out that the *closer* this auxiliary individual is to the new individual, the *smaller* are the resulting interval queries and the *faster* is the computation. In order to find an auxiliary individual which is *as close as possible* to the new individual, the M-front keeps an internal K-d tree data structure to perform approximate *nearest neighbor* search. In case of DE, the entire K-d tree mechanism can be avoided, since the trial individual, which is newly generated is close either to the target individual or to some other individual, depending on the particular DE strategy being used [18]. The M-front can be also used as a stand-alone archive for algorithms which use *unbounded archives* such as [30] or [26].

Experiments confirm that our method can outperform the state of the art Jensen-Fortin's divide and conquer algorithm up to certain population sizes. The performance of our method scales well, especially for a large number of objectives. Since our approach is dynamic in nature, the non-dominated individuals are known at *all times* which is a significant advantage over the state of the art method.

There are algorithms which use more precise methods, such as the *hypervolume* [65], to estimate the quality of individuals in the population. These algorithms can yield better solutions than algorithms which use less precise mechanisms [56]. However, the main problem with such algorithms is that the hypervolume is extremely costly to compute for big populations and large numbers of objectives [59]. In addition, even if the hypervolume could be computed fast, there would still be the need to determine the non-dominated individuals because the hypervolume is computed from them.

This chapter is based on a significant revision and extension of our previous work [18], in which we were restricted to *differential evolution* [45] algorithms. Later we generalized our method to any multi-objective evolutionary algorithm (MOEA) which uses non-dominated sorting [19]. In addition, as we mentioned before, now our approach can be applied to handle archives of MOEAs. We have also improved the implementation of our method. In our original work we use *skip lists* to keep the individuals sorted by each objective. Now we use simple *linked lists* and a hash-table to retrieve the positions of individuals in the linked lists. This results in faster insertions and removals, smaller memory usage and simpler implementation. The K-d tree is a data structure which gets unbalanced after many insertions and deletions. In our previous work we mitigated this problem by rebuilding the K-d tree from scratch after a fixed number of insertions and deletions. Now we propose a new, more frugal mechanism which detects if the K-d tree is unbalanced. We also now include a comparison with the state of the art Jensen-Fortin's algorithm on both practical and conceptual level. Lastly, we added a theoretical section which derives the average case expected computational complexity on a random model.

The organization of this chapter is as follows: first, we introduce a general framework of our approach. This framework introduces the concept of an archive and explains its usage in detail, while the concrete implementation of one such archive is described in the subsequent section. Next, the computational complexity of the proposed approach is theoretically explored. In the subsequent section we provide a conceptual comparison with Jensen-Fortin's algorithm while the experimental comparison is shown in the final section. The experimental section also contains a comparison with fast non-dominated sorting.

## 6.2 PROPOSED METHOD

For the purposes of this chapter we re-define the notion of an individual so that we may formulate our ideas more clearly.

We define an *individual* to be a pair $(\mathtt{id}, X)$ where $\mathtt{id} \in \mathbb{N}$ is a *unique identifier* and $X \in \mathbb{R}^n$ is a *decision vector*. This way we can distinguish between several individuals with the same value in the *decision* space.[2]

To avoid confusion when using subscripts and to simplify notation we use the following convention: when we have an individual $a = (id_a, X_a)$ then instead of writing $f_i(X_a)$ to denote the value of $i$-th objective for individual $a$, we simply write $f_i(a)$. Similarly, instead of $F(X_a)$ we simply write $F(a)$ to denote the objective vector of $a$.

### 6.2.1 *Applicability*

Our method requires some modifications to the usual computational flow of the MOEA. To illustrate this, we restrict ourselves to an evolutionary algorithm following the scheme in Algorithm 5. We demonstrate our method on this generic version of a MOEA. Popular algorithms such as NSGA-II [12] or GDE3 [38] mentioned earlier both follow this scheme.

**Algorithm 5:** MOEA that can use our method

```
   Input: Initial population size $N$
   Output: Approximation of the Pareto front by a population $P$
1  initialize population $P = \{a_1, ..., a_N\}$
2  while arbitrary stopping condition not satisfied do Evolutionary loop
3      while arbitrary stopping condition not satisfied do Generating phase
4          generate new individual $a'$
5          insert $a'$ into $P$
6          remove arbitrary dominated $a \in P$ if needed
7      end
8      non-dominated sorting
9      remove worst non-dominated fronts
10     trim $P$ back to size $N$ using a secondary criterion
11 end
12 report $P$
```

The algorithm has a generating phase on lines 3 to 7 and a survival selection phase on lines 8 to 10. In order for an algorithm to benefit from our method we need to modify *both phases*. We get an *equivalent algorithm* described in Algorithm 6.

The individual-by-individual steady-state generation and insertion of individuals, which is specific to DE, may be confusing, but indeed also algorithms which generate their individuals in large chunks, such as the NSGA-II, can be rearranged to conform to Algorithm 5. The mechanism which generates a new individual on line 4 is completely arbitrary. Therefore for NSGA-II the step on line 4 can be just

[2] In our implementation the id is simply a C++ pointer to the vector X in memory. In order for the id to remain valid, the individuals do not change their addresses in the memory once they are created.

*taking* the new individual from the offspring population. NSGA-II does not remove individuals in the generating phase, but some algorithms such as GDE3 do. Therefore we allow for such removals on line 6.

Note that in the generating phase we can remove only individuals which are dominated. This is a limitation of our approach. Later we shall explain why we need this limitation.

### 6.2.2 *Overnondomination*

It is a well known fact that there is a tendency for a large proportion of the population to become *non-dominated* when the number of objectives increases. This may also happen for 2 and 3 objective problems, especially in the later stage of the search, depending on the problem [3]. We call this phenomenon *overnondomination*.

Overnondomination is usually a bad thing. It causes some MOEAs to stagnate and even recede from the Pareto front. As we mentioned in the introduction it also causes problems for novel non-dominated sorting methods that try to infer domination relationships, such as the deductive sort.

In this work on the other hand, we use overnondomination *to our advantage*. We use it to bridge the dynamic non-dominance problem to the non-dominated sorting problem.

### 6.2.3 *Using an archive to avoid non-dominated sorting*

In the following we use the term *archive* to mean a data structure which holds a set of *mutually non-dominated individuals*. Later we provide an exact implementation of an archive which we call the *M-front*, but all results in this section are applicable to any archive.

Algorithm 5 uses non-dominated sorting to determine which individuals get discarded after the end of the generating phase. What we need to realize is that the algorithm *does not need to know all the fronts*. It only needs enough fronts so that they contain *at least* $N$ *individuals*.

If the population suffers from overnondomination such as in the right side of Figure 37, there is a good chance that the algorithm only needs to know the *first non-dominated front* i.e. the non-dominated part of the population.

Our method is to keep track of the *non-dominated* part of the population at *all times*. We do this by keeping it in *an archive* $A$. We update the archive with each single change to the population so that it contains *only non-dominated* individuals. We can see this in lines 5 to 10 of Algorithm 6.

If we know into which front an individual $a$ belongs, we say that $a$ has a *determined front*. All the individuals in the archive have *determined fronts*, since they belong to the *first* front.

We can now see the explanation of our limitation of removing only dominated individuals in the generation phase of Algorithm 5. If we would remove a non-dominated individual $a$ from the archive $A$, there is a possibility that some individual $a' \in P \setminus A$ (in $P$ but not in $A$) which was dominated by $a$ might become *no longer dominated* and we would need to insert it into $A$ to keep the integrity of the archive. Then we would need a mechanism to detect such individuals $a'$ efficiently. In this work however, we limit ourselves to the case where non-dominated individuals cannot be removed from the population if there are some dominated individuals. If $P \setminus A$ is empty, then we are free to remove even *non-dominated* individuals without breaking the integrity of the archive.

**Algorithm 6:** Generic MOEA using our method

```
   Input: Initial population size $N$
   Output: Approximation of the Pareto front by a population $P$
 1 initialize population $P = \{a_1, ..., a_N\}$
 2 determine non-dominated individuals in $P$
 3 construct archive $A$ from the non-dominated individuals
 4 while arbitrary stopping criterion not satisfied do Evolutionary loop
 5     while arbitrary stopping criterion not satisfied do Generating phase
 6         generate new individual $a'$
 7         insert $a'$ into $A$ and $P$
 8         remove dominated individuals from $A$
 9         remove arbitrary $a \in P \setminus A$ if needed
10     end
11     if $A$ contains more than $N$ individuals then
12         remove all individuals not in $A$ from $P$
13         trim $A$ and $P$ to size $N$ using a secondary criterion
14     else
15         while # individuals with determined front $< N$ do
16             determine next non-dominated front
17         end
18         remove individuals with undetermined fronts
19         trim $P$ back to size $N$ using a secondary criterion
20     end
21 end
22 report $P$
```

When it comes to discarding the worst non-dominated fronts at the end of the evolutionary loop of Algorithm 6, we have a good chance that we *do not need to perform any non-dominated sorting at all* (line 11). We just discard all the individuals which are not in the archive.

If the archive contains fewer than $N$ individuals (line 14), we need to determine additional fronts. We do this using the simple method

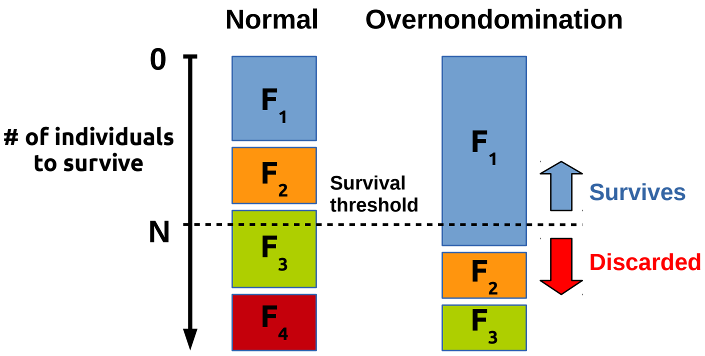


Figure 37: Survival selection based on non-dominated sorting.

described in Algorithm 7. We initialize an empty set $S$ to hold candidates for non-dominated individuals (line 1). For each individual $a_i$ in the population of size $k$ we determine whether the individual is dominated by an individual from $S$ (lines 3 to 9). If it is not dominated (line 10) we add this individual to $S$ and check which individuals in $S$ it *dominates*. These individuals are then removed from $S$ (line 14). We run this procedure repeatedly until enough individuals have their front determined.

Even if the archive does not contain more individuals than we need for the next generation (at least $N$) our task is greatly reduced since we need to compute the additional non-dominated fronts from a smaller set $P \setminus A$.

The computational complexity of our approach depends on how fast we can perform insertions and removals from the archive. In the following section we shall provide a fast archive whose usage can result in average case complexity of $O(M^2N^{2-\frac{1}{M-1}})$ which we prove in Section IV.

**Algorithm 7:** Determine non-dominated individuals

```
Input: Population $P = \{a_1, \dots, a_k\}$
Output: $S$ - set of non-dominated individuals $S \subseteq P$
$S \leftarrow \varnothing$
for $i := 1$ to : $k$ do
    bool ai_non_dominated $\leftarrow$ true
    forall the $a \in S$ do
        if $a \prec a_i$ then
            ai_non_dominated $\leftarrow$ false
            break
        end
    end
    if ai_non_dominated then
        insert $a_i$ into $S$
        forall the $a \in S$ do
            if $a_i \prec a$ then
                remove $a$ from $S$
            end
        end
    end
end
report $S$ as the non-dominated front
```

## 6.3 implementation of the archive

### 6.3.1 *Characterization*

In the previous section we argued that the problem of reducing the cost of non-dominated sorting can be approached if we had a sufficiently fast data-structure that manages a mutually non-dominated part of the population. Here we describe such a data structure, which we call an M-front. In the following sections, we gradually build up the main ideas behind the M-front.

The M-front data structure is a container which holds a *set* (in the mathematical sense) of *individuals*. It has the important invariance property that *all individuals it contains are mutually non-dominated*. This means that if we insert a new individual into the data structure, we need to determine and remove all individuals which have *become dominated*. Next we shall explain how to do this efficiently.

### 6.3.2 *Geometric motivation*

We illustrate our ideas on a two-dimensional (2 objectives) example. The circles in Figure 38 represent individuals in an archive of mutually non-dominated individuals. We insert an individual `new` into the archive $A$. In order to preserve the invariant of $A$ we need to find out:

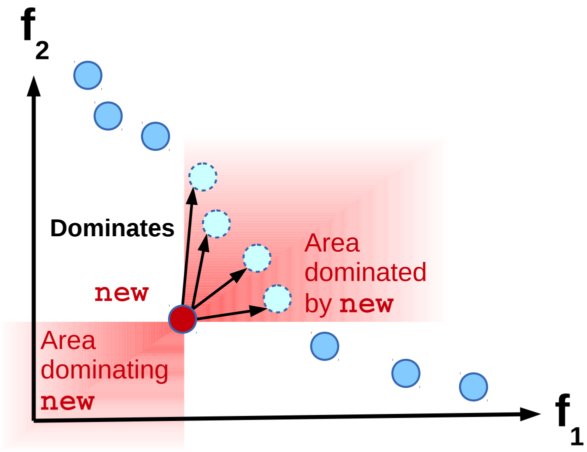


Figure 38: Insertion into a 2-dimensional archive.

1. if `new` *dominates* any individuals in $A$
2. if `new` *is dominated* by any individuals in $A$

Note that a positive answer to one of these questions implies a negative answer to the other, but there is a case when both answers are negative.

In the geometrical sense, we need to find out which individuals are in the **areas** *dominated by* or *dominating* `new`. These areas are rectangles aligned with the axes. The task of finding all the vectors which lie in such a rectangle is called an *orthogonal range query*. This is a well researched subject in the area of computational geometry and many clever techniques were developed to perform this task efficiently. Some are described in [10].

However the specific nature of our problem allows us to use a different approach. As we shall see in the following section, we can *transform* the *orthogonal range query* into an *interval query*, which is simpler. This is thanks to the specific shape of the orthogonal queries which come from the domination computation.

### 6.3.3 *Transformation of orthogonal queries*

First we select an *arbitrary* individual in the archive. We call this individual the **reference individual** and mark it with the symbol `ref`.

Next we compare the `ref` and `new` for dominance. There are three possible outcomes:

1. `ref` dominates `new`

2. `new` dominates `ref`

3. `ref` and `new` are mutually non-dominated

The first case is simple. We simply abort inserting `new`.

