Departamento de Matemática Aplicada I

MODELO DE COMPUTACIÓN EVOLUTIVO PARA REDES SOSTENIBLES, EFICIENTES Y RESISTENTES

Pedro Miguel Mendes Guerreiro

Mayo 2017

UNIVERSITY OF SEVILLE

Department of Applied Mathematics I

EVOLUTIONARY COMPUTATIONAL MODEL FOR SUSTAINABLE, EFFICIENT AND RESILIENT NETWORKS

A dissertation presented

by

Pedro Miguel Mendes Guerreiro

in partial fulfilment of the requirements for the degree of

Doctor by the University of Seville

Doctor Alberto Márquez Pérez
(Ph.D. in Mathematics,
Professor in the Department of
Applied Mathematics I from
University of Seville)

Doctor Mário Carlos Machado Jesus
(Ph.D. in Computer Science,
Professor in the Department of
Civil Engineering from
University of Algarve)

Seville, May 2017

Abstract

We present a new approach to adapt the differential evolution (DE) algorithm so that it can be applied in combinatorial optimization problems.

The differential evolution algorithm has been proposed as an optimization algorithm for the continuous domain, using real numbers to encode the solutions, and its main operator, the mutation, uses a arithmetic operations to create a mutant using three different random solutions.

This mutation operator cannot be used in combinatorial optimization problems, which have a domain of a discrete and finite set of objects. Based on this concept, we present an idea of representing each solution as a set, and replace the arithmetic operators in the classic DE genetic operators by set operators. Using a well known *NP-hard* problem, the traveling salesman problem (TSP), as an example of a combinatorial optimization problem, we study different possibilities for the mutation operator, presenting the advantages and disadvantages of each, before setting with the best one.

We also explain the modifications made to adapt the algorithm for a multi-objective optimization algorithm. Some of these modifications are inherent to the different type of problems, other modification are proposed to improve the algorithm. Amongst the later modification are using more than one population in the evolution process. We also present a new self-adaptive variation of the multi-objective optimization algorithm, although this is not limited to the multi-objective case, and can be used also in the single-objective.

Para a minha filha Inês.

Acknowledgements

Aos meus pais e à minha irmã, obrigado pelo amor, carinho e apoio que sempre me deram. À minha filha, um pedido de desculpas pelas ausências.

To all my friends, thank you for understanding and respecting my "madness".

To the Institute of Engineering, in particular to the Department of Civil Engineering and to the *Centro de Simulação e Cálculo* a special thanks for the resources made available.

A very special "Thank You" for my directors, Professor Mário Jesus and Professor Alberto Márquez, for their invaluable support, incentive, assistance, constructive critics, and most of all, for their friendship. I would not be writing this paragraph if it weren't for you both. Thank you!

Contents

Chapter 1

Introduction

Although computer science is an area of knowledge with a history that can be traced to the usage of the abacus, circa 2700-2300 BC, modern day computer science, using what we now consider "computers", emerged in the 20th century, after the second World War, with what is considered to be the first electronic general purpose computer, the ENIAC.

Since the ENIAC, we have come a long way, from using valves to build the computer, we now use integrated circuits. This led to what is commonly known as Moore's Law [61], that said that the number of transistors in an integrated circuit would double every two years. Although not being really a "law", but a mere observation, this proved true for more that forty years, leading to an exponential growth of the processing power of modern day computers, allowing a modern day cell phone to have more processing power than a twenty years old computer.

This evolution in the processing power of modern day computers allowed scientists from many different areas to get better and faster results, particularly those that need real computing power to get results, such as simulation, visualization or optimization, just to name a few.

In this thesis we will focus on optimization, which the Oxford dictionary defines as

"The action of making the best or most effective use of a situation or

resource."

This definition characterizes optimization as the procedures needed to use in a situation or resource to improve it, and this can be used in most, if not all, areas of knowledge, such as logistics, engineering, economics or medicine. However, we can only improved something if we start from an initial state, unfortunately, the inherent complexity of each area, in addition to the different problem parameters and/or restrictions, can make very difficult to reach a simple solution, let alone a good one.

Suppose you decide to grab a backpack, and travel through Europe without any type of plan. You just know all the cities you want to visit before returning home, and you don't want to pass through each city more than one. This is an example of a problem known as the traveling salesman problem (TSP). One way to go about it, is to write all cities you want to visit in a notebook, and then just visit the cities in the order you wrote them. Of course you would accomplish your goals, but very doubly, the solution found would be the best one, assuming the only global criteria to consider is to do this travel as fast as possible (allowing some time to visit the each city, of course).

Select the city you visit next using other than just "gut feeling", for instance always going to the closest city to the one you are in, would, most probably, result in a better solution, but some time and calculations would need to be made to decide which is the city closest to the current one. If you add some more restrictions, for instance, you must use only the train as the means of transport, maybe the initial solution is not even a valid solution, as it could happen that there isn't any direct train between two cities you have decided to visit in that order. Other than just a handful of cities to visit, finding a good solution to this type of problems by hand calculation is very difficult, if not impossibly, as this problem is known to not having any algorithm that allow for a solution to be found in feasible time [28, 6].

However, if the solution is the best or the most effective, is depended on the interpretation given to certain parameters, such as the cost needed to improve it,

and even if time or eventual computational restrictions could restrict from reaching the optimal, or even local, solution.

If, besides to the initial criteria, in this case to do the travel as fast as possible, you add some other criteria, such as to spend as little as possible and/or the total distance travelled to be the lowest as possible, you just "upgraded" from a single-objective optimization problem, to a multi-objective one, where each objective must be considered to evaluate the solution. The problem with this type of problems is that the objectives are, usually, contrary to one another, i.e., for the travel to be as fast as possible, probably more money is needed, this way contradicting the objective to spend as little as possible.

This contradictory objectives makes the solution of the optimization algorithm for this type of problems to not be unique, but rather be a set of trade-off solutions, meaning that to improve one objective, at least one of the others need to be worsened. From this set of solutions, the decision maker (the traveler, or you in this case) would then choose the one that looked the best, according to some personal preference, but knowing that from all solutions to choose from, none are globally worse than any other. These solution are called Pareto solutions and the set of Pareto solutions is called the Pareto set.

We can thus say that an optimization problem, needs an evaluation function that will determine the "value" of each solution. This evaluation function can be defined as

$$\mathbf{f} : \mathcal{S} \rightarrow \mathcal{Z}$$

where $\mathcal{S}$ is the feasible domain, and $\mathcal{Z}$ is the objective space. Usually, $\mathcal{Z} \subset \mathbb{R}^m$, where m are the number of objectives to be considered in the problem. When $m = 1$ we said to have a single-objective optimization problem, otherwise is a multi-objective optimization problem.

We can classify an optimization problem in two mains classes: continuous or

discrete. In the first case, $\mathcal{S}$ is continuous, in the second, $\mathcal{S}$ can only assume discrete values. In the latter case are the combinatorial optimization problems, which are the focus of this thesis.

A combinatorial problems is one where the domain of the problem is both a discrete and finite set of some kind of objects, that can be combined in different forms. In combinatorial optimization problems, the objective is to find the optimal combination to answer a certain problem. Basically, combinatorial optimization consist in finding the optimal solution, from a finite set of possible solutions. When explained like that, it looks simple, just find all of possible combinations and select the best. But as seen in our travel example, as the size of the set grows larger, the size of the number of possible combinations "explodes", and there is no efficient algorithm capable of determining the optimal solution from this enormous set of solutions. This type of problems belong to a class called *NP-hard* problems [28, 6], and many combinatorial optimization problems belongs to this class.

To circumvent the problem of finding the best possible solution, is common practice to use an approximation algorithm to determine a sufficient enough approximation to the best solution. Most of these approximation algorithms based on stochastic processes to find this approximate solution, and this non-deterministic processes are the reason for which no guarantee can be made there accurateness of the solution found.

A very known type of approximation algorithms are evolutionary algorithms (EA) which are search algorithms that use techniques that emulate the natural evolutionary processes, as proposed in the XIX century by Charles Darwin's "survival of the fittest" theory. Darwin said that, in nature, individuals compete with one another over scarce resources, and the better fitted individuals dominate the less fitted, and have an higher probability of passing their genes to their descendants, while the weaker individuals will, most likely, wither and fail. The basic idea in evolutionary algorithm is the same, i.e., the fittest individuals, according to some predefined measure, dominate the ones with a lower fitness, and while

the first survive to have offspring, the latter “die” without descendents to carry their genes to the next generation, accomplishing through this, the “survival of the fittest” theory.

John Holland is one of the pioneers of the evolutionary algorithms concept, and his genetic algorithms (GA) [38] are probably the most used evolutionary algorithm, with many variants proposed.

Genetic algorithms are population-based evolutionary algorithm, that, in a nutshell, are a global search technique that successively improves a solution (also know as an individual), using three main (genetic) operators: crossover, mutation and selection (or replacement). Every individual in the population “mate” with another one through the crossover operator, creating an offspring in the process, that inherits the genes from both parents. This offspring is afterwards subject to an eventual mutation, to introduce some random gene, that could result either in an improvement or in a complete failure. The final replacement operator allows for the best to survive to the next generation, this way propagating the best genes (characteristics), and discarding the less promising ones.

One variant of the initial genetic algorithm is know as differential evolution (DE), proposed by Storn and Price [82, 83]. Although sometimes said to be an genetic algorithm, because its genetic operators having the same name, its inner working makes it a different type of evolutionary algorithm. Differential evolution, although also being a population-based evolutionary algorithm that has as its genetic operators the mutation, the crossover and the selection (or replacement), has quite a different approach to these operators. While in genetic algorithms, the first, and main, operator is the crossover, in DE is task is up to the mutation operator, that basically, creates a mutant by adding the scaled difference between two individuals (or vectors, in DE world) to a third. Then this mutant, through the crossover operators, mates with a individual in the population, creating what in DE concept is called a trial individual. Basically, in differential evolution, this two operators create an offspring using a strange “four” fathers type of mating. Finally,

this trial individual (or offspring) is compared with is "father", and if proves to be better, replaces it for the next generation.

Although the crossover and the replacement operator are similar to the genetic algorithm counterpart, the mutation operator is a strange operator, as is it quite different from any other mutation in the evolutionary algorithm realm, but is also the main strength in the differential algorithm, and also the reason for the "differential" in the algorithm name.

But another conceptual difference between genetic algorithms and differential evolution is the fact that when the genetic algorithms were introduced, they worked with bits representing each gene in the individuals, while the differential evolution were developed to use real numbers. What this means is that differential evolution were developer to be used in continuous optimization problems, not discrete ones, in fact, DE cannot be used in discrete optimization problems without some adjustments.

If the differential evolution algorithm would not have proved itself to be a very simple, efficient and robust algorithm, it would not been subject to the trouble of using it in combinatorial optimization problems. As this was not the case, since very earlier in DE life, some approaches have been introduced to allow its the usage in this type of problems. However, most approaches have tried to cram the problem in the differential evolution domain, instead of "bringing" DE to the combinatorial optimization realm. The different between this is that if the problem is somehow crammed into DE, some (or all) of the problem characteristics are lost, and the algorithm probably will not take any advantage of them. If is done the other way around, this characteristic would not be lost, and the algorithm could use them to allow a better search of the solutions. The problem with this latter approach is that the differential evolution operators, particularly the mutation operator, uses arithmetic operators that cannot be easily translated to the discrete domain.

The focus of this thesis is precisely introducing a new technique to use in the

differential evolution operators, to allow its usage in combinatorial optimization problems, both for single-optimization problems and for multi-optimization ones.

This thesis is organized as follows: after an introduction to optimization problems and to the differential evolution algorithm made in this chapter, the next chapter will expand this introduction with basic knowledge to understand the next chapters. In chapter three we present our suggestions to allow the usage of differential evolution in single-objective combinatorial optimization problems, presenting and comparing them to other approaches. We also made a study of DE's parameters for this type of problems, before concluding with our final algorithm, and suggested parameters. Chapter four will further enhance our idea for the multi-objective case, explaining the needed adaptations, for this type of problems. We also introduce the idea of using more than one population for the evolutionary process, exploring different variations of this, with the pros and cons. A self-adaptive version of the algorithm is also introduced here, and the results are compared with and without this self-adaptive algorithm to see if it is worth it. Chapter five will finish this thesis with then conclusions about the work developed, and possible aspects for future research.

Chapter 2

Preliminaries

This chapter will contain some fundamental concepts used throughout the rest of this thesis. It will start by explaining the Differential Evolution algorithm, with its variations and operators, followed by the definition of what is a multi-objective optimization problems and how can they be solved using evolutionary algorithms. In the end, a presentation of combinatorial optimization problems, and the example of the traveling salesman problem.

2.1 Differential evolution

Evolutionary Algorithms (EA) are search algorithms, based on ideas from nature and genetics, using techniques designed to simulate the natural evolutionary processes, as proposed in the "survival of the fittest" theory, by Charles Darwin in the XIX century. In it Darwin say that in nature, individuals compete with one another over scarce resources, and the best individuals, according to some core characteristic, dominate the weaker ones, and as such have an higher probability of passing their genes to their descendants, while the weaker individuals will, most likely, wither and fail. The basic idea in EA is the same, i.e., the fittest individuals, according to some predefined measure, dominate the ones with a lower fitness, and while the first thrive and prevail through their descendants, the latter

are discarded, and through this, the "survival of the fittest" is accomplished.

Although there are evolutionary algorithms are that are not population-based, we will one consider the population-based evolutionary algorithms, i.e., the ones that work with multiple individuals (called a population), and evolve this population using biological inspired mechanisms, such as reproduction, mutation, selection, etc. Although the term individual is used frequently in EA, it's interchangeably with candidate solution, solution or even phenotype, this latter from the genetic domain. As in nature the individuals are composed of genes, this term is also used in EA, but so are chromosomes, elements or genotype. Throughout this thesis, this term will be used interchangeably.

Differential Evolution (DE) [82, 81, 83, 72] is an evolutionary algorithm introduced in 1995 by Storn and Price, as a global optimization algorithm for continuous spaces. It first appeared in a technical report, and it quickly proved itself as one of the most competitive evolutionary algorithms, not only due to its robustness and efficiency, but also because of its simplicity and easy to implement.

As most evolutionary algorithms, DE is a population based optimization algorithm, that starts with a population of candidate solutions created randomly in the domain of the problem, and evolve them until a stopping criteria is reached. Each candidate solution is a point in a domain defined by some preset bounded values, and to each is given a index from 0 to $Np - 1$, where Np is the number of individuals in the population. Np is one of the three control parameters of this algorithm, being the others the crossover rate (or probability) CR, and the scale factor F.

DE, like Genetic Algorithms (GA), evolve the initial population using a mutation and a crossover operators, before selecting which individuals will survive to the next generation, replacing the current generation population. Then this next generation population will undergo the same mutation and crossover, and so on and so forth, until the stopping criteria is reached. The basic algorithm for DE can be seen on algorithm 2.1.

Algorithm 2.1 Algorithm of Differential Evolution.

```
1: population ← initialization( )
2: while not end criteria do
3:     mutant ← mutate( population )
4:     trial ← crossover( mutant )
5:     population ← replace( population, trial )
6: end while
```

Although the EA terminology can be used to refer to individuals, genes, etc, in DE, as it was conceived to be used in a real, continuous space, is common to refer to an individual as a vector, or refer to a chromosome as a parameter, or element.

The Differential Evolution algorithm, although similar to Genetic Algorithms, have some differences, as it doesn't use a binary representation for the chromosomes, as some simple GA do, and, more important, the core difference is in DE's operators, particularly the mutation operator, being this also the reason for its name. In GA, the mutation can be a simple "flip" operator, where a single gene can be flipped from one value to another, according to some mutation probability, and the "main" part of the evolution is done by the crossover, but in DE, the mutation has a crucial part in the evolution, so much that DE's authors used its main principle to name the algorithm: DE's mutation calculates the difference between two candidate solutions, and then adds this weighted difference to another candidate solution, this way exploring the search space of the problem. The "Differential" in the name of the algorithm comes from the "difference" calculated in the mutation operator. This mutated solution is then mixed with another candidate solution, creating this way a trial solution. If this trial vector, when compared with the corresponding target vector in the population, yields a better fitness, then it replaces the target vector for the next generation. Each DE process will be now explained in more detail:

2.1.1 Initialization

In DE, the initial population of d-dimensional vectors are chosen using an uniform random generator for each parameter of the vector. Each vector in the population is defined by

$$\mathbf{x}_{i,g} \ , \quad i = \{1, ..., Np\} \tag{2.1.1}$$

where Np is the number of vectors in the population, and g is the current generation, and $\mathbf{x}_{i,g} = \{x_{1,i,g}, x_{2,i,g}, ..., x_{d,i,g}\}$. Each element of the vector can be bounded by a lower and upper values, according to

$$x_j^{lower} < x_{j,i,g} < x_j^{upper} \ , \quad j \in 1, ..., d \tag{2.1.2}$$

defining this way the domain of the problem. The random uniform generator is used to cover as much as possible the search space of the problem, as the more the initial population is spread in the search domain, the easiest for the operators to cover it, and reach the desired optimum value.

2.1.2 Mutation

As referenced earlier, the main difference between DE and other GA is the mutation operator, as in DE it is based on a difference of vectors, rather than a random-based mutation as in other GA. Equation (2.1.3) shown the formula used to create a mutation vector $\mathbf{v}_{i,g}$ for generation g:

$$\mathbf{v}_{i,g} = \mathbf{x}_{r1,g} + F \cdot (\mathbf{x}_{r2,g} - \mathbf{x}_{r3,g}) \, . \tag{2.1.3}$$

For each vector i, we start by selecting three random vectors $r1, r2, r3 \in \{1, 2, ..., Np\}$, all different from each other and different from current vector ($r1 \neq r2 \neq r3 \neq i$). The scale factor $F \in [0, 2]$ is a constant parameter that controls the amplification of the difference that will be added to the base vec-

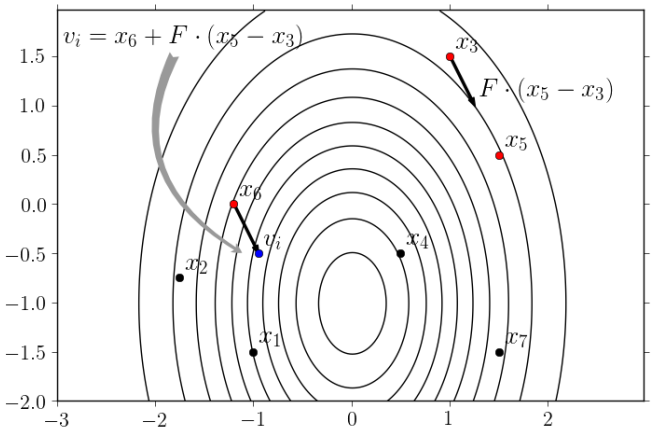


Figure 2.1: Mutation operator in Differential Evolution: three different vectors were selected randomly ($\mathbf{x}_6$, $\mathbf{x}_5$ and $\mathbf{x}_3$), and by applying equation (2.1.3), the mutant vector $\mathbf{v}_{i,g}$ is calculated.

tor $\mathbf{x}_{r1,g}$, controlling this way the influence of the difference in the new found mutant. In figure 2.1 we can see a representation of the process, for a two dimensional problem. Three different vectors were randomly selected in the population $\mathbf{x}_6 = [-1.2, 0.0]^T$, $\mathbf{x}_5 = [1.5, 0.5]^T$ and $\mathbf{x}_3 = [1.0, 1.5]^T$, respectively for $\mathbf{x}_{r1}$, $\mathbf{x}_{r2}$ and $\mathbf{x}_{r2}$ in equation (2.1.3), and assuming $F = 0.5$, the weighed difference between $\mathbf{x}_5$ and $\mathbf{x}_3$ would be

$$F \cdot (x_{r2} - x_{r3}) = 0.5 \cdot \left(\begin{bmatrix} 1.5 \\ 0.5 \end{bmatrix} - \begin{bmatrix} 1.0 \\ 1.5 \end{bmatrix} \right) = \begin{bmatrix} 0.25 \\ -0.50 \end{bmatrix},$$

and adding this to the base vector $\mathbf{x}_6$, would result in the mutant vector

$$\mathbf{v}_i = \begin{bmatrix} -1.2 \\ 0.0 \end{bmatrix} + \begin{bmatrix} 0.25 \\ -0.50 \end{bmatrix} = \begin{bmatrix} -0.95 \\ -0.50 \end{bmatrix}.$$

Several variations were formulated for DE, especially changing the mutant operator. These are normally identified as *DE/x/y/z*, where:

x defines the base vector to be mutated, usually *rand* for a random selected vector, or *best* to use the best vector in the current population;

y is the number of differences between vectors to be calculated;

z defines the crossover scheme, usually *bin* for a binary crossover, or *exp* to use an exponential crossover.

Using this nomenclature, equation (2.1.3) is commonly refereed as *DE/rand/1/bin*, or *DE/rand/1/exp*, depending on the crossover operator used. To avoid this "repetition" reference, is frequent in literature to found references to DE variations without an explicit crossover, for instance, use using only *DE/rand/1* to refer to the variation defined in equation (2.1.3). This is because DE variations have a close connection to the mutation operator, and not so much to the crossover. Also the different crossovers could be applied regardless of the mutation variation, so to avoid the "repetition" mentioned earlier, it is easier to refer to the variation without mentioning the crossover variation.

DE/rand/1 define the most common DE variant, as was first introduced by Storn and Price in 1995 [82], and, although over the years more variations where proposed, the most common ones, according to Das et al. [17], are still those proposed by the authors, in [81, 83, 41, 72]:

DE/rand/1

$$\mathbf{v}_{i,g} = \mathbf{x}_{r1,g} + F \cdot (\mathbf{x}_{r2,g} - \mathbf{x}_{r3,g}) \tag{2.1.4}$$

DE/best/1

$$\mathbf{v}_{i,g} = \mathbf{x}_{best,g} + F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}) \tag{2.1.5}$$

DE/rand/2

$$\mathbf{v}_{i,g} = \mathbf{x}_{r1,g} + F \cdot (\mathbf{x}_{r2,g} - \mathbf{x}_{r3,g}) + F \cdot (\mathbf{x}_{r4,g} - \mathbf{x}_{r5,g}) \tag{2.1.6}$$

DE/best/2

$$\mathbf{v}_{i,g} = \mathbf{x}_{best,g} + F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}) + F \cdot (\mathbf{x}_{r3,g} - \mathbf{x}_{r4,g}) \tag{2.1.7}$$

DE/current-to-best/1

$$\mathbf{v}_{i,g} = \mathbf{x}_{i,g} + F \cdot (\mathbf{x}_{best,g} - \mathbf{x}_{i,g}) + F \cdot (\mathbf{x}_{r1,g} - \mathbf{x}_{r2,g}) \tag{2.1.8}$$

In all of the above, $\mathbf{x}_{best,g}$ represent the best vector in the current generation, and $\mathbf{x}_{r4,g}$ and $\mathbf{x}_{r5,g}$ refer to two more random selected vectors, needed when using two differences between vectors, as in *DE/rand/2*, and *DE/best/2*.

2.1.3 Crossover

In GA, the crossover is where some genes of each parent are combined to form a new child. This child is exclusively composed by elements from either of the parents. In DE, the crossover can be either a binary crossover or a exponential crossover and each vector is crossed with the correspondent mutant vector, calculated previously. Although when Storn and Price introduced DE in 1995 [82] they used the exponential crossover, in an article from 1997 [83] they presented the binary crossover, and the latter is more commonly used in the literature.

Binary crossover

For each index vector i, the binary crossover is defined by

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & \text{if } \mathrm{rand}_j(0,1) \leq CR \text{ or } j = j_{rand} \\ x_{j,i,g} & \text{otherwise} \end{cases}, \quad j = 1, ..., d \tag{2.1.9}$$

where $\mathrm{rand}_j(0,1)$ is a uniform random function, resulting in a number between 0 and 1, $CR \in [0,1]$ is a constant parameter that defines the crossover rate. If the random generated value is lower than the crossover rate, the corresponding trial parameter comes from the mutant vector, otherwise is inherited from the current vector. j_{rand} is a random index, different for each vector, and ensures that the

resulting trial vector has, at least, one parameter from the mutant vector and, as such, is always different from the current vector.

Exponential crossover

The main difference between the binary crossover and the exponential one is that while in the binary crossover, the vectors are examined each parameter at a time, the exponential crossover work with blocks of parameters. For each vector index i, the trial vector $\mathbf{u}_{i,g}$ is defined by

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & j = \langle n \rangle_d, \langle n+1 \rangle_d, ..., \langle n+L-1 \rangle_d \\ x_{j,i,g} & \text{for all other } j \end{cases}, \ j = 1, ..., d \qquad (2.1.10)$$

where $\langle \cdot \rangle_d$ define the modulo function with modulus d and n is the initial index, chosen randomly. The number of parameters to be exchanged is given in L, and is selected from the interval $[1, d]$, according to algorithm 2.2. As can be seen, the value of L will be incremented until either there are no more parameters ($L \geq d$) or the random value calculated is greater than the crossover rate (CR). Both n and L are randomly calculated for each vector $\mathbf{u}_{i,g}$.

Algorithm 2.2 Algorithm to determine the length L of the block for the exponential crossover.

```
1: L ← 0
2: repeat
3:    L ← L + 1
4: until rand() < CR and L < d
```

2.1.4 Replacement

The replacement operator in DE is a simple greedy one, where each trial vector is compared with the corresponding target vector in the population, and the best one replaces the current generation vector, to the next generation, and is given by

$$\mathbf{x}_{i,g+1} = \begin{cases} \mathbf{u}_{i,g} & \text{if } f(\mathbf{u}_{i,g}) \leq f(\mathbf{x}_{i,g}) \\ \mathbf{x}_{i,g} & \text{otherwise} \end{cases} \tag{2.1.11}$$

where $f(\cdot)$ is the objective function to optimize. If the values are equal, the trial vector replaces the target one, and by doing this, some variation could be introduced in the population, although not changing the overall fitness value, because depending on the function to optimize, different vectors could return equal fitness values.

2.2 Multi-objective optimization

Multi-objective optimization (MO) problems, as the name suggests, are problems where more than one objective must be taken into account in the optimization process, and each objective can be either maximized or minimized, subject to a number of restrictions.

Suppose buying a new TV. TVs came in different sizes, with different prices and a miscellaneous of features, some more important than others, at least for most of us. For instance, nowadays, one might only consider a new TV set if it has the latest and greatest resolution, but to others, this might not be that much importance, and instead consider if the TV set can be connected to the Internet, or has wireless connection. With a plethora of features to consider, there are even those who consider the price when buying a TV set, go figure. This is clearly a multi-objective problem, where each characteristic of the TV is an objective in our problem, and the main goal is to decide which TV has the best set of characteristics, i.e., to optimize our problem. Of course there is no "one-size-fits-all" thing here, otherwise this would not be a problem, much less a multi-objective one, has everybody would buy that best "all-around" TV set. Instead, that plethora of characteristics must be taken into account, and each person must decide to which of them give more importance and which of them (if any) have no importance at all.

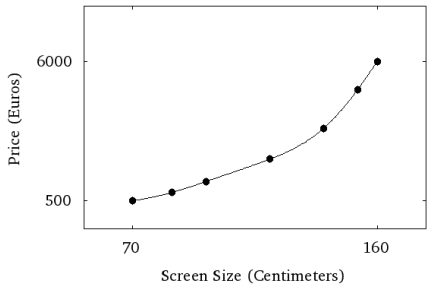


Figure 2.2: An hypothetical representation of the best values for price and size of TV sets.

What this means is that a multi-objective optimization problem, instead of a single objective one, does not give the best answer, but instead give the a set of solutions, each of them better than the others at one objective, but worse at others. It's up to the decision maker to look at those solutions and decide in the final (optimal) solution.

For simplicity, lets considering only the size and the price of the TVs, and imagine one wanting to buy the biggest TV possible, spending the least amount of money. Suppose that after some market analysis, a resume is made, shown in figure 2.2, with the best values for different sizes and prices. From those values can be seen that if considering only the price, one will end up with a smaller TV set, but on the other hand, if one goes for the largest TV, the price to pay would also be quite "large". For each person, the answer to the question "Which TV should I buy?" can be a different one, based on their experience, personal restrictions, and/or outside factors, and more often than not, is necessary to make some trade-offs in the decision process. In this example, as in most real life examples, there is no clear best answer, and some compromise is necessary to reach the decision.

2.2.1 Problem definition

Assuming, without loss of generalization, that all objectives are to be minimized (otherwise we could always use the duality principle [22]), the general form for a multi-objective optimization problem can be defined by

$$\text{minimize} \quad \mathbf{f}(\mathbf{x}) \tag{2.2.1}$$

subject to

$$g_i(\mathbf{x}) \leq 0 \ , \quad i = 1, ..., n \tag{2.2.2}$$

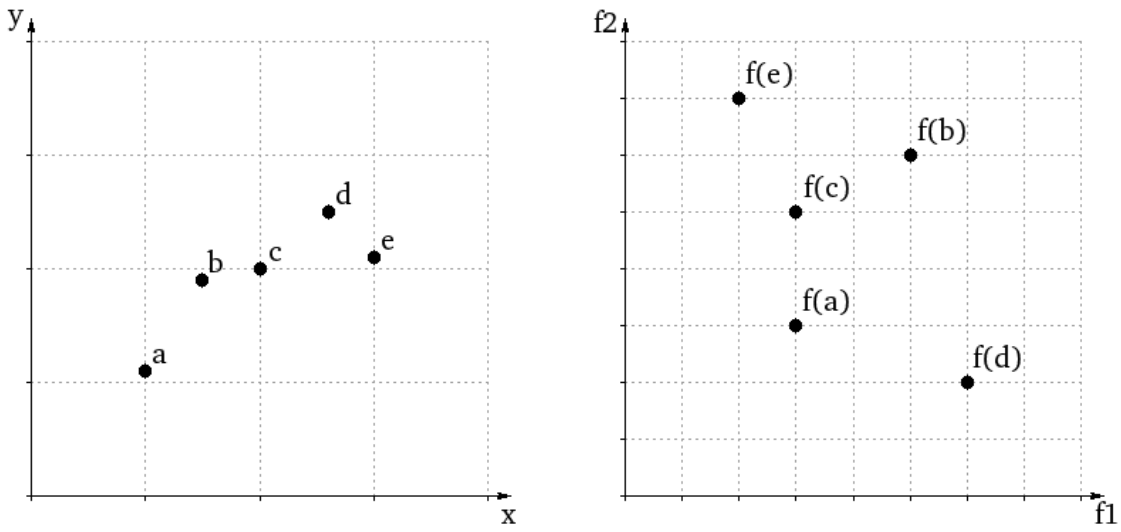
$$h_j(\mathbf{x}) = 0 \ , \quad j = 1, ..., p \tag{2.2.3}$$

$$x_k^L < x_k < x_k^U \ , \quad k = 1, ..., d \tag{2.2.4}$$

where $\mathbf{f} : \mathbb{R}^d \rightarrow \mathbb{R}^m$ is a multidimensional function, defined by $\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_m(\mathbf{x})]$, $\mathbf{x} = [x_1, x_2, ..., x_d]^T$ defines a vector of decision variables, each bounded between a lower x_k^L and an upper x_k^U value, and $g_i, \ h_j : \mathbb{R}^d \rightarrow \mathbb{R}$ are the constraint functions to be satisfied by the problem.