Let us look at the second case which is illustrated in Figure 39. We already know that `new` belongs into the archive since it cannot be dominated by any other individual. We still need to determine *all* the individuals which are dominated by `new`.

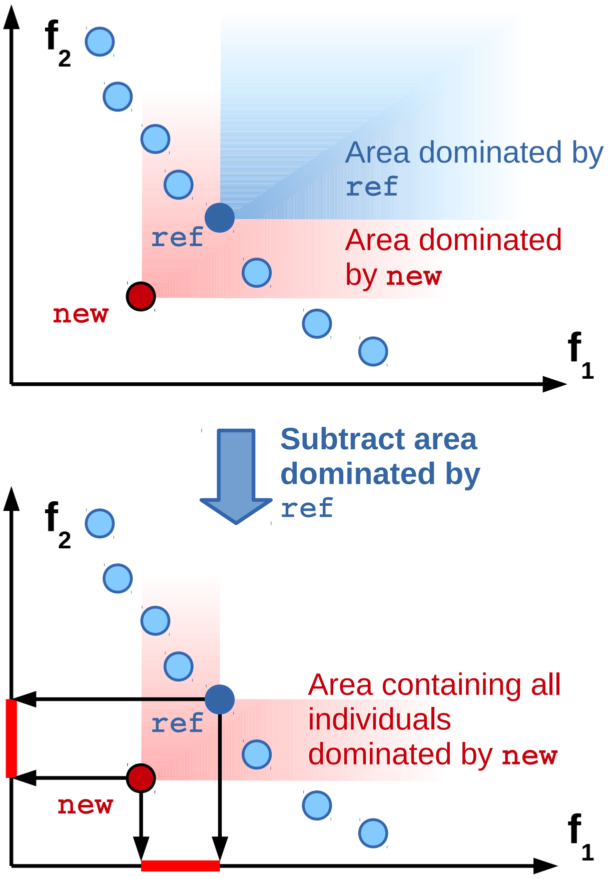

$f_2$
Area dominated by ref
ref
Area dominated by new
new
$f_1$
Subtract area dominated by ref
$f_2$
ref
Area containing all individuals dominated by new
new
$f_1$

Figure 39: Transformation of an orthogonal query into interval queries (case where `new` dominates `ref`).

We see that the area *dominated by* `ref` *does not* contain any individuals from the archive, since this would violate the invariance. Therefore we can *subtract* this area from the area dominated by `new` and we get two *intervals* which contain all the individuals which are dominated by `new`.

If an individual $a$ is dominated by $\texttt{new}$ then:

$$\begin{aligned} & f_1(a) \in [f_1(\texttt{new}); f_1(\texttt{ref})] \\ \text{or} \quad & f_2(a) \in [f_2(\texttt{new}); f_2(\texttt{ref})] \end{aligned}$$

The 2-dimensional case where new dominates ref is misleadingly simple. We see that the individuals which lie in at least one of the intervals are *exactly* the ones that are dominated by the new. In more than 2 dimensions the individuals which lie in at least one of the intervals form a *superset* of the individuals dominated by new and therefore we need to compare all of them for dominance.

Let us move to the last case. The ref and new are mutually non-dominated. Here we need to find out both *if* new itself is dominated and if it is not, *which* individuals new dominates. We do this by constructing the areas which are dominating and dominated by new, choosing a ref, constructing these areas for ref and subtracting them from the areas for new. This process is illustrated in Figure 40.

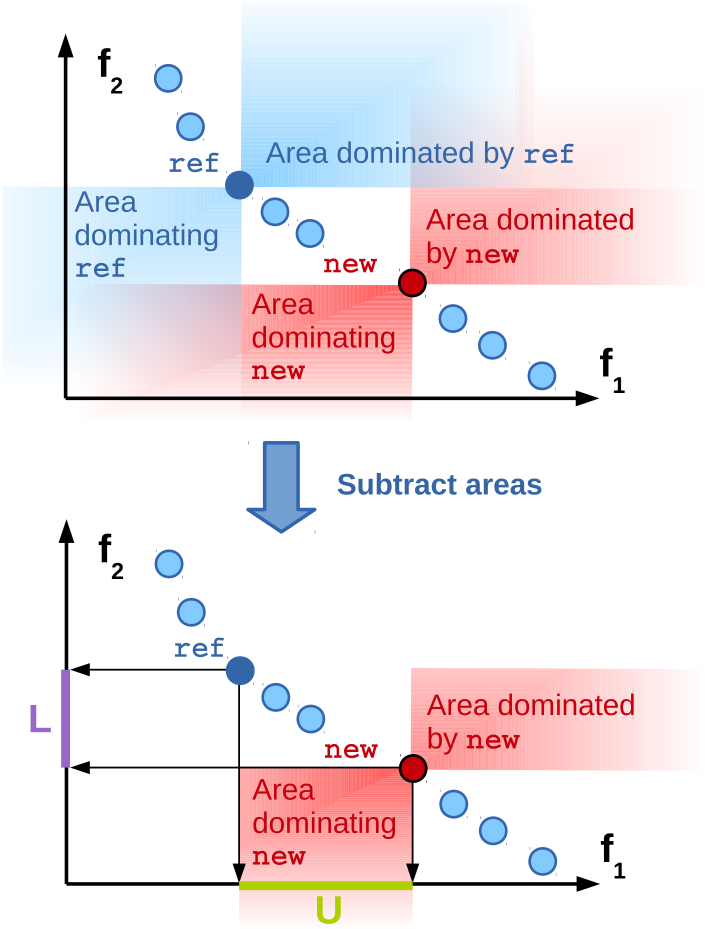


Figure 40: Transformation of an orthogonal query into interval queries (case where new and ref are mutually non-dominated).

Instead of searching the orthogonal areas which are left by this subtraction, we search the intervals marked by letters $\mathrm{U}$ and $\mathrm{L}$ in Figure 40. More formally:

If an individual $a$ dominates $\mathtt{new}$ then:

$$f_1(a) \in [f_1(\mathtt{ref}); f_1(\mathtt{new})] =: U$$

and if it is dominated by $\mathtt{new}$ then:

$$f_2(a) \in [f_2(\mathtt{new}); f_2(\mathtt{ref})] =: L.$$

We can see that there are individuals which belong to these intervals, but neither dominate nor are dominated by $\mathtt{new}$. Therefore we need to compare all of them for dominance with $\mathtt{new}$. In the future we hope to find a way how to avoid having to do this. Next, we formalize our findings in a more rigorous manner.

### 6.3.4 *Reference sets*

In the previous section we described a method to transform the areas that need to be searched when we insert a new individual into the archive. Here we formalize our ideas. First we define the *area* that needs to be searched.

**Definition 1** (Reference areas and reference sets)**.** *Let* $A = \{a_1, \ldots, a_N\}$ *be a set of mutually non-dominated individuals. Let* $\mathtt{new}$ *be an individual which does not belong to* $A$ *and let* $\mathtt{ref}$ *be an arbitrary individual from* $A$*.*

*The* upper reference **area** *for individual* $\mathtt{new}$ *induced by individual* $\mathtt{ref}$ *is the set* $RA_U(\mathtt{new}, \mathtt{ref}) \subseteq \mathbb{R}^M$ *given by:*

$$RA_U(\mathtt{new}, \mathtt{ref}) := \bigcup_{\substack{i \text{ such that} \\ f_i(\mathtt{ref}) < f_i(\mathtt{new})}} \left\{ Y \in \mathbb{R}^M \mid y_i \in [f_i(\mathtt{ref}); f_i(\mathtt{new})] \right\}. \qquad (8)$$

*We call the set of all individuals in* $A$ *whose objective vector lies in* $RA_U(\mathtt{new}, \mathtt{ref})$ *the* upper reference **set** *of individual* $\mathtt{new}$ *induced by individual* $\mathtt{ref}$*. We shall denote it by:*

$$RS_U(\mathtt{new}, \mathtt{ref}) := \{a \in A \mid F(a) \in RA_U(\mathtt{new}, \mathtt{ref})\}. \qquad (9)$$

*Conversely, the* lower reference **area** *for individual* $\mathtt{new}$ *induced by individual* $\mathtt{ref}$ *is the set* $RA_L(\mathtt{new}, \mathtt{ref}) \subseteq \mathbb{R}^M$ *given by:*

$$RA_L(\mathtt{new}, \mathtt{ref}) := \bigcup_{\substack{i \text{ such that} \\ f_i(\mathtt{new}) < f_i(\mathtt{ref})}} \left\{ Y \in \mathbb{R}^M \mid y_i \in [f_i(\mathtt{new}); f_i(\mathtt{ref})] \right\}. \qquad (10)$$

*Analogously we have the* lower reference **set***:*

$$RS_L(\mathtt{new}, \mathtt{ref}) := \{a \in A \mid F(a) \in RA_L(\mathtt{new}, \mathtt{ref})\}. \qquad (11)$$

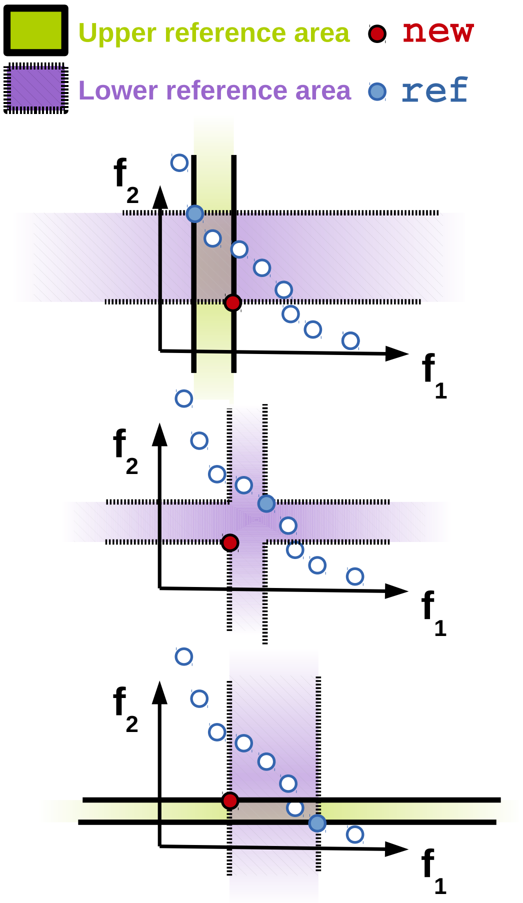


Figure 41: Reference areas induced by different choices of **reference individual**.

This definition is slightly more strict than in our previous work [18]. We can see an illustration of both upper and lower reference areas for several different choices of reference individual in Figure 41.

The following theorem says that when we are inserting a new individual into the archive, we need to compare it only to individuals from the upper and lower reference set.

**Theorem 1** (Properties of reference sets). *Let* $A = \{a_1, \ldots, a_N\}$ *be a set of mutually non-dominated individuals. Let* `new` *be an individual which does not belong to* $A$ *and let* `ref` *be an arbitrary individual from* $A$.

*Then:*

1. *if* `new` *dominates some* $a \in A$ *then*

$$a \in RS_L(\mathtt{new}, \mathtt{ref}),$$

2. *if some* $a \in A$ *dominates* `new` *then*

$$a \in RS_U(\mathtt{new}, \mathtt{ref}).$$

We shall prove only the first statement. The proof of the second statement is analogical.

*Proof.* We assume that $\mathtt{new}$ dominates some $a$. The situation is illustrated in Figure 42.

First we establish that $F(\mathtt{new}) \neq F(\mathtt{ref})$. If $F(\mathtt{new}) = F(\mathtt{ref})$ were true, this would imply that $\mathtt{ref}$ dominates $a$, which is in conflict with the assumption that all individuals in $A$ are mutually non-dominated.

Now let us handle the trivial case where $F(a) = F(\mathtt{ref})$. Since $\mathtt{new}$ dominates $a$, there must exist an objective $f_i$ such that $f_i(\mathtt{new}) < f_i(a) = f_i(\mathtt{ref})$. Therefore the union on the right side of (10) is not empty and contains $F(a)$.

Now we establish that there exists an objective $f_i$ such that:

$$f_i(a) < f_i(\mathtt{ref}) \tag{12}$$

If the opposite were true, then either $F(a) = F(\mathtt{ref})$ would hold, or $\mathtt{ref}$ would dominate $a$. We already handled the first case and the second is in conflict with the assumption of the theorem.

Since $\mathtt{new}$ dominates $a$, we have:

$$f_i(\mathtt{new}) \leqslant f_i(a). \tag{13}$$

By combining (12) and (13) we get:

$$f_i(\mathtt{new}) < f_i(\mathtt{ref}) \text{ and } f_i(a) \in [f_i(\mathtt{new}); f_i(\mathtt{ref})],$$

which means that $a$ belongs to the lower reference set. □

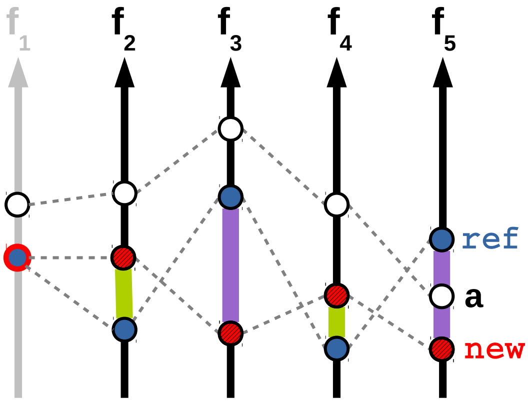


Figure 42: Illustration of the proof of Theorem 1. $f_1, \ldots, f_5$ are the axes of individual objective functions.

### 6.3.5 *M-list*

#### 6.3.5.1 *Data structure*

Now we describe how to construct the reference sets for a given pair (new, ref) *efficiently*.

To construct a reference set, we need to find all the individuals whose objective values lie in certain intervals. If we want to determine all the items whose attribute lies in a certain interval, it is helpful to have that data *sorted by that attribute*. We need to search according to all objectives, therefore we keep the population *sorted by each objective*. We keep a *sorted doubly linked list* for each objective. An illustration of these lists is shown in Figure 43. The upper and lower reference sets can be constructed simply by iterating between the positions of ref and new.