The solutions that satisfy the constraint function of the problem and the variable bounds constitute a *feasible decision variable space* $\mathcal{S} \subset \mathbb{R}^d$. Contrary to single-objective optimization problems, in multi-objective optimization problems there are multiple functions to optimize, and each of then return a value. As a result, each solution of a multi-objective problem is a point in a multi-dimensional *objective space*, $\mathcal{Z} \subset \mathbb{R}^m$. The decision space contains the solutions to the problem, and each of these solutions are mapped, through the optimization function, in the objective space, allowing some order of the solutions.

Suppose a two dimensional, two-objectives optimization problem, i.e., $\mathbf{f} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$. In figure 2.3 is an example of a possible set of solutions in the decision variable space (left), and the corresponding points in the objective space (right) for this type of problem. For instance, **d** defines a solution on the decision space, represented in the figure on the left. Evaluating this vector using the two objective functions (f_1 and f_2), each of them would result in a value, and a possible final



(a) (b)

Figure 2.3: The same problem in two domains: In (a) are solutions in the decision variable space and in (b) are the respective points in the objective space.

point in the objective space would be $\mathbf{f}(\mathbf{d})$, represented on the right.

The objective space is where each solution will be compared to each other to find which are worth considering and which can be discarded, but the domain of the solution itself is the decision space. Unless stated otherwise, the term solution will be used to refer to both the solutions in the decision space and their respective points in the objective space, being obvious that, for instance, when referring to a "solution in the objective space", the referred solution is in the decision space, it is its point that is in the objective space.

2.2.2 Pareto dominance

In figure 2.3(b) that there is no clear best solution, i.e., almost all of them are better in one objective but worse at the other. Assuming that no objective has a greater importance than the other, there is not a simple answer to find which solution is optimal, as each solution has to make some compromise to the others. But, for instance, solution **b** is clearly an uninteresting solution, as there is solution **a** better than **b** in both objectives. In multi-objective optimization problems,

solutions like **b** are called dominated solutions, and those like **a** are dominating solutions. This domination concept was first introduced by Vilfredo Pareto (see Stadler [80]), and is commonly known as the *Pareto dominance*. Formally, it can be defined by [80, 22, 99, 42]:

Definition 2.1. Given two solutions $\mathbf{x}, \mathbf{y} \in \mathcal{S}$, $\mathbf{x}$ is said to *weakly dominate* $\mathbf{y}$ (denoted by $\mathbf{x} \preceq \mathbf{y}$) if

$$\forall_{i \in \{1,\dots m\}} : f_i(\mathbf{x}) \leq f_i(\mathbf{y}) \tag{2.2.5}$$

Definition 2.2. Given two solutions $\mathbf{x}, \mathbf{y} \in \mathcal{S}$, $\mathbf{x}$ is said to *dominate* $\mathbf{y}$ (denoted by $\mathbf{x} \prec \mathbf{y}$) if

$$\begin{cases} \forall_{i \in \{1,\dots,m\}} : f_i(\mathbf{x}) \leq f_i(\mathbf{y}) \\ \exists_{i \in \{1,\dots,m\}} : f_i(\mathbf{x}) < f_i(\mathbf{y}) \end{cases}. \tag{2.2.6}$$

Definition 2.3. Given two solutions $\mathbf{x}, \mathbf{y} \in \mathcal{S}$, $\mathbf{x}$ is said to *strongly dominates* $\mathbf{y}$ (denoted by $\mathbf{x} \prec\prec \mathbf{y}$) if

$$\forall_{i \in \{1,\dots m\}} : f_i(\mathbf{x}) < f_i(\mathbf{y}). \tag{2.2.7}$$

Definition 2.4. Given two solutions $\mathbf{x}, \mathbf{y} \in \mathcal{S}$, $\mathbf{x}$ is said to be *incomparable* to $\mathbf{y}$ (denoted by $\mathbf{x} \sim \mathbf{y}$) if

$$\mathbf{x} \not\preceq \mathbf{y} \text{ and } \mathbf{y} \not\preceq \mathbf{x}. \tag{2.2.8}$$

These relations state that if solution $\mathbf{x}$ strongly dominates solution $\mathbf{y}$, then $\mathbf{x}$ is better that $\mathbf{y}$ in all objective; if solution $\mathbf{x}$ dominates solution $\mathbf{y}$, then $\mathbf{x}$ is not worse then $\mathbf{y}$ in all objectives, and is better then $\mathbf{y}$ in at least one objective; if solution $\mathbf{x}$

weakly dominates solution **y**, that **x** is not worse than **y** in all objectives; finally, if solution **x** is not worse than **y** in all objectives neither is **y** is worse then **x**, than solutions **x** and **y** are incomparable. Clearly, if solution **x** strongly dominates solution **y**, it also dominates and weakly dominates it, i.e., $\mathbf{x} \prec\prec \mathbf{y} \implies \mathbf{x} \prec \mathbf{y} \implies \mathbf{x} \preceq \mathbf{y}$.

As the weak domination relation is reflexive ($\mathbf{x} \preceq \mathbf{x}$), antisymetric ($\mathbf{x} \preceq \mathbf{y} \wedge \mathbf{y} \preceq \mathbf{x} \implies \mathbf{x} = \mathbf{y}$) and transitive ($\mathbf{x} \preceq \mathbf{y} \wedge \mathbf{y} \preceq \mathbf{z} \implies \mathbf{x} \preceq \mathbf{z}$), it defines a non-strict partial order, while the domination and the strong domination relations define a strict partial order, as they are irreflexive ($\mathbf{x} \not\prec \mathbf{x}, \quad \mathbf{x} \not\prec\not\prec \mathbf{x}$), transitive ($\mathbf{x} \prec \mathbf{y} \wedge \mathbf{y} \prec \mathbf{z} \implies \mathbf{x} \prec \mathbf{z}, \quad \mathbf{x} \prec\prec \mathbf{y} \wedge \mathbf{y} \prec\prec \mathbf{z} \implies \mathbf{x} \prec\prec \mathbf{z}$) and asymmetric ($\mathbf{x} \prec \mathbf{y} \implies \mathbf{y} \not\prec \mathbf{x}, \quad \mathbf{x} \prec\prec \mathbf{y} \implies \mathbf{y} \not\prec\not\prec \mathbf{x}$).

Consider solution **a** in in figure 2.3b: it strongly dominates solution **b**, because for all objectives, the values of solution **a** are better then those of solution **b**. Solution **a** is also dominates solution **c**, because they have the same value for one objective (f_1), but for the other objective, solution **a** has a better result. On the other hand, comparing solution **a** with solution **d** has an interesting aspect: solution **a** is better than solution **d** in the first objective (f_1), but is worse in the second objective (f_2), i.e., they are incomparable. As none dominates the other, they are called *non-dominated* solutions.

Using a pair-wise comparison, every solution in a finite set of solutions can be compared to each other, to determine which solutions dominates which; which solutions is dominated by which, and which are the solutions that are non-dominated in relation to one another. The purpose of this is, in the end, to obtain a set with the solutions that are non-dominated in relation to one another. All solutions in this set doesn't dominate any other solution in it, and for all solutions not in this set, there exists at least one solution in the set that dominates it.

In figure 2.4 is a resume for comparing two solutions: In relation to solution **x**, any solution in the upper-right quadrant is dominated by solution **x**, in the lower-left are the solutions that dominate solution **x**, and in the upper-left and

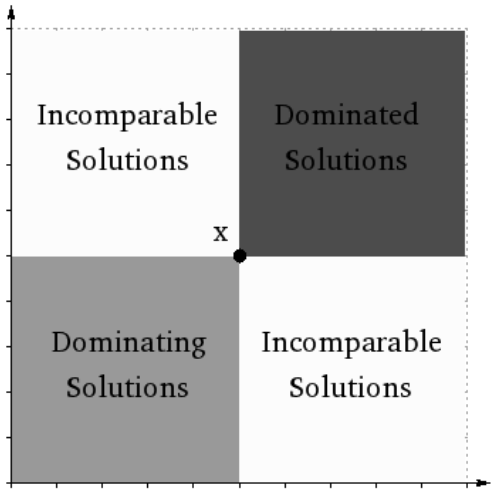


Figure 2.4: Dominating, dominated and incomparable solutions to X, in the objective space.

lower-right are the solutions that cannot be compared to solution **x**, i.e., they are incomparable to **x**.

Definition 2.5. From a set of solutions P, the *non-dominated set* of solutions $P^\star$ are those that are not dominated by any other solution in P:

$$P^\star = \{\mathbf{x} : \mathbf{y} \not\prec \mathbf{x},\ \forall_{\mathbf{x},\mathbf{y}\in P}\}\,. \tag{2.2.9}$$

Representing these non-dominated set of solutions in the objective space would result in a *non-dominated front*. If the set P is the decision space $\mathcal{S}$, i.e., $P = \mathcal{S}$, then this non-dominated set is called the *Pareto-optimal set*, and each solution $\mathbf{x} \in P^\star$ is a *Pareto-optimal solution*. Likewise, the representation of the Pareto-optimal set in the objective space is called the *Pareto-optimal front*.

In figure 2.5, solutions $\{\mathbf{e}, \mathbf{c}, \mathbf{a}, \mathbf{d}\}$ define the non-dominated set, and the line connecting them is the non-dominated front. The grey area is the dominated region, i.e., for every solution in the dominated region, there exists at least solution in the non-dominated set that dominates it, as is the case of solution **b**, that is dominated by both solutions **c** and **a**.

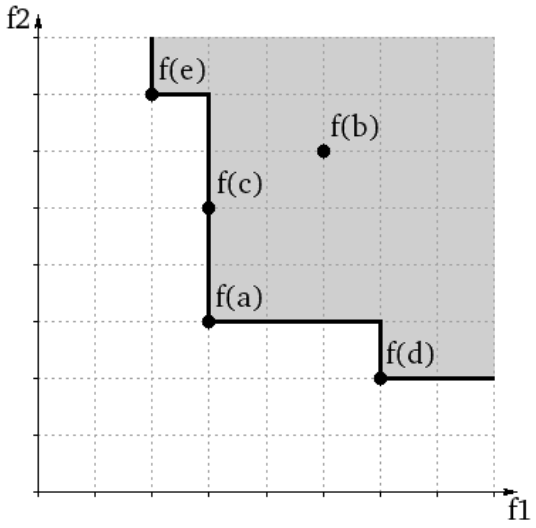


Figure 2.5: Non-dominated set $\{\mathbf{e}, \mathbf{c}, \mathbf{a}, \mathbf{d}\}$ and non-dominated front (black line).

If the outcome of a multi-objective optimization problem is a non-dominated set of solutions, henceforth called *approximation set*, how to compare two approximation sets to decide which is better? Although most times is not easy to answer that question, by extending the concept of Pareto dominance to sets of solution, a relation can be given between two approximation sets [99, 42]:

Definition 2.6. Given two approximation sets A, B, A is said to *weakly dominate* B (denoted by $A \preceq B$) if

$$\forall_{\mathbf{y} \in B} \exists_{\mathbf{x} \in A} : \mathbf{x} \preceq \mathbf{y}. \tag{2.2.10}$$

Definition 2.7. Given two approximation sets A, B, A is said to *better than* B (denoted by $A \lhd B$) if

$$A \neq B \wedge \forall_{\mathbf{y} \in B} \exists_{\mathbf{x} \in A} : \mathbf{x} \preceq \mathbf{y}. \tag{2.2.11}$$

Definition 2.8. Given two approximation sets A, B, A is said to *dominate* B (denoted by $A \prec B$) if

$$\forall_{\mathbf{y}\in B}\exists_{\mathbf{x}\in A} : \mathbf{x} \prec \mathbf{y}. \tag{2.2.12}$$

Definition 2.9. Given two approximation set A, B, A is said to *strongly dominate* B (denoted by $A \prec\prec B$) if

$$\forall_{\mathbf{y}\in B}\exists_{\mathbf{x}\in A} : \mathbf{x} \prec\prec \mathbf{y}. \tag{2.2.13}$$

2.2.3 Evolutionary multi-objective optimization

As seen previously, in multi-objective optimization is common for all objectives to be equally important, and because of this, the methods used to find the optimal solution in single-objective problems cannot be used here, as using single-objective algorithms to find solutions for multi-objective problems would result in optimal solutions for one objective, disregarding all other objectives, but solutions to multi-objective problems, as already discussed, need some compromise to be made between each objectives. To find this type of solutions, an algorithm for a multi-objective optimization problem, need, itself, to solve another multi-objective optimization problem, as there are two (possibly conflicting) objectives it needs to achieve:

1. *Proximity*, i.e., needs to find an approximation set of solutions as close as possible to the Pareto-optimal set,

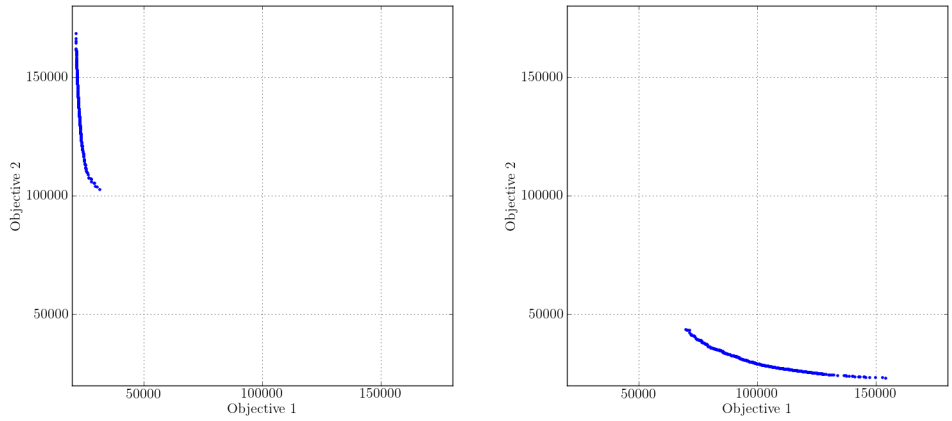
2. *Diversity*, i.e., the approximation set solutions found need to be as diverse as possible, in order to represent the entire Pareto-optimal front. If there are too many solutions in certain zone of the Pareto front and none or few solution in other zones, the approximation set would not correctly represent the Pareto-optimal front.

To find an approximation to the Pareto-optimal front, first, a set of solutions is needed, to then determine the non-dominated set. A very common way to find

them is using Evolutionary Multi-Objective Optimization (EMO) algorithms, also referred in the literature as Multi-Objective Evolutionary Algorithms (MOEA). Their intrinsic characteristics are that main reason to use them for this type of problems, but also their main drawback: the main advantage of using EMO is the fact that, as they are usually population-based algorithms, they evolve a population of solutions in each generation, and use this population to find the non-dominated set, this way approaching the Pareto-optimal front in each generation. On the other hand, as they are heuristic algorithms, they search the decision space for feasible solutions, evolving and improving them in each generation of the process, with the final objective of approaching the Pareto-optimal front. But, in the end, there is no guarantee that the final non-dominated set of solutions are the Pareto-optimal solutions, only that they were best non-dominated set from all generated solutions.

The other problem is guaranteeing that the set of solutions are diverse enough to represent the entire range of the Pareto-optimal front. If an approximation to the Pareto front is found, but is not diverse enough to cover the entire Pareto-optimal front, or at least a very good part of if, them this approximation is skewed, as most likely a good enough decision cannot be made using it. Consider figure 2.6: although representing an approximation to the Pareto front, neither figure 2.6(a) nor figure 2.6(b) are diverse enough to represent the entire front, because there isn't a good distribution of the solutions regarding both objectives. In figure 2.6(a) the approximation front is skewed for good values in the first objective, disregarding completely the second objective, and in figure 2.6(b) is the other way around, only the second objective has good approximations.

Over the years, several EMO algorithms have been developed to accomplish these two goals, and they can be divided according to the technique used to do it. A thoroughly discussion of some can be found in [93, 87], but the most common are the three presented next.



(a) (b)

Figure 2.6: Two approximation fronts that don't represent the entire Pareto-optimal front.

The most common technique uses the Pareto dominance to find solutions approximated to the Pareto-optimal set, and computed a distance between each solution in the non-dominated set to maintain the diversity in this set. State-of-the-art algorithms, such as the non-dominating sorting genetic algorithm II (NSGA-II) [23] and the strength Pareto evolutionary algorithm 2 (SPEA2) [96] use this type of technique. NSGA-II, in particular, is one of the most used algorithms for benchmarking and comparing with new algorithms [29, 11], and was, inclusive, the most used algorithm, in a recent survey by von Lücken et al. [87]. This algorithm uses a two stage approach to select which individuals to keep for the next generation: first it joins in the population with the offspring generated by the usual evolutionary operators (crossover and mutation), then it sorts all individuals into fronts according the the Pareto dominance, and then uses a crowding distance to maintain the diversity of the solutions. After this two measures, it selects the population for the next generation by selecting first the individuals in the first front, then those on the second, and so on, until either it reaches the size of the population, or until the number of missing individuals in the population is less that the size of the next front. In the first case, the population is found, and the algorithm proceeds to the next generation, otherwise, it selects from the next front, the number of missing individuals, based on their crowding distance, i.e., select first the

individuals in a less crowded region of the objective space, until it reaches the size of the desired population.

The SPEA2, on the other hand, uses an archive population to keep the non-dominated individuals, and the population to generate the offspring. First, it calculates the strength of each individual in both population and archive. The strength is the number of individuals that each one dominates, and then assigns a fitness to each individual, that is the combined strengths of its dominators. To preserve the diversity, it uses an adaptation of the kth nearest neighbour method, in which, for each individual, the distances to each other individual is calculated and sorted in a list. The density of each individual is an inverse function of the kth element of that list, being k, according to the authors, the square root of the size of the sample, i.e., population and archive. Finally, this density is added to the fitness. To create the archive population for the next generation, first, using the union of the old archive with the population, it selects the non-dominated individuals and copy then to the new archive. As the size of the archive population is fixed, three cases can occur: If the non-dominated individuals fit exactly in the archive, then this step is done, and the algorithm proceeds to calculate the next generation; If the size of the archive is too small, the add the best dominated individuals from the union until archive is complete; If the size of the archive is too large, some non-dominated individuals must be removed. This removal is done one individual at a time, selecting each time the individual with the minimum distance to another individual, until all exceeding elements are remove from the archive.

Other technique is to decompose the multi-objective problem into several single-objective problems (SOP), using a vector of weights for each objective, that can be solved using any traditional algorithm for single-objective optimization. The solutions for the objective problems are used to construct the non-dominated set, and the vector with the weights controls the diversity of the solutions. The multi-objective evolutionary algorithm based on decomposition (MOEA/D) [75]

is an example of an algorithm using this technique. The core any decomposition algorithm is how to decompose the multi-objective problem into several single-objective. In [75], the authors proposed three approaches: a simple weight sum approach, a Tchebyshev approach, or a boundary intersection approach. In MOEA/D, the population of is composed of the best solutions found so far for each problem, and the non-dominated solutions a kept in an external population. The MOEA/D algorithm starts by creating a set of weight vectors, one for each sub-problem to be considered, and calculates the pairwise distance between them. This distance defines the neighbours of each weight vector. Then, for each sub-problem, select two random neighbours and use their solutions to create an offspring using usual genetic operators. This offspring is then used to update all the current sub-problem's neighbours solution, using the decomposition formula. The external population is also updated accordingly, i.e., place the offspring in the external population if no individual in it dominate the offspring, and remove any individuals in the external population dominated by the offspring, and repeat everything for the next sub-problem.

The final technique is to use a scalar indicator, developed to measure the "quality" of the solutions obtained, as will be presented later (see section 2.2.4). Using this technique, a fitness is assigned to each solution, and this fitness guides the search for better solutions. Unary indicators give a value to each solution using some performance criteria, and binary indicators compare two solutions to measure their relative quality. These type of algorithms usually don't need any additional method to maintain diversity, as the indicator already guarantee it. The indicator-based evolutionary algorithm (IBEA) [95] was the first to use this idea, and the idea is a simple one: Define a fitness function based on an Pareto compliant indicator (see 2.2.4) and using this function assign a fitness to each individual in the population. Apply any usual genetic operators to create an offspring population, and join both populations. To select the individuals to the next generation, just choose the individual with the smallest fitness, remove it from population, and

update the remaining individual's fitness accordingly, repeating these steps until the size of the population is adequate.

2.2.4 Performance measures

When solving single-objective optimization problems, it is relatively easy to compare two outputs, and see which is "best", as the output is a single solution corresponding to the minimum or maximum of a single-objective function. But when the problem is multi-objective, the output is not a single solution, is an approximation set of solutions.

In addition, it is very unlikely that a EMO will reach the Pareto-optimal set, either because the number of Pareto-optimal solutions is to high, or because determining a single Pareto-optimal solution is a NP-hard problem, or simple because the intrinsic heuristic of EMOs. Instead, as the outcome of an EMO is an approximation to the Pareto-optimal set, some type of measure is needed to qualify this approximation, either to compare it with the Pareto-optimal set, if available; or compare with other approximation sets from the same multi-objective optimization algorithm; or even to compare approximation sets between different multi-objective optimization algorithms.

Using the Pareto dominance, given in definitions 2.6 to 2.9, a partial order can be used to compare different approximation sets, in a natural way. However, although using it, one can say that, for instance, approximation set A is better than approximation set B, this "betterness" cannot be quantified. Does it have more solutions in it? Is it closer to the Pareto-optimal front? Does it have a better diversity?

To answer this type of questions, other type of quantifiers were introduced, commonly called performance measures. Zitzler et al. [94, 42] classified the different quantifiers into three different groups, according to the method used: the *dominance ranking* method, the the *quality indicator* method, or the the *attainment*

function method.

The dominance ranking assigns a ranking value to each approximation set, based on comparison between approximation sets. Although there are several ways to determine the rank to an approximation set based on the dominance concept, Zitzler et al. [94] suggested that counting the number of sets by which each approximation set is dominated by (Fonseca and Flemming [26]), extended by the Pareto dominance gives a better ranking with less ties. Suppose $A_1, ..., A_r$ and $B_1, ..., B_r$ are approximation sets, resulting from running two different multi-objective optimization algorithms r times (for simplicity, lets assume both algorithms were executed an equal number of times). The rank of approximation set A_i is determined by the number of approximation sets $B_j,\ j = 1, ..., r$ that dominates A_i:

$$\operatorname{rank}(A_i) = \left|\left\{B : B \prec A_i, \forall_{B \in \{B_1,...,B_r\}}\right\}\right|. \tag{2.2.14}$$

Analogously, the same is used for calculating the ranks for the approximation sets generated by the other algorithm, and the result is two sets, one for each algorithm, with the rank of each approximation set when compared with the other algorithm, like $\{\operatorname{rank}(A_1), ..., \operatorname{rank}(A_r)\}$, and $\{\operatorname{rank}(B_1), ..., \operatorname{rank}(B_r)\}$. Finally, using a statistical rank test, such as the Mann-Whitney rank sum test or the Kruskal-Wallis rank test when there are more than two optimizers to compare, can be used to determine if there is a significant difference between the algorithms.

This method, although being simple and computationally inexpensive, only gives very general information about the relative performance of the optimizers, and others methods should be used to complement it, unless a significant difference can be demonstrated between the optimizers using it alone [94].

Another method is the attainment function. This method is based on goal-achievement probability, meaning the probability that at least one element of an approximation set X attains the goal-point $\mathbf{z} \in \mathbb{R}^m$ in a single run [31, 25]. Defin-

ing *attain* by

Definition 2.10. Given an approximation set X, and a solution in the objective space $\mathbf{z}$, X is said to *attain* $\mathbf{z}$ (denoted $X \trianglelefteq \mathbf{z}$) if

$$\exists_{\mathbf{x} \in X} : \mathbf{x} \preceq \mathbf{z}. \tag{2.2.15}$$

Although in practice the attainment function is unknown, it can be estimated using data from several independent executions from an multi-objective optimization algorithm. The *empirical attainment function*, used in this method, is defined by

$$\alpha_r(\mathbf{z}) = \frac{1}{r} \sum_{i=1}^{r} \mathbb{I}(X_i \trianglelefteq \mathbf{z}), \tag{2.2.16}$$

where $X_i,\ i = \{1, ..., r\}$ is the ith approximation set of r runs of a multi-objective optimization algorithm and $\mathbb{I}(\cdot)$ is the indicator function, returning zero if the argument is false, or one otherwise. Basically, this empirical attainment function will estimate the relative frequency on which each objective point in the objective space was weakly dominated by an approximation set in r runs. Statistical tests can now be made using these results, to compare the results of two multi-objective optimization algorithms. This method also allows to compare two different multi-objective optimization algorithms visually, and even see the differences between them, i.e., the area were one algorithm behaves better then the other [54], which can be important to understand how an algorithm behave, in the developing phase.

However, its computational cost is very high, which is the disadvantage of this method.

Finally, the most used method in literature [94] are the quality indicators. These came in various "shapes" and "forms", but the basic idea behind them all is to assign a real value to approximation sets. Theoretically, a n-ary indicator can

be defined by

$$I\left(A_1, ..., A_n\right) : \mathcal{S}^n \rightarrow \mathbb{R}, \tag{2.2.17}$$

and, although there is no limit to n, in literature is not frequent to appear anything higher then binary indicators, being the unary, by far, the most frequent [99, 77]. An explanation for this could be the fact that unary indicators assign a value to each approximation set, regardless of any other approximation set; and the number of results given by high order indicators. Suppose r approximation sets are to be compared: using an unary indicator, r results will be obtained; in a binary indicator, as a pairwise comparison is used, will end up with $r(r-1)$ results [99]; and so on.

Suppose two approximation sets A and B, a good unary indicator should not only be able to establish an order between them, but the difference between the indicators should also reveal the difference in the quality between the approximation sets.

As each unary indicator is only able to measure one characteristic of the approximation set, for instance, the distance to the Pareto-optimal front or the diversity within the approximation set, different unary indicators, measuring different characteristics, can give different values to the same approximation set, possibly resulting in a different classification of the approximation sets. For this reason, when comparing two approximation sets using a unary indicator, one has to always refer to the indicator used, i.e., "approximation set X is better than approximation set Y, according to indicator I".

An explanation of the most used performance measures will be presented next.

Hypervolume

The *hypervolume* metric, also known as S-metric, was proposed initially by Zitzler and Thiele [97, 98] as the portion of the objective space weakly dominated by

an approximation set, and is to be maximized. To calculate the hypervolume, an additional point in the objective space must be used, to bound the approximation set. Assuming a two-objective problem (the concept can be canonically extended to multiple dimensions), for a maximization problem, is common to use the origin ($(0, 0)$ in an two-objective problem), but for a minimization problem, a point "outside" the extreme values of the approximation set is required. For instance, if X is an approximation set with cardinality n and $\mathbf{x}_k \in X,\ k = \{1, ..., n\}$, the point $(\max(f_1(\mathbf{x}_1), ..., f_1(\mathbf{x}_n)) + 1, \max(f_2(\mathbf{x}_1), ..., f_2(\mathbf{x}_n)) + 1)$, defines a point strongly dominated by all solutions in the approximation set.

As it depends on the magnitude of the values, it should be normalized, and becomes a ratio:

$$I_{HVR}(A) = \frac{I_{HV}(A)}{I_{HV}(R)}, \tag{2.2.18}$$

where A is an approximation set, R is a reference set (or the Pareto-optimal set, if available), and $I_{HV}(\cdot)$ defines the hypervolume for a given approximation set.

This indicator measures in both the proximity and the diversity and an approximation set, since the higher the value, the close is the approximation set from the Pareto-optimal set and the higher its diversity along the Pareto front.

The hypervolume is also the only known unary indicator to strictly comply to Pareto dominance, i.e., $\forall_{A,B \in \mathcal{S}} : A \prec B \implies I_{HV}(A) > I_{HV}(B)$, but this depends on using the same bounding point for both approximation sets, and that bounding point being strongly dominated by all solution in both approximation sets.

Recent surveys [87, 77] show that this indicator is, by far, the most used in the specialized literature, despite the very computational cost, as it is exponential on the number of objectives.

Generational distance

The *generational distance* [85, 86] is an indicator that measures the distance of an approximation set to a reference set, either the Pareto-optimal set or a very close approximation, and is defined by

$$I_{GD}(A,B) = \frac{1}{|A|}\left(\sum_{\mathbf{a}\in A} d\,(\mathbf{a})^p\right)^{1/p}, \quad d(\mathbf{a}) = \min_{\mathbf{b}\in B} \|\mathbf{f}(\mathbf{a}) - \mathbf{f}(\mathbf{b})\| \tag{2.2.19}$$

where A and B are an approximation sets. Although the authors initially proposed $p = 2$, this was later changed to $p = 1$, for a simpler computation and interpretation, as now the generational distance is the average of the Euclidean distances between each solution in A and the nearest solution in B, meaning the lower the result of the indicator, the closer approximation set A is to B. Instead of two approximations sets, a reference set R can be used, and the indicator becomes $I^1_{GD}(A) = I_{GD}(A,R)$.

This indicator measures only the proximity, and does not comply to Pareto dominance, i.e, an approximation set A could be better than B, but the generational distance of A (in respect to a reference set R) is not necessarily lower than that of B, i.e., $A \lhd B \implies I^1_{GD}(A) \leq I^1_{GD}(B)$. It is also sensitive to the cardinality of the approximation set, as an approximation set with a lower cardinality can obtain a better (lower) indicator value than another set, with a higher cardinality, but in which all solutions dominate the solutions on the first set.

Although its usage is decreasing [77], it is still very used in the specialized literature.

Inverted generational distance

The *inverted generational distance* [14] was proposed to improve the generational distance, and its idea was to simple reverse the order of the approximation sets, i.e., it calculates the distance between each solution in the reference set and its

nearest solution in an approximation set, averaging it over the cardinality of the reference set, mathematically

$$I_{IGD}(A, R) = I_{GD}(R, A). \tag{2.2.20}$$

As it computes the difference based on the reference set, it is not sensitive to the cardinality of the approximation set, as GD, it measures both the diversity and the proximity, and is simple to compute. According to the literature its usage has been growing [77], albeit recently been proved that it does not comply to Pareto dominance [39].

ϵ-indicator

The *epsilon indicator family* are a set of indicators introduced by Zitzler et al. [99], and they determine the minimum value needed to transform a dominating reference set, into a set that is weakly dominated by another set. Is called a family because there are four variations of the same concept: it can be a unary or binary indicator, and can be additive or multiplicative. In any case, the fundamental concept is the ϵ-dominance, defined as

Definition 2.11. Given two solutions $\mathbf{x}, \mathbf{y} \in \mathcal{S}$, $\mathbf{x}$ is said to *ϵ-dominate* $\mathbf{y}$ (denoted by $\mathbf{x} \preceq_\epsilon \mathbf{y}$) if

$$\forall_{i \in 1,\ldots,m} : f_i(\mathbf{x}) \leq \epsilon \cdot f_i(\mathbf{y}). \tag{2.2.21}$$

Using the ϵ-dominance, the multiplicative binary epsilon indicator, is defined by

$$I_\epsilon(A, B) = \inf_{\epsilon \in \mathbb{R}} \{\forall_{\mathbf{b} \in B} \exists_{\mathbf{a} \in A} : \mathbf{a} \preceq_\epsilon \mathbf{b}\} \tag{2.2.22}$$

and the unary counterpart is defined by replacing the second approximation set B by a reference set R

$$I_\epsilon^1(A) = I_\epsilon(A, R). \tag{2.2.23}$$

The additive versions are defined analogously, replacing the product in definition 2.11 by an addition, and using $I_{\epsilon+}$ and $I_{\epsilon+}^1$ to represent the binary addition and the unary addition epsilon indicators, respectively.

This indicator is compliant to the Pareto dominance, i.e., $\forall_{A,B\in\mathcal{S}} : A \lhd B \implies I_\epsilon^1(A) \leq I_\epsilon^1(B)$ and cheap to compute, although being sensitive to scaling.

R indicators

The R indicators [35, 34, 40] are based on a utility function, in a sense, transforming the decision maker's preference information into an indicator. Suppose set of weight parameters vectorsΛ and a utility function $u_\lambda(\mathbf{z})$, that transforms a solution into a scalar value, using the specified parameter vectorλ, where $\lambda = \{\lambda_1, ..., \lambda_m\} \in \Lambda$. The utility function is usually defined using the weighted Tchebyshev function:

$$u_\lambda(\mathbf{z}) = \max_{i\in\{1,...,m\}} \lambda_i \cdot |z_i^\star - z_i|\,, \tag{2.2.24}$$

where $\mathbf{z} \in \mathcal{Z}$ is an objective solution, $\mathbf{z}^\star$ is the ideal point, i.e., $z_i^\star = \min_{\mathbf{z}\in\mathcal{Z}} z_i,\ i \in 1, ..., m$. Using this utility function, the $R2$ indicator can be defined by:

$$I_{R2}(A, B) = \frac{1}{|\Lambda|}\sum_{\lambda\in\Lambda}\left(\min_{\mathbf{b}\in B} u_\lambda(\mathbf{b}) - \min_{\mathbf{a}\in A} u_\lambda(\mathbf{a})\right), \tag{2.2.25}$$

and the R3 as:

$$I_{R3}(A, B) = \frac{1}{|\Lambda|}\sum_{\lambda\in\Lambda}\frac{\min_{\mathbf{b}\in B} u_\lambda(\mathbf{b}) - \min_{\mathbf{a}\in A} u_\lambda(\mathbf{a})}{\min_{\mathbf{b}\in B} u_\lambda(\mathbf{b})}. \tag{2.2.26}$$

If, instead of two approximation sets, a reference set R is used, they became unary operators, defined by $I_{R2}^1(A) = I_{R2}(A, R)$, and $I_{R3}^1(A) = I_{R3}(A, R)$. Further details can be found in the original papers [35, 34, 40].

In every case, the set Λ should contain a sufficiently large number of uniformly weighted vectors λ, with $\forall_{i\in\{1,\ldots,m\}} : \lambda_i \geq 0$ and $\sum_{i=1}^{m} \lambda_i = 1$.

This indicators cover both the proximity and the diversity of the approximation set and comply to Pareto dominance.

2.3 Combinatorial optimization

The name combinatorial optimization (CO) is derived from combinatorial problems, in which the domain of the problem is both a discrete and finite set of some kind of objects, that can be combined in different forms. In combinatorial optimization problems, the objective is to find the optimal combination to answer a certain problem. Basically, combinatorial optimization consist in finding the optimal solution, from a finite set of possible solutions. When explained like that, it looks simple, just find all of possible combinations and select the best. In a small set of possible combinations this is feasible, the problem is, as the sets grows larger, the size of the number of possible combinations "explodes", and there is no efficient algorithm capable of determining them all. This type of problems belong to a class called *NP-hard* problems [28, 6], and most combinatorial optimization problems belongs to this class.

One famous problems in combinatorial optimization is the knapsack problem, in which given a set of items, each with is own weight and value, find which items to put in the knapsack, maximizing the value but keeping the weight under a predefined limit. Suppose we have just 3 items, with their respective value and weight, as illustrated in figure 2.7, and we wanted to store those items in a knapsack with a weight limit of 40Kg. In this very simple example, with just three items, the possible combinations are easily calculated: Not forgetting that not all items are necessarily going into the knapsack, the different combinations are $\sum_{i=1}^{3} \binom{3}{i}$, or 7 possible combinations, showed in table 2.1, with the respective

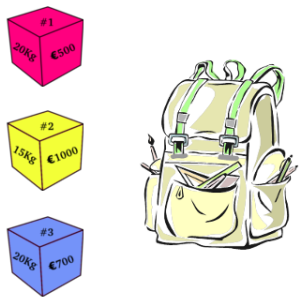


Figure 2.7: Example of a knapsack problem: How to optimize the knapsack capacity, maximizing the carried value?

Items Packed	Weight	Value
#1	20Kg	€500
#2	15Kg	€1000
#3	20Kg	€700
#1,#2	35Kg	€1500
#1,#3	40Kg	€1200
#2,#3	35Kg	€1700
#1,#2,#3	55Kg	€2200

Table 2.1: Possible ways to solve the knapsack problem, with respective weight and value.

weight and value.

From this, very few, possible solutions, one can easily see that 1) the last combination is not feasible, because it exceeds the limit of the knapsack; and 2) the one that maximizes the value and keep the weights within the bounded limits is the solution that uses items #2 and #3, giving a maximum value of €1700, and weighing 35Kg.

This very simple example was just to illustrate a combinatorial optimization problem, in which we could make a simplified analysis due to very low number of possible solutions, but that served to illustrate the combinatorial optimization problem: with only three feasible solutions, its easy to find the optimal one, but remember that if instead of 3 items we had just a few more, say 10, the number of combinations would raise from 7 to 1023, which would not be easily solved by hand, albeit not impossible, but the more items, the harder would the problem

became.

As said earlier, this is a characteristic of *NP-hard* problems: they are solvable for a low number of inputs, but there is no provable efficient algorithm to solve them for any number of inputs.

2.3.1 Travelling salesman problem

Probably, the most famous NP-Hard combinatorial optimization problems is the traveling salesman problem (TSP), commonly stated as "*Given a number of cities and a distance between each, find the shortest path that travels through each city exactly once and ends in the same city where it started*". This is probably the most studied combinatorial optimization problem, and is commonly used to benchmark optimization algorithms. There exists algorithms to find an exact solution to small instances of TSPs, but as the number of cities increase, the number of possible solutions raise exponentially, making impossible to apply brute-force algorithms to find the optimal solution in an acceptable time.

The core application of TSP is transport problems, but it has many applications to other real life problems. In [47, 57] many are introduced, amongst them:

Drilling printed circuit boards [30]: After the circuit boards is "printed", holes need to be made for each electronic component to be soldered. As these components may have connections of different sizes, different sized holes are needed. Each time a different size hole is needed, the machine needs to go to the toolbox and change the size of the drill, wasting time. This can be viewed as consecutive TSP, one for each hole size, where the "cities" are the position of the holes to be drilled, and the edges are the distance to move from hole to hole, starting (and ending) in the toolbox, to change to the next drill size. The objective is to minimize the travel time of the machine head.

Satellite positioning [7]: To study and explore the universe, there are satellites in orbit, free from Earth's atmosphere interference. Particularly, there are

missions involving more than one satellite, that need to move in coordination. Each time they need to be repositioned to study to a different position in the sky, precious fuel is needed for the satellite's rockets to move his position. The objective is to minimize the fuel needed to move from one position to the next, knowing that the mission is over as soon as one satellite runs out of fuel.

Warehouse merchandise distribution [5]: Very large warehouses have automatic crane mechanisms to collect and store material. When an item is ordered, the crane goes to the item position and brings it to the respective loading platform, and the inverse when items are delivered to the warehouse. In this type of warehouses, the crane usually don't stop all day, going back and forth between the item's position and the loading platforms. Considering each arc (i, j) as the possibility to deliver order j after delivering order i, this can be formulated as a TSP problem, aiming to minimize the travel of an empty crane.

X-ray crystallography [9]: To study the structure of crystals is measured their reflection to x-rays, using hundred's of thousands of positions. Although the measure is fast, the motors to move the x-ray machine, always takes some time to do it. The order of the positions is not important, but the time of the measure does, at it depends on repositioning the x-ray machine. Therefore, the problem is finding the best sequence to minimize the repositioning time.

As simple as an TSP problem can be understood, also a simple algorithm to solve it can be enumerated: Just calculate all permutations of nodes, and find the optimal solution. This is called a brute force algorithm and the caveat of this type of algorithms is the *NP-hardness* of the problem, which means that although this approach can be used for small dimension problems (with a small number of nodes), for real world problems this would be unfeasible, as the time required to compute all permutations of nodes would be unreasonable.

The TSP problem can be defined using a graph, i.e., a data structure capable of representing the cities and their connections. Consider the graph $G = (N, E, w)$, where N is a set of nodes or vertices, E is a set of edges $E = \{\{u, v\} : u, v \in N\}$,

where $\{u, v\}$ represent connection between node u and node v, and a weight function $w : E \to \mathbb{R}_+$, associating each edge to a non negative weight $w(u, v)$, representing the distance, the cost, the time, etc., of going from node u to node v. To simplify the notation, is frequent to use the w_{uv} to mean $w(u, v)$.

If the edges are symmetric, i.e., $\forall_{u,v \in N} : w_{uv} = w_{vu}$, the graph is said to be undirected, and the TSP is said to be a *symmetric traveling salesman problem* (STSP), or simply TSP; otherwise, if at least one of the edges is not symmetric $\forall_{u \in N} \exists_{v \in N} : w_{uv} \neq w_{vu}$, the notation used is (u, v) to represent an edge starting in node u and ending in node v, the graph is said to be directed, and, accordingly, the TSP is an *asymmetric traveling salesman problem* (ATSP).

When each node defines a point in $\mathbb{R}^d$ and the weight function is defined by the Euclidean distance $w_{uv} = \left(\sum_{i=1}^{d} (u_i - v_i)^2\right)^{1/2}$, the TSP is an *Euclidean TSP*, on which the triangle inequality is satisfied $\forall_{u,v,k \in N} : w_{uv} \leq w_{uk} + w_{kv}$. The Euclidean TSP is, obviously, symmetric.

When $w : E \to \mathbb{R}_+^n$, each edge as n values associated (time, cost, distance, etc.), and the problem is said to be an multi-objective traveling salesman problem (MOTSP).

Unless stated otherwise, when referring simply to TSP we are always referring to the Euclidean TSP.

Solving the TSP is a special case of the of finding an Hamiltonian circuit in a graph, which is a known NP-complete problem [28, 6]. An Hamiltonian circuit is a cycle that visits each node exactly once, as such, solving the TSP problem is finding the shortest path Hamiltonian circuit.

From the many formulations (see [70, 65] for a detail analysis), the most frequently used is the one defined by Dantzig et al. [13]. Assigning a binary variable x_{uv} to

$$x_{uv} = \begin{cases} 1 & \text{exists an edge between node } u \text{ and node } v \\ 0 & \text{otherwise} \end{cases}, \tag{2.3.1}$$

the TSP can be formulated as [57]:

$$\text{minimize} \sum_{u<v} x_{uv} w_{uv} \tag{2.3.2}$$

Subject to

$$\sum_{u<k} x_{uk} + \sum_{k<v} x_{kv} = 2 \quad (k \in N) \tag{2.3.3}$$

$$\sum_{u,v \in S} x_{uv} \leq |S| - 1 \quad (S \subset N,\ |S| > 1) \tag{2.3.4}$$

$$x_{uv} = 0 \text{ or } 1 \quad \{u, v\} \in E \tag{2.3.5}$$

where (2.3.3) is known as degree constraint and (2.3.4) as subtour elimination constraint. To put it simply, equation (2.3.3) guarantees that between any three nodes exists only two edges, and (2.3.4) says that the number of existing edges in a proper subset S of N is, at most, $|S| - 1$. Suppose a solution with a subtour S with $|S| < |N|$. If S defines a subtour, then $\sum_{u,v \in S} x_{uv} = |S|$, but this breaks constraint (2.3.4), as it would became $|S| \leq |S| - 1$, which is impossible, so the solution cannot contain the subtour S.

As previously stated, the TSP is a *NP-hard* problem, and for that reason, exact solvers, as the brute force approach, cannot be generally used to solve the TSP, and some approximation algorithms are needed. These algorithms don't return the exact optimum solution, but give an good enough approximation in a computational reasonable time. Amongst the several heuristic algorithms available to solve the TSP, we can divide them in two groups: the *tour building* ones, that create a tour (solution) to the TSP, and the *tour improving*, that try to improve a given initial solution. In the first group we can refer the *nearest neighbour* algorithm, that starting from a random node, always choose the nearest neighbour not yet visited, until gets back to the starting node; and the *Christofides' algorithm* [12]

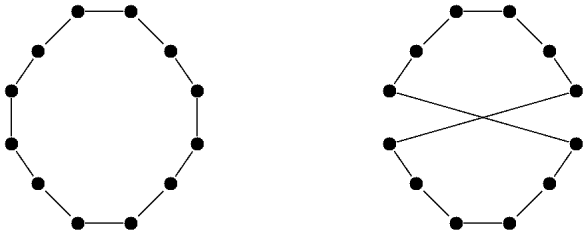


Figure 2.8: A 2-opt move: the initial solution (a), and after the move (b).

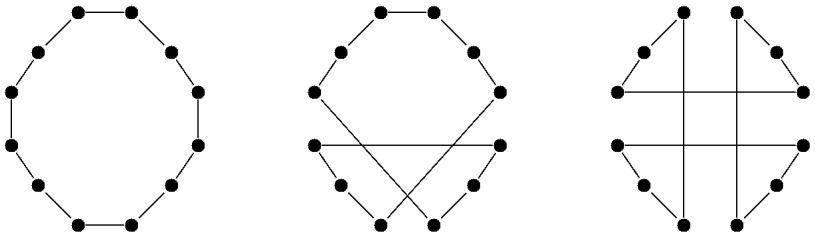


Figure 2.9: Two possible *k-opt* moves: An initial solution (a), and after a *3-opt* move (b) or a *4-opt* move (c).

that used the principle that, if removing an edge from a TSP results in a *spanning tree*[1], then starting with the *minimum spanning tree*[2] it possible to create the TSP (see [12] for further details). for the secong group, two possible algorithms are the *2-opt* [15], that swaps two edges if this swap result in an improved solution (see figure 2.8); and the *Lin-Kerninghan* algorithm [50], that is a generalization of the *2-opt* algorithm, where the number of swaps is dynamically calculated (see figure 2.9). As it should be clear, for any of the second group to work, a starting solution is needed, and the better the solution, the fastest this type of approach would return their "optimal" solution. The better heuristics to solve the TSP incorporate these two groups into a single algorithm, as the *Lin-Kernighan heuristic* (LKH) [36] and the *edge assembly crossover* (EAX) [62], state-of-the-art algorithms to solve the TSP.

The Lin-Kernighan algorithm, has was introduced in [50], is basically a *k-opt*

[1]A spanning tree (ST) is a particular case of a graph, in which all nodes are connected but there isn't any cycle.

[2]The minimum spanning tree (MST) is a tree that connects all the nodes with the minimum total weight.

variant, in which k is not a static parameter, but is dynamically determined in each iteration. In [36], the LKH implementation was formulated, in which the basic move is a 5-opt. The EAX algorithm is a genetic algorithm based approach, using the crossover as its main focus, as it uses two parents for create an offspring. The create this offspring, first creates a small set of cycles (called *AB-cycles*) using alternating edges from both parents, and then join these *AB-cycles* into a TSP by removing one edge from each *AB-cycle* and create two new edges to join each *AB-cycle*, until a feasible solution is reached (further details in [62].

Chapter 3

Differential evolution for combinatorial optimization

3.1 Introduction

As stated previously, the differential evolution algorithm was introduced to solve problems defined on real spaces, i.e., the domain of the problem is continuous. This type of problems is very different from combinatorial problems, where the domain is discrete, and many times the representation is not numeric but symbolic, meaning there is nothing we could optimize in this realm, as symbols cannot be subject to any arithmetic operators, as the ones defined in DE's operators. Some form of representation for this type of problems is needed, so they can be encoded using numeric terms, and in this form, be subject to some optimization.

Due to the discretization of the domain of the problem and the fact that most of them are *NP-hard*, combinatorial optimization problems are not easily tackled by most evolutionary algorithms, and a careful approach must be used, either in the codification of the individuals and in the operators used in the evolutionary process. In DE, both codification of the individuals and operators are defined in the real domain, and to be applied to combinatorial optimization problems, one or both of them need to be, somehow, converted to work in the discrete space.

Moreover, another thing to consider is the feasibility of the mutant individual, i.e., the result of combining the base individual with the difference between other two individuals should be a valid individual, and this is the main problem for most combinatorial optimization problems. If the result of the mutation operator is an unfeasible individual, a repair/replace mechanism is needed and this could influence substantially the success of the algorithm, more so than the algorithm itself, as Price et al recognized in [72].

Nevertheless, several approaches have been introduced over the years to allow the usage of discrete values in continuous methods. One of the first codification approaches was introduced even before DE itself, in 1994 by Bean [8], and is known as *random keys* encoding, and basically transforms a floating-point vector into their discrete counterpart by assigning each floating-point value to a discrete one based on their relative order, i.e. to the higher value was assigned the value 1, to the second higher the value 2, etc. Using this encoding, the algorithm worked with real values in their core, and this transformation was only applied when an evaluation was needed.

A similar approach was introduced by Nearchou [63], but he uses intervals to encode the floating-point values to discrete ones, instead of a direct translation. Litchblau [48, 49] developed the *relative position indexing*, which is also based on the random keys approach, but instead of using random floating-point values, it creates them by dividing each element of the individual by their largest value. Onwubolu [66, 68, 67, 69] gave us the *forward/backward transformation*, which transforms the discrete values to/from floating-point using a mathematical formula. Lampinen and Zelinka [46], introduced a mixed integer-discrete-continuous optimization, further developed by Zelinka [92], resulting in a *discrete set handling* approach, that basically substitutes the discrete value by their index, and optimize the index, instead of the value itself.

DE's authors implicitly recognized that the algorithm working only for continuous domains was a shortcoming, and in [72] Price et al. suggested two meth-

ods based on matrices, for it to work in combinatorial problems: The first uses a permutation matrix as the difference between two vectors, and is known as the *permutation matrix* approach; the other is the *adjacency matrix* approach, which encodes the solutions using adjacency matrices, instead of vectors, then uses a logical operator to calculate the difference between two matrices. Although being introduces by the authors of the algorithm, this approaches were never, if ever, much used. Prado et al. [71] suggested what they dubbed as a general approach, where the difference between two vector to be the list of swaps of elements needed to transform one into the other, usually referred to *differential list of movements*.

Recently, several authors proposed set-based approaches to handle the discrete space in DE. The common with all of the approaches from Maravilha et at. [55, 56], Liu and Maeda [51, 52], Liu et al. [53] and Guerreiro et al. [32, 33] is the usage of set operations (union, intersection, etc.) instead of arithmetic ones, with different custom rules to accomplish the scale factor in the mutation. These are but some of the most used implementations to use DE in combinatorial optimization problems, an extensively listing can be found in [18, 17].The ones introduced above will be further explained in the next section.

3.2 Previous approaches

3.2.1 Permutation matrix approach

The basic idea behind DE is adding some mutation (the difference between two vectors) to another vector, and this, as seen extensively, cannot be translated directly into the discrete domain. Price et al. [72] introduced the permutation matrix approach, as this matrix can be used as the “difference” between two discrete vectors, because it gives the “transformation” from one into the other. Multiplying this permutation matrix to another vector, transforms it, in accordance to the permutation matrix.

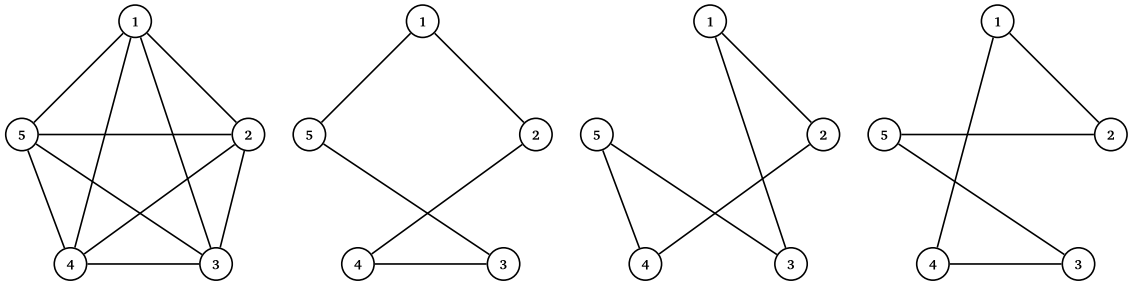


Figure 3.1: An example of a TSP with five cities. In (a) is the domain of the problem, (b)-(d) shows three possible solutions.

Suppose an instance of traveling salesman problem with five cities, labeled 1 though 5. The full domain of the problem is a complete graph[1] shown in figure 3.1(a). In this domain, several solutions are possible, for instance those shown in figures 3.1(b) to (c). Representing this example solutions using vectors, would result in

$$\mathbf{x}_1 = \begin{bmatrix} 4 \\ 2 \\ 1 \\ 5 \\ 3 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 1 \\ 3 \\ 5 \\ 4 \\ 2 \end{bmatrix}, \quad \text{and} \quad \mathbf{x}_3 = \begin{bmatrix} 4 \\ 1 \\ 2 \\ 5 \\ 3 \end{bmatrix}.$$

To calculate the mutation operator given in equation 2.1.3, first the difference between x_{r2} and x_{r3} need to be calculated, and this "difference" is the permutation matrix. In this example, the permutation matrix would result in

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \quad \text{where} \quad P \cdot \mathbf{x}_3 = \mathbf{x}_2. \tag{3.2.1}$$

Before adding this permutation matrix to the base vector $\mathbf{x}_1$, in accordance equation 2.1.3, a scale factor F should be "multiplied" to the permutation matrix. Price et al. suggested an algorithm to exchange some columns of the permutation matrix using a probability based on a δ factor, that would have an effect similar

[1] A complete graph is a graph where every node is connected to every other node through one edge.

to scaling the permutation matrix. If $\delta = 0$ the permutation matrix would be the identity matrix, and this would result in no permutation to be applied, and if $\delta = 1$ the permutation matrix would remain the same as previously calculated. The δ parameter will have here the same effect F has in the original formula for the continuous domain, the only difference being that δ represents a probability, as such $\delta \in [0; 1]$, where F, theoretically, could take any real value, although in practice was defined as $F \in [0; 2]$. The whole procedure to find the permutation matrix and scale it, can be observed in algorithm 3.1, where any two vectors **x** and **y** are given (assuming they have equal number of elements) and the final scaled permutation matrix is returned. Refer to [72] to a full analysis of the algorithm.

Algorithm 3.1 Computing the scaled permutation matrix.

Input: x, y
Output: P

1: $n \leftarrow$ sizeof(x)
2: $P \leftarrow O_{n,n}$ // Create a zero matrix with order n
3: // Compute the permutation matrix P
4: **for** $i = 1..n$ **do**
5: $j \leftarrow \{j: x_i = y_j, \forall_{j=1..n}\}$ // position of x_i in y
6: $P_{i,j} \leftarrow 1$
7: **end for**
8: // "Multiply" the permutation matrix by a scale factor probability δ
9: **for** $j = 1..n$ **do**
10: **if** $P_{j,j} = 0$ and rand()$> \delta$ **then**
11: $i \leftarrow \{i: P_{i,j} = 1, \forall_{i=1..n}\}$ // find row where $P_{i,j}$ =1
12: swap_rows(i, j)
13: **end if**
14: **end for**

We can then multiply the base vector $\mathbf{x}_{r1}$ by this scaled permutation matrix, that will exchange the elements of the base vector using the permutations defined between $\mathbf{x}_2$ and $\mathbf{x}_3$.

In the previous example, suppose $\delta = 1$, and the result of multiplying the scaled permutation matrix P to the base vector $\mathbf{x}_1$would result in

$$P \cdot \mathbf{x}_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 2 \\ 1 \\ 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 5 \\ 4 \\ 1 \end{bmatrix}.$$

Recall that, as the vectors are given in columns, to change the rows in a vector, it has to be left multiplied by the permutation matrix. If the vectors were lines, the multiplication would be a right one.

This method has the drawback of not identifying rotated vectors, i.e., suppose this two different vectors:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 4 \\ 5 \\ 3 \end{bmatrix}, \text{ and } \mathbf{y} = \begin{bmatrix} 5 \\ 4 \\ 2 \\ 1 \\ 3 \end{bmatrix}.$$

When representing them vectors in a graph, both would result in the same solution, presented in figure 3.1(c). If both vectors represent the same solution, their "difference" should be zero, but calculating the permutation matrix between them results in

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

which is clearly not zero. For combinatorial optimization problems, this means that the method is "wasting" computational resources, calculating and using different vectors to represent the same solution.

3.2.2 Adjacency matrix approach

The adjacency matrix approach was developed in [72], by Price et al., as a way to circumvent the drawback of the previous approach, i.e., to recognize rotated, but otherwise equal solutions. The reasoning behind this approach is to use an adjacency matrix to represent the solutions, instead of vectors. Using this latter

representation, the difference between two vectors could not be zero, albeit the two vectors represented the same solution ($\mathbf{x} = [1, 2, 4, 5, 3]^T$ and $\mathbf{y} = [5, 4, 2, 1, 3]^T$ represent the same solution). If the solutions are encoded using adjacency matrices, the representation of both solutions would be the same, and its difference would be zero.

To calculate the difference between two matrices, they used

$$\mathbf{A}_i \oplus \mathbf{A}_j = (\mathbf{A}_i + \mathbf{A}_j) \mod 2$$

to represent the modulo 2 addition, also known as the eXclusive-OR (XOR) logical operator, and then defined the difference matrix as

$$\Delta_{i,j} = \mathbf{A}_i \oplus \mathbf{A}_j \tag{3.2.2}$$

which is analogous to the difference between two vectors in continuous case.

Considering two possible solutions for the TSP, their adjacency matrix could be

$$\mathbf{A}_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \text{ and } \mathbf{A}_2 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix},$$

and calculating the difference between these two matrices, i.e., the difference matrix as defined in equation (3.2.2), would result in

$$\Delta_{1,2} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

The solutions and the respective difference matrix are represented in figure 3.2, where can be seen that, in practice, the difference matrix results only in the edges that existed in one of the matrices but not in both, i.e., the removal of the common elements, which is, obviously, how the logical XOR operator works.

But this approach is not without problems either, as can be seen in figure 3.2(c),

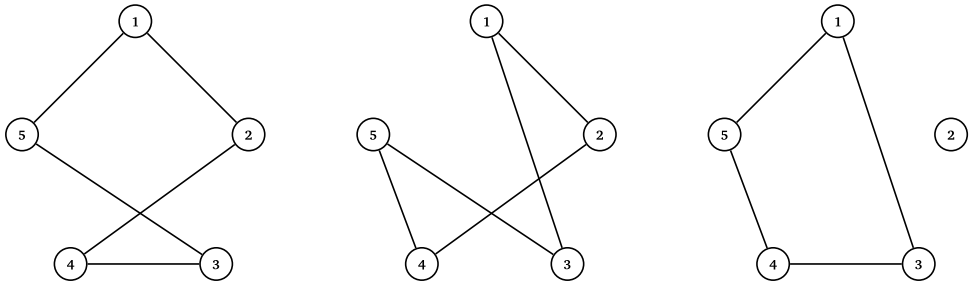


Figure 3.2: A representation of two solutions for the TSP (a)-(b), and the respective difference matrix (c).

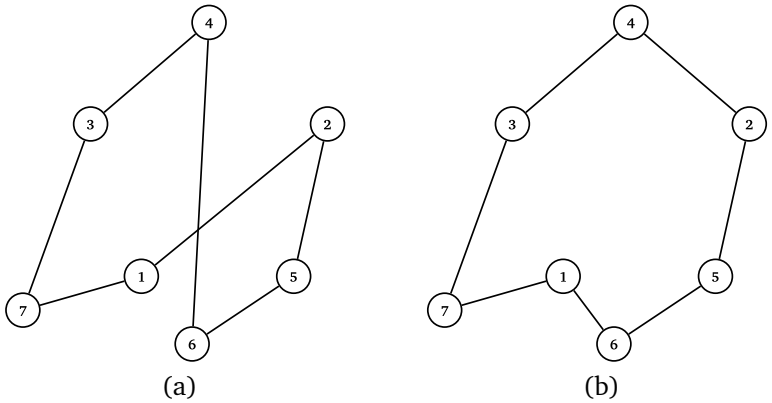


Figure 3.3: Example of a 2-opt move.

where the difference matrix is clearly an unfeasible solution. Although this could be overlooked, as this is but an intermediary result, adding a valid base matrix ($\mathbf{x}_{r1}$, in equation (2.1.3)) to this difference matrix, would, very likely, not result in a valid solution, meaning a repair mechanism must be used to fix this invalid solution. One possible solution given by the authors is discarding this invalid solution, and perform a 2-opt exchange in the base matrix. The 2-opt local search chooses two edges (not incident) to remove, and reconnects the affected four cities, each with the neighbour of the other one, guaranteeing a feasible solution (see figure 3.3).