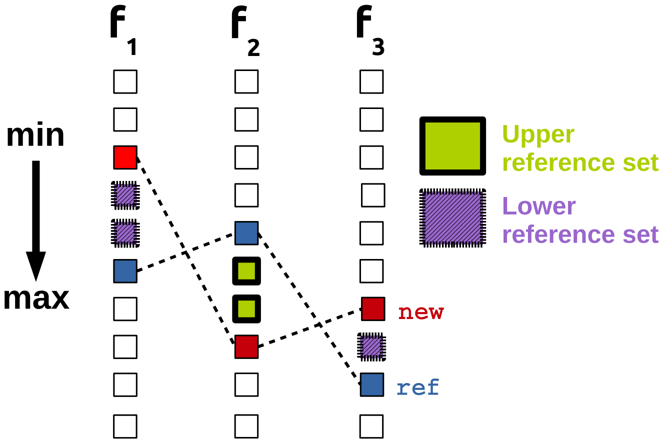


Figure 43: Constructing reference sets using linked lists.

However, the lists do not support random access. There is also no mechanism for fast insertion in logarithmic time such as with RB trees or skip-lists. This is again a slight modification to our previous work [18] where we used skip-lists. In order to remove or insert an individual into these lists while maintaining the ordering we use an alternative mechanism. We maintain a *hash-table* that maps the id's of the individuals in the archive to an *object which holds the positions of that individual in each list*. [3] We call the resulting data structure consisting of M linked lists and a hash-table an **M-list**.

#### 6.3.5.2 *Insertions and removals*

The M-list supports these two fundamental operations:

[3] We implemented this in C++ using a std::unordered_map mapping Individual* pointers to an object that held M iterators of type std::list::iterator, one for each list.

- `remove(a)` removes an arbitrary individual
- `insert(new, ref)` inserts a new individual using a reference individual

Removal of an arbitrary individual is simple. We just:

1. Retrieve the positions of $a$ from the hash-table
2. Remove the entry for $a$ from the hash-table
3. Remove the entry for $a$ from each list

In the average case retrieval and removal from the hash-table is an $O(1)$ operation. Removal from a linked list is also an $O(1)$ operation. Since we have $M$ lists, removal is a $O(M)$ operation.

On the other hand, the `insert` procedure is significantly more complex. This procedure is illustrated in Figure 44 and described in Algorithm 8. The key idea is the combination of the creation of reference sets with finding the correct position for the `new` in each list.

In the description of the algorithm we use the programming concept of an *iterator*. An iterator marks the *position* of an element in a data structure. If `it` is an iterator, then by `*it` we denote the *item* at that position. In our example iterator `it` marks the position of a certain individual in a linked list and `*it` is the individual itself. We establish the convention that the linked lists in the M-list are sorted in an *ascending* order.

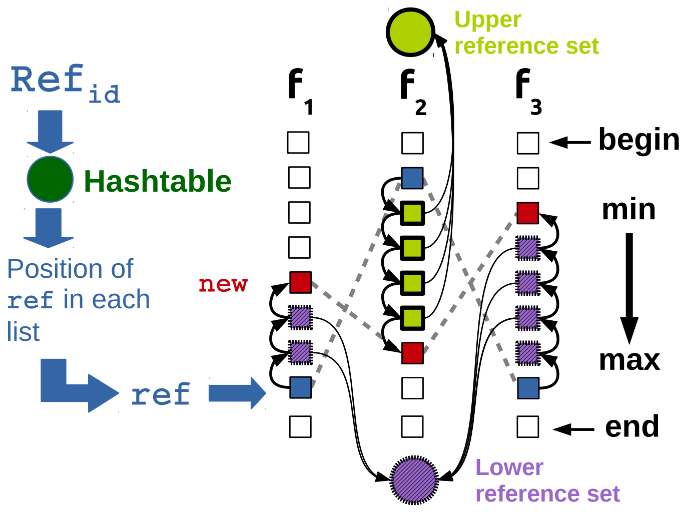


Figure 44: Insertion into the M-list.

The insertion is illustrated in Figure 44 and described in Algorithm 8 which has a helper procedure described in Algorithm 9. When we insert a new individual into the M-list, we first check if it is dominated by the reference individual (Algorithm 8 line 2). If this is the case, we know immediately that the new individual cannot be inserted into the archive and abort the insertion. If new dominates ref (line 7) we know that new is not dominated by any individual in the M-list. We store this information for future optimization (line 8).

If new is not dominated by ref, we proceed with constructing the reference sets. We initialize the upper and lower reference sets,

$$\mathrm{RS}_U(\texttt{new},\texttt{ref}) \text{ and } \mathrm{RS}_L(\texttt{new},\texttt{ref}),$$

as empty. Then for each list we must determine the correct position for new. This is accomplished by Algorithm 9.

We initialize an iterator to the position of ref in the current list. If the value of the current objective $f_i$ is the same for new and ref, we know that objective is not relevant for creation of the reference sets. We also know that the position of ref is also the correct position of new. In this case we just insert new to a neighboring position to ref and move to the next objective (line 4).

If $f_i(\texttt{new}) < f_i(\texttt{ref})$, we know that $f_i$ is relevant for the creation of the *lower* reference set. There may be more than one individual $a$ with $f(a) = f_i(\texttt{ref})$ in the list, so we increment[4] the iterator to the last position where this holds since we want to capture *all* individuals from the interval $[f_i(\texttt{new}); f_i(\texttt{ref})]$. We then start to decrement the iterator, which moves it toward the place where $f_i(\texttt{new})$ belongs in the list. Simultaneously we are creating the proper reference set (line 9). After each decrement, we check if $f_i(*\texttt{it}) \geqslant f_i(\texttt{new})$, which is equivalent to the *lower* reference set requirement: $f_i(*\texttt{it}) \in [f_i(\texttt{new}); f_i(\texttt{ref})]$. If this condition holds, we insert the individual at position it into the lower reference set. Once the condition fails, we know that we have found the right place to insert new. We insert new and move to the next list (line 12).

If $f_i(\texttt{new}) > f_i(\texttt{ref})$ the situation is symmetrical. We perform the exact opposite of all operations from the previous case while constructing the *upper* reference set.

Once the insertion of new into the lists and the construction of the reference sets is complete we create a position object holding the position of new in each list. We insert this object into the hash-table (line 15) so that we can retrieve new from the lists in constant time.

Then we check if new dominates some individuals in the M-list by comparing new to each individual in the *lower* reference set. If we find such an individual, we remove it from the M-list immediately (removal from M-list is fast as mentioned above) and set the new_non_dominated flag. If there is an individual that is dominated by new, that means that new is not dominated by any individual in the entire M-list and we can skip the following step.

[4] In Figure 44 incrementing may be seen as a *downward* movement by one box.

**Algorithm 8:** M-list::insert(ref, new)

```
Input: ref, new
Output: R - set of removed individuals from the M-list
$R \leftarrow \varnothing$
if ref dominates new then
    insert new into R
    return R
end
bool new_non_dominated ← false
if new dominates ref then
    new_non_dominated ← true
end
$RS_U(new, ref) \leftarrow \varnothing$
$RS_L(new, ref) \leftarrow \varnothing$
retrieve the position object of ref from H
position object of new =
M-list::insert_to_lists(new, position object of ref)
insert position object of new into H
forall the $a \in RS_L(new, ref)$ do
    if new dominates a then
        remove a from the archive
        insert a into R
        new_non_dominated ← true
    end
end
if new_non_dominated then
    return R
end
forall the $a \in RS_U(new, ref)$ do
    if a dominates new then
        remove new from the archive
        insert new into R
        return R
    end
end
return R
```

**Algorithm 9:** M-list::insert_to_lists(new, Position object of ref)

```
Input: new, Position object of ref
Output: Position object of new
for $i := 1$ to : $M$ do
    initialize iterator it to the position of ref in $list_i$
    if $f_i(new) = f_i(ref)$ then
        insert new before or after it
        add position to the position object
    if $f_i(new) < f_i(ref)$ then
        increment it to the last position where
        $f_i(*it) = f_i(ref)$
        while $f_i(*it) \geqslant f_i(new)$ do
            insert *it into $RS_L(new, ref)$
            decrement it
        end
        insert new right after it
        add position to the position object
    if $f_i(new) > f_i(ref)$ then
        decrement it to the last position where
        $f_i(*it) = f_i(ref)$
        while $f_i(*it) \leqslant f_i(new)$ do
            insert *it into $RS_U(new, ref)$
            increment it
        end
        insert new right before it
        add position to the position object
    end
end
return position object of ref
```

Last of all we check if new is dominated by some individual in the archive. We do this by comparing new to each individual in the *upper* reference set.

#### 6.3.5.3 *Important programming details*

We conclude this subsection with an important tip to implement an efficient M-list.

In the previous section we did not explain how to implement the data structures symbolizing the reference sets. At first sight this does not look like a significant problem. The most obvious solution is to use a container that represents a *mathematical set*. For example the `std::set` or `std::unordered_set` data structure from the C++ standard library. In general, a container that is implemented either as a sorted list or as a hash-table. This would assure that

- We can perform insertions quickly
- We know if the item being inserted already is in the set

Using a computer program profiler[5] we found out that the operations involving insertions and traversal of reference sets are critical to the performance of the entire algorithm.

Armed with this knowledge we tried to implement the reference sets using several alternatives. We tried to use programming classes which model the mathematical concept of a set using either a *hash-table* or a *red-black* tree internally.[6] We also tried our own implementation of hash-table with size growing according to powers of two and linear probing collision resolution.

Lastly we tried to implement the set as a simple array based stack, while relaxing the unique entry property of a mathematical set. If we insert a particular individual more than once, it will be in the set more than once. This may happen if an individual is between the `new` and `ref` in more than one list in the M-list.

Surprisingly, the stack implementation outperformed all other implementations in almost all instances. The additional cost of having to compare some individuals for dominance more than once was outweighed by the speed of the stack data structure. Consequently, we perform all our experiments using the stack data structure.

### 6.3.6 *K-d trees*

#### 6.3.6.1 *Nearest neighbor problem*

In this section we turn ourselves to the question of selecting a *reference individual*. We understand intuitively that if the reference individual is *close* to the new individual, the reference set induced will be *small* and we will not have much work comparing the new individual to all individuals in it. In our previous work [18] we provide experimental justification for this. Therefore we try to choose the reference individual *as close as possible* to the newly inserted individual.

Formally, we define this problem of finding a close reference individual as follows. Given an M-list ML, a metric $d$ and an individual being inserted $\mathtt{new} \notin \mathrm{ML}$ we hope to find $\mathtt{ref}$ satisfying:

$$\mathtt{ref} \in \mathrm{ML} \text{ such that } \forall a \in \mathrm{ML} : d(\mathtt{ref}, \mathtt{new}) \leqslant d(a, \mathtt{new})$$

For example, if the metric is the $L_1$-distance in the objective space, i.e., $d(x, y) := \sum_{i=1}^{M} |f_i(x) - f_i(y)|$, this measures the sum of widths of the reference areas in each objective, as illustrated in Fig. 39. Then, a

[5] Callgrind.

[6] We tried the `std::set` and `std::unordered_set` containers from the gcc 4.8.1 implementation of the C++ Standard Library. The `std::set` is implemented as a red-black tree and `std::unordered_set` is implemented as a hash-table with a prime size and separate chaining collision resolution.

reference point close to new w.r.t. this metric tends to provide a small reference set. This task is a well studied problem with name *nearest neighbor search*.

Importantly, our objective here is not to find out the nearest individual in a strict manner, but a relatively close reference point. Especially, we do not want to spend much more time to find out the exact solution than to perform the operations described in the previous section. Therefore, an approximation approach to the nearest neighbor search is a good candidate for our purpose.

We employ the K-d tree approach [10] to perform an approximate nearest neighbor search. K-d tree is a hierarchical data structure which supports fast *nearest neighbor* and *approximate nearest neighbor* queries. The K-d tree structure has been reported to have a good performance when the *dimension of the problem is small*. Dimensions until about 8 are considered very small in the nearest neighbor community, whereas the MOEA community considers problems with more than 4 objectives as many-objective. Therefore, we maintain a K-d tree *along* the M-list.

We will not describe the general mechanism of the K-d tree since there are many publications that provide an excellent description (see e.g. [50]). We describe only the details of the particular K-d tree implementation that we tested ourselves and to which our experimental results apply.

#### 6.3.6.2 *Implementation details*

With each insertion to the M-list we need to perform an insertion into the K-d tree and the same goes for removals. Because we need to use the K-d tree in such a dynamic manner, we use a slightly modified version. We keep the data *only in the leaves* of the tree. This results in somewhat simpler removals and insertions into the tree.

The approximate nearest neighbor procedure proceeds exactly as the standard exact nearest neighbor procedure, but allows for only 4 evaluations of metric $d$. Once these 4 evaluations have been spent, the procedure returns the closest individual found so far. We chose 4 evaluations ad hoc, since it seemed to perform well in many problem instances.

#### 6.3.6.3 *Re-balancing the K-d tree*

The major problem with using K-d trees is that it is primarily a *static* data structure. That is, it is not well suited for the dynamic character of the multi-objective optimization. When many insertions and deletions are performed, the tree tends to become *unbalanced*. To our best knowledge there is no efficient method to detect the fact that the tree is unbalanced and to perform the re-balancing. Therefore we developed a heuristic to determine if the tree is out of balance and if it is, we simply destroy it and construct it again.

Our mechanism to detect the loss of balance of a K-d tree is as follows. At the beginning of the algorithm, we compute the bounding hyper-box of the population in the M-list (illustrated in Figure 45) and denote it by $H_{\text{old}}$. During optimization, we periodically compute the bounding hyper-box $H_{\text{new}}$ of the population which is currently in the M-list. Then we compute the ratio of the volume of the *intersection* of these boxes to their *union*:

$$\alpha := \frac{\text{vol}(H_{\text{new}} \cap H_{\text{old}})}{\text{vol}(H_{\text{new}} \cup H_{\text{old}})} \; . \tag{14}$$

A value of $\alpha$ from (14) near 1 means that the boxes are almost identical, while a value close to 0 means that they are quite different. This can happen either by the box $H_{\text{new}}$ becoming too small, too big, or moving away from $H_{\text{old}}$. If the value drops bellow a predefined level (we have chosen 0.5 ad hoc), we rebuild the tree from scratch and replace $H_{\text{old}}$ by $H_{\text{new}}$.