This approach correctly identifies rotated solutions as equals but, as the authors recognize, most solutions generated are invalid, and the good results obtained using this method are mainly due to the 2-opt repair mechanism, rather than the DE algorithm itself.

3.2.3 Relative position indexing

The basic idea behind the relative position indexing (RPI) approach, as introduced by Lichtblau [48, 49], is to transform the discrete values into floating-point by dividing each discrete value by the maximum, use the classic DE operators as usual, and transform them back to discrete based on their relative position in the vector.

Suppose three possible solution for our five cities TSP were given by

$$\mathbf{x}_1 = \begin{bmatrix} 4 \\ 2 \\ 1 \\ 5 \\ 3 \end{bmatrix}, \quad \mathbf{x}_2 = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 4 \\ 3 \end{bmatrix}, \quad \text{and} \quad \mathbf{x}_3 = \begin{bmatrix} 3 \\ 4 \\ 1 \\ 2 \\ 5 \end{bmatrix}.$$

To transform these solutions into floating-point, each element in each solution must be divided by the maximum value, which is 5 in this case, resulting in

$$\mathbf{x}'_1 = \frac{\mathbf{x}_1}{5} = \begin{bmatrix} 0.8 \\ 0.4 \\ 0.2 \\ 1.0 \\ 0.6 \end{bmatrix}, \quad \mathbf{x}'_2 = \frac{\mathbf{x}_2}{5} = \begin{bmatrix} 0.2 \\ 0.4 \\ 1.0 \\ 0.8 \\ 0.6 \end{bmatrix}, \quad \text{and} \quad \mathbf{x}'_3 = \frac{\mathbf{x}_3}{5} = \begin{bmatrix} 0.6 \\ 0.8 \\ 0.2 \\ 0.4 \\ 1.0 \end{bmatrix}$$

that would be used in the classic DE operators. For instance, suppose a scale factor $F = 0.8$, applying the mutation operator, as defined in equation (2.1.3), to these solutions, would result in

$$
\begin{aligned}
\mathbf{v}'_i &= \mathbf{x}'_1 + F \cdot (\mathbf{x}'_2 - \mathbf{x}'_3) \\
&= \begin{bmatrix} 0.8 \\ 0.4 \\ 0.2 \\ 1.0 \\ 0.6 \end{bmatrix} + 0.8 \cdot \left(\begin{bmatrix} 0.8 \\ 0.4 \\ 0.2 \\ 1.0 \\ 0.6 \end{bmatrix} - \begin{bmatrix} 0.6 \\ 0.8 \\ 0.2 \\ 0.4 \\ 1.0 \end{bmatrix} \right) \\
&= \begin{bmatrix} 0.8 \\ 0.4 \\ 0.2 \\ 1.0 \\ 0.6 \end{bmatrix} + 0.8 \cdot \begin{bmatrix} -0.4 \\ -0.4 \\ 0.8 \\ 0.4 \\ -0.4 \end{bmatrix} = \begin{bmatrix} 0.48 \\ 0.08 \\ 0.84 \\ 1.32 \\ 0.28 \end{bmatrix}.
\end{aligned}
$$

This mutant vector could then be transformed to the discrete domain by, starting with the lowest value (0.08 in this example), assigning to its position (0.08 is in the second position in the vector) the label 1, then assign to the position of the second lowest (0.28) the label 2, and so on and so forth, resulting in the vector $\mathbf{v}_i = [3, 1, 4, 5, 2]^T$. This transformation would always return a valid individual, except if two floating-point values are equal. In this case some reconstruction would be needed, or the vector would be discarded and another one would be calculated.

Also, unlike in the real domain, where any minor perturbation to a solution would not produce any major changes, here, any minor perturbation in the floating-point vector could result in a completely different solution, for instance, changing the previous v_i solution by adding to it $[0.21, 0.11, -0.2, 0, -0.1]^T$, and then converting to discrete values, would produce $[4, 2, 3, 5, 1]^T$, which is a completely different solution.

Although this approach allows the usage of DE operators without modifications, it basically shuffles the elements creating a new permutation and also fails to identify rotated but otherwise equal solutions ($[0.71, 0.54, 0.47, 0.29, 0.34]^T$ and $[0.96, 0.15, 0.67, 1.42, 1.37]^T$ are completely different, when but when translated, produce the same solution, albeit rotated).

3.2.4 Forward/backward transformation

The forward/backward transformation (FBT), also known as the Onwubolu's approach [66, 68, 67, 69] is another method to use DE in combinatorial optimization. As the previous method, this also implements a transformation of the discrete values into floating-point, apply the DE operators in the real domain, and convert them back to discrete.

As the name implies, the method consist of two steps: the first is the *forward transformation*, where the discrete solutions are converted in floating-point values by applying

$$\mathbf{x}'_i = -1 + \alpha \mathbf{x}_i \tag{3.2.3}$$

where α is a small number. In [68] the author suggested the value $\alpha = \frac{500}{10^3 - 1}$ as a good ratio.

Suppose the TSP example given previously, where one of the solutions was $\mathbf{x} = [4, 2, 1, 5, 3]^T$. Applying equation (3.2.3) to the first value 4, we would get $-1 + \frac{500}{10^3-1} \cdot 4 = 1.002002$, and applying it to all values would result in the vector $x' = [1.002002, 0.001001, -0.499499, 1.502503, 0.501502]^T$.

After this transformation, the usual DE operators are applied, using the real vectors, and, as the discrete values are needed to evaluate the solution, the trial vector, resulting from these operators, must be transformed back to discrete values. This is done in the second step, the *backward transformation*, which transforms the floating-point vector back to discrete values, according to

$$\mathbf{u}_i = \text{round}\left((1 + \mathbf{u}'_i)\, \alpha^{-1} \right). \tag{3.2.4}$$

Suppose that after applying the DE operators, we got the trial vector $\mathbf{u}' = [0.901902, -0.099099, -0.299299, 1.602603, 0.401401]^T$. As this cannot be directly evaluated, it needs to be converted back to discrete using equation (3.2.4). Converting the first value 0.901902, would result in

$\text{round}\left((1+0.901902)\times\left(\frac{500}{10^3-1}\right)^{-1}\right) = 4$, meaning the first value is 4. Applying equation (3.2.4) to the rest of the elements would result in the vector $\mathbf{u} = [4, 2, 1, 5, 3]^T$.

Although the backward transformation returns integer values, it does not guarantee to always return valid solutions. For instance, transforming the vector $[0.2, 0.5, 1.2, 1.8, 0.3]^T$, back to discrete would produce $[2, 3, 4, 6, 3]^T$, which is not a valid solution to our problem, as not all nodes are represented in the path, and some of them are repeated. The invalid solutions need to be discarded or fixed using some repairing operations, as suggested in [68, 67, 69]. Also, it fails to identify rotated, but otherwise equal, solutions, as the RPI approach.

3.2.5 Sub-range encoding

In 2006, Nearchau and Omirou proposed the sub-range approach [64] to transform floating-point values to a discrete domain. This approach is based on the random-key encoding proposed by Bean [8], but used an interval to transform the floating-point values. In a d-dimensional problem, the vector $[1, 2, \ldots, d]^T$ would be divided into d intervals, called sub-ranges, by dividing each value by d, defining, this way, the upper value for each range, in the form

$$\mathbf{SR} = \begin{bmatrix} 1/d \\ 2/d \\ 3/d \\ \cdots \\ d/d \end{bmatrix}.$$

Suppose our five cities TSP, where each city is labeled 1 through 5. To create the sub-range vector, first create a vector with the values 1 through 5, and then divide each value by 5 (the dimension of the problem), resulting in $\mathbf{SR} = [0.2, 0.4, 0.6, 0.8, 1.0]^T$. Using this approach, a initial discrete solution would be translated to a floating-point value by generating random numbers, each in the range defined by the respective discrete value. Suppose the solution

$\mathbf{x} = [3, 4, 1, 2, 5]^T$: As the first value is 3, it defines the index of the maximum value for the random number, in this case the maximum value is 0.6. A random value is now generated between $[0.4; 0.6]$, say 0.46, and this is the floating-point number to represent the discrete value 3. Applying this to all discrete values would result in the floating-point vector. Algorithm 3.2 represents this procedure.

Algorithm 3.2 Translate a discrete solution *x* into a floating-point vector *fpv*, using the sub-range encoding.

Input: *x, SR*
Output: *fpv*

```
1: n ← sizeof( x )
2: for i = 1..n do
3:    max ←SR[ x[ i ] ]
4:    if x[ i ] = 1 then
5:       min ← 0
6:    else
7:       min ← SR[ x[ i ] -1 ]
8:    end if
9:    fpv[ i ] ← rand( min, max )
10: end for
```

After applying the usual DE operators, the trial vector need to be converted back to the discrete domain, to be evaluated. The procedure is represented in algorithm 3.3, and goes like this: For each element of the floating-point vector, search the sub-range vector for the lowest value in it still higher than the floating-point element. The discrete value is the index of the value found. Suppose the trial vector $\mathbf{u}'_i = [0.58, 0.97, 0.14, 0.47, 0.39]^T$. Taking the first element, 0.58, and searching in **SR**, for the lowest value higher than 0.58 is 0.6, which is the third element of **SR**, so the first discrete element is 3. Doing the same for the second element, 0.97, would result in the discrete value 5, and so on, until the final result would be $\mathbf{u}_i = [3, 5, 1, 3, 2]^T$. Although being discrete and in the correct range of values, there could exists duplicated values, as in this case. To fix this unfeasible solution, the authors proposed a simple fix, consisting in removing the duplicated values (maintaining the first appearance), and randomly selecting unused elements to each empty slot. In the previous example, we would delete the

duplicated 3 $\left(\mathbf{u}_i = [3, 5, 1, \_, 2]^T\right)$, and fill the empty position with a randomly select unused element. In this case the only unused element is 4, so the final solution would be $\mathbf{u}_i = [3, 5, 1, 4, 2]^T$.

Algorithm 3.3 Translate a floating-point vector *fpv* into a discrete solution *x*, using the sub-range encoding.

Input: *fpv*, *SR*
Output: *x*
1: $n \leftarrow$ sizeof(*fpv*)
2: $d \leftarrow$ sizeof(*SR*)
3: **for** $i = 1..n$ **do**
4: $k \leftarrow \min_k$(*fpv*[*i*] $\leq$ *SR*[*k*]), $\forall_{k=1..d}$
5: *x*[*i*] $\leftarrow k$
6: **end for**

As this approach uses floating-point values and only decodes them to discrete values when the fitness value is needed, we can apply the usual operators from DE, but otherwise, this approach is another shuffle generator, suffering from the same problem as the previous two, as it fails to recognize rotated solutions.

3.2.6 Discrete set handling

Another approach is the discrete set handling (DSH), introduced by Lampinen and Zelinka in 1999 [46] as a mixed integer-discrete-continuous optimization method, but expanded further and named discrete set handling by Zelinka in 2009 [92]. In this approach each solution is represented by the indexes of the respective discrete element, instead of the discrete values themselves. Suppose another five cities TSP, where the cities are named A, B, C, D, E, but could be $1, 2, 3, 4, 5$, or any other symbol. In table 3.1 we can see the index for each of our cities. Suppose we had the solution $A - E - D - B - C$, using the respective indexes for each city, this solution would be encoded as $\mathbf{x}_i = [1, 5, 4, 2, 3]^T$.

But as classic DE were not defined for integer domain problems, some changes need to be done to the algorithm to surpass this. According to the authors, its simple to use integer values with DE: first, the initial population should be cre-

city	A	B	C	D	E
index	1	2	3	4	5

Table 3.1: Five cities, with their respective indexes, for the discrete set handling.

ated, using whatever domain the problem is defined upon, and then it should be encoded using the respective indexes, as demonstrated earlier. Then the classic DE operators are used, for the mutation and crossover, generating floating-point values in the process. This values must then be converted back to valid indexes, by truncating the floating-point values to its nearest integer value. If out-of-range values exist, they should be replaced by random generated values, within the defined boundaries; and if some indexes are repeated, they must be repaired by removing the repeated indexes and randomly replacing them with the missing indexes. This convert/repair procedure is shown in algorithm 3.4, where the first cycle converts the floating-point values to integer, leaving "empty" positions when a duplicated value is found, and the second randomly selects the missing indexes and inserts them in the "empty" positions. Line 15 find the missing indexes by removing the indexes already in the solution from a set of all indexes, and then shuffles them, to randomize the list.

Suppose the trial vector $\mathbf{u}'_i = [1.25, 7.68, 3.98, 4.47, 1.78]^T$. Applying the method in algorithm 3.4, value 1.25 would be truncated to 1, and placed in the first position of the discrete solution, then 7.68, as its greater them the maximum value, which is 5, is replaced by a random value, between 1 an 5, suppose 3, and so on, until, after the first cycle, the result could be $\mathbf{x} = [1, 3, \_, 4, \_]^T$. The missing indexes $2, 5$ are then randomly placed in the missing positions of the solution, and in the final a possible feasible discrete solution could be $\mathbf{x}_i = [1, 3, 5, 4, 2]^T$.

As others approaches presented previously, this approach also fails to recognize rotated, but otherwise equal, solutions, and, as it doesn't take into account the underneath problem and its combinatorial design, is another shuffle generator.

Algorithm 3.4 Convert/repair floating-point vectors using discrete set handling.

Input: *fpv*, *min*, *max*
Output: x

```
1: I ← ∅
2: for i = 1..sizeof( fpv ) do
3:    v ← truncate( fpv[ i ] )
4:    // control out-of-range values
5:    if v < min or v > max then
6:       v ← truncate( rand( min, max ))
7:    end if
8:    // if the value is not repeated
9:    if v not in x then
10:      x[ i ] ← v
11:   else
12:      I ← I ∪ i                          //save indexes without values
13:   end if
14: end for
15: missing ← { min, ..., max } \ x  // defining the missing indexes
16: suffle( missing )                        // shuffle the missing set
17: for i = 1..sizeof( I ) do
18:   x[ i ] ←pop( missing )       // removes the first item from missing
19: end for
```

3.2.7 Differential list of movements

Most of the previous approaches, basically, shuffe the order of the elements in the solutions, without taking in consideration the core aspect of DE, which is the difference between vectors in a $\mathbb{R}^n$ space. In 2010, Prado et al. introduced the differential list of movements (DLM) [71], in which they presented a new idea to express the difference of two solutions in the discrete space.

They define the difference between two solutions as a list of exchanges that must be made in the sequence of the elements in a solution, to transform it in the other solution. The differential list of movements is defined by

$$M_{j \to i} = \mathbf{x}_i \ominus \mathbf{x}_j, \tag{3.2.5}$$

where $\ominus$ is a binary operator that receives two solutions and returns a list of movements representing the transformation from $\mathbf{x}_j$ to $\mathbf{x}_i$. To scale this list, it needs to be "multiplied" by the scale factor F, defined by

$$M'_{j\to i} = F \otimes M_{j\to i}, \tag{3.2.6}$$

where $\otimes$ is a binary operator, that receives a list of movements and a constant and returns $n = \lceil F \cdot |M_{j\to i}| \rceil$ elements of the initial list, with $|\cdot|$ being the size of the list. The idea behind this operator is to select F percent of the values from the list of movements $M_{j\to i}$, meaning, for instance, if $F = 0.5$ and the list of movements have 10 elements, the resulting of $F \otimes M_{j\to i}$ would be a list with 5 elements (50% of the original list). Prado et at. gave several definitions for the product of a list of movements by a constant $F \in [0, 1]$, refer to [71] for a formal definition of them all.

Applying the scaled list of movements to a given solution $\mathbf{x}_k$ would swap the elements of $\mathbf{x}_k$ using the given list, and is defined by

$$\mathbf{x}'_k = \mathbf{x}_k \oplus M'_{a\to b} \tag{3.2.7}$$

where $\oplus$ is another binary operator that receives one solution and a list of movements and by applying the movements defined in the list to the initial solution, results in a transformed solution.

Using equations (3.2.5)-(3.2.7),we could define a new mutation operator for DE, as

$$\mathbf{v}_i = \mathbf{x}_{r1} \oplus F \otimes (\mathbf{x}_{r2} \ominus \mathbf{x}_{r3}), \tag{3.2.8}$$

and algorithm 3.5 present the whole procedure.

Lets illustrate these operators with an example: suppose $F = 0.5$ and the solutions

Algorithm 3.5 Mutation operator, using a differential list of movements.

Input: x_1, x_2, x_3, F
Output: v

```
// Create the differential list of movements between x_2 and x_3
dlm ← ∅
n ←sizeof( x_2 )
for i = 1..n do
    j ← {j: x_{2,i} = x_{3,j}, ∀_{j=1..n} }        // position of the ith element of x_2, in x_3
    if i ≠ j then
        dlm ← dlm ∪ ( i, j )                       // insert the indices for the swap
        swap( x_3, i, j )                          // swaps the elements in x_3
    end if
end for
// scale the differential list of movements
qt ← ceiling( F * sizeof( dlm ))
sdlm ← sample( dlm, qt )// Removes qt random elements from dlm
// Add the scaled list to the base solution
v ← x_1
for k = 1..sizeof( sdlm ) do
    ( i, j ) ← sdlm( k )
    swap( v, i, j )
end for
```

$$\mathbf{x}_{r1} = \begin{bmatrix} 4 \\ 2 \\ 1 \\ 5 \\ 3 \end{bmatrix}, \quad \mathbf{x}_{r2} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}, \quad \text{and} \quad \mathbf{x}_{r3} = \begin{bmatrix} 5 \\ 1 \\ 4 \\ 2 \\ 3 \end{bmatrix}.$$

First, calculating $\mathbf{x}_{r2} \ominus \mathbf{x}_{r3}$, would result in a list of exchanges necessary to transform $\mathbf{x}_{r3}$ into $\mathbf{x}_{r2}$ in the following way: The first element of $\mathbf{x}_{r2}$ is 1, which is in the second position in $\mathbf{x}_{r3}$. So we need to exchange the first and the second element of $\mathbf{x}_{r3}$, creating the list $M_{r3 \to r2} = \{(1, 2)\}$, and the intermediate vector would be $\mathbf{x}'_{r3} = [1, 5, 4, 2, 3]^T$. Next in $\mathbf{x}_{r2}$ is 2, which is in the fourth place in $\mathbf{x}_{r3}$, so we need to swap the second with the fourth element, adding it to the list of movements $M_{r3 \to r2} = \{(1, 2), (2, 4)\}$, and resulting in a new $\mathbf{x}'_{r3} = [1, 2, 4, 5, 3]^T$. Applying this throughout all $\mathbf{x}_{r3}$ would produce $M_{r3 \to r2} = \{(1, 2), (2, 4), (3, 5), (4, 5)\}$. Now for scaling this list, $F \otimes M_{r3 \to r2}$, F percent elements of $M_{r3 \to r2}$ are randomly selected. Since $F = 0.5$, the result could be $M'_{r3 \to r2} = \{(3, 5), (2, 4)\}$. Finally, applying these

movements to $\mathbf{x}_{r1}$ would result in exchanging the third with the fifth element, resulting in the intermediate solution $\mathbf{x}'_{r1} = [4, 2, 3, 5, 1]^T$, and then the second with the fourth, resulting in the final mutant vector $\mathbf{v}_i = [4, 5, 3, 2, 1]^T$.

This approach doesn't need any repair mechanism and it actually makes DE operators aware of the problem, but as others, it fails to identify rotated, but otherwise identical solutions, as the difference between two equal, but rotated, solutions is not an empty list of movements, as it should if it recognized them as equals.

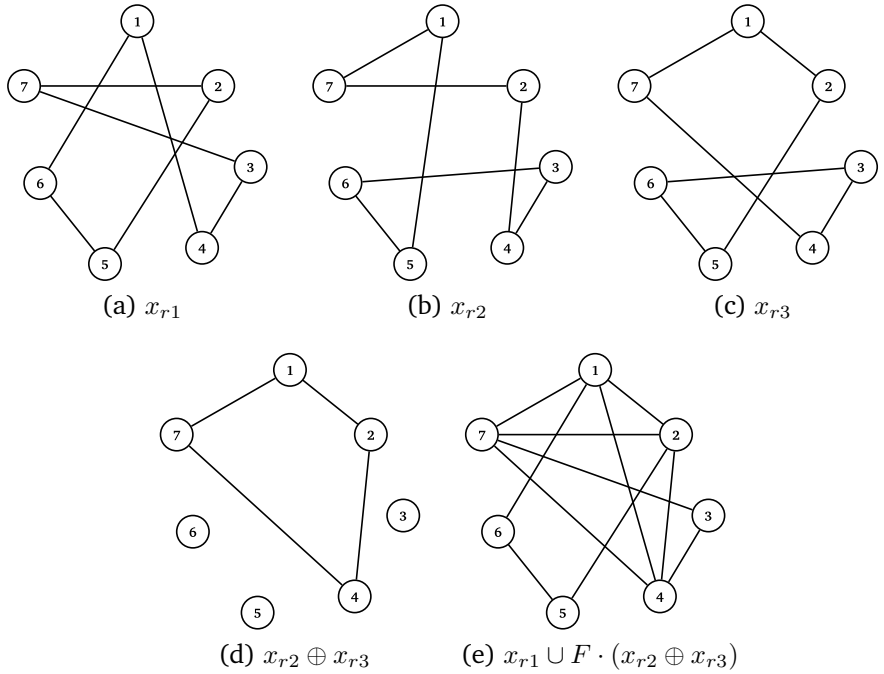
3.2.8 Set-based approaches

In 2013, 4 different groups of authors proposed different set-based approaches to use differential evolution in combinatorial optimization problems. Maravilha et al. [55, 56] proposed an approach in which the arithmetic operators in equation (2.1.3) are replaced by set operators:

$$v_i = x_{r1} \cup F \cdot (x_{r2} \oplus x_{r3}) \tag{3.2.9}$$

where $\oplus$ represents the eXclusive-OR logical operator, or set symmetrical difference, $\cup$ in the union if two sets, and the multiplication used the definitions from Prado et al. [71], for the DLM approach. This process is represented if figure 3.4, where in (a) to (c) are three possible solutions, (d) is the difference between solutions (b) and (c), and the final mutant solution is in (f).

As seen in the example, the resulting mutant solution, most of the times, is unfeasible, and the authors proposed a repair mechanism incorporated in the crossover operator: Basically, the crossover would consist in solving the subproblem defined in $v_i \cup x_i$, where x_i is the corresponding individual in the population. In practice, this means the mutation operator is used to create an "easier" problem, by removing edges from the original TSP domain, and then an exact method is used to solve this "easier" problem, creating a valid trial individual in



(a) x_{r1} (b) x_{r2} (c) x_{r3}

(d) $x_{r2} \oplus x_{r3}$ (e) $x_{r1} \cup F \cdot (x_{r2} \oplus x_{r3})$

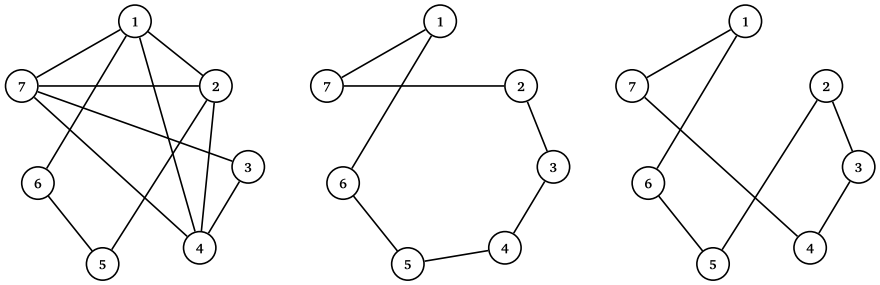
Figure 3.4: Illustration of the set-based proposal by Maravilha et at. [55]. In (a)-(c) are three possible solutions for a seven cities TSP. The differential set $x_{r2} \oplus x_{r3}$ is represented in (d), and assuming, for simplicity, $F = 1$, the final result of the mutation operator is represented in (e). In (d) is represented the complete domain for this seven cities TSP.

the process. The process is illustrated in figure 3.5.

Liu and Maeda [51, 52] proposed an approach where they use a mixture of set operators and redefined arithmetic ones to create the mutant vector, according to

$$v_i = \omega \times x_{r1} + \mathrm{rand}() \times (x_{r2} - x_{r3}) \tag{3.2.10}$$

where ω and $\mathrm{rand}() \in [0; 1]$ are random values, generated for each element, the$-$ operator is the set relative complement (or set difference), $\times$ is defined as an assignment of a probability to an element, and $+$ is similar to the union operator, but if two elements are equal but with different probabilities, it selects the one with higher probability. Basically, the mutation operator works with each element being defined by something like $0.6(1, 2)$, meaning the element $(1, 2)$ with a probability to be chosen of 0.6. For instance, if $\omega \times x_{r1} = \{0.4\,(1, 2)\,, 0.6\,(3, 4)\,, 0.2\,(1, 4)\,, 0.5\,(2, 3)\}$ and $\mathrm{rand}() \times (x_{r2} - x_{r3}) = \{0.2\,(1, 3)\,, 0.8\,(2, 3)\}$, the result of the "ad-



(a) Mutant solution v_i (b) Current solution x_i (c) Trial solution u_i

Figure 3.5: Illustration of the crossover for the set-based proposal by Maravilha et at. [55]. In (a) is the mutant solution obtained in the mutation operator, in (b) is a possible solution in the current population, and in (c) is a possible trial solution, obtained from the edges in (a) and (b).

dition" operator would be $\{0.4(1,2), 0.6(3,4), 0.2(1,4), 0.2(1,3), 0.8(2,3)\}$. In every generation there will be defined a random probability $\alpha \in [0;1]$ to each individual. Using this probability, every element in the respective individual will enter a tournament, and if the element's probability is not smaller than α, the element is reserved to construct the mutant, otherwise is discarded.

To construct the mutant individual, elements from the reserved set are selected based on their probability. If a feasible solution is not reached and there are no more elements in the reserved set, choose the missing elements using an heuristic, for instance, the nearest neighbour.

Liu et al. [53] proposed another set-based approach, where they divided the domain E of the combinatorial optimization problem in a d-tuple $(E^1, ..., E^d)$, where each $E^i,\ i \in \{1, ..., n\}$ is the domain of the ith dimension of the search space, and d is the dimension of the search space. Each solution x for the problem is also a d-tuple $\left(x^1, ..., x^d\right)$, where $x^i \in E^i$. For instance, in a TSP, the domain E of the problem are all edges connecting two nodes, each E^i is composed by the edges that are connected to node i, and each x^i is composed by the two incident edges to node i. Then they redefine the mutation operator as

$$v_i = x_{r1} + F \times (x_{r2} - x_{r3}) \tag{3.2.11}$$

where the $-$ operator is the set relative complement, and $\times$ is a binary operator between a constant parameter F and the differential set, resulting in a scaled differential set, defined by

$$SDS^j = F \times DS^j = \begin{cases} DS^j & \text{if } \text{rand}(0,1) < F \\ \varnothing & \text{otherwise} \end{cases}, j = 1, ..., d, \tag{3.2.12}$$

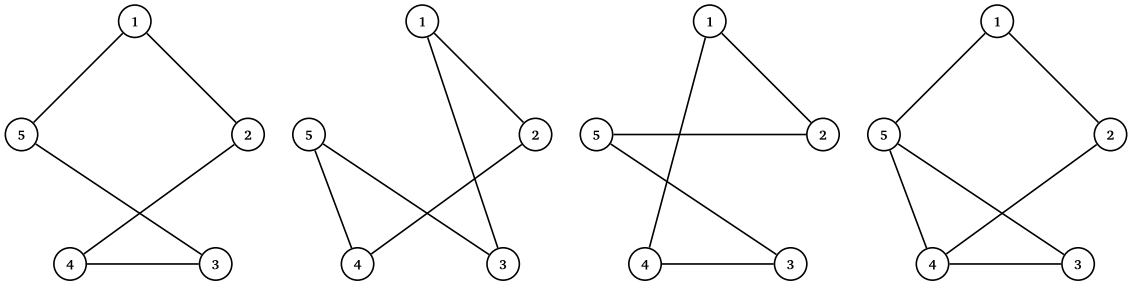
where $DS^j \subseteq x_{r2}^j$ is the j-dimension of the differential set. The scaled differential sets SDS resulting from the previous operation are then "added" to the base solution, using

$$v_i^j = x_{r1}^j + SDS^j = \begin{cases} x_{r1}^j & \text{if } SDS^j = \varnothing \\ SDS^j & \text{otherwise} \end{cases}, j = 1, ..., d. \tag{3.2.13}$$

Suppose a complete graph with five nodes, representing the domain of the TSP. The domain E would be all edges in the graph, and each $E^1, ..., E^i, ..., E^5$ would be composed by the edges incident to node i. If the three selected solutions for the mutation are those represented in figure 3.6, their representation would be

$$x_{r1} = \left\{ \begin{array}{c} (1,2) \\ (1,5) \\ (2,4) \\ (3,4) \\ (3,5) \end{array} \right\}, \quad x_{r2} = \left\{ \begin{array}{c} (1,2) \\ (1,3) \\ (2,4) \\ (3,5) \\ (4,5) \end{array} \right\} \text{ and } \quad x_{r3} = \left\{ \begin{array}{c} (1,2) \\ (1,4) \\ (2,5) \\ (3,4) \\ (3,5) \end{array} \right\},$$

and for each, of them, x_k^i $k \in \{1, ..., 3\}$ would be, respectively,



(a) x_{r1} (b) x_{r2} (c) x_{r3} (d) $x_{r1} + F \times (x_{r2} - x_{r3})$

Figure 3.6: Three possible solutions for a TSP with five cities (a)-(c), and the resulting mutation solution (d).

$$
\begin{array}{lll}
x^1_{r1} = \{(1,2),(1,5)\} & x^1_{r2} = \{(1,2),(1,3)\} & x^1_{r3} = \{(1,2),(1,4)\} \\
x^2_{r1} = \{(1,2),(2,4)\} & x^2_{r2} = \{(1,2),(2,4)\} & x^2_{r3} = \{(1,2),(2,5)\} \\
x^3_{r1} = \{(3,4),(3,5)\}\ , & x^3_{r2} = \{(1,3),(3,5)\}\ , \text{ and} & x^3_{r3} = \{(3,4),(3,5)\}\ . \\
x^4_{r1} = \{(2,4),(3,4)\} & x^4_{r2} = \{(2,4),(4,5)\} & x^4_{r3} = \{(1,4),(3,4)\} \\
x^5_{r1} = \{(1,5),(3,5)\} & x^5_{r2} = \{(3,5),(4,5)\} & x^5_{r3} = \{(2,5),(3,5)\}
\end{array}
$$

Calculating the differential set, i.e., $DS = x_{r2} - x_{r3}$, results in $DS = \{(1,3),(2,4),(4,5)\}$, with $DS^1 = DS^3 = \{(1,3)\}$, $DS^2 = \{(2,4)\}$, $DS^4 = \{(2,4),(4,5)\}$ and $DS^5 = \{(4,5)\}$. Applying equation (3.2.12), assuming $\text{rand}_j(0,1) = \{0.6, 0.2, 0.7, 0.3, 0.5\},\ \forall_{j \in \{1..5\}}$, and $F = 0.5$, the respective scaled differential set would be

$$
\begin{array}{l}
SDS^1 = \emptyset \\
SDS^2 = \{(2,4)\} \\
SDS^3 = \emptyset \\
SDS^4 = \{(2,4),(4,5)\} \\
SDS^5 = \emptyset
\end{array}
\quad ,
$$

and by applying equation (3.2.13), the final mutant solution would be $v = \{(1,2),(1,5),(2,4),(3,4),(3,5),(4,5)\}$, represented in figure 3.6(d).