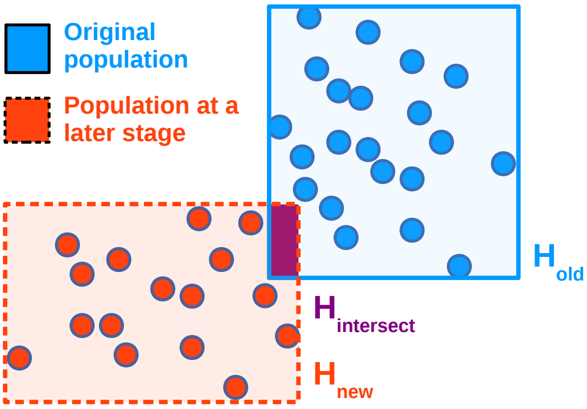


Figure 45: Computing $\alpha$ in (14).

A nice feature of the $\alpha$ indicator in (14) is that $\alpha \in [0; 1]$ and that it is invariant to the scaling of the axes.

### 6.3.7 *M-front*

There is another reason why we chose the K-d tree in particular. As [10, p.101] suggests, the construction of the K-d tree can get expensive because of the need to compute the medians at each node to split the data uniformly. On the same page, the author suggests pre-sorting the data with respect to each dimension, in order to avoid the costly computation of the medians. The M-list is a data structure where the data *already is sorted* with respect to each dimension, which reduces the computational cost even further. To express this affinity of the M-list and K-d tree, we call the combined structure the M-front.

An insertion into the M-front is described in Algorithm 10. When we insert `new` into the M-front, a reference individual is chosen as the *approximate nearest neighbor* to `new` (line 1). `new` is next inserted into both parts of the M-front, i.e. the M-list and the K-d tree. The individuals that are removed from M-list (they may contain also `new`) must be also removed from the K-d tree (line 5).

**Algorithm 10:** Insertion into the M-front

```
   Input: M-front internals: K-d tree $K$, M-list ML, new
   Output: $R$ - set of removed individuals from the M-front
1  ref ← retrieve approximate nearest neighbor to new using $K$
2  insert new into $K$
3  $R$ ← insert(ref, new) into ML
4  forall the $a \in R$ do
5      remove $a$ from $K$
6  end
7  return $R$
```

We can also remove arbitrary individuals from the M-front by removing them from the M-list and from the K-d tree. This is particularly useful when we want to prune the set of non-dominated individuals. Here one can take advantage of the M-front internals. For example, one method to prune non-dominated individuals is the *partitioned quasi-random selection (PQRS)* [26]. The computational cost of this procedure is decreased if the population is sorted with respect to each objective. Hence the M-list can be used to decrease the cost of PQRS. Similarly, one can take advantage of the K-d tree, which is a data structure suitable for efficient nearest neighbor computation, to reduce the computational cost of pruning procedures which perform these computations, such as *the M nearest neighbors pruning* [37].

Source code for all mentioned data structures can be found in [15].

## 6.4 COMPUTATIONAL COMPLEXITY

In this section we theoretically explore the computational complexity, i.e. the number of required operations, especially floating point number comparisons, in our algorithm. Since the core of our method is the insertion into the M-front, we investigate the computational complexity of one such insertion.

### 6.4.1 *Best and worst case complexity of our method*

The M-front is composed of two data structures, namely the M-list and the K-d tree. Each insertion starts with the retrieval of the reference individual from the K-d tree. If the K-d tree containing N items is balanced, then approximate nearest neighbor queries can be performed in O(ln(N)) time. Therefore if we restrict our usage of the K-d tree to computing just the approximate nearest neighbors, the cost of retrieving the reference individual is O(ln(N)). After the reference individual is retrieved, it is compared to the newly inserted individual for dominance. If the reference individual dominates the newly inserted individual, the insertion is aborted, leaving the computational cost at O(ln(N)). If the reference individual does not dominate the new individual, its position object is retrieved from the hash-table in the M-list. This operation is O(1) in the *average* case and O(N) in the *worst* case.

Afterwards, the reference sets are constructed while inserting the new individual into the linked lists of the M-list. There is a non negligible chance that the upper and lower reference sets are *empty* or contain only the reference individual itself. In this case there are no more operations needed. This is especially likely if there is a reference individual which is particularly closer to the new individual than the other individuals in the M-front. We can see an illustration of a sequence of 5 best case insertions in Figure 46. Each time the new individual is paired with its reference individual in the archive. All insertions yield empty upper and lower reference sets. Intuitively we see that the probability of such a sequence is not infinitesimally small, but it may actually happen. Especially if the underlying MOEA is evolving the population by perturbing one individual at a time. The individual being perturbed may serve as a reference individual, saving thus the cost of the K-d tree.

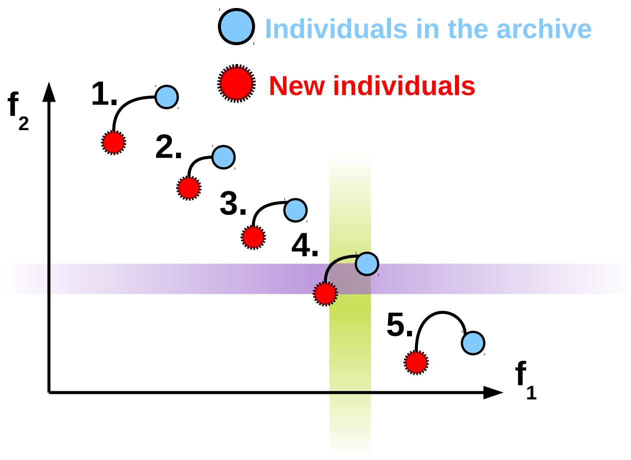


Figure 46: Sequence of insertions with empty reference sets.

In the worst case, the reference sets contain all the individuals in the archive. In this case there are N domination comparisons needed, i.e. $O(MN)$ floating point number comparisons, and the complexity is the same as the brute force comparison.

An insertion could cause the K-d tree to be considered unbalanced. In that case it needs to be rebuilt. The cost of rebuilding a K-d tree is $O(MN\ln(N))$ [10]. However if the tree is checked for imbalance periodically each $T$ insertions and if $T \geqslant N$ then there is at most 1 *rebuilding* of the tree for $N$ insertions. Then the average cost for one insertion is $O(M\ln(N))$ which does not change the worst case complexity.

### 6.4.2 *Average case complexity of our method*

We model the *average case* complexity of our method using the concept of *random variables* from probability theory. Similarly as when establishing the best case complexity, we estimate the computational complexity of inserting a single individual into an M-front archive. We model the population using *random vectors*. Therefore the complexity that we estimate is itself a random variable. We estimate its *expected value* and *asymptotic properties*.

We try to use familiar naming conventions but from now on, we deal with *random* individuals.

**Definition 2** (Random individual)**.** *A* random individual $a$ *is the ordered pair* $(\mathrm{id}_a, Y_a)$ *where:*

1. $\mathrm{id}_a \in \mathbb{N}$ *is the identifier,*
2. $Y_a = (y_{a,1}, \ldots, y_{a,M}) \in \mathbb{R}^M$ *is a random* vector.

The vector $Y_a$ is supposed to model the *objective vector*. In this section we are not interested in the decision vectors at all. We are just trying to model a snapshot of the population in a MOEA.

The most important property of an M-front is that all the individuals within are *mutually non-dominated*. We model the population of the M-front using the following definition:

**Definition 3** (Random front)**.** *Let* $\mathrm{RF} = \{a_1, \ldots, a_n\}$ *be a set of random individuals. If the probability that there are individuals* $a_i, a_j \in \mathrm{RF}$ *such that* $a_i$ *dominates* $a_j$ *is zero, then we call* RF *a* random front.

We shall estimate the computational complexity of inserting an individual into an M-front whose population is a certain specific type of a *random front*.

**Definition 4** (Uniform front)**.** *Let Pf* : $\mathbb{R}^{M-1} \to \mathbb{R}$ *be a function that is* strictly decreasing *with respect to each variable.*

*Let* $RF_{Pf} = \{a_1, \ldots, a_n\}$ *be a set of random individuals whose objective vectors are independent identically distributed (i.i.d.) random vectors with the following distribution: Each vector's first* $M-1$ *components are i.i.d.*

*uniform random variables on* $[0;1]$. *The last component is the value of function Pf of the first* $M-1$ *components*

$$Y = (y_1, \dots, y_{M-1}, Pf(y_1, \dots, y_{M-1})) \ .$$

*We call* $RF_{Pf}$ *a* uniform front *with shape Pf.*

The mutual non-domination of individuals is guaranteed thanks to the decreasing nature of the shape function *Pf*, as is formally described in the following theorem.

**Theorem 2** (Correctness of uniform front)**.** *A uniform front is indeed a* random front *in the sense of Definition 3.*

The proof can be found in the Appendix A.

The uniform random front models the individuals in an M-front. Let us now investigate the computational cost of an insertion into such an M-front. As we explained earlier, this cost is proportional to the number of individuals in the reference sets. Therefore we estimate the *expected cardinality of the reference sets.* In order to keep things as simple as possible, we shall assume that the inserted individual is in the *center* of the uniform random front. That is:

$$Y_{\mathsf{new}} = (0.5, \dots, 0.5, Pf(0.5, \dots, 0.5)) \ .$$

Furthermore we shall add some assumptions on the shape function of the front *Pf*.

**Theorem 3** (Expected cardinality of the reference sets)**.** *Let* $Pf : \mathbb{R}^{M-1} \to \mathbb{R}$ *be a*

- *Lipschitz function with respect to the maximum metric with constant* $L$, *i.e.* $\forall X, Y \in \mathbb{R}^{M-1}; |Pf(X) - Pf(Y)| \leqslant L \cdot \max_{i \in [\![1;M-1]\!]} |x_i - y_i|$, *and*
- *there exists an* $S \in \mathbb{R}$ *such that the probability density function* $f_{Pf}$ *of the random variable* $Pf(y_1, \dots, y_{M-1})$ *where* $y_i \sim U[0;1]$ *are i.i.d., is bounded by* $S$.

*Let* $d_{M-1} : \mathbb{R}^M \times \mathbb{R}^M \mapsto [0;\infty)$ *be the maximum pseudo-metric defined by:*

$$d_{M-1}(X, Y) := \max_{i \in [\![1;M-1]\!]} |x_i - y_i| \ .$$

*Let* $RF_{Pf}$ *be a* uniform random front *with shape Pf containing* $N$ *individuals,* `new` *be a newly inserted individual with objective vector*

$$Y_{new} = (0.5, \dots, 0.5, Pf(0.5, \dots, 0.5))$$

*and* `ref` *be a reference individual chosen as the closest individual to* `new` *with respect to* $d_{M-1}$.

*Let the number of individuals which fall either into the upper or lower reference sets denoted by* $C_{M,N}$. *Then* $E[C_{M,N}]$ *exists for all* $M,\ N \in \mathbb{N}$, $M > 1$, *and*

$$E[C_{M,N}] \in O(MN^{1-\frac{1}{M-1}}) \qquad (15)$$

*for a fixed* $M$.

The proof can be found in Appendix B.

The requirements imposed on *Pf* may seem complicated and heavy handed but we do it for the sake of simplicity of the proof. One example of such a function is an arbitrary linear function with all coefficients negative.

Since each dominance comparison has the computational cost of $O(M)$, we see that the *expected computational cost of inserting into an M-front* is

$$O(M^2N^{1-\frac{1}{M-1}}) \qquad (16)$$

floating point comparisons.

### 6.4.3 *Summary*

We have estimated the computational cost of insertions into the M-front. The most costly operation is comparing against individuals in the reference sets. Therefore we were concerned with the *cardinality of the reference sets*.

If we look at the algorithm in Figure 6 and assume overnondomination, there are approximately the same number of individuals in the M-front as there are individuals in the initial population. Therefore the N in both contexts is roughly the same. In order to be able to compare the computational complexity to an algorithm that performs non-dominated sorting, we shall multiply the costs of inserting one individual (16) by N, since in the course of one generation of algorithm in Figure 6 there are N insertions. We summarize the computational complexities in Table 8. As we see in Table 8, our approach scales better with respect to M than Jensen-Fortin's algorithm in the average case. On the other hand Jensen-Fortin's algorithm scales better in terms of N for a fixed M.

## 6.5 COMPARISON WITH JENSEN-FORTIN'S METHOD

Jensen-Fortin's algorithm [36] [29] is one of the fastest non-dominated sorting algorithms so far. Our algorithm is different on many levels other than speed. In this section we shall go into depth on all the details in which the two algorithms differ.

Table 8: Computational complexities where N is the population size and M is the dimension.

| | **Jensen-Fortin** | **M-front** |
|---|---|---|
| Best case | $O(MN)$ | $O(MN)$ or $O(MN\ln(N))$ using the K-d tree |
| Average case | $O(N\ln^{M-1}(N))$ | $O(M^2N^{2-\frac{1}{M-1}})$ |
| Worst case | $O(MN^2)$ | $O(MN^2)$ |

### 6.5.1 *Description of Jensen-Fortin's algorithm*

We mentioned Jensen-Fortin's algorithm in the introduction. Here we shall describe it in a little more detail. Jensen-Fortin's algorithm is based on Kung's algorithm [40]. This algorithm computes the set of non-dominated individuals in a population. Instead of repeatedly using Kung's algorithm to compute non-dominated fronts one by one, Jensen chose a cleaner approach which constructs all non-dominated fronts in one run.

Unfortunately the algorithm did not work well in the general case where more than one individual has the same value for some objective. This was recently fixed by Fortin et al. From now on, we shall work only with the Fortin's version of the algorithm. We call this algorithm the Jensen-Fortin's algorithm. The algorithm uses a divide-and-conquer strategy. This is illustrated in Figure 47.

In the notation of Fortin et al., the algorithm has two main procedures, `Helper_A` and `Helper_B`, and two splitting procedures, `Split_A` and `Split_B`. After some preprocessing, the algorithm calls the procedure `Helper_A` which does essentially all the work. This procedure splits the problem into problems of smaller size and merges the results using the `Helper_B` procedure which is itself a recursive divide-and-conquer algorithm. `Helper_B` splits the problem again using the `Split_B` procedure. The problem is further divided until either the dimension M or the problem size N gets reduced to 2, which are handled as final cases.

### 6.5.2 *Conceptual comparison*

The main difference between the two methods is that Jensen's method is a *procedure* while our method is essentially a *data structure*. Our data structure keeps track of the non-dominated individuals at *all times* and this knowledge is updated with each change in the population.