As the resulting mutant solution is not a feasible solution, must be fixed in the

crossover operator, defined by

$$u_i^j = \begin{cases} \text{learn\_from}\left(v_i^j\right) & \text{if } rand_j(0,1) < CR \text{ or } j = j_{rand} \\ \text{learn\_from}\left(x_i^j\right) & \text{otherwise} \end{cases} \quad j = 1, ..., d \qquad (3.2.14)$$

where j_{rand} is a random integer $\in \{1, ..., d\}$, $rand_j\,(0,1)$ is a random value defined for each dimension j, and the learn procedure is defined by selecting elements from the given set $(\text{either } v_i^j \text{ or } x_i^j)$, or from the respective sub-domain E^j, if no more elements are available to create a valid solution.

All these approaches correctly identify rotated solutions, as the set operator used to defined the difference between two solutions always returns empty if they are equal, i.e., if both solutions are the same, and also use the domain of the problem in the new defined operators, but all of them generate unfeasible solutions, and a repair mechanism needs to be used to reach a final, valid, solution.

The fourth set-based approach is the work of this thesis, and will be presented in the next section.

3.3 Set-based operators

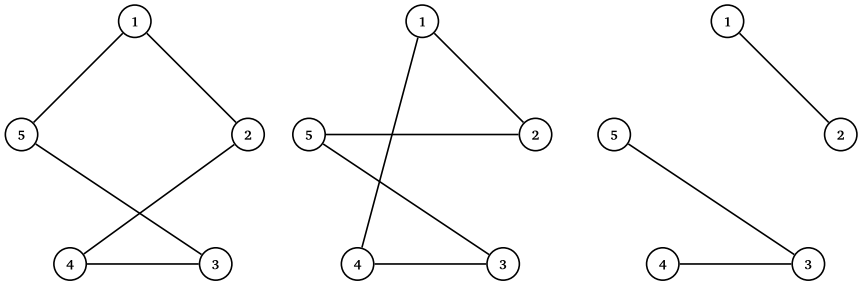
As seen in the previous section, the main problem for using DE for combinatorial optimization problems is also the core of DE optimization process, i.e., the mutation operator. The problem is that the mutation operator defined in equation (2.1.3) have no meaning in combinatorial problems, and almost all approaches given above try to solve the combinatorial problem using the real space, instead of trying to find a discrete domain operators to use in DE. The folowing proposal try to do exactly that.

3.3.1 Representation

As seen previously, the mutation operator defined in equation (2.1.3) cannot be used in combinatorial problems without some sort of adaptation, either by translating the problem to real domain, or by redesigning the operator to work in the problem's domain. The latter approach should be more suitable, as, if done correctly, allows to use the DE steering mechanism in the domain of the problem, as it does with problems in the continuous domain.

Combinatorial optimization problems, by definition, have a discrete, enumerated and finite set as its domain, and if possible, this domain should be used in the problem, not DE's usual real space. And if in the classical DE, defined for continuous domain problems, real-domain arithmetic operations are used, to be possible to consider Sets for the domain of the problem, operations from the set-domain should be used, instead of arithmetic ones. These operations are the union, the intersection, the relative complement (or difference) and the symmetric difference.

Of course this means that the problem must be represented in a way to work with these operations. Considering the TSP as an example of a combinatorial problem, it can be represented in a graph, and the usual way to represent solutions for the problem is to either use the nodes or the edges of the graph as elements of each solution. In literature, is more frequent to see TSP represented as nodes, where each solution consist of the nodes that compose that solution. Two possible solutions for a TSP could be $x_1 = \{4, 2, 1, 5, 3\}$ and $x_2 = \{1, 2, 5, 3, 4\}$, and, for instance, calculating the intersection between these two solutions would result in a set with all nodes, although the solutions are not equal. This happens because Sets, by definition, have no order, and as such, the two solutions given above are one and the same, as they have the same elements (nodes). As this would occur to any solution when using node representation, our approach represents the solutions using edges. In figure 3.7 is illustrated the solutions above using edges, and the resulting intersection operation. The intersection would only result



(a) x_1 (b) x_2 (c) $x_1 \cap x_2$

Figure 3.7: Two solutions for a TSP using edge encoding (a)-(b), and the respective intersection (c).

in a set with all elements from both solutions if both solutions were exactly equal.

3.3.2 Mutation

Considering set-based operations for the mutation operator, the basic idea was to use some type of building blocks between the solutions, and use these building blocks to "construct" a new solution. As seen previously, the intersection between two sets results in the common elements between them, and so this set operator was used to find the afore-mentioned building blocks, defining this way the difference between two solutions as

$$DF = x_{r2} \cap x_{r3},$$

obtaining, this way, the common elements between the two sets. The multiplication operator must be defined between a scalar and a set, and we used one given in [71]:

Definition 3.1. Given a scalar $F \in [0; 1]$ and any set S, the multiplication of a scalar by a set is represented by $F \otimes S$, and this operation results in another set $S' \subseteq S$, containing $\lceil F \times \mid S \mid \rceil$ random elements from S, where $\mid \cdot \mid$ represents the cardinality of the set.

Finally, to replace the addition arithmetic operator, the closer concept in Sets is the union, as this operator will create a new set, with all elements from both sets.

However, the resulting solution would, probably, be an unfeasible solution, as it would have more elements than needed. The repair mechanism would depend on the problem at hand, but for the TSP, could consist in selecting a starting edge from those incident to the nodes with lowest degree, and then select edges incident to the last, using those in $x_{r1} \cup F \otimes (x_{r2} \cap x_{r3})$, until either a feasible solution is found, or no more edges can be added without breaking a problem constraint, in which case this repair is discarded, and the base solution $x_{r1,g}$ is used.

Using these operations, we defined in Guerreiro et al. [32], the set-based mutation operator as

$$v_{i,g} = x_{r1,g} \uplus F \otimes (x_{r2,g} \cap x_{r3,g}) . \tag{3.3.1}$$

Algorithm 3.6 represents this mutation operator, using the above formula, with the proposed repair mechanism. Note that this repair mechanism is not an heuristic to find a TSP on purpose, as the goal is for the work to be done by the mutation, with some help from the repair, not for the repair to do the work and the mutation to be nothing but an excuse to have something to be repaired.

To test if this idea was feasible, it was tested on some instances from the TSPLIB [76], using a scale factor $F = 0.8$ and a crossover $CR = 0.9$, with a population Np twice as much as the dimension of the problem instance used. We used the TSP not with the idea of introducing "the next best thing" to solve the TSP, but because it is the most studied combinatorial optimization problem, and the one usually used to compared results. We also implemented the DLM approach from Prado et al. [71] and the sub-range approach from Nearchau and Omirou [64], with the same parameters, to serve as comparison. As a stopping criteria was defined the full convergence of the population, meaning when all solutions are equal, or when no evolution occurred after 30 generations, meaning all solutions were local optima.

Due to the core functionality of all evolutionary algorithms, the results cannot be taken individually, but rather averaging the results from a number of runs of

Algorithm 3.6 Algorithm for the mutation operator, with a repair mechanism for the TSP problem.

Input: x_1, x_2, x_3, F
Output: v

1: $DF \leftarrow x_2 \cap x_3$
2: $qt \leftarrow$ sizeof(DF) - ceiling($F *$ sizeof(DF))
3: **while** $qt > 0$ **do**
4: remove a random edge from DF
5: $qt \leftarrow qt - 1$
6: **end while**
7: $union \leftarrow x_1 \cup DF$
8: $tail \leftarrow$ random node from those in $union$ with lowest degree
9: $head \leftarrow$ node from $tail$'s neighbours with the lowest degree
10: $v \leftarrow$ { $tail$, $head$ }
11: remove { $tail$, $head$ } from $union$
12: $tail \leftarrow head$
13: $neighbours \leftarrow$ nodes neighbour of $tail$ in $union$
14: **while** $neighbours \neq \emptyset$ **do**
15: $head \leftarrow$ node in $neighbours$ with the lowest degree
16: $v \leftarrow v \cup$ { $tail$, $head$ }
17: remove all edges incident to $tail$ from $union$
18: $tail \leftarrow head$
19: $neighbours \leftarrow$ nodes neighbour of $tail$ in $union$
20: **end while**
21: **if** v is not valid **then**
22: $v \leftarrow x_1$
23: **end if**

	ulysses22			berlin52		
	Generation	Best	Error	Generation	Best	Error
Sub-range	491	10827.3	54.39%	953	22407.4	197.10%
DLM	3002	7016.0	0.04%	15069	9038.2	19.84%
Set-based	126	7054.1	0.59%	213	8703.2	15.40%

Table 3.2: Results for some TSP instances, using $F = 0.8$, $CR = 0.9$ and a population twice as much as the dimension of the problem.

the algorithm. The results shown in table 3.2 are the average of 10 runs for each instance of the problem, always with the same parameters. As can be seen, the results obtained by the sub-range approach cannot be compared neither with the DLM, not with our set-based approach, with always an error higher than 50% of the optimum value. Both DLM and the set-based approach got very close to the optimum on the smaller instance, but DLM did it at the expense of a much higher number of generations, when compared with the set-based approach. A similar result can be observed on the other instance regarding the number of generations, although here the set-based approach reached a better result then the DLM.

From those results we could conclude that this approach was not bad, but some more development were needed, mainly because this operator, has was defined in equation (3.3.1), had some downfalls. In the beginning of the evolutionary process, when, theoretically, all solutions are different, the result of the intersection of any two solutions will be an empty (or almost) set, and the union of any solution with an empty set will result in itself, meaning that the mutation operator will result, most of the times, in the initial base solution $x_{r1,g}$. As the generations evolve, the solutions will, again theoretically, became more and more equal, i.e., will have more common elements as each generation goes by, and the result of an intersection between any two solutions will be a set almost identical to the two initial solutions, and by the same principle, the union of this set with the initial base solution, will result, again, in a solution very close to the base one $x_{r1,g}$, as most elements will be already in it (remember all solutions will be, theoretically, very similar). This means that in the start and at the end of the evolutionary pro-

cess, equation (3.3.1) will result almost always in the base solution, in practice, meaning that there will be no real "mutation" applied.

Another thing to consider is the cardinality of the mutant individual, created by equation (3.3.1). When using any set operation, the one thing one can take for granted is that the cardinality of the resulting set will not be the same as the initial ones. As in most combinatorial problems the cardinality of each solution is fixed, it's easily to see that the mutant solutions resulting from equation (3.3.1) will have a different cardinality than the solutions used to create it. As always when having an unfeasible solution, it can be discarded or repaired. In this case, if the mutation operator results, almost always, in an unfeasible solution, if this solution is discarded, there would never be any evolution, so the other option is to repair this unfeasible solution, and this is probably the reason for the good results, and not the DE algorithm itself.

Improved mutation

As the previous approach had some downfalls, some variations of it were formulated, and studied, to create a more generic approach. An empirical study was made and from several conceivable formulas, those that, in the end, would result in empty, or almost, sets were removed from consideration, and presented bellow are the ones that worth a closer look.

$$v_{i,g} = x_{r1,g} \cap F \otimes (x_{r2,g} \cup x_{r3,g}) \tag{3.3.2}$$

$$v_{i,g} = x_{r1,g} \cap F \otimes (x_{r2,g} \cap x_{r3,g}) \tag{3.3.3}$$

$$v_{i,g} = x_{r1,g} \cup F \otimes (x_{r2,g} \setminus x_{r3,g}) \tag{3.3.4}$$

$$v_{i,g} = x_{r1,g} \cup F \otimes (x_{r2,g} \cup x_{r3,g}) \tag{3.3.5}$$

$$v_{i,g} = x_{r1,g} \cup F \otimes (x_{r2,g} \cap x_{r3,g}) \tag{3.3.6}$$

$$v_{i,g} = x_{r1,g} \cup F \otimes (x_{r2,g} \vartriangle x_{r3,g}) \tag{3.3.7}$$

$$v_{i,g} = x_{r1,g} \setminus F \otimes (x_{r2,g} \setminus x_{r3,g}) \tag{3.3.8}$$

$$v_{i,g} = x_{r1,g} \setminus F \otimes (x_{r2,g} \vartriangle x_{r3,g}) \tag{3.3.9}$$

In all formulas, $\vartriangle$ represents the symmetric difference $x_{r2,g} \vartriangle x_{r3,g} = (x_{r2,g} \setminus x_{r3,g}) \cup (x_{r3,g} \setminus x_{r2,g})$ and the definition for $\otimes$ has been given previously. The rest are the usual set operators, union, intersection and relative complement. To simplify, lets ignore the generation index g, and consider $F = 1$, this way removing this parameter from the subsequent analysis.

Inspecting equations (3.3.2)-(3.3.9), two situations can be seen: (3.3.4) to (3.3.7), due to their outer union operator, would end up a cardinality higher than the desired one, and the other equations would end up with a lower cardinality, because of the outer intersection and relative complement operators. Either way, some repairing mechanism would almost always be needed (except when the solutions are exactly the same), when using set-based operations, but this effect is should be minimum the closer to the desired cardinality the mutant solution is.

Considering the beginning of the evolution, where almost all solutions are different from one another, equations (3.3.2) and (3.3.3) would result in a close to zero cardinality, as the intersection of a base solution x_{r1} with any other (very different) solutions, would result empty, and due to this, the repair mechanism is a

major factor in the initial evolutions, using these formulas. As the evolution goes by, the initial (almost) empty solution would get closer to the base solution x_{r1}, and in the end of the evolution stage, the resulting mutant solution's cardinality would be very close to the desired one, because, near the end of the evolution, all solutions would be (almost) equal, and when this happens, equation (3.3.2) and (3.3.3), result in a solution very close to x_{r1}, although with a lower cardinality (except if the three solutions are exactly the same). In this case, the repair mechanism would have a minor effect, as it wouldn't have much to repair.

Equations (3.3.4), (3.3.5) and (3.3.7), as stated above, will always end up with a higher cardinality than any of the initial used solutions, but a closer analysis results in two possible initial cases: for equations (3.3.5) and (3.3.7), if all solution are mostly different, then $|\,(x_{r2,g} \triangle x_{r3,g})\,| \leq |\,(x_{r2} \cup x_{r3})\,|$, because if x_{r2} is mostly different from x_{r3}, than $(x_{r2} \setminus x_{r3})$ would result similar to x_{r2}, and $(x_{r3} \setminus x_{r2})$ would result similar to x_{r3}, and from this, $(x_{r2,g} \triangle x_{r3,g})$ would be similar to $(x_{r2} \cup x_{r3})$, so their cardinality would be almost double of the desired. Due to this, equations (3.3.5) and (3.3.7) would be, in fact, closer to $v_i = (x_{r1} \cup x_{r2} \cup x_{r3})$, and the resulting solution's cardinality would be almost three times as much as each solution. The second initial case is equation (3.3.4), where the set relative complement $(x_{r2} \setminus x_{r3})$ would result in a cardinality slightly lower than the one of x_{r2}, and this make the resulting final union to have a cardinality double of the desired one. In either case, in the beginning of the evolution, the resulting mutant solution has a cardinality that is three or two times higher, which means the repair mechanism must be used to create feasible solutions. Towards the end of the evolution, when all solution should be almost identical, another two cases are possible: in equation (3.3.5), the inner union operator $(x_{r2} \cup x_{r3})$ would result in a cardinality slightly higher to either of them, and when joining this to the base solution using the outer union, the mutant solution would have, again, a slightly higher cardinality than any of then; for the second case, consider equations (3.3.4) and (3.3.7). If all solutions are almost equal, $\varnothing \subseteq (x_{r2} \setminus x_{r3}) \subseteq (x_{r2,g} \triangle x_{r3,g})$, because the elements

that are in one solution and not the other are close to zero, and so the resulting set is almost empty, meaning their cardinality is almost zero. The union of any solution with an (almost) empty set results in a solution very similar to the initial one, and as such, these equations would return a mutant solution that is close the base solution x_{r1}, but with a cardinality somewhat higher. In both cases, the final mutant solution would be very similar to the base solution x_{r1}, but with a higher cardinality, and this means that although some repair is needed, its effect would be minimum, as the mutant solutions wouldn't need much repair.

Finally equations (3.3.6), (3.3.8) and (3.3.9). In the start of the evolutionary process, the result is a mutant solution with a cardinality close to the one of the base solution x_{r1}, either because: 1) in (3.3.6) the union of any solution with an almost empty set (resulting from the inner intersection), produces a solution with a cardinality slightly higher than the initial solution; or 2) in (3.3.8), the relative complement $(x_{r2} \setminus x_{r3})$, results in a cardinality a little lower than $|x_{r2}|$, and then the final operation results in a solution with a cardinality slightly lower than the one in x_{r1}; or finally 3) in (3.3.9) the inner symmetric difference $(x_{r2} \setminus x_{r3}) \cup (x_{r3} \setminus x_{r2})$ would result in a solution similar to $(x_{r2} \cup x_{r3})$ (see justification for (3.3.7), above), and the final relative complement would return a cardinality slightly lower than $|x_{r1}|$. In all possible cases, the mutant solution would a cardinality very close to the desired one, which means, that the repair mechanism wouldn't have much effect here. Towards the end of the evolutionary process, when all solution are very similar to one another, again all of them would result in an mutant solution with a cardinality close to the base solution x_{r1}, due to: 1) in (3.3.6), the inner intersection would result in a cardinality close to, although slightly lower, either solution, and then the outer union would return a cardinality a little higher than the one of x_{r1}; or 2) in (3.3.8) and (3.3.9), the inner operations would result in a close to zero cardinality because when two solution are almost equal, there are very few (if any) elements not in both of them, and the cardinality of a relative complement of an almost empty set in any set,

results in the cardinality of this latter set, in this case, the base solution x_{r1}. Again, in all cases, the cardinality of the mutant solution is very close to the desired one, which means that the repair mechanism would not have much impact. However, this three formulas behave differently from the others in the main core of the evolutionary process. All the others would either start with a high cardinality and would descend into the desired one, or start with a very low cardinality and would ascend into the desired one, but these three start close to the desired cardinality but then, (3.3.8) and (3.3.9) will lower it, because when the solution start getting somewhat similar, those formulas would remove some elements from the base solution, resulting in a lower cardinality, or in (3.3.6), as some elements will be added to the base solution due to the union operator, the resulting solution will have a higher cardinality. In a nutshell, these three formulas start with an acceptable cardinality, then will either lower or raise it, hopefully not much, and finally, end with an acceptable cardinality, making, in theory, these three formulas those that will have the least influence on the repair mechanism in the mutant solution. All this analysis is resumed in table 3.3, where are presented the expected cardinality of each solution in the beginning and in the end of the evolutionary process, a well as the general evolution of the cardinality in between the two extremes.

In figure 3.8 are represented the actual measured cardinality for two instances of the TSPLIB. These results are the average cardinality of 5 executions for each generation's solutions, with a limit of 500 generations for the smaller instance and 1000 generations for the larger one. The black horizontal line represent the desired cardinality to either instance.

As expected, no equation returns the expected cardinality, meaning none a return feasible mutant solution, and all must be subject to a repair mechanism. It can be observed also, that all formulas behave as predicted, regardless of the dimension of the problem, as show when comparing figure 3.8(a) and figure 3.8(b).

Somewhat surprising is the evolution of equations (3.3.2) and (3.3.3), that were expected to converge to a cardinality close to the desired one, but here

Operation	$x_{r1} \neq x_{r2} \neq x_{r3}$	$\cdots$	$x_{r1} \approx x_{r2} \approx x_{r3}$
$x_{r2} \cup x_{r3}$	$\leq \mid x_{r2} \mid + \mid x_{r3} \mid$	$\setminus$	$\geq \mid x_{r2} \mid$ or $\geq \mid x_{r3} \mid$
$x_{r2} \cap x_{r3}$	$\geq \mid \varnothing \mid$	$/$	$\leq \mid x_{r2} \mid$ or $\leq \mid x_{r3} \mid$
$x_{r2} \setminus x_{r3}$	$\leq \mid x_{r2} \mid$	$\setminus$	$\geq \mid \varnothing \mid$
$x_{r2,g} \vartriangle x_{r3,g}$	$\leq \mid x_{r2} \mid + \mid x_{r3} \mid$	$\setminus$	$\geq \mid \varnothing \mid$
Equation (3.3.2)	$\geq \mid \varnothing \mid$	$/$	$\leq \mid x_{r1} \mid$
Equation (3.3.3)	$\geq \mid \varnothing \mid$	$/$	$\leq \mid x_{r1} \mid$
Equation (3.3.4)	$\leq \mid x_{r1} \mid + \mid x_{r2} \mid$	$\setminus$	$\geq \mid x_{r1} \mid$
Equation (3.3.5)	$\leq \mid x_{r1} \mid + \mid x_{r2} \mid + \mid x_{r3} \mid$	$\setminus$	$\geq \mid x_{r1} \mid$
Equation (3.3.6)	$\geq \mid x_{r1} \mid$	$\frown$	$\geq \mid x_{r1} \mid$
Equation (3.3.7)	$\leq \mid x_{r1} \mid + \mid x_{r2} \mid + \mid x_{r3} \mid$	$\setminus$	$\geq \mid x_{r1} \mid$
Equation (3.3.8)	$\leq \mid x_{r1} \mid$	$\smile$	$\leq \mid x_{r1} \mid$
Equation (3.3.9)	$\leq \mid x_{r1} \mid$	$\smile$	$\leq \mid x_{r1} \mid$

Table 3.3: Expected cardinality for the different variations of the set-based mutant operator, in the beginning of the evolutionary process (left) and at the end (right), and the expected evolution, as the generations go by (middle).

shown to stagnate, and never reaching this value. In particular, equation (3.3.3) don't show any type of convergence, nor in the cardinality, nor in the optimum value, otherwise it would have stopped before reaching the generation limit, as does equation (3.3.2), that don't converge to the desired cardinality, but shows a faster convergence to either the global or a local optimum, as will be shown in the results, latter. This behaviour by equation (3.3.3) could be explained by either it having a very, very, slower convergence, as in figure 3.8(a) it seems to be slowly raising; or due to the algorithm used in the selection of the solutions to be used in the formulas, explained below.

As expected, equations (3.3.6), (3.3.8) and (3.3.9) have the best values for the cardinality, and equations (3.3.4), (3.3.5) and (3.3.7) shows the predictable higher cardinality in the beginning, declining toward the desired one, although this convergence is much slower on the larger instance than on the smaller one.

These results show, as expected, that the results of all but equations (3.3.6), (3.3.8) and (3.3.9), depend much on the repair mechanism, and this three are the ones that show a less dependence of this mechanism, resulting in an evolution

closer to a "pure" DE (without any repair) as possible, using set-based operators. Apart from these, equation (3.3.2) will also be subject to further study, mainly because it has a faster convergence, when compared to the other equations, although this probably is caused by the repairing mechanism, and not the formula itself.

As a last, but not least, note is the fact that from these last four, only two respect the meaning of "difference" in the arithmetic domain, meaning that when two numbers are equal, their difference is zero. In equations (3.3.2) and (3.3.6), the difference between two equal sets, is not an empty set, but rather the same set, because the intersection and the union are idempotent operations, i.e., for any set A, $(A \cap A) = (A \cup A) = A$. The relative complement and the symmetric difference used in equation (3.3.8) and (3.3.9), on the other hand, respect the principle of the arithmetic difference, i.e., when two sets are equal the result is the empty set $(A \setminus A) = (A \triangle A) = \emptyset$.

From this analysis, the formula for the mutation that most respects the DE's principles are equations (3.3.8) and (3.3.9), as both respect the "difference" operator, and both return sets that need a minimal repair. Remember the idea is not to solve a specific problem, but to have an operator that is the least intrusive in the algorithmic process as possible, leaving the task to find the optimum for whatever problem, for the algorithm as a whole.

Selecting the random solutions for the mutation

Another issue to have in consideration in the mutation operator is the selection of the solutions x_{r1}, x_{r2}, and x_{r3} to use. In classic DE, equation (2.1.3) only obligates their indexes to be different, i.e., $r1 \neq r2 \neq r3 \neq i$, and in the continuous space, this should be sufficient, because the values of x_k should be different from one another, depending on the precision defined. But in combinatorial optimization problems, the probability that two random selected solutions are equal, is much higher than in the continuous space, especially further down in the evolution. If

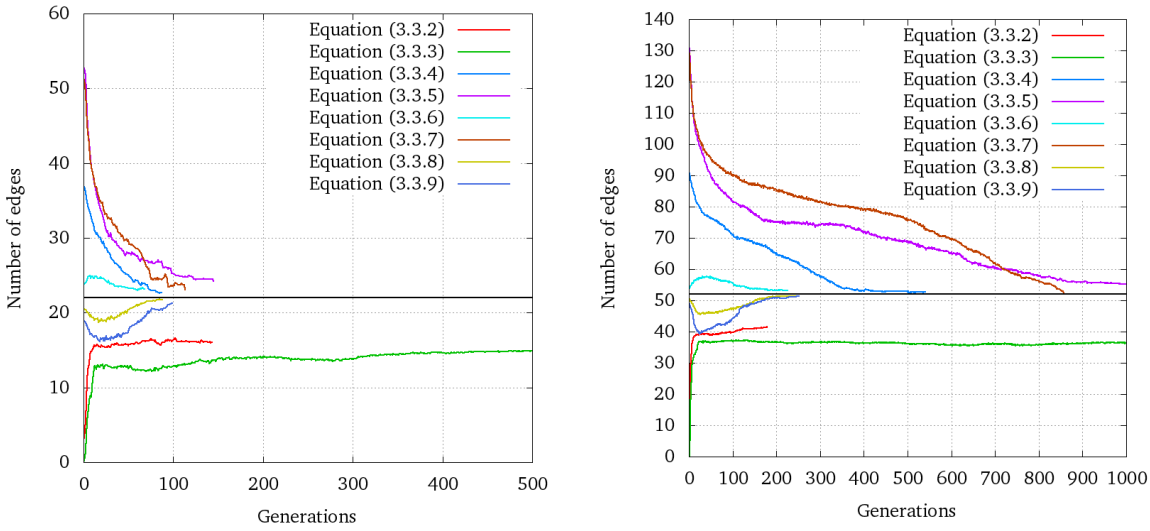


Figure 3.8: Evolution of the average cardinality of the mutant solutions in two instances of the TSPLIB. In (a) a 22 dimensions problem (ulysses22) and in (b) a 52 dimensions one (berlin52). The black horizontal line represent the desired cardinality of each problem.

this is not taken into account, the evolution will probably stagnate because there will be no different solutions to generate a difference. To prevent this, the selection method was changed, with a tighter restriction to have the three solutions really different from one another, regardless of their indexes, i.e., $x_{r1} \neq x_{r2} \neq x_{r3} \neq x_i$. Of course it will get to a stage in the evolution where almost all solutions are equal, and maybe there aren't four different solutions in the population. To prevent this possible stalemate, this restrictions was relaxed to try to find a different solution (if an equal one is already selected) a predefined number of times. If no different solution is found in any of these tries, use the last solution selected. This prevents the possible infinite loop, where the algorithm would try to find a different solution, when there aren't different solutions to find. Algorithm 3.7 implements this procedure. In it, the inner while loop tries to find a solution different from the ones already found for a maximum of 50 times, if it doesn't succeed, gives up and uses the last solution found.

Algorithm 3.7 Algorithm for selecting three different random solutions.

Input: *population*, *i*
Output: *random_solutions*
1: initialize *random_solutions* as an empty list
2: **for** $r = 1..3$ **do**
3: $k \leftarrow 1$
4: **repeat**
5: *sol* $\leftarrow$ select a random solution in *population*
6: $k \leftarrow k + 1$
7: **until** *sol* $\notin$ *random_solutions* and *sol* $\neq$ *population*[*i*] or $k > 50$
8: insert *sol* in *random_solutions*
9: **end for**

Repairing the solutions

Although the usage of the repair mechanism is as minimum as possible, it is still needed, and some changes were made, regarding the previous version. Instead of the, somewhat, complex reconstruction done previously, now is using a simple greedy mechanism, where the elements (edges) with the lowest value are preferred to repair the solution, instead of others with an higher value. To prevent this complexity, before the algorithm starts, all edges are analyzed, and an ordered list using the value of each edge is created. When a reconstruction is needed, it uses this list, starting from the lowest value edge to the highest, using the first possible edge that don't break the constrains, then the second lowest, and so on, until the solution is repaired.

This was done mainly to remove unneeded complexity in the reconstruction, and has two implications on the algorithm: Although improving the solutions, as it serves as a local search, it also could impede the algorithm from exploring thoroughly the search space, steering it very quickly to some local sub-optimum. To prevent this, a random probability was used, where the repair as a 50/50 hypotheses to use a greedy algorithm or being completely random.

In algorithm 3.8 is a high level description of the repair procedure. As said previously, this mechanism is problem dependent, meaning the repair for one type of problems of not the same for other type of problem, with different constrains.

Algorithm 3.8 A possible repair algorithm for the TSP.

Input: *mutant, ordered_edges*
Output: *solution*

```
1: initialize solution as an empty list
2: edges_not_in_solution ← ordered_edges
3: order edges in mutant according to their value
4: // first pass to eliminate edges in mutant that break constrains
5: for all edges in mutant do
6:    remove edge from mutant
7:    if inserting edge into solution don't break any constrain then
8:       insert edge into solution
9:       remove edge from edges_not_in_solution
10:   end if
11: end for
12: if rand() < 0.5 then
13:   shuffle( edges_not_in_solution )
14: end if
15: // Second pass to insert the missing edges
16: i ← 1
17: repeat
18:   edge ← edges_not_in_solution[ i ]
19:   if inserting edge into solution don't break any constrain then
20:      insert edge into solution
21:   end if
22:   i ← i + 1
23: until solution is valid
```

3.3.3 Crossover

The crossover also needed some changes in order to work for combinatorial optimization problems, as equation (2.1.9) cannot be applied to a set-based mutation, mainly because it needs some order in the elements to apply the crossover, and as we've seen, sets by definition are not ordered. Even if we somehow, managed to order the elements of a set, the result would, most likely, not be a valid solution. If we had the solution in figure 3.7(a) as x_i and in figure 3.7(b) as v_i, they could be represented as $x_i = \{(1,2),(2,4),(3,4),(3,5),(1,5)\}$ and $v_i = \{(1,2),(2,5),(4,5),(3,4),(1,3)\}$. Using equation (2.1.9), one possible result could be $u_i = \{(1,2),(2,5),(3,4),(3,5),(1,3)\}$, and this is not a feasible solution.