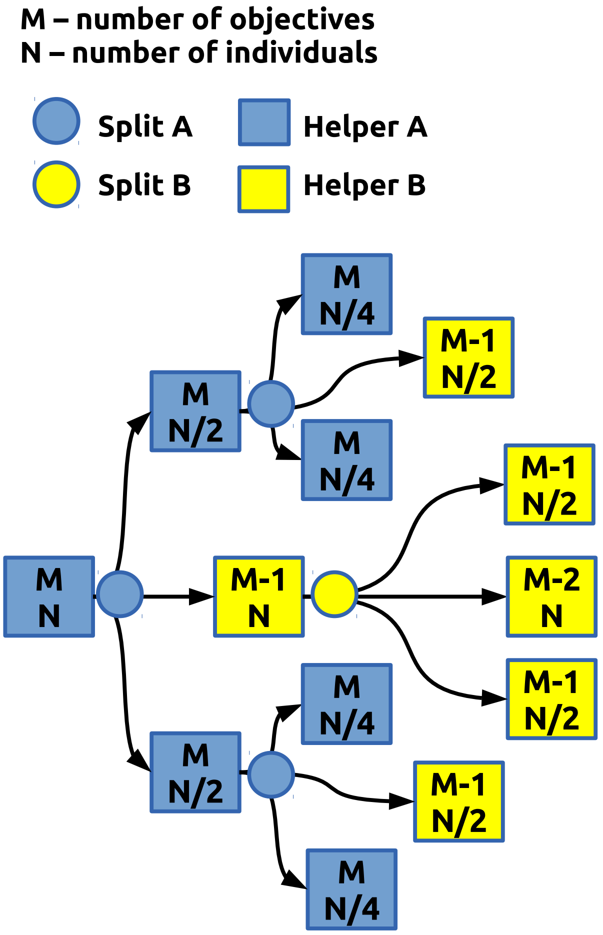


Figure 47: Jensen-Fortin's algorithm.

This is not possible with Jensen's algorithm. Once a single individual changes the entire computation needs to be executed again.

The advantage of Jensen's method is that it computes *all* the non-dominated fronts, while our method computes only those fronts that *are needed by the trimming procedure*.

### 6.5.3 *Computational speed*

In the section on experimental results we show that Jensen's algorithm performs well on *large populations*, while our algorithm works well with *a large number of objectives*.

Another main difference is that Jensen's algorithm performs the entire computation at once, while our method allows the cost to be distributed along the entire run of the algorithm. As soon as an individual's objective value is evaluated, we can insert it into the M-front.

### 6.5.4 *Flexibility*

The Jensen's algorithm is fairly difficult to modify. It took 10 years since the original publication by Jensen for someone to undertake the task to generalize the algorithm to be able to handle the case of multiple individuals sharing the same objective value. No other modifications are known.

On the other hand, the core of our algorithm is just the M-list. The way in which reference individuals are chosen depends entirely on the user. The user is free to choose any strategy to select the reference individual. There are probably many data structures more sophisticated than the K-d tree. Some algorithms compute the nearest neighbor in the objective space for their own purposes. This computation can be reused in inserting an individual into the M-list. A notable example of such an algorithm is the DEMO/obj algorithm [47]. Other algorithms perturb the population one individual at a time. In this case the unperturbed individual is likely to be close to the perturbed one, and may serve as the reference individual. One example of such a MOEA is *differential evolution*. We used this approach in our previous work [18].

The M-list itself can be modified to use a different type of dominance, such as the $\epsilon$-dominance. This may be also possible with Jensen's algorithm, but it is not very straightforward.

Lastly, the M-front is a standalone archive which can be used in algorithms such as [58] that store their non-dominated individuals.

### 6.5.5 *Parallelization*

The `Helper_A` procedure splits the problem into three subproblems which need to be solved in a particular order. Even though divide-and-conquer algorithms are usually easy to parallelize, there is a sequential dependence in `Helper_A`.

The three problems created by `Helper_B`, on the other hand, can be executed in any order. Therefore we suppose that Jensen's algorithm can be easily parallelized.

Our algorithm performs many insertions and removals each of which locks the M-front. Parallelization within each transaction is possible but because the insertion is a relatively small operation we are not sure if significant speedup can be achieved.

### 6.5.6 *Summary*

The differences between Jensen-Fortin's algorithm and our method are summarized in Table 9.

When deciding which algorithm to use for one's application, the most important question one needs to answer is whether all the non-

Table 9: Comparison of the Jensen-Fortin's algorithm and our method.

| | **Jensen-Fortin** | **M-front** |
|---|---|---|
| Average complexity | $O(N\ln^{M-1}(N))$ | $O(M^2N^{2-\frac{1}{M-1}})$ |
| Best performance on | High $N$ | High $M$ |
| Main concept | Procedure | Data structure |
| Computes all fronts | Yes | No |
| Flexibility | No | Yes |
| Parallelization | Yes | Yes |

dominated solutions should be known at all times. If this is so, one should use our approach, since there is no way to update the non-dominated set once one individual changes. If however, the underlying algorithm needs to know the non-dominated individuals only at some specified instant in the course of optimization, the user should consider the population size and the dimensionality of the problem and consult the experimental data in the flowing section.

## 6.6 EXPERIMENTAL RESULTS

### 6.6.1 *Experimental setup*

In order to test the performance of our algorithm we have implemented the GDE3 (Generalized Differential Evolution) MOEA [38] using three non-dominated sorting methods:

- Our method (M-front)
- Jensen-Fortin's algorithm
- Deb's fast non-dominated sorting

The three algorithms produce identical outputs. Only thing that is different is the speed. We ran the algorithm on a variety of DTLZ1 [13] and WFG9 [34] problems.

We chose WFG9 in particular because it is multi-modal and non-separable and therefore we hope that it resembles a large number of real world problems.

We chose DTLZ1 because its objective functions are relatively steep. This means that the evaluation of the initial randomly initialized population is quite far from the *true Pareto front*. The randomly initialized population has objective values in the ranges of hundreds while the true Pareto front is a simplex within the hyper-box $[0; 0.5]^M$. During the run of the MOEA, the population needs to travel a significant distance. This should test our K-d tree re-balancing mechanism.

The GDE3 algorithm has two main parameters. The crossover operator and the scaling factor F. We have chosen *exponential crossover* and a value of 0.2 for both F and the *crossover probability* Cr. We chose the parameters according to empirical results by Kukkonen [48] and after an informal off-line calibration of the algorithm.

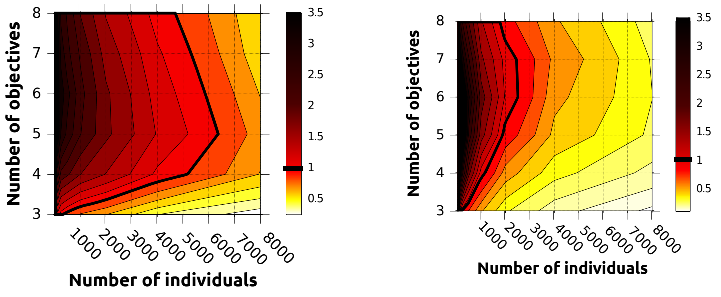
We ran experiments with various *initial population sizes* N and various *numbers of objectives* M. The population sizes start at 50 individuals and increase in an almost geometric progression up to 8000 individuals. We chose such big populations to clearly demonstrate at which population size the asymptotic superiority of Jensen-Fortin's algorithm prevails and our method is outperformed. Large population sizes are especially useful for many objective problems where the number of individuals needed to approximate the Pareto front with a fixed precision grows *exponentially* with the number of objectives [3].

The number of objectives ranges from 3 to 8, while the number of variables is always 15. For each configuration we ran the algorithm for 500 generations, 10 times with different random seeds, and averaged the results. The experimental setup is summarized in Table 10.

Table 10: Experimental setup.

| | |
|---|---|
| Crossover | exponential |
| Cr | 0.2 |
| F | 0.2 |
| initial population size N | 50, 125, 250, 500, 1000, 2000, 4000, 8000 |
| number of objectives M | 3, 4, 5, 6, 7, 8 |
| number of generations | 500 |
| number of runs | 10 |
| number of variables | 15 |

We implemented all algorithms in C++ and compiled them using the gcc 4.8.1 compiler using the aggressive compiler optimization flag -O2. We ran the experiments on a desktop PC with an Intel® Core$^{TM}$ i7-2600 CPU @ 3.40GHz x 8 processor running Ubuntu 13.04 operating system with Linux 3.8.0-19.29 kernel. We ran the experiments one at a time, with no other programs running.



(a) WFG9

(b) DTLZ1

Figure 48: Ratio of average total non-dominated sorting time: $\frac{\text{Jensen-Fortin}}{\text{M-front}}$.

### 6.6.2 *Comparison with Jensen-Fortin's algorithm*

#### 6.6.2.1 *Total computation time*

First we measured the total wall clock computation time *spent just on non-dominated sorting* using the C++11 <chrono> library. Instead of presenting the *absolute* times, which we believe to be more susceptible to change from platform to platform, we present *ratios* of the average time used by Jensen-Fortin's algorithm *divided* by the average time used by our method. Since we have the same number of runs for both algorithms, the presented ratios become:

$$\text{ratio} = \frac{\sum_{k=1}^{10} JF_k}{\sum_{k=1}^{10} MF_k}$$

where $JF_1, \ldots, JF_{10}$ are the times taken by Jensen-Fortin's algorithm and $MF_1, \ldots, MF_{10}$ are the times taken by the M-front method. All results are summarized in Table 11 and Table 12. All results have been tested for significance using the Wilcoxon signed rank test on the significance level 0.05. A number in boldface means that our method *significantly* outperformed the competitor, while a number in italics means that our method has been *significantly* outperformed for that particular configuration. A number without boldface or italics means that the results were not significantly different. We use the same notation in Table 14, where we compare our results with fast non-dominated sorting.

Numbers greater than one mean that our method is faster in this instance. For the sake of perspective, we provide the average times used by Jensen-Fortin's algorithm for the *most complex* and *least* complex problem setup in Table 15. The numbers in parentheses are the standard deviations across the 10 runs.

We can see that for 3 objectives the Jensen-Fortin's method is faster in almost all instances. For 4 objectives *and higher* our method catches up and outperforms the Jensen-Fortin's algorithm for all population

Table 11: Ratio of average time for non-dominated sorting: $\frac{\text{Jensen-Fortin}}{\text{M-front}}$ for the WFG9 problem.

| M | N = 50 | 125 | 250 | 500 | 1000 | 2000 | 4000 | 8000 |
|---|---|---|---|---|---|---|---|---|
| | Total (ratios) | | | | | | | |
| 8 | **3.13** | **3.50** | **2.79** | **2.54** | **2.05** | **1.57** | **1.10** | *0.56* |
| 7 | **2.79** | **3.77** | **3.08** | **2.67** | **2.28** | **1.72** | **1.21** | *0.59* |
| 6 | **2.79** | **3.61** | **3.36** | **2.98** | **2.41** | **1.86** | **1.29** | *0.68* |
| 5 | **2.96** | **3.78** | **3.17** | **2.78** | **2.33** | **1.87** | **1.36** | *0.76* |
| 4 | **2.27** | **2.70** | **2.23** | **2.10** | **1.82** | **1.58** | **1.14** | *0.68* |
| 3 | *1.00* | **1.29** | 1.09 | 0.96 | *0.82* | *0.69* | *0.44* | *0.28* |
| | In the last generation (ratios) | | | | | | | |
| 8 | **2.38** | **3.68** | **2.87** | **2.60** | **2.17** | **1.68** | **1.14** | *0.59* |
| 7 | **3.30** | **4.61** | **3.04** | **2.72** | **2.33** | **1.78** | **1.24** | *0.61* |
| 6 | **2.15** | **4.07** | **3.45** | **3.08** | **2.45** | **1.89** | **1.33** | *0.70* |
| 5 | **2.64** | **3.42** | **3.10** | **2.67** | **2.31** | **1.87** | **1.41** | *0.79* |
| 4 | **2.94** | **3.01** | **2.21** | **2.01** | **1.79** | **1.69** | **1.27** | *0.79* |
| 3 | 0.77 | **1.37** | 1.04 | 1.04 | *0.89* | *0.86* | *0.62* | *0.48* |

sizes up to 4000 individuals. There the asymptotic superiority of Jensen-Fortin's algorithm becomes apparent.

The smaller the population the better our method performs in comparison to Jensen-Fortin's algorithm. However, this trend breaks down for 50 individuals. This is probably due to the fixed cost that the K-d tree carries along.

The reader should note that the initial population sizes do not correspond *exactly* to the size of the domination sorting problem being solved. The GDE3 algorithm produces a population that has somewhere between N and 2N individuals, which needs to be trimmed to N individuals. Therefore the size of the problem being solved is slightly bigger.

We can interpolate our experimental results so that the pattern is more visible. In Figure 48 we can see a contour plot which interpolates our experimental results. Population sizes in our experiments grow exponentially, but Figure 48 tries to give us a better understanding of the overall pattern. The isocurve for 1, that is the hypothetical

Table 12: Ratio of averages $\frac{\text{Jensen-Fortin}}{\text{M-front}}$ for the DTLZ1 problem.

| M | N = 50 | 125 | 250 | 500 | 1000 | 2000 | 4000 | 8000 |
|---|---|---|---|---|---|---|---|---|
| | Total non-dominated sorting time (ratios) | | | | | | | |
| 8 | **1.96** | **2.97** | **2.53** | **2.03** | **1.40** | *0.90* | *0.57* | *0.29* |
| 7 | **2.20** | **3.97** | **3.27** | **2.61** | **1.64** | **1.11** | *0.65* | *0.33* |
| 6 | **2.32** | **5.08** | **4.23** | **3.23** | **1.94** | **1.15** | *0.59* | *0.34* |
| 5 | **2.44** | **5.30** | **4.40** | **2.78** | **1.69** | *0.94* | *0.52* | *0.32* |
| 4 | **1.91** | **3.93** | **2.89** | **1.89** | **1.10** | *0.62* | *0.37* | *0.23* |
| 3 | 1.11 | **1.48** | 1.01 | *0.82* | *0.57* | *0.35* | *0.21* | *0.13* |
| | Non-dominated sorting time in the last generation (ratios) | | | | | | | |
| 8 | **2.15** | **3.66** | **2.24** | **2.30** | **1.47** | *0.89* | *0.62* | *0.32* |
| 7 | **2.07** | **4.27** | **3.36** | **2.67** | **1.78** | 1.04 | *0.67* | *0.30* |
| 6 | **2.17** | **4.28** | **4.57** | **3.18** | **1.91** | **1.16** | *0.58* | *0.34* |
| 5 | **2.01** | **4.03** | **4.00** | **2.70** | **1.96** | **1.19** | *0.74* | *0.40* |
| 4 | **1.65** | **3.05** | **3.11** | **2.58** | **1.88** | **1.26** | *0.81* | *0.51* |
| 3 | 0.90 | **1.58** | 1.26 | **1.32** | 1.01 | *0.82* | *0.54* | *0.42* |

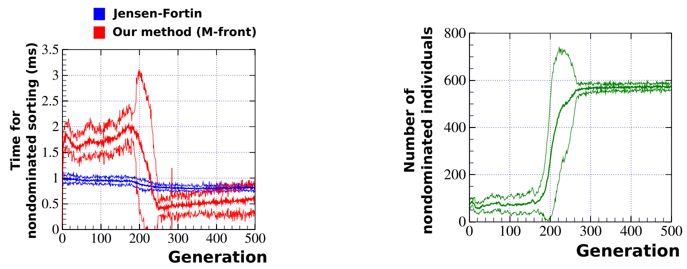
curve on which the two algorithms perform with the same speed, is shown in bold black.