Instead of using the classic DE crossover and then repair these trial solutions, an operator that creates feasible solutions is preferred, and a different ones can be used depending on the problem. For the TSP example, the ordered crossover (OX), introduced by Davis [19] was used, as its a proven crossover for combinatorial problems, and for the TSP in particular. The ordered crossover works by selecting two points to cut the parent individuals, and then create the offspring by coping the elements between the two cut points of each parent to the respective offspring, and then select from the other parent the missing elements. Suppose the two parents and the cut points "|" are defined as

$$p1 = (\ 1 \mid 2\ \ 4 \mid 3\ \ 5\ )$$

$$p2 = (\ 1 \mid 2\ \ 5 \mid 4\ \ 3\ ).$$

To create the offspring, first the values between the cutting points of each parent would be copied to the respective offspring

$$o1 = (\ \_ \mid 2\ \ 4 \mid \_\ \ \_\ )$$

$$o2 = (\ \_ \mid 2\ \ 5 \mid \_\ \ \_\ ),$$

and then the empty spaces would be filled using the missing elements from the

other parent, starting after the second cut point, and wrapping around when the end is reached, and the end result would be

$$o1 = (\ 5 \mid 2\ \ 4 \mid 3\ \ 1\ )$$

$$o2 = (\ 4 \mid 2\ \ 5 \mid 3\ \ 1\ ).$$

As in DE only one offspring is needed, they are both evaluated and the best one is selected as the trial solution.

Two notes about using this operator in a set-based approach: first, as can be seen in the example above, a node representation was used, instead of the edge representation given earlier, because this operator only works with nodes, not with edges, more on this latter; and second, as implied by the name, this operator needs the order of the elements to work, and as said earlier, in Sets there is no order.

This means that some further work is needed to use this crossover, namely a translation to a node representation before applying it, and then translate the trial solution back to edge representation, to continue the evolution. Although this means that some execution time is "lost" doing this operation, the results obtained using this crossover were better than those using other crossover operators that use edge representation, for instance the generalized partition crossover (GPX) [89] or alternating edge crossover (AEX) [59]. In table 3.4 are the results for these three crossovers, and four variations of the mutation operator, using a TSPLIB problem instance, with 22 nodes (ulysses22). The results shown that the OX crossover gives the better results across the board, especially with equation (3.3.2) and (3.3.9) variations, the first being the best for all crossovers, but this is probably due to the reconstruction mechanism.

3.3.4 Parameter analysis

As said earlier, DE has three control parameters, that should be tuned as efficiently as possible, to get the best results from the algorithm. Unfortunately, as there is

	OX			GPX			AEX		
Equation	Gen	Best	StDev	Gen	Best	StDev	Gen	Best	StDev
(3.3.2)	1549	**7013.0**	0.0	1128	7032.6	37.8	1675	7032.6	37.8
(3.3.6)	426	7189.3	78.3	59	9570.3	963.0	39	9694.1	530.9
(3.3.8)	133	7063.0	47.2	49	9424.8	1157.2	37	10352.7	1132.0
(3.3.9)	274	7014.6	5.1	7620	8245.8	899.4	92	8051.1	520.6

Table 3.4: Results for OX, GPX and AEX crossovers, and different mutation variations, using ulysses22 instance from TSPLIB.

no such thing as "the best algorithm" capable of always solving every problem better/faster/efficiently (see the no free launch theorems, by Wolpert and Macready [90]), also there is no "magic values" for the parameters, always giving the best results for every problem. What this means is that, if the algorithm used a certain type of problems, or in a specific problem, performs in a certain form (say, for instance, always find the optimum value after an average of x generations) using some parameters, using the same set of parameters, in a different problem, or a different set of problems, are not guaranteed to have the same performance (it could never find the optimum, or find it in an average of y generations, with $y \ll x$ or $y \gg x$).

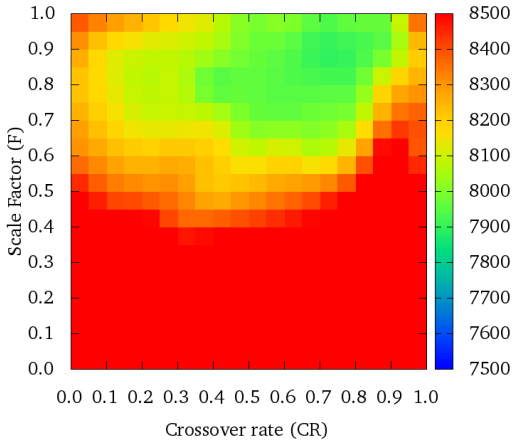
Despite this, both Storn and Price [83] and later Lampinen and Storn [45], said DE is not very sensitive to its control parameters, and choosing good values for them is not difficult to do, in order to obtain good results. In fact, Lampinen and Storn go further and affirm that is much more difficult to choose the parameters in order to fail convergence at all [45]. As a rule of thumb, they suggested, as first approach, to use a population size between $5d$ and $10d$, where d is the dimension of the problem, a scale factor $F = 0.5$, and a crossover rate should be an high value, usually $CR \geq 0.9$, to speed up the convergence. This recommendations, however, were based solely on continuous problems, combinatorial problems are a completely different type of problems, and the recommendations may not hold.

Using an the berlin52 instance from TSPLIB, several executions were made for each variation of the control parameters, to study their behavior in this type of

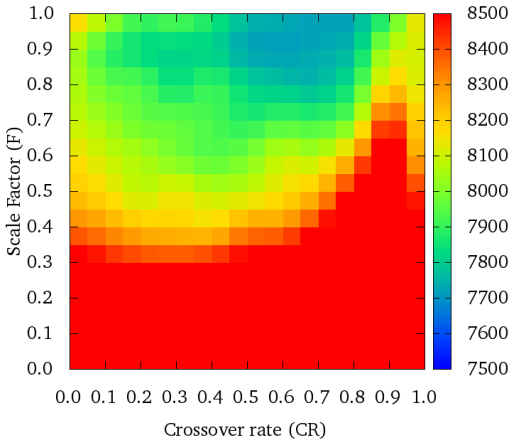
problems. The scale factor F and the crossover CR were given values from 0.0 to 1.0, in 0.1 increments, and the population size Np was tested for $2d$, $5d$ and $10d$, respectively, $Np = 104$, $Np = 260$ and $Np = 520$. Each combination of parameters were run 10 times for a maximum of 100 generations, for each of the mutation formulations given in equations (3.3.2), (3.3.6),(3.3.8) and (3.3.9).

In figure 3.9 is represented a heatmap with the average fitness values for different combinations of F/CR, considering: in figure 3.9(a) using a constant population size $Np = 104$, for all mutation formulations; in figure 3.9(b) using a constant population size $Np = 260$ for all mutation formulations; and in figure 3.9(c) using a constant population size $Np = 520$ and all formulations; and in figure 3.9(d) all different mutation formulations and all population sizes. Observing (a), (b) and (c), i.e., studying the evolution with a constant, but increasing, population size, it can be seen that for a "small" population size (a), the best results are achieved with high values for both $F \in [0.7; 1.0]$ and $CR \in [0.3; 0.9]$, although a very large crossover $CR > 0.9$ shows to worsen the results. This tendency is verified as the population size increases, in (b) and (c), but as it does so, also increase the range of acceptable values for F and CR. Also, as the population grows, CR shows to be somewhat sensitive to F, as the lower the F parameter goes, the lower the CR can go (always with $F > CR$), but higher values for F also need high values of CR, showing a peak around $CR = 0.9$. This shows an algorithm sensitive to F and CR parameters with a lower population size, but this sensitivity decreases as the population size increases. In figure (3.9)(d) is the average results regardless of the population sizes, and, as expected, shows this tendency, with the best results are centered around $F \in [0.8; 1.0]$ and $CR \in [0.6; 0.8]$.

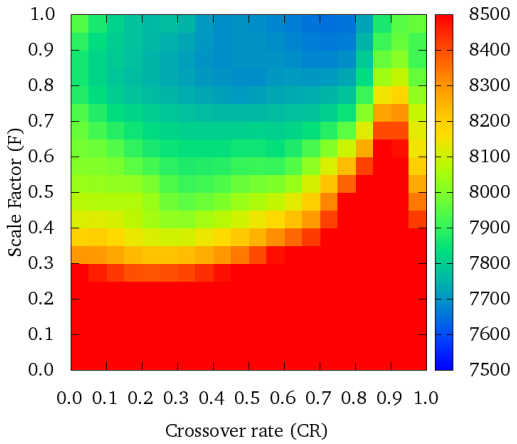
This is confirmed in figure 3.10, where is shown the evolution of the average fitness for all mutation formulation and population sizes. In 3.10(a) for the crossover CR, showing the overall tendency of better results with an higher value of F, regardless of the CR value, and on the contrary, as the value for F lowers, the results are worse, independent of the CR value. However, when $F = 1.0$, some



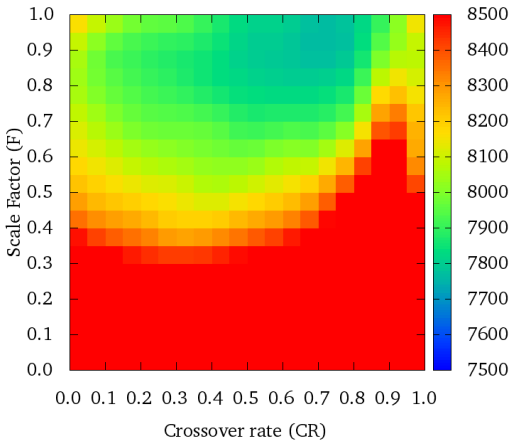
(a) All mutations, $Np = 104$.



(b) All mutations, $Np = 260$.



(c) All mutations, $Np = 520$.



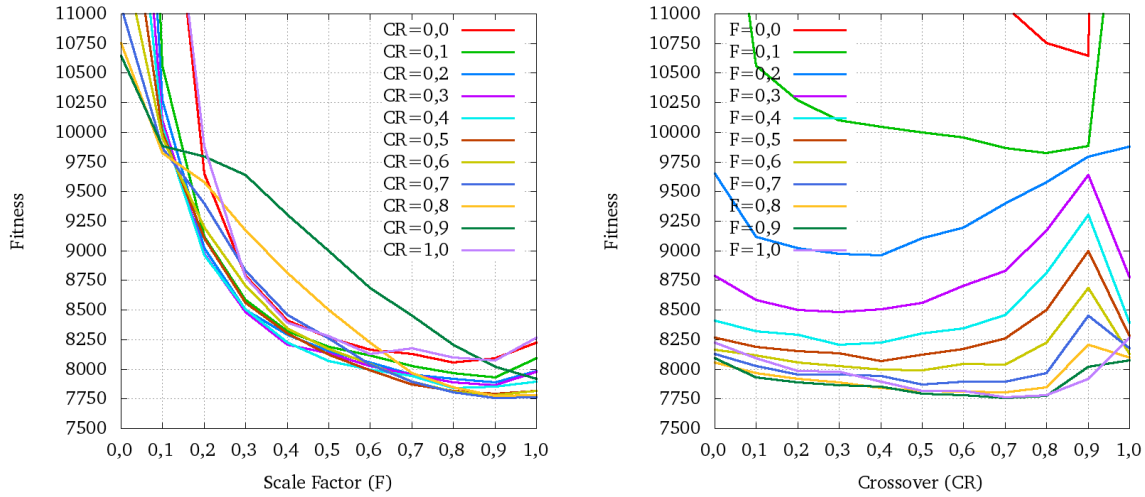
(d) All mutations, all Np.

Figure 3.9: Average fitness for different values of the scale factor (F) and crossover (CR) parameters. In (a), (b), and (c) averaging the values from different mutation formulas, with a constant population size of, respectively, $Np = 104$, $Np = 260$ and $Np = 520$; in (d) averaging the values through different population sizes and mutation formulations.

values for the CR, especially the ones with a lower value, have a slight worse results then for $F = 0.9$. The fact that, when $F = 0.0$, the results are simply not acceptable is due to the mutation operator is not being used, and as the mutation is the core factor in DE to evolve the population, this means that only the crossover operator is working, and solely, is incapable to give good results, at least in the maximum number of generations given. In 3.10(b) is presented the reverse, i.e., the evolution of the fitness for different values of the scale factor F, shows the more uniform spread of the crossover for any given value of F. Unlike for F, CR shows acceptable values, independent of the scale factor value, except of very low values for F. This figure shows that, generally, for each, increasing, value of F, the best fitness value is obtained for a different value of CR, also increasing, i.e., the fitness values are getting better as both F and CR are getting higher., reaching the best values when $F \in [0.9; 1.0]$ and $CR \in [0.7; 0.8]$. With values of $CR > 0.8$ the results get worse, except in the extreme $CR = 1.0$, where the results get better again. This could be explained by the operation of crossover operator itself. Notice that 3.10(b) shows similar results for $CR = 0.0$ and for $CR = 1.0$, regardless of the scale factor used. In both these extreme values, the crossover, in fact, does no crossover whatsoever, is just gives the current individual in the population or the mutant individual (refer to equation (2.1.10)).

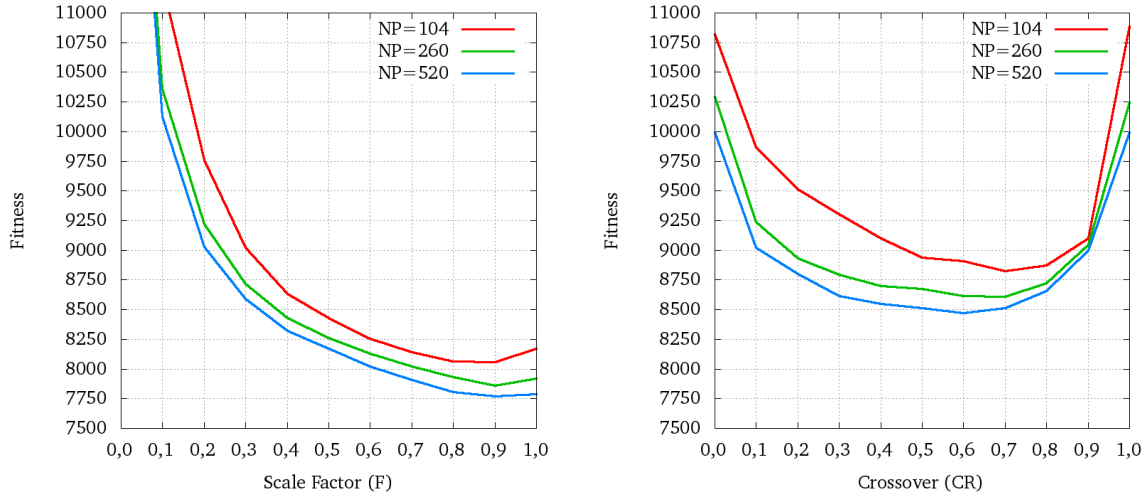
In figure 3.11 is the average fitness for the three distinct population sizes, for different scale factor values in 3.11(a), and crossover 3.11(b), were can be seen, as expected, that as the population size increases, the fitness results are better in both cases.

Considering now the different mutation formulations, in figure 3.12 are the heatmaps for the evolution the parameters for each pair of population size and mutation formula. From top to bottom are equations (3.3.2), (3.3.6), (3.3.8) and (3.3.9), and from left to right are population sizes of $Np = 104$, $Np = 260$ and $Np = 520$. Starting with the bottom two lines, corresponding to equation (3.3.8) and (3.3.9), the results are similar to those presented earlier, however, equation



(a) (b)

Figure 3.10: Evolution of the average fitness values for: (a) different crossover CR values with constant same scale factor F, and (b) different scale factors F values with a constant crossover CR.



(a) (b)

Figure 3.11: Average fitness for different population sizes, for (a) different scale factor F, and (b) different crossover CR.

(3.3.9) seems to accept a wider range parameters. In the second line, although similar to those discussed, deserves a special mention, as it gives acceptable results with broader range of parameters, especially F, but for good results the range is much narrower than the other two, and also need a bigger population size. Finally, the first line shows a somewhat different map. As all the others, has a wide range of acceptance to the CR value with incidence in high values, particularly with lower population sizes, the difference is the much greater sensitivity to the F parameter, as anything lower than $F = 0.7$ produces very bad results, regardless of either CR or population size.

Finally, in table 3.5 are the fitness for the different mutation equations and population size, and also the averaged results, ordered by this averaged results. For each equation, along with the fitness result is the order in which the parameter's fitness was classified using the respective parameters. It's clear from this, short, table that there isn't a perfect combination of parameters resulting in the best values for every mutation equation formulated. For instance, the best combination of parameters for equation (3.3.2) only got the 12^{th} result using equation (3.3.6), and the best parameters for equation (3.3.8) were the second best for (3.3.9), but only the 24^{th} for equation (3.3.6).

As there isn't a good set of parameters all around, the next big thing is to use the average, with all its advantages and disadvantages, and he best average result was 7757.2, using a scale factor $F = 0.9$, and a crossover rate $CR = 0.7$. As for the population size, from the tests made, the larger the population the better the results, as is clear from figure 3.11, but this is not a reason to set a population size astronomically high, as with this larger population size comes an increase in computation time and/or space.

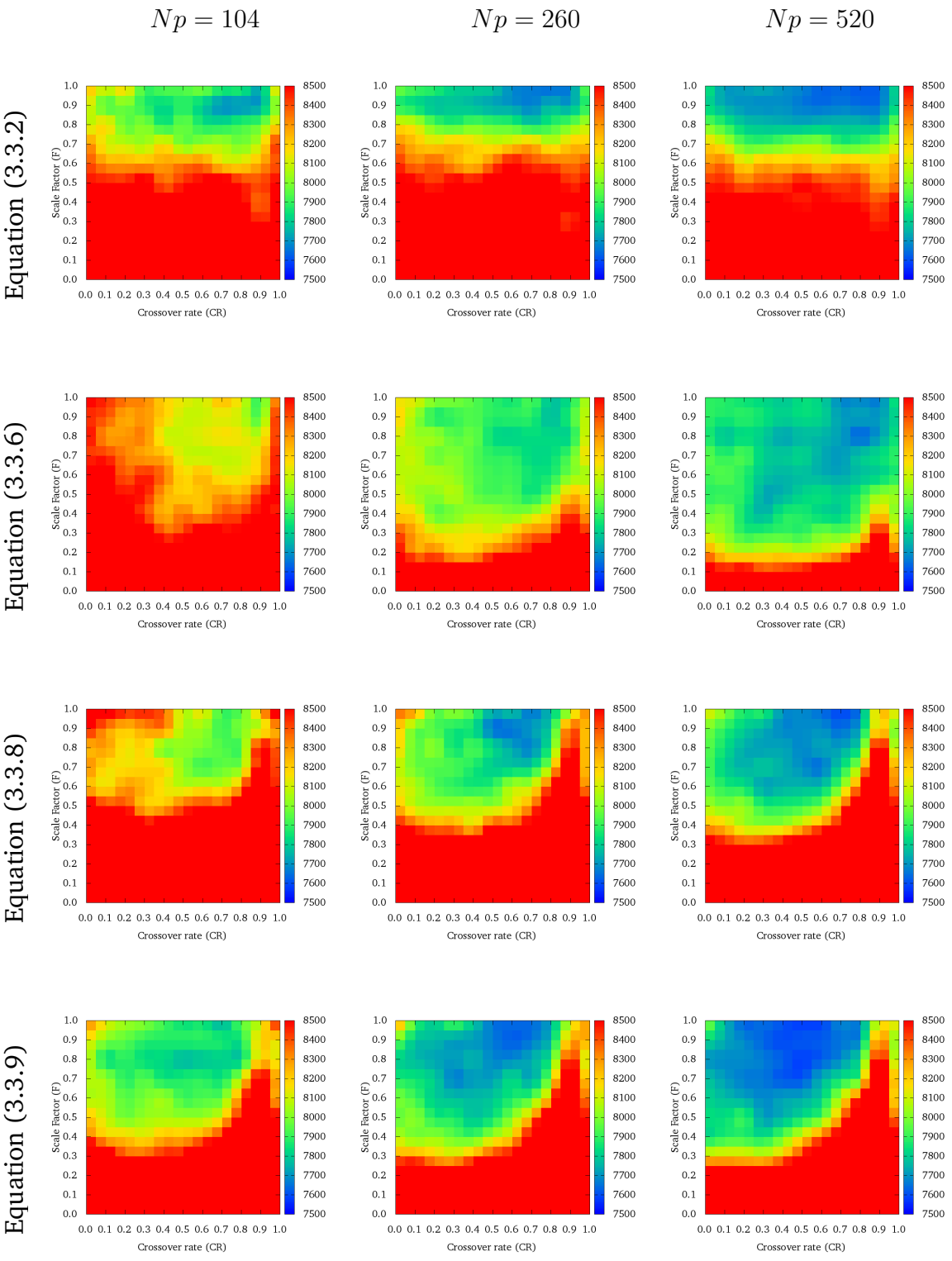


Figure 3.12: Average fitness for different values of the scale factor F and crossover CR parameters. In the left column are population sizes of $Np = 104$, the middle column is for $Np = 260$ and the right columns is for values of $Np = 520$. From top to bottom are the mutation formulations used in equation (3.3.2), (3.3.6), (3.3.8) and (3.3.9), respectively.

		Equation (3.3.2)		Equation (3.3.6)		Equation (3.3.8)		Equation (3.3.9)		Average
F	CR	#	Fitness	#	Fitness	#	Fitness	#	Fitness	Fitness
0.9	0.7	1	7651.1	12	7905.3	2	7742.7	11	7729.5	7757.2
1.0	0.7	7	7731.9	24	7966.2	1	7691.5	2	7672.3	7765.5
0.9	0.8	4	7682.2	4	7849.7	11	7842.1	14	7736.1	7777.5
1.0	0.8	6	7723.1	7	7865.2	5	7767.2	22	7780.0	7783.9
0.9	0.6	5	7698.2	20	7937.5	7	7788.5	7	7712.0	7784.1

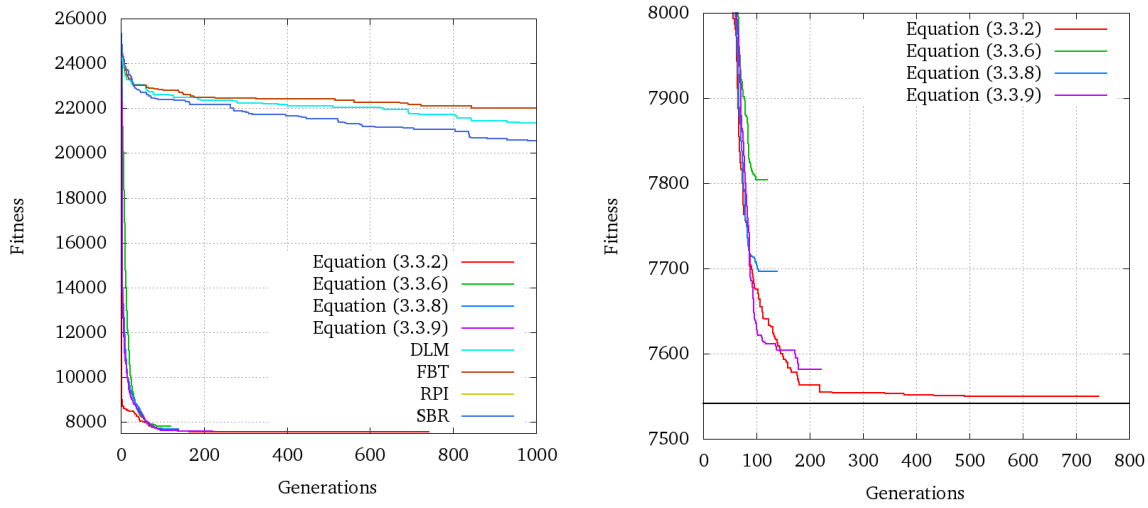
Table 3.5: Best average fitness by mutation formulation and respective parameters.

3.3.5 Results

Using the operators given in this section, an DE algorithm was implemented in Python, using the DEAP [27] and the SCOOP [37] libraries. The first library implemented some basic evolutionary algorithms, with common libraries and auxiliary toolboxes to implement other EA. The SCOOP library allows to use distributed processing in the code in a very straight-fashion way. Since nowadays, almost every computer has multi-core processors[2], this processing power should not be wasted, and the SCOOP libraries allows an easy distribution of the computing processes though these CPU cores, or even to other computers, on a network.

The code as run using several instances of TSPLIB, of different sizes. For each instance was used a population size of five times the dimension of the problem, as this was a good enough size, without being too excessive in computational resources, a scale factor of $F = 0.9$ and a crossover rate of $CR = 0.7$. Each instance was run 10 times for the four different mutation formulas, and the stopping criteria was 1000 generations, or when all the elements in the population converge to the same solution. For comparison, the relative position indexing (RPI), the forward/backward transformation (FBT), the subrange approach (SBR) and the differential list of movements (DLM) were also run, using the same parameters. In figure 3.13(a) is the evolution of the average fitness, for the berlin52 instance. As easily seen, all the other approaches show a very slower convergence, if any,

[2]Multi-core processors are single computer chips with several central processing units (CPU) in it.



(a) (b)

Figure 3.13: Evolution of the search for the optimum. In (a) for all approaches, in (b) a closer look of only our formulations.

that any of equations (3.3.2), (3.3.6), (3.3.8) or (3.3.9). In figure 3.13(b) is a closer look of only these formulations, and the black line represents the optimum value. Here can be seen that, for this instance, the best result was obtained by equation (3.3.2), but at the cost of many more generations.

Table 3.6 shown the average of the best values obtained for each problem, as well as the standard deviation, the number of generations needed, and the relative error to the optimum value. As the figures already have shown for a particular problem, the best value is obtained using equation (3.3.2), however, as already discussed, this results are probably heavily influenced by the repair mechanism. Nevertheless, all other variations of the formulation show acceptable results, with relative errors to the optimum, usually below 5%, with equation (3.3.9) showing better results than the other two, both in the average as also in the standard deviation, as the problems grow in dimension. The other approaches, clearly need many more generations to have a possibility to achieve similar values. For instance, a second look at table 3.2 on page 75, shows that for the berlin52 problem, the DLM approach needed about 15000 generations to archive worse

results then those obtained after only about 500 generations for equation (3.3.2), and less then 150 generations for any of the others three.

For the reasons already explained, although equation (3.3.2) shows the best results, is not the one suggested to serve as base for the set-based approach, but rather equation (3.3.9), because is one of the least affected by the repair mechanism, and also don't show as much premature convergence as any of the other two.

		ulysses16	ulysses22	eil51	berlin52	eil76	kroA100
Equation (3.3.2)	Avg	6859,6	7020,0	436,2	7550,5	551,4	21670,3
	StDev	1,9	22,1	6,7	26,9	2,6	239,9
	Gen	69	84	762	466	825	912
	Rel.Err	0,01%	0,10%	2,39%	0,11%	2,49%	1,82%
Equation (3.3.6)	Avg	6867,3	7049,0	443,7	7804,3	556,3	22458,5
	StDev	13,5	38,0	6,3	198,4	9,8	569,2
	Gen	63	55	110	101	172	141
	Rel.Err	0,12%	0,51%	4,15%	3,48%	3,40%	5,53%
Equation (3.3.8)	Avg	6862,4	7020,0	442,2	7697,7	562,0	22132,7
	StDev	4,7	22,1	9,3	218,7	9,2	362,7
	Gen	60	80	111	99	148	134
	Rel.Err	0,05%	0,10%	3,80%	2,06%	4,46%	4,00%
Equation (3.3.9)	Avg	6869,6	7020,6	448,2	7582,0	554,0	22036,3
	StDev	20,8	22,0	9,7	42,2	5,5	358,7
	Gen	110	78	122	144	307	198
	Rel.Err	0,15%	0,11%	5,21%	0,53%	2,97%	3,54%
DLM	Avg	7260,0	9324,8	1220,9	21336,1	1960,0	131416,8
	StDev	145,7	384,6	16,7	569,1	33,8	1739,9
	Gen	1000	1000	1000	1000	1000	1000
	Rel.Err	5,85%	32,96%	186,60%	182,90%	264,31%	517,50%
FBT	Avg	8173,3	10389,7	1204,6	22003,4	1875,9	130779,0
	StDev	263,0	234,8	31,8	337,6	261,3	1395,0
	Gen	1000	1000	1000	1000	1000	1000
	Rel.Err	19,16%	48,15%	182,77%	191,74%	248,68%	514,51%
RPI	Avg	7663,6	9394,8	1162,8	20558,9	1949,0	130983,3
	StDev	276,2	364,4	34,9	594,6	16,1	2170,3
	Gen	372	930	1000	1000	1000	1000
	Rel.Err	11,73%	33,96%	172,96%	172,59%	262,27%	515,47%
SBR	Avg	7258,7	8214,2	743,3	12991,7	1162,6	71445,6
	StDev	239,2	287,1	45,3	590,7	58,8	4093,8
	Gen	568	885	1000	1000	1000	1000
	Rel.Err	5,83%	17,13%	74,48%	72,26%	116,10%	235,71%

Table 3.6: Average results for different TSP instances, using different approaches.

Chapter 4

Multi-objective differential evolution for combinatorial optimization

In the previous chapter was introduced a new methodology to use the differential evolution algorithm in combinatorial optimization problems. In this chapter will be shown how to adapt that methodology to use in multi-objective combinatorial optimization problems, focusing on how to keep the best solutions from generation to generation and how to select them, using more than one population to evolve the solutions and using self-adaptive parameters.