When we look at the results for DTLZ1 in Table 12 we see a similar pattern. The asymptotic superiority of Jensen-Fortin's method is immediately visible. Moreover we see that for large populations the results are more favorable for Jensen-Fortin's method. Already for 2000 individuals the performance of the two algorithms are tied, while for 4000 and 8000 individuals Jensen-Fortin's algorithm is faster. On the other hand, for small population sizes and high number of objectives, our method is faster.

#### 6.6.2.2 *Computation time in the last generation*

On first sight it may seem strange that the results for 3 objectives are so unfavorable for our method. We shall now examine this situation in detail.

Let us look at the computational time spent on non-dominated sorting *in each generation*. In Figure 49a we can see the averaged computational times for the DTLZ1 problem with 3 objectives and 500 in-



(a) Decrease in the average computational time of our method around generation 200.

(b) Increase in the average number of non-dominated individuals around generation 200.

Figure 49: Influence of overnondomination on computational speed of our method for DTLZ1 - 500 individuals - 3 objectives. All series are surrounded by a band of ±2 standard deviations.

dividuals. Our method is significantly slower, but around generation 200 it accelerates and becomes faster than Jensen-Fortin's algorithm.

The reason for this behavior is that the proportion of non-dominated individuals is relatively small in the first 200 generations. We can see this in the lower part of Figure 49b. When the number of non-dominated individuals is small, our algorithm needs to resort to using the algorithm in Figure 7 to determine additional fronts. Once the number of non-dominated individuals is greater than the initial population size, 500 individuals in this case, the GDE3 algorithm has enough non-dominated individuals in the M-front and does not need to invoke the auxiliary algorithm (Figure 7) to compute an additional non-dominated front.

We can examine how strong this effect is, by comparing the average computational times *only in the last generation*. The intuition is that by the last generation the population has almost converged and the proportion of the non-dominated individuals is high.

These results are summarized in the bottom of Table 11 and Table 12. By comparing the data for the last generation to the total data we can see that the most significant differences are visible for 3 objectives. The reason is that for 3 objectives the overnondomination phenomenon is not yet present.

### 6.6.3 *Comparison with fast non-dominated sorting*

Here we present only the results for the WFG9 algorithm in order to save space, since the core of our experimental section is the comparison with Jensen-Fortin's algorithm. The results for DTLZ1 were slightly worse, but similar to those for WFG9. This can be inferred from looking at Table 11 and Table 12.

We choose the same methodology of presenting our data as when comparing with Jensen-Fortin's algorithm.

In Table 14 we see that our algorithm outperforms the fast non-dominated sorting in *all* problem instances. With few exceptions the relative performance of our algorithm *increases with population size* and *decreases with number of objectives*. This is in accordance with our estimation of average computational complexity.

Table 13: Ratio of averages $\frac{\text{Fast non-dominated sorting}}{\text{M-front}}$ for the WFG9 problem.

| | Total non-dominated sorting time (ratios) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| M | N = 50 | 125 | 250 | 500 | 1000 | 2000 | 4000 | 8000 |
| 8 | **1.30** | **2.06** | **2.38** | **3.23** | **3.81** | **4.11** | **4.17** | **3.54** |
| 7 | **1.23** | **2.37** | **2.80** | **3.65** | **4.75** | **5.35** | **5.49** | **4.53** |
| 6 | **1.33** | **2.51** | **3.43** | **4.82** | **6.19** | **7.40** | **7.94** | **7.09** |
| 5 | **1.59** | **3.18** | **4.11** | **6.05** | **8.61** | **11.36** | **13.69** | **13.28** |
| 4 | **1.69** | **3.60** | **5.04** | **8.65** | **13.64** | **20.53** | **26.39** | **28.10** |
| 3 | **1.78** | **5.11** | **7.94** | **13.56** | **21.89** | **33.49** | **37.80** | **43.58** |
| | Total number of domination comparisons (ratios) | | | | | | | |
| 8 | **3.8** | **5.1** | **5.6** | **5.9** | **6.4** | **6.8** | **7.2** | **7.6** |
| 7 | **4.6** | **6.4** | **7.2** | **7.9** | **8.5** | **9.1** | **9.6** | **10.1** |
| 6 | **6.0** | **8.7** | **10.0** | **11.0** | **12.0** | **13.1** | **14.3** | **15.8** |
| 5 | **8.2** | **12.3** | **15.0** | **17.6** | **20.2** | **23.1** | **26.3** | **29.6** |
| 4 | **11.4** | **20.5** | **29.8** | **37.2** | **44.9** | **51.0** | **57.9** | **63.9** |
| 3 | **22.4** | **46.4** | **68.8** | **80.3** | **89.9** | **90.1** | **89.5** | **88.7** |

To quantify the importance of overnondomination, we also present the comparison of computational times for the last generation. By comparing these results with the total times in Table 14, we can see that the biggest difference is for 3 objectives. This is consistent with our previous analysis.

The fast non-dominated sorting algorithm performs *domination comparisons* between pairs of individuals. Our algorithm also performs *domination comparisons* when it compares the inserted individual to the individuals in the reference sets. We can count the number of these comparisons which are executed during the entire run of the optimizer. This way we get a measure which is *independent* from the underlying hardware and programming language, since all implementations should perform exactly the same steps.

Table 14: Ratio of averages $\frac{\text{Fast non-dominated sorting}}{\text{M-front}}$ for the WFG9 problem *in the last generation.*

| | Non-dominated sorting time (ratios) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| M | N = 50 | 125 | 250 | 500 | 1000 | 2000 | 4000 | 8000 |
| 8 | 0.97 | **1.99** | **2.34** | **3.26** | **3.92** | **4.29** | **4.25** | **3.52** |
| 7 | **1.43** | **2.70** | **2.66** | **3.64** | **4.77** | **5.43** | **5.58** | **4.54** |
| 6 | 1.03 | **2.66** | **3.38** | **4.89** | **6.22** | **7.52** | **8.15** | **7.11** |
| 5 | **1.37** | **2.80** | **4.00** | **5.78** | **8.57** | **11.40** | **13.92** | **13.66** |
| 4 | **2.11** | **3.81** | **4.94** | **8.26** | **13.58** | **22.22** | **29.45** | **32.35** |
| 3 | 1.38 | **5.64** | **7.80** | **15.22** | **24.80** | **42.93** | **55.14** | **75.79** |
| | Number of domination comparisons (ratios) | | | | | | | |
| 8 | **4.3** | **5.5** | **5.9** | **6.2** | **6.7** | **7.1** | **7.3** | **7.7** |
| 7 | **5.1** | **6.9** | **7.8** | **8.3** | **8.6** | **9.2** | **9.7** | **10.3** |
| 6 | **6.3** | **9.3** | **10.4** | **11.2** | **12.2** | **13.3** | **14.6** | **16.1** |
| 5 | **8.6** | **12.3** | **16.2** | **17.7** | **20.6** | **23.8** | **27.6** | **31.5** |
| 4 | **11.8** | **20.3** | **32.3** | **40.2** | **50.6** | **62.2** | **74.7** | **89.9** |
| 3 | **25.0** | **50.7** | **88.3** | **125.2** | **178.9** | **240.8** | **341.8** | **469.6** |

We can see the comparison in terms of domination comparisons in the bottom part of Table 14. We can see that these results are strongly correlated to the results in terms of computational time. This favors the hypothesis that a major part of the computational time is consumed by domination comparisons.

It is also interesting to see that the results in terms of domination comparisons are more favorable for our algorithm than the results in terms of computational time. In other words our algorithm does not reach its full potential. For example with 3 objectives and 1000 individuals our algorithm requires **89.9** times *fewer* domination comparisons, but overall it is only **21.89** times faster. This is caused by all the additional data structures that our algorithm needs to maintain. This includes primarily the K-d tree, but also the sorted lists, the reference sets, and the hash-table. The results for the number of dominance comparisons can be improved using *exact* nearest neighbor computation within the M-front.

To see the importance of overnondomination in our algorithm we provide the ratios of domination comparisons needed in the last generation in the bottom right part of Table 14. Again the differences between the *total* numbers of domination comparisons and domination comparisons in the *last* generation are most pronounced for 3 objectives, where the overnondomination is weakest.

### 6.6.4 *Confirmation of theoretical results*

In the chapter on computational complexity, we tried to model the population in an M-front using random vectors and then use the techniques from the theory of probability to model the computational cost of operations on the M-front. During this modeling, we have made a number of assumptions on the distribution of the individuals in the population. These assumptions are relatively simple in order to make the proofs of our theorems simple. Now we shall confront this model to the experimental data.

We have shown that an average insertion into the M-front needs $O(MN^{1-\frac{1}{M-1}})$ domination comparisons where $N$ is the number of individuals in the M-front. Assuming overnondomination, there are roughly as many individuals in the M-front as is the initial population size. We can try to substitute the initial population size in our experimental data for $N$.

The number of insertions in one generation is exactly the same as the initial population size. Therefore the computational cost is roughly $O(MN^{2-\frac{1}{M-1}})$ domination comparisons, where $N$ is the initial population size.

For a fixed $M$ this gives the power function:

$$f(N) = \alpha N^{\beta}. \tag{17}$$

We tried to fit the general curve given by (17) to our experimental results on the WFG9 problem using the least squares method. In almost all cases the approximation was appropriate. First let us look at the data for the *number of domination comparisons in the last generation* which is presented in Table 16.

We can see that the estimated $\beta$ coefficients are not very far from their theoretical counterparts. There is a slight tendency for the experimental coefficients to be higher. On the other hand the (less important) $\alpha$ coefficients seem to be more or less constant with respect to $M$. We believe that this is due to the fact that we used the *maximum* metric in our theoretical computation, but in experiments we used the more strict *Manhattan* metric.

Next we present the comparison in terms of actual wall clock time. The comparison of fitted and theoretical coefficients for the *elapsed time in the last generation* is in Table 17.

Here we see that the $\beta$ coefficients are slightly underestimated for *all* values of $M$. Nevertheless, the theoretical and estimated coeffi-

Table 15: Average absolute time taken by Jensen-Fortin's algorithm for the most complex and least complex problem setup (milliseconds).

| | Total | |
|---|---|---|
| | $M = 3, N = 50$ | $M = 8, N = 8000$ |
| WFG9 | 52.7061 (0.818) | 615 008 (11619.6) |
| DTLZ1 | 26.5054 (0.383) | 259 788 (6472.1) |
| | In the last generation | |
| WFG9 | 0.1024 (0.0045) | 1279.94 (36.14) |
| DTLZ1 | 0.0522 (0.0039) | 581.20 (18.98) |

Table 16: Fitting of curve $f(N) = \alpha N^{\beta}$ to experimental data: Number of domination comparisons in the last generation (WFG9).

| M | β theoretical | β estimate | α estimate |
|---|---|---|---|
| 8 | 1.857 | 1.902 | 1.170 |
| 7 | 1.833 | 1.890 | 0.978 |
| 6 | 1.800 | 1.848 | 0.923 |
| 5 | 1.750 | 1.768 | 0.958 |
| 4 | 1.667 | 1.625 | 1.122 |
| 3 | 1.500 | 1.442 | 1.095 |

Table 17: Fitting of curve $f(N) = \alpha N^{\beta}$ to experimental data: Time elapsed in the last generation (microseconds) (WFG9).

| M | β theoretical | β estimate | α estimate |
|---|---|---|---|
| 8 | 1.857 | 1.818 | 0.111 |
| 7 | 1.833 | 1.832 | 0.078 |
| 6 | 1.800 | 1.704 | 0.145 |
| 5 | 1.750 | 1.607 | 0.196 |
| 4 | 1.667 | 1.503 | 0.226 |
| 3 | 1.500 | 1.304 | 0.500 |

cients follow a similar pattern as we increase the number of objectives.

## 6.7 CONCLUSION

We have presented a new method to decrease the cost of non-dominated sorting. The main idea is to use a special data structure which holds and *updates* the knowledge of non-dominated individuals during the run of a MOEA. We have provided one such data structure which we call the M-front.

Jensen-Fortin's algorithm is the latest, and in terms of computational complexity, fastest algorithm. We have demonstrated that although the M-front does not provide an improvement in terms of average computational complexity, it is faster than Jensen-Fortin's algorithm on some problems for a broad range of population sizes and numbers of objectives. For very big populations the asymptotic superiority of Jensen-Fortin's algorithm eventually prevails but our algorithm can perform faster up to a fairly big population size (approximately 4000 individuals). Besides that, our algorithm scales very well with the number of objectives. This is because our algorithm uses the fact that a large proportion of the population tends to be non-dominated in such instances to its advantage.

All performance results aside, an important advantage of our method is that the non-dominated individuals are *known at all times*. If one individual changes, the change in the non-dominated front is immediately recorded. The core of our algorithm is the M-front which can be reused as a cost-effective data structure for MOEAs which archive their non-dominated individuals.

In future we should explore different implementations of the fast archive. An implementation using *segment trees* [10] seems to be a promising candidate. Also, replacing the K-d tree in the M-front with a more sophisticated nearest neighbor data structure may bring further improvement.

# 7 CONCLUSION AND DISCUSSION

This thesis had two goals. First, to improve our understanding of differential evolution in the multi-objective realm, and second to improve the computational efficiency of these algorithms.

In the first part of this thesis we explored the issue of *rotational invariance*. The degree to which a differential evolution algorithm is rotation invariant depends on the crossover probability parameter. While the only value for which DE is rotationally invariant is 1, many authors use values which are much smaller. It is therefore of great interest to see how the performance for such choices of parameters changes when the problem axes are rotated. In chapter 4 we studied the effects of rotation on bi-objective problems. We found out that the change in performance is significant even for small rotations. There is a consistent *drop* in performance on *separable* problems while the qualitative properties of the change for *non-separable* problems are much less predictable. Unexpectedly, for *multi-modal* problems, *low* values of crossover probability perform better through the observed spectrum of rotations. This is an interesting finding, but more research is needed to either confirm or refute this claim. We do not have an explanation for this phenomenon yet.

Next, we confronted the issue of *parameter control*. The relative sensitivity and difficulty to properly tune the DE parameters has motivated researchers to develop methods to automatically adjust these parameters during the optimization run. However, these methods were presented only as parts of unified multi-objective optimizers and therefore they were impossible to compare. In Chapter 5 we isolated the underlying parameter control mechanisms from various deterministic, adaptive, and self-adaptive algorithms and implanted them into a unified algorithm. We then tested this algorithm on a set of known benchmark problems as well as one new problem. We measured the performance of these methods in terms of hypervolume, as well as their behavior in measuring *which parameters* they found. We found out that on the usual benchmark problems even the simple mechanisms can lead to results comparable with parameter tuning. On the new problem, which we proposed exactly because it can be optimized only by a small set of parameters, the *self-adaptive* methods were the only ones that managed to find a satisfactory Pareto front for all objective dimensionalities. After examining the progress of the parameters used by the adaptive methods we found out that each algorithm evolves its parameters in a more or less problem independent way. This is a potential vulnerability of adaptive methods and it implies that their relative success may be accidental. Development of adaptive methods which are competitive with the self-adaptive ones is an interesting subject for future work.

In the second part of this thesis we presented a mechanism to reduce the computational cost of multi-objective DE. We concentrated on reducing the cost of the *non-dominated sorting, diversity estimation* and *archiving*. The main idea is to use a special data structure which holds and *updates* the knowledge of non-dominated individuals during the run of the algorithm. We have provided one such data structure which we call the M-front. This data-structure maintains the knowledge of non-dominated individuals up to date at all times. This reduces the cost of non-dominated sorting, since the first non-dominated front does not need to be determined. Its application to archiving is straightforward and moreover its internal structure allows several diversity estimation procedures to be performed more efficiently. The M-front can be applied to most multi-objective population based algorithms, but the particular workings of DE can be exploited to achieve even greater speedups. In future we should explore different implementations of the fast archive. An implementation using *segment trees* [10] seems to be a promising candidate. Also, replacing the K-d tree in the M-front with a more sophisticated nearest neighbor data structure may bring further improvement.

We conclude that even a restricted field, such as multi-objective differential evolution opens many questions and many directions for innovation.

Part IV

# APPENDICES

Here we prove theorems from Chapter 6.

# A

# APPENDIX TEST

## A.1 PROOF OF THEOREM 2

*Proof.* Suppose that we have two individuals $a, b \in RF_{Pf}$, whose objective values are:

$$Y_a = (y_{a,1}, y_{a,2}, \dots, y_{a,M-1}, Pf(y_{a,1}, y_{a,2}, \dots, y_{a,M-1}))$$
$$Y_b = (y_{b,1}, y_{b,2}, \dots, y_{b,M-1}, Pf(y_{b,1}, y_{b,2}, \dots, y_{b,M-1}))$$

For $a$ to dominate $b$ it is necessary that $y_{a,i} \leqslant y_{b,i}$ for all $i$ and $y_{a,j} < y_{b,j}$ for at least one $j$, for $i,\ j \in [\![1; M]\!]$. However, if there is such a $j < M$ we have

$$y_{a,M} = Pf(y_{a,1}, \dots, y_{a,M-1}) > Pf(y_{b,1}, \dots, y_{b,M-1}) = y_{b,M}$$

and if there is not a such $j < M$, meaning $y_{a,i} = y_{b,i}$ for all $i < M$, it implies

$$y_{a,M} = Pf(y_{a,1}, \dots, y_{a,M-1}) = y_{b,M}.$$

Hence $a$ cannot dominate $b$. This ends the proof. □

## A.2 PROOF OF THEOREM 3

We shall need the following three lemmas in our proof.

**Lemma 1** (First order statistic)**.** *Let $x_1, \dots, x_n$ be i.i.d. random variables and $F$ be the cumulative distribution function (cdf) of these variables. Then the cdf of the random variable*

$$x_{\min} := \min(x_1, \dots, x_n) \tag{18}$$

*is given by:*

$$F_{\min}(x) = 1 - (1 - F(x))^n \quad . \tag{19}$$

The $x_{\min}$ from (18) is called a *first order statistic*. More information on order statistics can be found in[9].

**Lemma 2** (Expected distance of closest point)**.** *Let $X_1, \dots, X_N$ be i.i.d. random* vectors *with uniform distribution on $[0; 1]^M$. Let $X_{\min}$ denote the random vector which is closest to the center $c = (0.5, 0.5, \dots, 0.5)$ of the hyper-box $[0; 1]^M$ with respect to the* maximum *metric $d$. Then, the expected distance of $X_{\min}$ from the center $c$,*

$$D_{M,N} := E[d(X_{\min}, c)] \quad , \tag{20}$$

*can be expressed in a closed form as:*

$$D_{M,N} = \frac{N}{2} B(N, 1 + \frac{1}{M}) \ , \tag{21}$$

*where* $B(x, y) := \int_0^1 t^{x-1}(1-t)^{y-1}\,dt$ *is the beta function.*

*Proof.* Let us first construct the cumulative distribution function (cdf) $F_1$ for the distance $d(X_i, c)$ of an arbitrary $X_i$ from $c$. By definition, the value of a cdf in $x \in \mathbb{R}$ is the probability that the random variable is smaller than $x$. The set of all vectors whose distance to $c$ is less than $x$ with respect to the maximum measure forms a cube with edge length of $2x$. Then, since $X_i \sim U[0;1]^M$, the probability of $X_i$ falling into such a cube is equal to the volume of the cube, namely $(2x)^M$. That is, $F_1(x) = (2x)^M$ for $x \in [0; 0.5]$.

Then, since the distance of the closest individual to $c$ is in fact the minimum of the distances, we have from (19) that the cdf of $d(X_{\min}, c)$ is

$$F_N(x) = 1 - (1 - (2x)^M)^N \quad \text{for any } x \in [0; 0.5].$$

Since $d(X_{\min}, c)$ is nonnegative, its expectation is computed by

$$\begin{aligned} D_{M,N} &= \int_0^{1/2} (1 - F_N(x))dx = \int_0^{1/2} (1 - (2x)^M)^N dx \\ &= \frac{1}{2M} \int_0^1 (1-t)^N t^{\frac{1}{M}-1} dt = \frac{1}{2M} B(N+1, \frac{1}{M}) \\ &= \frac{N}{2} B(N, \frac{1}{M} + 1) \ . \end{aligned}$$

This completes the proof. □

The asymptotic properties of this expected distance are summarized in the following theorem:

**Lemma 3** (Asymptotic properties of $D_{M,N}$)**.** *Let* $D_{M,N}$ *be the expected distance from* (20)*. Then for any* $M \geqslant 1$*:*

$$\lim_{N \to \infty} N^{\frac{1}{M}} D_{M,N} = \frac{1}{2} \Gamma(1 + \frac{1}{M}) \ . \tag{22}$$

*Here* $\Gamma$ *is the Gamma function.*

*Proof.* Note that

$$B(N, \frac{1}{M} + 1) = \frac{\Gamma(N)\Gamma(\frac{1}{M} + 1)}{\Gamma(N + \frac{1}{M} + 1)} \ .$$

Using the asymptotic property of the Gamma function, derived from Stirling's formula:

$$\lim_{n \to \infty} \frac{\Gamma(n + x)}{n^x \Gamma(n)} = 1, \quad \text{for any } x \in \mathbb{R},$$

we have

$$\begin{aligned} N^{\frac{1}{M}} D_{M,N} &= \frac{N^{\frac{1}{M}+1}}{2} B(N, \frac{1}{M}+1) \\ &= \frac{N^{\frac{1}{M}+1}\Gamma(N)}{\Gamma(N+\frac{1}{M}+1)} \frac{\Gamma(\frac{1}{M}+1)}{2} \\ &\xrightarrow{N\to\infty} \frac{\Gamma(\frac{1}{M}+1)}{2} \quad . \end{aligned}$$

This ends the proof. □

Now we are ready to prove Theorem 3.

*Proof of Theorem 3.* A random individual $a \in RF_{Pf}$ belongs to a reference set induced by `ref` and `new` iff the following condition is satisfied for some $i \in \{1, 2, \ldots, M\}$:

$$y_{a,i} \in [y_{\mathsf{new},i}; y_{\mathsf{ref},i}] \text{ or } y_{a,i} \in [y_{\mathsf{ref},i}; y_{\mathsf{new},i}]. \tag{23}$$

Define sets

$$\begin{aligned} A(x) &:= \{y \in [0;1]^{M-1} | \exists i \in [\![1; M-1]\!],\ y_i \in [0.5-x; 0.5+x]\} \\ B(x) &:= [Pf(0.5+x, \ldots, 0.5+x); Pf(0.5-x, \ldots, 0.5-x)] \end{aligned}$$

and a random variable $\delta = d_{M-1}(Y_{\mathsf{ref}}, Y_{\mathsf{new}})$. Since $Pf$ is strictly decreasing w.r.t. each element and $Y_{\mathsf{new}} = (0.5, \ldots, 0.5, Pf(0.5, \ldots, 0.5))$, the reference area induced by $Y_{\mathsf{ref}}$ is a subset of $A(\delta) \times B(\delta)$. Therefore, letting $\check{C}_{M,N}$ denote the number of individuals whose first $M-1$ components exist in $A(\delta)$ and $\hat{C}_{M,N}$ denote the number of individuals whose last component exists in $B(\delta)$, we have $C_{M,N} \leqslant \check{C}_{M,N} + \hat{C}_{M,N}$. Since the inequality inherits when the expectation is taken, we find

$$E[C_{M,N}] \leqslant E[\check{C}_{M,N}] + E[\hat{C}_{M,N}] \quad . \tag{24}$$

Hence, it suffices to show the right-hand side is bounded by a desired order.

First, we consider $E[\check{C}_{M,N}]$. Let $\mathbb{I}\{X\}$ be the indicator which is 1 if the event $X$ happens and 0 otherwise. For the simplicity of notation, we let $Z_a$ be the first $M-1$ components of $a \in RF_{Pf}$. Then,

$$\begin{aligned} E[\check{C}_{M,N}] &= E\big[\textstyle\sum_{i=1}^{N} \mathbb{I}\{Z_i \in A(\delta)\}\big] \\ &= \textstyle\sum_{i=1}^{N} E\left[\mathbb{I}\{Z_i \in A(\delta)\}\right] \quad . \end{aligned}$$

Remember that

$$\begin{aligned} \mathtt{ref} &= \operatorname*{argmin}_{i \in [\![1;N]\!]} d_{M-1}(Z_i, Z_{\mathsf{new}}) \\ \delta &= d_{M-1}(Z_{\mathsf{ref}}, Z_{\mathsf{new}}) = \min_{i \in [\![1;N]\!]} d_{M-1}(Z_i, Z_{\mathsf{new}}). \end{aligned}$$

The inside of the summation is then

$$\begin{aligned}
&\mathrm{E}[\mathbb{I}\{Z_i \in A(\delta)\}] \\
&= \mathrm{E}[\mathbb{I}\{i = \mathtt{ref}\} + \mathbb{I}\{i \neq \mathtt{ref}\}\mathbb{I}\{Z_i \in A(\delta)\}] \\
&= 1/N + \mathrm{E}[\mathbb{I}\{i \neq \mathtt{ref}\}\mathbb{I}\{Z_i \in A(\delta)\}] \\
&= 1/N + \mathrm{E}[\mathbb{I}\{i \neq \mathtt{ref}\}\mathbb{I}\{Z_i \in A(\min_{i \in [\![1;N]\!]} d_{M-1}(Z_i, Z_{\mathrm{new}}))\}] \\
&= 1/N + \mathrm{E}[\mathbb{I}\{i \neq \mathtt{ref}\}\mathbb{I}\{Z_i \in A(\min_{j \in [\![1;N]\!] \setminus \{i\}} d_{M-1}(Z_j, Z_{\mathrm{new}}))\}] \\
&\leqslant 1/N + \Pr[Z_i \in A(\min_{j \in [\![1;N]\!] \setminus \{i\}} d_{M-1}(Z_j, Z_{\mathrm{new}}))] \quad . \qquad (25)
\end{aligned}$$

For the last inequality we used $\mathrm{E}[\mathbb{I}\{X\}\mathbb{I}\{Y\}] \leqslant \mathrm{E}[\mathbb{I}\{X\}]\mathrm{E}[\mathbb{I}\{Y\}]$ and $\mathrm{E}[\mathbb{I}\{X\}] = \Pr[X]$. Note that on the right-most side $Z_i$ is independent of

$$\min_{j \in [\![1;N]\!] \setminus \{i\}} d_{M-1}(Z_j, Z_{\mathrm{new}}).$$

Since $Z_i$ is uniformly distributed in $[0;1]^{M-1}$, given

$$\delta_i = \min_{j \in [\![1;N]\!] \setminus \{i\}} d_{M-1}(Y_j, Y_{\mathrm{new}})$$

the above probability is the proportion of the volume of $A(\delta_i)$ to $[0;1]^{M-1}$, which is bounded above by $2(M-1)\delta_i$. Then,

$$\mathrm{E}[\check{C}_{M,N}] \leqslant 1 + 2(M-1) \textstyle\sum_{i=1}^{N} \mathrm{E}[\delta_i] \quad . \qquad (26)$$

Next, we consider $\mathrm{E}[\hat{C}_{M,N}]$. For the simplicity of notation, we let $W_a$ be the last component of $a \in RF_{Pf}$. Then, By definition

$$\begin{aligned}
\mathrm{E}[\hat{C}_{M,N}] &= \mathrm{E}\big[\textstyle\sum_{i=1}^{N} \mathbb{I}\{W_i \in B(\delta)\}\big] \\
&= \textstyle\sum_{i=1}^{N} \mathrm{E}\,[\mathbb{I}\{W_i \in B(\delta)\}] \quad .
\end{aligned}$$