4.1 Introduction

In the previous chapter was introduced a new methodology to use the differential evolution algorithm in combinatorial optimization problems, using the TSP as an example, has this is one of the most, if not the most, known and used problem in combinatorial optimization. But, although the TSP example is very understandable, in real live the TSP usually has more to it. Suppose a salesman, not only wanted to minimize the time spent to visit all his customers, but also wanted to minimize the cost of the travel, knowing that some roads have tolls, and/or that he will have to sleep some nights on a hotel. This type of situation is said to be

a multi-objective optimization problem, as there are more than one (two in this example) objectives to be optimized, and frequently these objectives are contradictory to one another, as in the TSP example, where, for instance, if the salesman wanted to minimize the time, he probably would use tolls roads, as this allow usually faster travel, but in doing so, he would spend more money, this way raising the second objective value. And the same happens the other way around. Depending on the dimension of the problem, inspecting and evaluating the different hypothesis to determine the best, is very difficult if not impossible, as even with all the hypotheses at hand, it is very unlikely that the one optimum solution are found. In multi-objective optimization problems, the algorithms usually don't find the optimal solution, simple because there isn't one, they rather return a set of solutions, each one better at one objective and worse at others. This set of non-dominated solutions are called the Pareto set, and their representation is called the Pareto front, as seen previously.

The decision about the best solution is not in the algorithm itself, but on an external entity, called the decision maker, usually a human, that will inspect the solutions in the Pareto front and decide which is the "best" solution, according to his preferences. When the algorithm finds the Pareto set without any information about the decision maker preferences is called a *posteriori* approach, as only after the algorithm finishes its execution does the decision maker intervenes in the process. In some cases, it is possible to incorporate the decision maker preferences into the algorithm, making, this way, the solutions found by algorithm biased towards the decision maker preferences, simplifying his latter decision. This are called *priori* approaches, as the decision maker's preferences are incorporated in the algorithm before its execution. A third type are the *interactive* approaches, where after a certain number of iterations, the algorithm gives some information to the decision maker, and ask him for some preferences to be used for the next iterations, and so on and so forth.

Each of these approaches have advantages and disadvantages, and choosing

one over the others may vary from problem to problem and even from decision maker to decision maker. In evolutionary multi-objective optimization algorithms, usually the posteriori approach is the one used, meaning that the algorithm will be executed without any external information about the preferences of the decision maker, and because of this, the set of solutions found, must be as close to the Pareto-optimal set as possible, and at the same time, maintain a diversity in the solutions in that set for them to correctly represent the entire Pareto front, otherwise the decision maker may not have enough information, or even good information, to make his decision.

The common methodologies to accomplish this dual-objective of proximity/diversity, is using the Pareto dominance concept, decomposing the multi-objective problem into several single-objective, or using quality indicators. Each of these techniques have already been introduced and discussed in section 2.2.3, and although all these techniques origin different evolutionary multi-objective optimization algorithms, our objective is to use the differential evolution algorithm introduced in the previous chapter, and modify it to solve multi-objective combinatorial optimization problems.

Is not as DE as not been used previously to solve multi-objective optimization problems before, several authors have provided adaptations to DE to this type of problems, as the Pareto differential evolution (PDE) [1], the generalized differential evolution 2 (GDE2) [44] and GDE3 [43], the multi-objective differential evolution (MODE) [24] or the differential evolution for multi-objective optimization (DEMO) [78], just to name a few (see [58, 18] for further information), but these have been constructed to work in the real domain, not for combinatorial optimization, and although some principles are similar, others need to be adjusted and/or modified.

The first thing to consider is what to to with the non-dominated set of solutions, as they can be either stored in a separate population, or be in the main population. As the former approach was used, this is the first modification to the

algorithm needed. The repair mechanism also needs consideration, as its based on a greedy algorithm, and for multi-objective problems, this procedure is not the most effective, as the results would be skewed towards some really good values for some objective, but this type of problems is about reaching good trade-off solutions, not exceptionally good for some objective and very bad at others.

Although the Pareto dominance concept is probably the most used in evolutionary multi-objective optimization algorithm to replace the population from one generation to the next, other approaches are possible. Examining the way this procedure works, one first alternative is to use the convex hull instead of the Pareto dominance, which is a similar concept, with the advantage that several efficient algorithms exist to determine the convex hull. This will further explained and expanded using a concave hull algorithm, based on the one used for the convex hull, allowing for an efficient detection and usage of this type of hull in multi-objective problems.

Other techniques used in evolutionary algorithms are the usage of more than one population in the evolutionary process and the usage of some type of self-adaptation of the parameters. While the former is usually implemented when using in parallel computation, this is not a requirement, and we will implement it without using parallel computing. Self-adapting the parameters, although not a requirement of multi-objective problems, in fact most implementations, if not all, have originated in single-objective problems, is probably more important in the former than in the later, as if in single-objective problems the parameters can usually be tuned using a similar problem whose optimal solution is known, in multi-objective problems, there is no optimal solution to compare with, all non-dominated solutions are valid, and tuning the parameters is actually more difficult in this type of problems. Both of these techniques will be tested, using some existing implementations along with other ideas, and the results will be analyzed, to see which techniques can be used and which are worth it.

In the rest of the chapter the different implementation of these different ap-

proaches will be described, and the results discussed, to conclude with the suggested final implementation of the evolutionary multi-objective combinatorial optimization. As an example of this type of problem, the multi-objective traveling salesman optimization problem will be used in the examples.

4.2 Saving the non-dominated solutions

Unlikely single-objective problems, that usually (but not always) have just one population of solutions to evolve, in multi-objective optimization, is frequent to use an external archive, a second population, to keep the non-dominated set of solutions. This solutions are separated from the general population, and are not used in the evolutionary process. When a non-dominated solution is found in the general population, is copied to the archive, and any existing solution in it, dominated by this new solution have to be deleted.

The differential evolution algorithm don't contemplate any archive population in its evolutionary process, so an adaptation has to be made to allow it to save the non-dominated solutions. This adaptation consists in, after the initial population is created, the archive is filled with the non-dominated solutions in the initial population. Then the algorithm evolves this population as usual, and after replacing the population for the next generation, it updates the archive with the non-dominated solutions in the new population. The non-dominated solutions are usually determined using the Pareto dominance concept. This process goes on during the entire evolution, and in the end, the archive contains the Pareto set of solutions for our multi-objective optimization problem. Algorithm 4.1 shows the differential evolution algorithm with an archive population. In line 2, the archive is created using an empty list and the initial population, the empty list being the initial (empty) archive.

Algorithm 4.1 Differential evolution algorithm with an archive to save the non-dominated solutions.

```
1: population ← initialization( )
2: archive ← update( empty, population )
3: while not end criteria do
4:    mutant ← mutate( population )
5:    trial ← crossover( mutant )
6:    population ← replace( population, trial )
7:    archive ← update( archive, population )
8: end while
```

4.3 Adapting the repair mechanism

In the previous chapter, a repair mechanism was constructed to repair the unfeasible solutions created by the set-based mutation. But this repair mechanism, although not completely, has a greedy component in it, meaning the "best" element is selected at a given point so repair a solution. If in a single-objective problem, this may be interesting, as it, possible, pushes the algorithm toward the optimum solution, in multi-objective there is no "best" element to use in the reconstruction, as each can have a better value for some objective but a worse for the others.

In fact, a greedy repair, would be very similar to a priori approach, in which the decision maker preferences had been incorporated in the algorithm, except that this preferences would be all-or-nothing, i.e., a greedy repair would focus exclusively on one objective, and the results would be similar to those in figure 2.6 in page 27. In fact, those Pareto front are the result of a greedy repair, focusing solely in the first objective (a) and in the second (b).

As a repair mechanism is a necessity in our set-based mutation, it cannot be dismissed, but neither can a greedy one be used. One approach was to dismiss the greedy repair, and use a full random repair, where each missing element in out solution would be filled by a random selected element, from the decision space domain.

Another more general idea was to make each solution work toward an objective, by incorporating an objective preference in each solution. Thus, each solution

are represented by

$$X_i = (x_i, objective)\,, \tag{4.3.1}$$

where X_i is the new extended solution, x_i is the actual solution, and $objective$ is the objective for that solution to focus, when a repair is needed. This allows for each solution to be repaired with the focus on its objective, but, as the mutation and crossover are indifferent to this, each solution will be combined with other solutions, with the same or other objective, this way incorporating good elements from other objective. However, as each solution need to have an objective, both the mutation and the crossover operators, need to defined a default objective for the mutant and the trial solution, respectively. The mutant solution's objective will be the same as the solution used as base solution in the mutation operation, as for the trial solution, it will inherit the objective of the current solution, i.e, the solution in the current population, not the mutant solution. The initial objective for each solution is assigned when the initial population is created, by randomly selecting one of the objectives of the problem.

Algorithm 4.2 shows the repair mechanism for multi-objective traveling salesman optimization problem, where *mutant* is the mutant solution to be repaired, *repair_objective* is the objective to be used in the repair process, and can be $0, 1, 2, ..., m$ or random, where values $\in \{1, 2, ..., m\}$ is the objective to be used in the repair, meaning a greedy repair will be used; 0 means the process described above, where each solution will be reconstructed using one objective, inherited from the base solution; and random means a completely random repair. *ordered_edges* is a list with as many elements as the number of objectives, and each element is a composed by a list of edges, ordered using only the respective objective. Suppose the multi-objective problem represented by the graph in figure 2.1, where each edge has two values, the first is the value to use in the first objective, the second is for the second objective. The first element of *ordered_edges* would

be the list of edges ordered according to the first objective, i.e., *ordered_edges*[*1*] = [*BC, AB, CD, AD*], and the second element would be composed by the edges ordered according to the second objective, *ordered_edges*[2] = [*AD, BC, AB, CD*]. As in the algorithm for single-objective problems, this list is created before the evolutionary process starts, and not changed in it, as these values are always the same. Lines 3-10 defines which element in *ordered_edges* will be used, according to the value of *repair_objective*.

Algorithm 4.2 A repair mechanism for the multi-objective TSP.

Input: *mutant, ordered_edges, repair_objective*
Output: *solution*

```
1: solution ← empty list
2: // initialize edges_not_in_solution according to repair_objective
3: if repair_objective = 0 then
4:    objective ← mutant.objective
5: else if repair_objective is random then
6:    objective ← chose a random objective
7: else
8:    objective ← repair_objective
9: end if
10: edges_not_in_solution ← ordered_edges[ objective ]
11: // first pass to eliminate edges in mutant that break constrains
12: order edges in mutant according to their value
13: for all edge in mutant do
14:    remove edge from mutant
15:    if inserting edge into solution don't break any constrain then
16:       insert edge into solution
17:       remove edge from edges_not_in_solution
18:    end if
19: end for
20: if repair_objective is random then
21:    shuffle( edges_not_in_solution )
22: end if
23: // Second pass to insert the missing edges
24: i ← 1
25: repeat
26:    edge ← edges_not_in_solution[ i ]
27:    if inserting edge into solution don't break any constrain then
28:       insert edge into solution
29:    end if
30:    i ← i + 1
31: until solution is valid
```

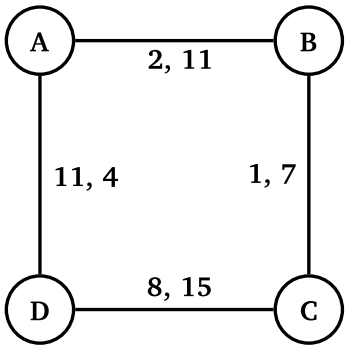


Figure 4.1: An example of a graph with two values for each edge.

Of course this means that "groups" of solutions will exist virtually in the population, each group being formed by all solutions with the same objective to repair. But with each solution in the population having his own objective, and due to the process used to mutate and repair the solutions, the objective of the ith solution can change from one generation to the next, this means that, theoretically, one group of solutions could overcame the all others and have all solutions in the population in it, i.e, all solutions with the same objective to repair, much like the greedy repair discussed earlier. However, due to the stochastic evolutionary process and to the DE mutation operator that creates a new solution using three others, this is unlikely to happen. In fact, represented in figure 4.2 are the number of solutions for each of the two objectives of a bi-objective problem for a population size of 200 individuals and a maximum of 100 generations. On the left are the values of one execution, and on the right are the averaged values for 10 executions. As can be seen, the number of solutions for each objective is not constant, in fact, the number keeps changing in each generation, meaning that the majority of the population goes toward one objective, then changes to the other, and so on and so forth, dynamically, and without any external influence. In an average of 10 executions, can be seen that the values are nearly divided halfway, with both objectives getting half the population.

The results after 500 generations of a multi-objective optimization TSP instance, can be seen in figure 4.3, using greedy repairs, one for each objective, a random repair, and the repair using "groups of solutions", where each solution is

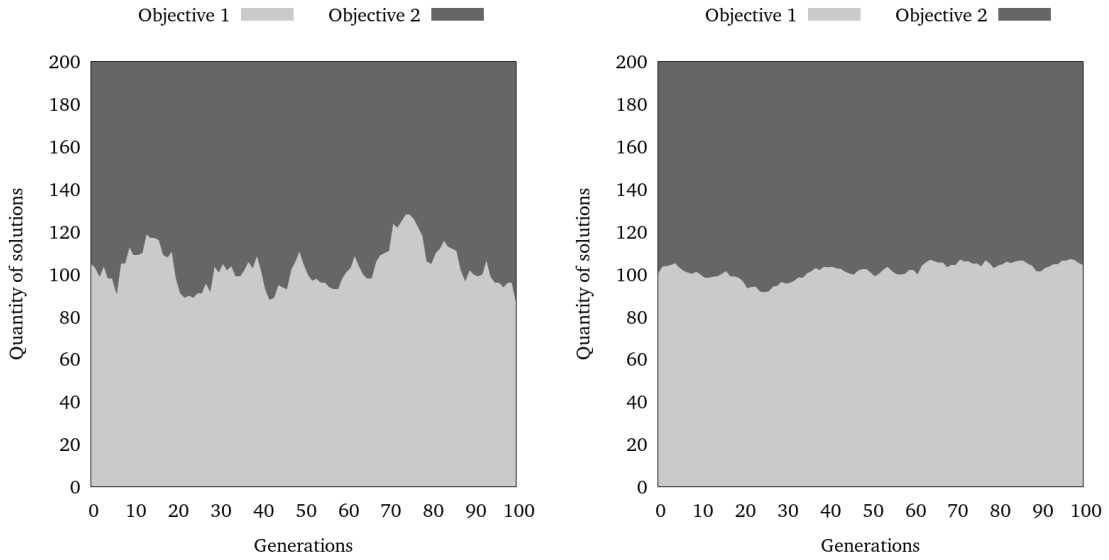


Figure 4.2: Number of solutions for each of the two objectives for a maximum of 100 generations of a bi-objective problem: on the left are the values for a single execution; on the right are the average values of 10 executions.

repaired considering only its own objective, regardless of the other solutions. Even using only the figure, without any further analysis, it's obvious that the greedy repair, as expected, obtained good results in the respective objective, but very bad results on the other, and is not a good repair mechanism, as for the other two, is also easy to conclude that the Pareto front reached using the "group" repair method is much better than the random repair, because the resulting Pareto front has clearly a better approximation to the Pareto optimal front; has a better diversity of the solutions, as they are spread through larger values for both objectives; and even has a larger number of solutions in the Pareto front.

4.4 Replacing the population

The main difference between a single-objective and a multi-objective optimization problem, is that a single-optimization problem returns a single solution, and a multi-objective returns a set with the best trade-off solutions, the Pareto set, as seen in the previous figure, where the Pareto set in represented using their images, i.e., the Pareto front. How to make the differential evolution algorithm to

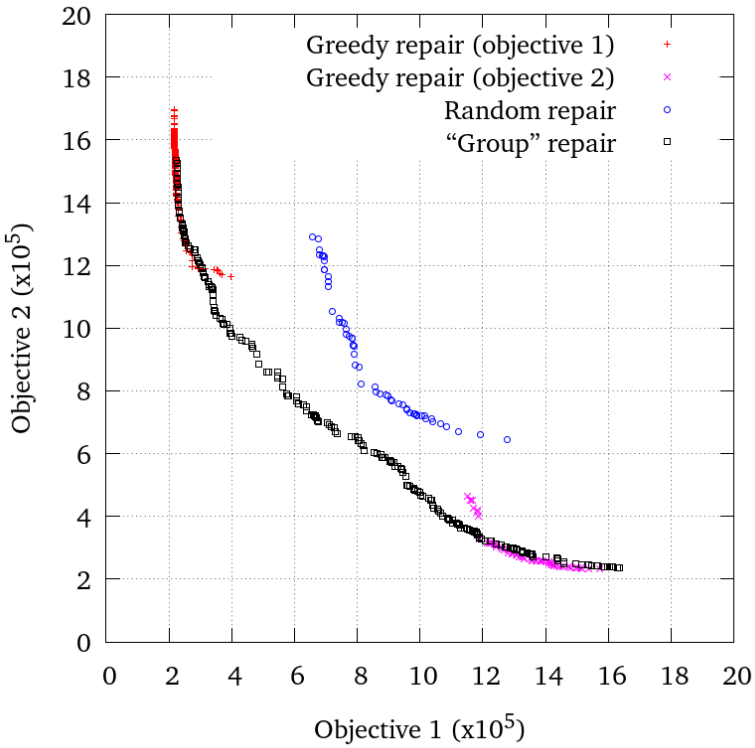


Figure 4.3: Pareto front after 500 generation, for the kroAB100 problem, using a random repair, and the repairing each solution considering its own objective.

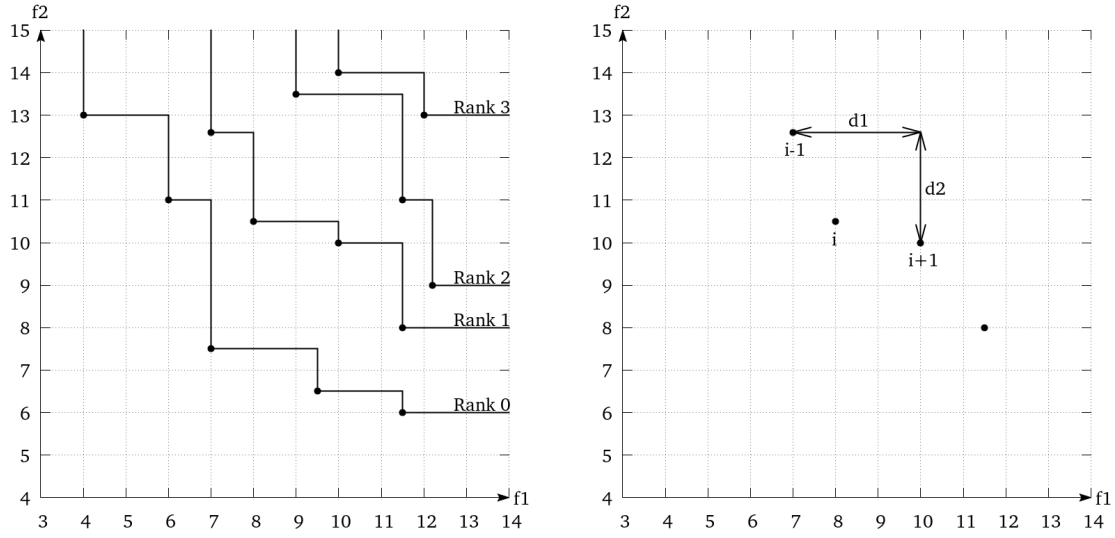
keep these solution between generations has already been discussed, not how to actually found them. Although three different approaches will be shown, the basic idea is the same: using all solutions in the population, rank the solutions according to some criteria, and assign each solution a diversity measure. The first approach uses the Pareto dominance concept, in the same form as the NSGA-II, to replace the population for the next generations, other uses the convex hull, and the third uses a concave hull.

Pareto dominance

The most used evolutionary multi-objective algorithm to compare new algorithms with, is the NSGA-II algorithm [87, 29, 11], and for this reason, its technique to replace the population from one generation to the next will be used in DE.

To select which solutions will be kept for the next generation, the NSGA-II al-

gorithm first assigns all solutions a ranking value, based on how many layers of non-dominated solutions, each solution has above it, and then uses a crowding distance, to assure a diversity in the population. To assign a rank to each solution, first it joins the current population with the offspring, and using this combined population, will determine the non-dominated solutions. This non-dominated solution will be assign the ranking zero, as it doesn't have any other non-dominated set above it. This set is then removed from the population, and a new non-dominated set is determined, and assigned to rank one, has this second set has one set that dominates it. This second set is then removed from the population, and the process continues until a ranking is assigned to all solutions. This process is illustrated in figure 4.4(a), showing the rank assigned to each solution. Although the figure shows all solutions ranked, practice it only needs to rank as many solutions as the number of solutions in the initial population, and the reason is simple: this procedure is used to select a population of size Np, from a population of size $2Np$ (remember the offspring were joined to the population), so when more than Np solutions are ranked, no more are needed, as the best Np solutions are already selected. Supposing all solutions are represented in figure 4.4(a), only half of them (seven in this case) needed to be ranked, meaning only solutions in the first two rankings (in this example) are needed, because the first two ranks together have nine solutions from the seven needed. Next is the calculating of the crowding distance, to assure a diversity in the population. The crowding distance is only calculated using solutions the the same rank, there is no crowding distance between solutions in different ranks. For a two dimensional problem, the crowding distance is calculated using the principle of the Pythagoras's theorem, adding the distance of the sides of the right triangle between the two neighbours of every solution, as shown in figure 4.4(b), to calculate the crowding distance of the ith solution. The concept is extended to upper dimensions. A detailed explanation of each of these procedures is in the NSGA-II algorithm, in [23].



(a) Ranking the Pareto fronts.

(b) Crowding distance

Figure 4.4: The NSGA-II selection using Pareto dominance: in (a) the ranking procedure, in (b) the crowding distance.

Convex hull

As seen in figure 4.4(a), the ranking of the non-dominated consists in finding the different layers of solutions, similar to the onion-peeling concept, except that, instead of peeling the "whole" onion, it was cut in four parts, and only the lower left part is peeled. A simple algorithm for the onion-peeling is using the convex hull to determine each layer, and the same principle can be used to rank the solutions, instead of the Pareto dominance.

Although almost all research in multi-objective optimization problems use the concept of Pareto dominance, the convex-hull has the advantage having efficient algorithms to determine it, and some researchers have used it before [60, 79, 91, 4].

From the different algorithms to determine the convex hull of a set of points, some work in multi-dimensional space, others only in two dimensions. Due to the complete convex hull not being needed to determine the Pareto front, a modified version of the Graham's scan algorithm was used, that determine only a section of the convex hull. The caveat is that this algorithm only works in two dimensions,

i.e., we are limited to work in bi-objective optimization problems, but if necessary, a different algorithm could be used, removing this limitation.

The modified algorithm works like this: First, the solutions are ordered in lexicographic order, using the first objective (assuming first objective is in the x coordinate), and if two solutions have the same value for the first objective, sort them by the second objective. Then find the solution with the minimum value for the second objective (assuming a minimization problem), and using Graham's scan, find the convex hull only using the solutions up to the one found. The reasoning behind this is simple, in a minimization problem, the solutions are in the third "quadrant", if thinking in the trigonometric circle, i.e., where are the lower values for both axis. If the search for the hull would be allowed to continue passing the solution with the minimum value for the second objective, obviously, those values would start raising, and those values would no longer be the minimum values for that objective. If the initial lexicographic order of the solutions are altered, this algorithm would works with problems other then minimization, it's just a matter of thinking to which "quadrant" the solutions will evolve, and sort accordingly.

This algorithm for the convex hull is used to determine the ranks for each solution much the same the Pareto dominance is used if the NSGA-II algorithm: the current population is joined with the offspring, and using this joint population, the appropriate section of the convex hull is determined and the solutions belonging to it are assigned rank zero and removed from the joint population. The next layer of the convex hull is determined, but this time is assigned the rank one. The process continues until at least as many solutions are ranked as the size of the initial population Np. The process is illustrated in algorithm 4.3. The sort procedure in line 4 depends in the type of problem, i.e., if the scan for the convex hull is to be made using the lower part, (problems where the second objective is to be minimized), the solutions must be sorted in ascending order, allowing a left to right scan, otherwise they are sorted in descending order (in problems where

the second objective is to be maximized), meaning the scan for the convex hull is made using the upper part, or right to left scan. Lines 7-15 allow to find the convex hull in only a subset of the entire population, depending on the type of problems. When both objectives are to be maximized (max-max), the Pareto front is located towards (∞, ∞), as such, the only solutions needed to be searched to find the respective section of the convex hull are those up to the maximum value of the y-coordinate (the second objective), otherwise the convex hull would start descending, and those are not the optimum solutions for a max-max problem. If the first objective is to be maximized but the second is to be minimized (max-min problem, meaning the Pareto front is towards $(\infty, -\infty)$), then the scan only needs to start in the solution with the minimum value of the y-coordinate (remember the solutions are sorted according to the x-coordinate), because those before have an higher value for the second objective, but a lower value for the first, which is not the type of optimum solutions for a max-min problem. Starting with the lowest y, means that it starts with the "best" solution considering the second objective, but as the value for the second objective gets worse, the value for the first (the x-coordinate) gets better, until it reaches the last solutions in the population, which is the one with the maximum value for the first objective. A similar analogy can be made for the other two. Apart from limiting the search of solutions in population from *start_idx* position to *end_idx* position, lines 16 to 24 constitute the Graham's scan algorithm to find the convex hull. Each convex hull section found is placed in the ranked_population list, meaning the first element of this list is the solutions with the lowest rank, in the second element are the solutions with the second lowest rank, and so on. The main loop makes this whole procedure repeats until more solutions than the population size Np are ranked.

As this ranking procedure is very unlikely to return the exact number of solutions needed, some need to be discarded. To select the solutions to discard, the crowding distance is used to assure that the solutions discarded are those in a more crowded location, and not isolated solutions, assuring, this way, a diver-

Algorithm 4.3 Rank solutions using convex hull.

Input: *population*
Output: *ranked_population*

```
1: ranked_population ← empty list
2: count ← 0
3: while count < Np do
4:    sort population according to type of problem        // min-min, min-max,
      max-max, max-min
5:    start_idx ← 1
6:    end_idx ← sizeof( population )
7:    if problem is max-max then
8:       end_idx ← index of maximum value of y-coordinate
9:    else if problem is min-max then
10:      start_idx ← index of maximum value of y-coordinate
11:   else if problem is min-min then
12:      end_idx ← index of minimum value of y-coordinate
13:   else if problem is max-min then
14:      start_idx ← index of minimum value of y-coordinate
15:   end if
16:   hull ← population[ 1 ]              // first element of population
17:   for solution in population[ start_idx : end_idx] do
18:      while sizeof( hull ) > 1 and turn( hull[ -2 ], hull[ -1 ], solution ) is
         clockwise do
19:         remove hull[ -1 ]
20:      end while
21:      if solution ≠ hull[ -1 ] then
22:         insert solution in hull
23:      end if
24:   end for
25:   remove all solutions in hull from population
26:   insert hull in ranked_population
27:   count ← count + sizeof( hull )
28: end while
```

sity in the population. The crowding distance is the same used in the NSGA-II algorithm, and described in [23].

Using the convex hull ranking and the crowing distance, the population is replaced for the next generation, according to algorithm 4.4. Due to the way the ranking procedure works, all solutions up to those in the last rank are guaranteed to survive to the next generation, and the first loop (lines 5-8) guarantees that, also counting the number of solutions still missing. From those solutions in the last rank, only those with the higher diversity should pass to the next generation, up to the number of missing solutions, and this is accomplished by calculating the crowding distance of all solutions in the last rank, sorting those solutions in reverse order using the crowding distance, i.e., those in less crowded areas are first, and selecting the missing number of solutions from the first sorted ones.

The main advantage of ranking the solutions using the convex hull instead of the Pareto dominance is the existence of efficient algorithms to compute the convex hull, on the other hand, one possible disadvantage is when the Pareto front is non-convex, the convex hull can have some difficulties, as the solutions in the non-convex area would get a worse rank, and may even be discarded, leading to a possible empty zone in the Pareto front.

Concave hull

But why go from creating the Pareto front using the Pareto dominance to the convex hull? There are a whole lot of possible solutions in the space between the two approaches. Consider the Pareto front in figure 4.5(a): all solutions presented there belong to the Pareto front, and will be in the first rank, using when using the Pareto dominance, but when thinking in the "big picture", i.e., on the final solution to be picked by the decision maker, it's obvious even to the naked eye, that some are better solutions, while others are worse. For instance, solutions similar to solution D, although belonging to the Pareto front, are very unlikely to be chosen

Algorithm 4.4 Replacing the population for the next generation, using convex hull ranking and crowding distance.

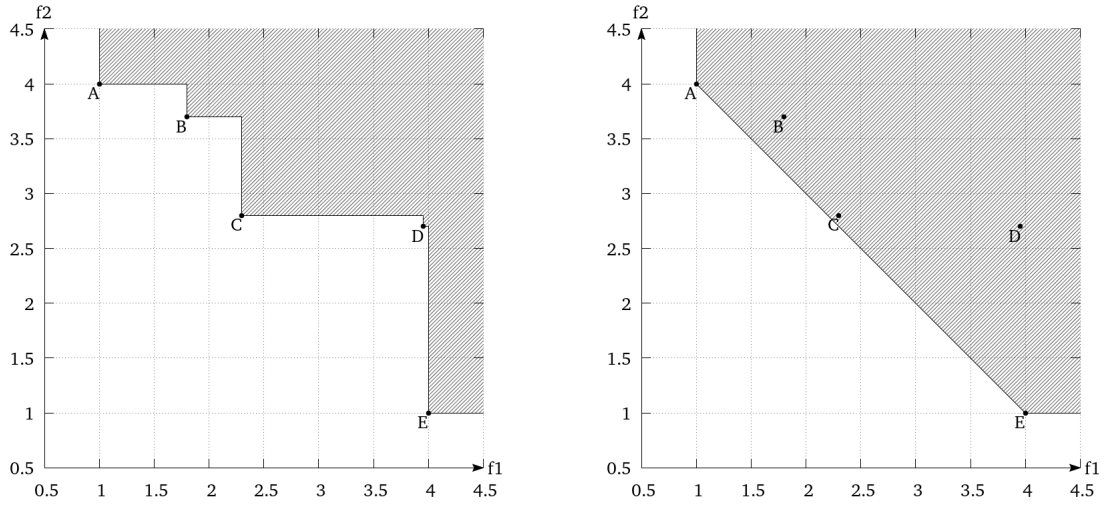
Input: *population*, *trial_solutions*
Output: *population*

```
1: count ← Np
2: insert trial_solutions into population
3: ranked_population ← rank population using convex hull
4: population ← empty list
5: for rank in 1 to sizeof( ranked_population ) - 1 do
6:    insert solutions in ranked_population[ rank ] to population
7:    count ← count - sizeof( rank )
8: end for
9: if count > 0 then
10:   calculate crowding distance for ranked_population[ -1 ]        // Last
      ranked_population
11:   sort ranked_population[ -1 ] according to reverse crowding distance
12:   insert first count solutions in ranked_population[ -1 ] into population
13: end if
```

by the decision maker, because both solutions C and E are almost as good as solution D in one objective, but are much better in the other. If the algorithm would "discard" those solutions from the Pareto front, not much information would be lost.