Analogously to (25), we have

$$\mathrm{E}\,[\mathbb{I}\{W_i \in B(\delta)\}] = 1/N + \Pr[W_i \in B(\delta_i)] \quad .$$

Note that on the right-most side $W_i$ is independent of $\delta_i$. Given $\delta_i$, the above probability reads

$$\begin{aligned}
&\Pr[W_i \in B(\delta_i) \mid \delta_i] \\
&= \textstyle\int_{Pf(0.5+\delta_i,\ldots,0.5+\delta_i)}^{Pf(0.5-\delta_i,\ldots,0.5-\delta_i)} f_{Pf}(w)dw \\
&\leqslant \textstyle\int_{Pf(0.5+\delta_i,\ldots,0.5+\delta_i)}^{Pf(0.5-\delta_i,\ldots,0.5-\delta_i)} S dw \\
&= S|Pf(0.5-\delta_i,\ldots,0.5-\delta_i) - Pf(0.5+\delta_i,\ldots,0.5+\delta_i)| \\
&\leqslant 2SL\delta_i.
\end{aligned}$$

Taking the expectation over $\delta_i$ and plugging it into the above inequalities we have

$$\mathrm{E}[\hat{C}_{M,N}] \leqslant 1 + 2SL \textstyle\sum_{i=1}^{N} \mathrm{E}[\delta_i] \quad . \qquad (27)$$

With (24), (26) and (27) we have

$$E[C_{M,N}] \leqslant 2 + 2((M-1) + SL) \textstyle\sum_{i=1}^{N} E[\delta_i] \ . \tag{28}$$

Note that $E[\delta_i]$ is nothing but $D_{M-1,N-1}$ in Lemma 2. Hence, we find

$$E[C_{M,N}] \leqslant 2 + 2((M-1) + SL)ND_{M-1,N-1} \ . \tag{29}$$

For the limit of $N \to \infty$, using Theorem 3 we obtain

$$\begin{aligned} &\lim_{N\to\infty} \frac{E[C_{M,N}]}{N^{1-\frac{1}{M-1}}} \\ &\leqslant \lim_{N\to\infty} \frac{2 + 2((M-1) + SL)ND_{M-1,N-1}}{N^{1-\frac{1}{M-1}}} \\ &= ((M-1) + SL)\Gamma(1 + \frac{1}{M-1}) \in O(M) \ . \end{aligned}$$

Therefore, we find $E[C_{M,N}] \in O(MN^{1-\frac{1}{M-1}})$. □

# BIBLIOGRAPHY

[1] H. A. Abbass. The self-adaptive Pareto differential evolution algorithm. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, volume 1, pages 831–836, May 2002.

[2] H. A. Abbass, R. Sarker, and C. Newton. PDE: a Pareto-frontier differential evolution approach for multi-objective optimization problems. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 971–978, 2001.

[3] Hernán E. Aguirre and Kiyoshi Tanaka. Working Principles, Behavior, and Performance of MOEAs on MNK-landscapes . *European Journal of Operational Research* , 181(3):1670–1690, 2007.

[4] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[5] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52, 2002.

[6] J. Brest, S. Greiner, B. Boskovic, M. Mernik, and V. Zumer. Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. *Evolutionary Computation, IEEE Transactions on*, 10(6):646–657, Dec 2006.

[7] Janez Brest, Borko Bošković, Sašo Greiner, Viljem Žumer, and Mirjam Sepesy Maučec. Performance comparison of self-adaptive and adaptive differential evolution algorithms. *Soft Computing*, 11(7):617–629, feb 2007.

[8] S. Das and P. N. Suganthan. Differential Evolution: A Survey of the State-of-the-Art. *Evolutionary Computation, IEEE Transactions on*, 15(1):4–31, Feb 2011.

[9] H. A. David and H. N. Nagaraja. *Basic Distribution Theory*, pages 9–32. John Wiley & Sons, Inc., 2005.

[10] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Second edition, 2000.

[11] Kenneth A. De Jong. Genetic Algorithms are NOT Function Optimizers. In *FOGA*, pages 5–17, 1992.

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Trans. Evol. Comp*, 6(2):182–197, apr 2002.

[13] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. Scalable Test Problems for Evolutionary Multiobjective Optimization. In *Evolutionary Multiobjective Optimization*, chapter Advanced Information and Knowledge Processing, pages 105–145. Springer London, 2005.

[14] Roman Denysiuk, Lino Costa, and Isabel Espírito Santo. Many-objective Optimization Using Differential Evolution with Variable-wise Mutation Restriction. In *Proceedings of the Conference on Genetic and Evolutionary Computation*, pages 591–598, New York, NY, 2013. ACM.

[15] Martin Drozdik. Data structures. https://bitbucket.org/martin_drozdik/datastructures, Oct 2014. source code.

[16] Martin Drozdik, Hernan Aguirre, Youhei Akimoto, and Kiyoshi Tanaka. Comparison of Parameter Control Mechanisms in Multi-objective Differential Evolution. January 2015. LION9 conference.

[17] Martin Drozdik, Hernan Aguirre, and Kiyoshi Tanaka. Attempt to Reduce the Computational Complexity in Multi-objective Dierential Evolution Algorithms. Symposium on Evolutionary Computation, Karuizawa, Japan, Dec 2012.

[18] Martin Drozdik, Hernan Aguirre, and Kiyoshi Tanaka. Attempt to Reduce the Computational Complexity in Multi-objective Differential Evolution Algorithms. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pages 599–606, New York, NY, USA, 2013. ACM.

[19] Martin Drozdik, Youhei Akimoto, Hernan Aguirre, and Kiyoshi Tanaka. Computational Cost Reduction of Non-dominated Sorting Using M-front. *Evolutionary Computation, IEEE Transactions on*, PP(99):1–1, 2014.

[20] Martin Drozdik, Kiyoshi Tanaka, and Hernan Aguirre. An Archive for Mutually Nondominated Individuals with Nearest Neighbor Based Updating. In *2013 Convention Record, The IEEE Shin-Etsu Session*, Niigata, Japan, 2013.

[21] Martin Drozdik, Kiyoshi Tanaka, Hernan Aguirre, Sebastien Verel, Arnaud Liefooghe, and Bilel Derbel. An Analysis of Differential Evolution Parameters on Rotated Bi-objective Optimization Functions. In *The Tenth International Conference on Simulated Evolution And Learning (SEAL 2014)*, volume 8886, chap-

ter Lecture Notes in Computer Science, pages 143–154. Springer, Dunedin, New Zealand, Dec 2014.

[22] Wlodzislaw Duch. What is Computational Intelligence and where is it going? In *Challenges for computational intelligence*, pages 1–13. Springer, 2007.

[23] Agoston Endre Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 3(2):124–141, 1999.

[24] Burak Eksioglu, Arif Volkan Vural, and Arnold Reisman. Survey: The Vehicle Routing Problem: A Taxonomic Review. *Comput. Ind. Eng.*, 57(4):1472–1483, nov 2009.

[25] Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley Publishing, 2nd edition, 2007.

[26] J. E. Fieldsend, R. M. Everson, and S. Singh. Using Unconstrained Elite Archives for Multiobjective Optimization. *Evolutionary Computation, IEEE Transactions on*, 7(3):305–323, June 2003.

[27] David B. Fogel. *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press, 1st edition, 1998.

[28] Lawrence J. Fogel. *Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[29] Félix-Antoine Fortin, Simon Grenier, and Marc Parizeau. Generalizing the Improved Run-time Complexity Algorithm for Non-dominated Sorting. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 615–622, New York, NY, USA, 2013. ACM.

[30] David Hadka and Patrick Reed. Borg: An Auto-adaptive Many-objective Evolutionary Computing Framework. *Evol. Comput.*, 21(2):231–259, 2013.

[31] Nikolaus Hansen, Raymond Ros, Nikolas Mauny, Marc Schoenauer, and Anne Auger. Impacts of Invariance in Search: When CMA-ES and PSO Face Ill-Conditioned and Non-Separable Problems. *Applied Soft Computing*, 11:5755–5769, 2011.

[32] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. second edition, 1992.

[33] V. L. Huang, S. Z. Zhao, R. Mallipeddi, and P. N. Suganthan. Multi-objective optimization using self-adaptive differential evolution algorithm. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 190–194, May 2009.

[34] S. Huband, P. Hingston, L. Barone, and L. While. A Review of Multiobjective Test Problems and a Scalable Test Problem Toolkit. *IEEE Transactions on Evolutionary Computation*, 10(5):477–506, 2006.

[35] Dario Izzo, Luís F. Simões, Marcus Märtens, Guido C. H. E. de Croon, Aurelie Heritier, and Chit Hong Yam. Search for a Grand Tour of the Jupiter Galilean Moons. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 1301–1308, New York, NY, USA, 2013. ACM.

[36] M.T. Jensen. Reducing the Run-Time Complexity of Multiobjective EAs: The NSGA-II and Other Algorithms. *IEEE Transactions on Evolutionary Computation*, 7(5):503, 2003.

[37] Saku Kukkonen and Kalyanmoy Deb. A Fast and Effective Method for Pruning of Non-dominated Solutions in Many-Objective Problems. In *Parallel Problem Solving from Nature - PPSN IX*, volume 4193, chapter Lecture Notes in Computer Science, pages 553–562. Springer Berlin Heidelberg, 2006.

[38] Saku Kukkonen and Jouni Lampinen. GDE3: The third evolution step of generalized differential evolution. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 1, pages 443–450, 2005.

[39] Saku Kukkonen and Jouni Lampinen. An Empirical Study of Control Parameters for The Third Version of Generalized Differential Evolution (GDE3). In *IEEE Congress on Evolutionary Computation*, pages 2002–2009, 2006.

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

[41] J. Liu and J. Lampinen. A Fuzzy Adaptive Differential Evolution Algorithm. *Soft Computing*, 9(6):448, 2005.

[42] Kent McClymont and Ed Keedwell. Deductive Sort and Climbing Sort: New Methods for Non-dominated Sorting. *Evol. Comput.*, 20(1):1–26, mar 2012.

[43] Efrén Mezura-Montes, Margarita Reyes-Sierra, and CarlosA. Coello Coello. Multi-objective Optimization Using Differential Evolution: A Survey of the State-of-the-Art. In UdayK. Chakraborty, editor, *Advances in Differential Evolution*, volume 143, chapter Studies in Computational Intelligence, pages 173–196. Springer Berlin Heidelberg, 2008.

[44] Rodrigo César Pedrosa Silva, Rodolfo Ayala Lopes, and Frederico Gadelha Guimarães. Self-adaptive Mutation in the Differ-

ential Evolution. In *GECCO*, GECCO '11, pages 1939–1946, New York, NY, USA, 2011. ACM.

[45] Kenneth Price, Rainer M. Storn, and Jouni A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Springer, New York, NY, 2005.

[46] A. K. Qin, V. L. Huang, and P. N. Suganthan. Differential Evolution Algorithm With Strategy Adaptation for Global Numerical Optimization. *Evolutionary Computation, IEEE Transactions on*, 13(2):398–417, April 2009.

[47] Tea Robič and Bogdan Filipič. DEMO: Differential Evolution for Multiobjective Optimization. In *Evolutionary Multi-Criterion Optimization (EMO 2005)*, volume 3410, pages 520–533. Springer Berlin Heidelberg, 2005.

[48] Kukkonen Saku. *Generalized Differential Evolution for Global Multi-Objective Optimization with Constraints*. PhD thesis, Lappeenranta Univ. of Technology, 2012.

[49] Ralf Salomon. Re-evaluating Genetic Algorithm Performance Under Coordinate Rotation of Benchmark Functions. *Biosystems* , 39(3):263–278, 1996.

[50] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[51] Hans-Paul Paul Schwefel. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[52] Oliver Schütze. A New Data Structure for the Nondominance Problem in Multi-objective Optimization. In *Proceedings of the 2Nd International Conference on Evolutionary Multi-criterion Optimization*, EMO'03, pages 509–518, Berlin, Heidelberg, 2003. Springer-Verlag.

[53] R. Storn. On the usage of differential evolution for function optimization. In *Fuzzy Information Processing Society, 1996. NAFIPS., 1996 Biennial Conference of the North American*, pages 519–523, Jun 1996.

[54] R. Storn and K. Price. Minimizing the real functions of the ICEC'96 contest by differential evolution. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 842–844, May 1996.

[55] Rainer Storn and Kenneth Price. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[56] Kiyoharu Tagawa, Hidehito Shimizu, and Hiroyuki Nakamura. Indicator-based Differential Evolution Using Exclusive Hypervolume Approximation and Parallelization for Multi-core Processors. In *GECCO*, GECCO '11, pages 657–664, New York, NY, USA, 2011. ACM.

[57] Jason Teo. Exploring dynamic self-adaptive populations in differential evolution. *Soft Computing*, 10(8):673–686, 2006.

[58] Markus Wagner and Frank Neumann. A Fast Approximation-guided Evolutionary Multi-objective Algorithm. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 687–694, New York, NY, USA, 2013. ACM.

[59] L. While, L. Bradstreet, and L. Barone. A Fast Way of Calculating Exact Hypervolumes. *Evolutionary Computation, IEEE Transactions on*, 16(1):86–95, Feb 2012.

[60] D. Zaharie. Control of Population Diversity and Adaptation in Differential Evolution Algorithms. In R. Matousek and P. Osmera, editors, *Proc. of Mendel 2003, 9th International Conference on Soft Computing*, pages 41–46, Brno, Czech Republic, jun 2003.

[61] Daniela Zaharie. Critical Values for the Control Parameters of Differential Evolution Algorithm. In *Proceedings of MENDEL 2002*, 2002.

[62] A. Zamuda, J. Brest, B. Boskovic, and V. Zumer. Differential evolution for multiobjective optimization with self adaptation. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 3617–3624, Sept 2007.

[63] Jingqiao Zhang and A. C. Sanderson. JADE: Adaptive Differential Evolution With Optional External Archive. *Evolutionary Computation, IEEE Transactions on*, 13(5):945–958, Oct 2009.

[64] Mingming Zhang, Shuguang Zhao, and Xu Wang. Multi-objective evolutionary algorithm based on adaptive discrete Differential Evolution. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 614–621, May 2009.

[65] Eckart Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Comput. Eng. Netw. Lab. Swiss Federal Instit. Technol. (ETH), Zurich, Switzerland, 1999.