On the other hand, when considering the convex hull, like in figure 4.5(b), only solutions A and E would be in the Pareto front, i.e., only those solutions would be in the rank zero. This way, much information would be lost, as, for instance, both solutions B and especially C have a good trade-off, when compared with the solutions in the first rank. Using the convex hull ranking approach, solutions B, C and D would all be in a second rank, although the information each of them brings to the final decision process is very different.

Based on this reasoning, we introduce a concave hull selection mechanism, where not all non-dominated solutions (using the Pareto dominance) will be in the same rank, but not all will be passed to a later rank. Look again at figure 4.5(a): solution D, when compared with solutions C and E, makes an angle of almost 90º, i.e., looking at the triangle formed by solutions C, D and E, is almost a right-angled triangle. The concept of Pareto dominance can be seen as right-



(a) Pareto dominance

(b) Convex Hull

Figure 4.5: A possible Pareto front using two different approaches.

angled triangles, where the hypotenuse is the segment between the two solutions, and there cannot be any solutions bellow the other legs of the triangle. On the other hand, looking at figure 4.5(b), and considering only solutions A, C and E, is easy to see that those solutions are almost collinear, i.e., the angle formed by them is close to being a straight angle. This means that solution C is a good solution, and should not be passed to a secondary rank, as it would be if using the convex hull. The idea is to select a threshold angle α, between 90º (Pareto dominance) and 180º (Convex hull), and if the angle formed by three consecutive solutions is higher than the threshold angle, the last solution is kept in the same rank. Figure 4.6(c) shows the solutions selected for the first rank using this procedure, and using $\alpha = 135º$. As can be seen, solution D is, correctly, passed on to a secondary rank, while solutions B and C are kept in the first rank, as they have good trade-off values. However, if the threshold angle were higher, for instance, $\alpha = 150º$, then solution B would be passed to a second rank, and only solutions A,C and E would be in the first rank.

The algorithm to determine the concave hull is similar to the convex hull, but instead of checking that every three solutions in the hull make a counter-clockwise

turn, the angle between them is used to decide to keep or discard a solution.

First, the solutions are ordered and only those in the appropriate quadrant are searched to find the hull, much the same way as for the convex hull, considering also the type of problem. As the solutions are ordered, from one solution to the next, the value for the first objective never gets worse: either is better or is equal, in this latter case, with a worse value for the second objective. For this reason, every counter-clockwise turn can be accepted, as a "turn around" would never happen due to the ordering and the section of solution used in the search. In the concave hull approach, the clockwise turns are not automatically discarded, as they indicated a concave angle, and some but not all concave angles are to be accepted. Algorithm 4.5 shows the implementation of the search for a section of a concave hull, and works like this: for all solutions to be searched, if the two last solutions in the hull with the new solution make a clockwise turn, but the angle between the three is lower than α, it means that the last solution in the hull is worse than the other two being considered, and must be removed from the hull. This removal process is repeat until either the turn is not clockwise, or the angle in higher than α. Consider state in figure 4.6(a), where the hull is composed by solutions A,B,C and D, and solution E is under evaluation to be included. Clearly, when looking at the turn that the three solutions under consideration made, is a clockwise turn, meaning is not automatically accepted, is necessary to determine the angle. Assuming $\alpha = 135^{o}$, clearly the angle made by the three solutions is lower (is close to 90º, as seen before), as as such the last solution in the hull (D) is removed, and the last three solutions (now B, C and E) are again evaluated. This time the turn is counter-clockwise (see figure 4.6(b)), and as such the process can continue, and solution E is inserted in the hull, as shown in figure 4.6(c).

However, as is, the search could end up finding hulls like those in figure 4.7(a), as every three consecutive solutions in it are either counter-clockwise turns, or when clockwise, their angle is higher then α (a value of $\alpha = 135^{o}$ were used). But clearly, if before an argument was made about solution D not being a good

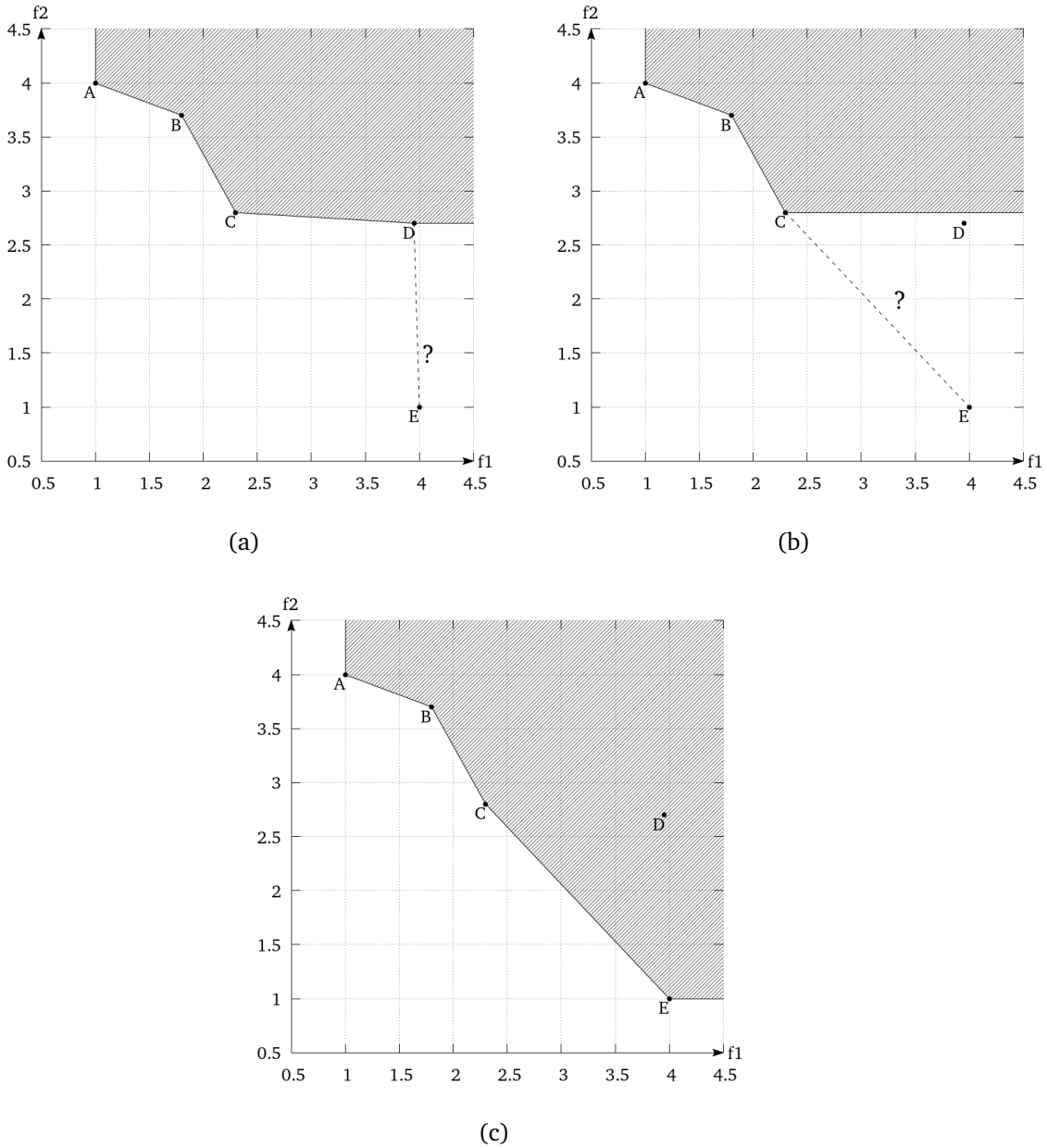


Figure 4.6: Concave hull search in progress. In (a), as CDE is a clockwise turn, does $\angle CDE$ is lower then the threshold? In (b) solutions BCE make a counter-clockwise turn. In (c) are the final hull.

Algorithm 4.5 Search for a section of the concave hull.

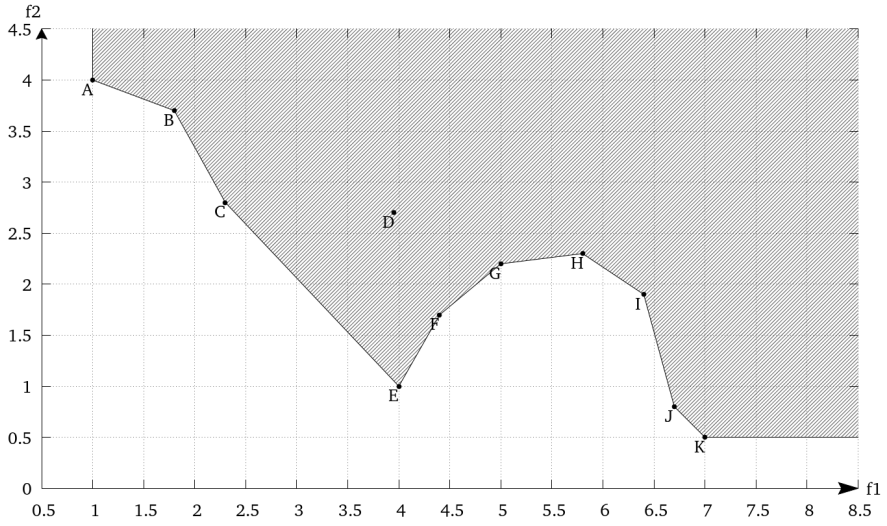
```
1: hull ← population[ 1 ]                    // first element of population
2: for solution in population[ start_idx : end_idx] do
3:    while sizeof( hull ) > 1 and turn( hull[ -2 ], hull[ -1 ], solution ) is clockwise
      and angle( hull[ -2 ], hull[ -1 ], solution ) < α do
4:       remove hull[ -1 ]
5:    end while
6:    if solution ≠ hull[ -1 ] then
7:       insert solution in hull
8:    end if
9: end for
```

final solution for the decision maker, solutions F, G, H and I are not good solutions either, as all have values for both objectives worse than solution E. Uninteresting solutions as those should be discarded, but as is, the algorithm for the concave hull selects them as the threshold angle is verified.

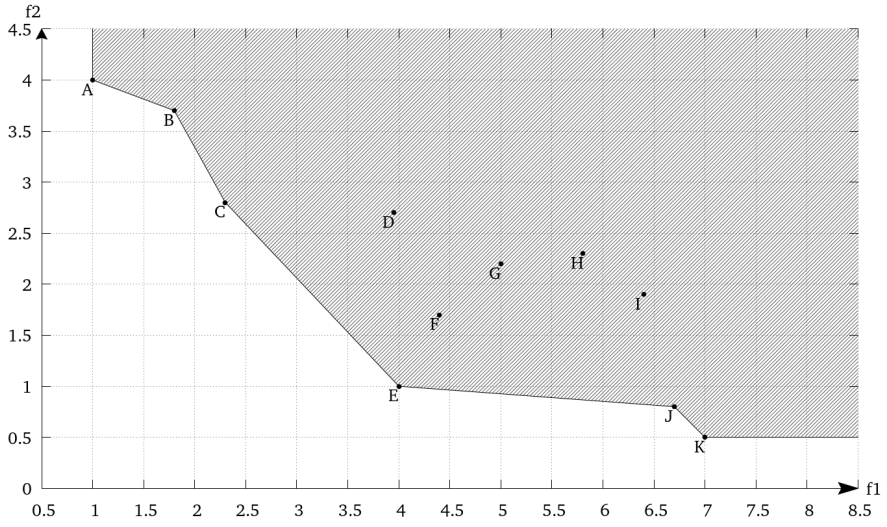
What all those solutions have in common, when looked under the Pareto dominance, is the fact that they all are dominated solutions, in this case, dominated by solution E. Relating the concave hull with the Pareto dominance, it's easy to see that the only solutions interesting to include in the concave hull are those in the non-dominated set, but not all solutions in the non-dominated set are to be included in the concave hull, as is the case of solution D.

Algorithm 4.6 shows the final algorithm to rank a set of solutions using the concave hull approach, where after the angle analysis, only those solutions that are not dominated by the last solution in the hull are to be included in it. This is much simpler to compute than the full Pareto dominance, as is only need see if the next solution is non-dominated relating to the last solution already in the hull, a full blown comparison of Pareto dominance is not needed.

When compared to the convex hull, this approach should allow a better approximation in the case of non-convex Pareto fronts, for obvious reasons, albeit needing one more parameter α, than any of the other two approaches. A good starting value for α is halfway through the Pareto dominance and the convex hull, around the 135º.



(a)



(b)

Figure 4.7: In (a) the problem with concave hull search. After correcting the search algorithm, it correctly identifies the concave hull in (b).

Algorithm 4.6 Rank solutions using convex hull.

Input: *population,* α
Output: *ranked_population*

```
1: ranked_population ← empty list
2: count ← 0
3: while count < Np do
4:    sort population according to type of problem        // min-min, min-max,
      max-max, max-min
5:    start_idx ← 1
6:    end_idx ← sizeof( population )
7:    if problem is max-max then
8:       end_idx ← index of maximum value of y-coordinate
9:    else if problem is min-max then
10:      start_idx ← index of maximum value of y-coordinate
11:   else if problem is min-min then
12:      end_idx ← index of minimum value of y-coordinate
13:   else if problem is max-min then
14:      start_idx ← index of minimum value of y-coordinate
15:   end if
16:   hull ← population[ 1 ]              // first element of population
17:   for solution in population[ start_idx : end_idx] do
18:      while sizeof( hull ) > 1 and turn( hull[ -2 ], hull[ -1 ], solution ) is
         clockwise and angle( hull[ -2 ], hull[ -1 ], solution ) < α do
19:         remove hull[ -1 ]
20:      end while
21:      if solution is not dominated by hull[ -1 ] then
22:         insert solution in hull
23:      end if
24:   end for
25:   insert hull in ranked_population
26:   remove solutions in hull from population
27:   count ← count + sizeof( hull )
28: end while
```

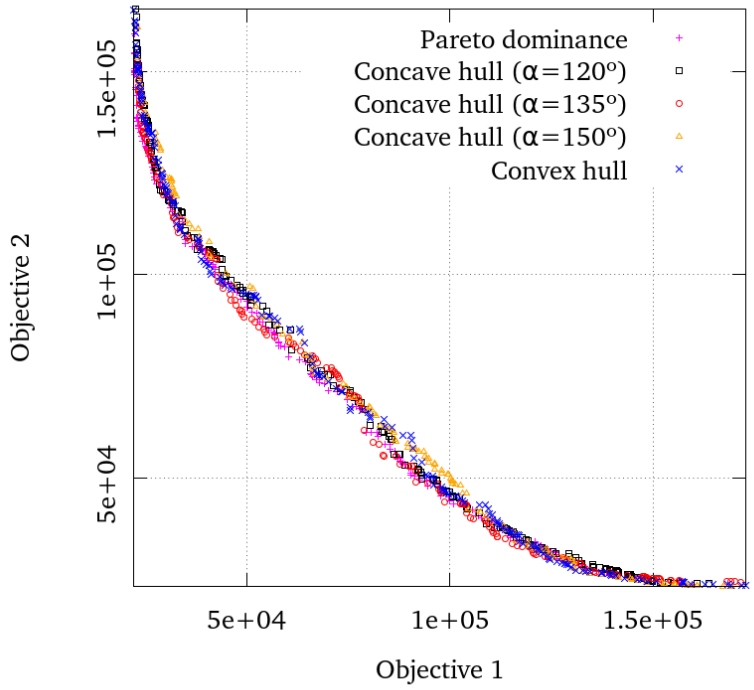


Figure 4.8: Pareto front obtained using different replacement operators.

In figure 4.8 are the Pareto fronts obtained for one execution of a multi-objective TSP, using all three approaches, and with different values for α in the concave hull approach. Unlike in the different repair mechanisms, here is not clear which is the best overall option, and probably different executions of the same configuration will return different values, as such further analysis is needed to conclude about the replacement operator. All three approaches will be available in the final evolutionary multi-objective optimization algorithm.

Further statistical analysis is needed to see which type of replacement operator is expected to perform better. In figure 4.9 are the boxplot of the results of the hypervolume quality indicator, for the kroAB100 problem, a multi-objective traveling salesman problem, using a population of 500 individuals, with $F = 0.9$ and $CR = 0.7$, with a maximum of 500 generations. It's clear that the Pareto dominance based replacement operator returns the best non-dominated solutions, with the convex hull (CVH) returning the worst and the concave hull (CCH) some-

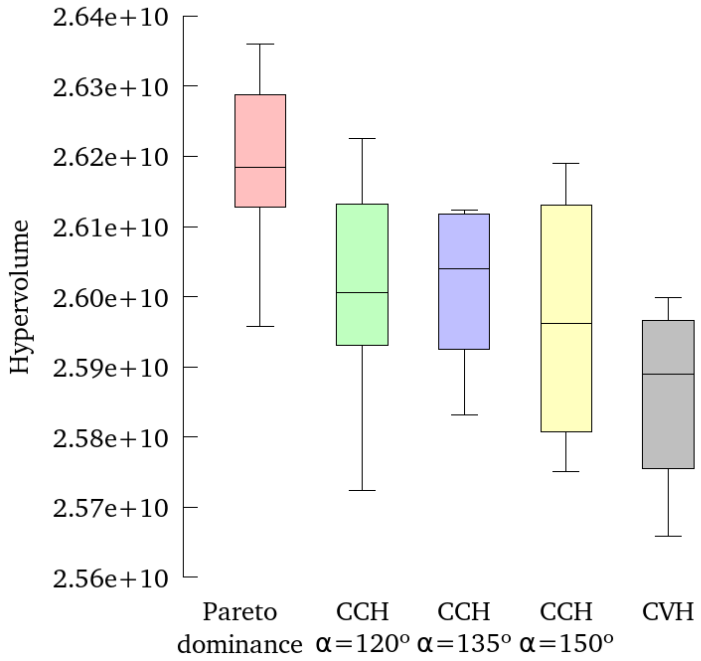


Figure 4.9: Hypervolume's boxplot for 10 executions of a multi-objective TSP, with a maximum of 500 generations, for replacement operators based on the Pareto dominance, on the concave hull (CCH) with different values for α, and on the concave hull (CVH).

where in the lower middle. However, just looking at different values for α, the middle 135º return the best non-dominated set of solutions, and the most regular, with the values for the different executions closer to each other than any other approach.

The hypervolume is a unary indicator that returns the value of the area covered by the Pareto front, considering one reference point (refer to section 2.2.4 for further details), and table 4.1 present the mean and average values the hypervolume for these approaches where is clear both the best non-dominated solutions, and the more regular solutions using the concave hull with $\alpha = 135º$, albeit with worse results. But have to be said that these conclusions are drawn based on the hypervolume indicator, other quality metrics may have different results. The cardinality of the Pareto front is very similar, with the Pareto dominance having as slightly higher number of solutions. However, in table 4.1 is also the mean and standard deviation of the time, in seconds, for ten executions of the algorithm, and here is

	Cardinality of	Hypervolume		Time	
	Pareto front	μ	σ	μ	σ
Pareto dominance	**206**	**2.6182E+10**	1.2086E+8	7704	422
Concave hull ($\alpha = 120^o$)	195	2.6011E+10	1.5873E+8	3979	1346
Concave hull ($\alpha = 135^o$)	194	2.6015E+10	1.1200E+8	**3748**	803
Concave hull ($\alpha = 150^o$)	194	2.5967E+10	1.7149E+8	3813	825
Convex hull	194	2.5860E+10	1.2823E+8	4995	731

Table 4.1: Statistical measures after for 10 executions of a multi-objective TSP, with a maximum of 500 generations, for the different approaches for a replacement operator.

clear that the complexity of the Pareto dominance is much higher than any of the hull-based approaches, and this is due to the existence of efficient algorithms for determining the hull, which is the main reason for even considering them in the first place. While the Pareto dominance needs, in average, slightly more than two hours for each execution of the algorithm, the concave hull needs a little more than 1 hour, and the convex hull lies somewhere in the middle, with an average around the hour and twenty minutes for each execution. From the hull-based variations, the one with $\alpha = 135^o$ is both the fastest and the one that returns the best non-dominated set. For reference, these results were on a Intel Core i7-5820K, 6 cores with hyper-threading, running at 3.30GHz and 32Gb of memory.

A final metric is the empirical attainment function, that allow the visualization of the results obtained by multiple executions of an algorithm, and even compare the outcomes of two algorithms to see exactly where they differ and by how much. López-Ibáñez et al. [54] developed a software package to plot these functions that we will use to compare these different approaches. In figure 4.10 are the side by side differences between the empirical attainment function of the Pareto domination approach with all the others. The dashed line represents the median attainment surface of each approach, and the bottom and upper line represent the best solutions attained in all executions on both approaches under consideration, and the worst solutions for any execution of both approaches, respectively, and the differences between approaches are the darker areas. In the top row we can

observe that the Pareto domination approach attains better results almost everywhere, but the concave hull, with both $\alpha = 120^o$ and $\alpha = 135^o$ got some results better in the lower right, i.e., for the objective 2. In the other two, both the concave hull with $\alpha = 150^o$ and the convex hull, attain almost always much worse results that the Pareto dominance approach, particularly the last one, as can be seen in the bottom right image.

4.5 Using multiple populations

In the multi-objective's adaptation of the repair mechanism, explained previously, each solution is repaired with the focus on only one objective, grouping this way, solutions that works toward the same objective. Although with a different motivation and implementation, in practice, this is a similar concept to having multiple populations.

Several authors have already used the differential evolution algorithm with multiple populations, for instance Tasoulis et al. [84], de Falco et al. [21, 20], Appoloni et al. [3] and Weber et al. [88], they all suggested using more than one population, but their focus was on parallel computing. All their implementations evolved the different sub-populations using different processors, i.e., each population evolves independently from all others, in its own processor. Of course, if each sub-population were to evolve completely isolated from all others, solution-wise, it would be no different than to use just one population, and execute the algorithm as many times as the number of populations. They all suggested a separate evolution of the sub-populations, but using some predefined rules to exchange solutions between sub-populations at certain times. For instance, Tasoulis et al. suggested that at each generation, the best solution of each sub-population can migrate to the next sub-population, in a network with a ring topology[1], according to a prede-

[1]Network topology is the way in which a set of computer nodes are connected to each other. The ring topology means that each computer is connected to two other nodes, forming a closed circuit.

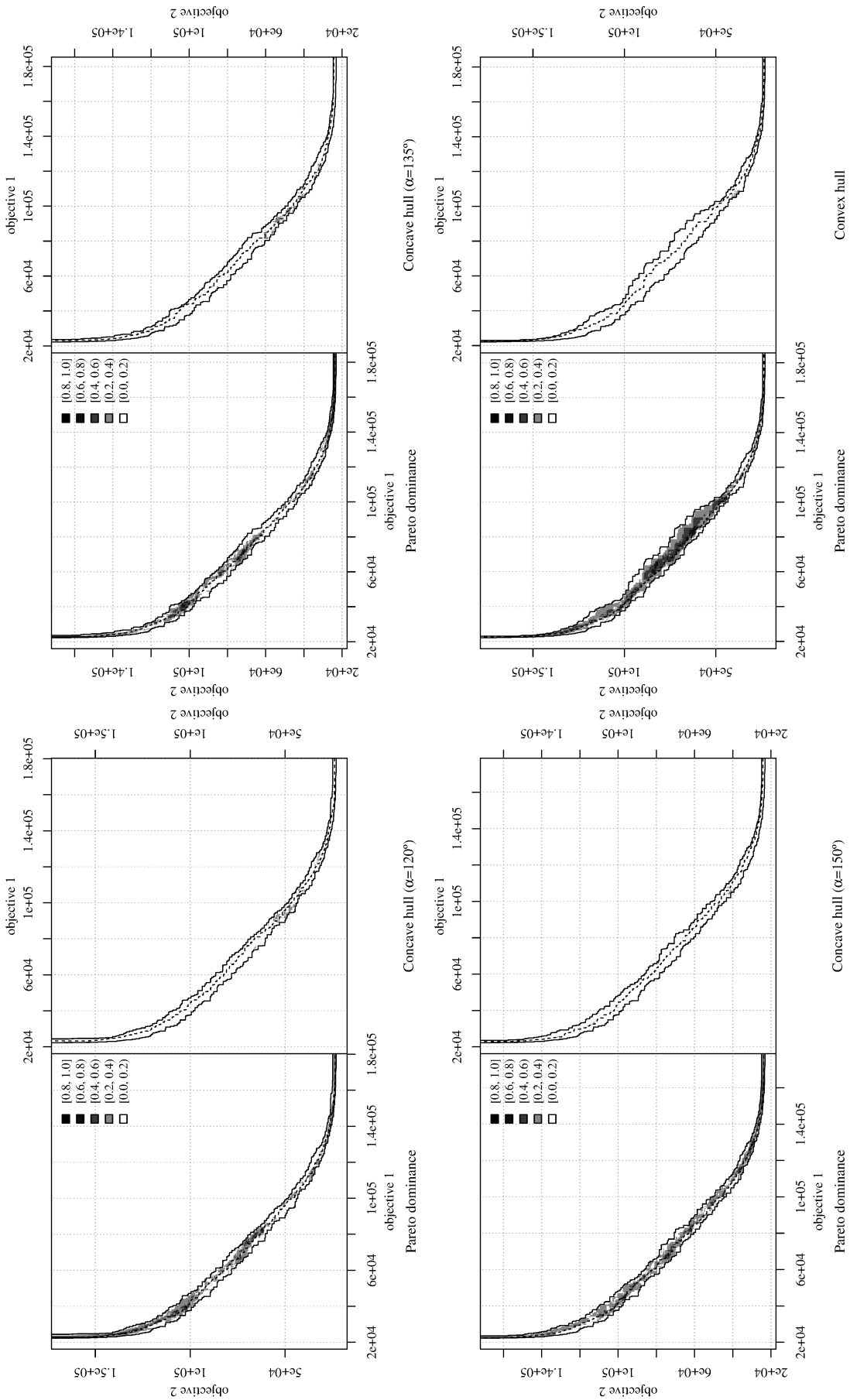


Figure 4.10: Differences in the estimated attainment function between the outcomes of the Pareto domination approach and all the others.

fined migration rate. This migration mechanism allows the sharing of information between sub-populations, and, if any sub-population find itself stuck in a local optimum, receiving some "foreign" solution may permit it to escape. The difference between the different implementations is the suggested network topology, and the rules to migrate solutions between the sub-populations.

Other authors [2, 16] used multiple populations not for parallel computing, but to use different parameters and/or mutation strategies in each sub-population, hoping for each sub-population to take advantage of the respective technique and propagate that to the other sub-populations when they migrate solutions.

Out motivation to experiment with multiple populations is neither parallel computing, nor to try different parameters or strategies. Distributed processing is already used in the single-objective algorithm to distribute the genetic operators between the different cores of the CPU, and the same implementation is to be maintained for the multi-objective algorithm. Also, as seen extensively, in multi-objective problems there is no "best" solution to be migrated between sub-populations.

Our purpose is to experiment a generalization of the repair mechanism, where each solution is repaired using its own objective, creating this way virtual groups of solutions, each group with his own repair objective. Based on this concept, the usage of multiple populations was implemented, using a sub-population to evolve each objective, the difference being that instead of migrating solutions from population to population, we decided to use completely independent populations. As this his, in fact, the same as using a greedy repair algorithm for each objective, at the end of the evolutionary process, the solutions are very good for each objective individually, but very bad for every other objective (see figure 4.3 on page 109), instead of using always separate populations, these are used in the beginning of the evolution, and at a certain δ point, all sub-populations are merged into one, and the evolutionary process continues with just this single population. The idea is

to initially reach good values for each objective regardless of the other objectives, and then, with all the solutions together in the same population, evolve those solutions to, hopefully, obtain good trade-offs between each objective.

We choose this rather unconventional mechanism to merge the different sub-populations instead of migrating solutions between them, for two reasons: first, a similar concept is already implemented in the repair mechanism, as each virtual group of solutions evolves using random chosen solutions from all population, allowing a dynamic migration from one group to another; and second, in evolutionary multi-objective algorithms there isn't one "best" solution to use to migrate to another sub-population, and using a random solution don't seemed justified as, again, this is similar to what is done in the repair mechanism.

As to when to join the different sub-populations into one, considering the evolutionary multi-objective optimization algorithm uses as a stopping criteria the maximum number of generations, we decided to base this decision on the number of generations already calculated, from the maximum number of generations to calculate. Several options was tested: from joining the solutions after 10% ($\delta = 0.1$), of the maximum number of generations have been calculated, after 20% ($\delta = 0.2$), after 30% ($\delta = 0.3$), to a maximum of after 40% ($\delta = 0.4$) of the number of generations. To simplify the writing, from now on, when referring to x% of the generations, we are referring to using separated populations for x% of the maximum number of generations given, after that number the sub-populations are joined into one, and the evolution proceeds.

Table 4.2 shows the mean and standard deviation of the hypervolume metric for 10 executions of a multi-objective TSP, after 500 generations. The first line ($\delta = 0.0$) serves as a base comparison, and means no sub-population were used, all solutions were always in the same (single) population. All others means that after x% of generations have been calculated, the solutions are joined into a single population and the evolution continues until the maximum number of generations is reached. From the values in the table we can see that, considering the

δ	Cardinality of Pareto Front	Hypervolume	
		μ	σ
0.0	206	2.6182E+10	1.2086E+8
0.1	227	2.6208E+10	0.9174E+8
0.2	227	2.6326E+10	0.8116E+8
0.3	210	**2.6377E+10**	1.2924E+8
0.4	225	2.6283E+10	1.0496E+8

Table 4.2: Statistical measures, for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using two separate sub-populations, one for each objective. The top line ($\delta = 0.0$) means no sub-populations were used, all solutions were always in the same population, an serves as comparison.

hypervolume metric, using sub-populations always gave better results than using one single population, reaching its peak when $\delta = 0.3$. When $\delta > 0.3$ the algorithm don't have enough generations left to reach good trade-off solutions, and the results start to deteriorate.

Also, if when $\delta = 0.3$ we have the best overall result, the $\delta = 0.2$ case, although with a slightly lower value, it has a much lower standard deviation, which means that the different results are more consistent through different executions. This is confirmed with a boxplot of the values, in figure 4.11, here the much higher diversity of $\delta = 0.3$ is clearly seen.

The differences between the estimated attainment functions, when using a single population or using two populations with the different options on when to merge the populations back into one is shown in figure 4.12. In it we can observe not only which technique are better, but where does the better results come from, for instance, in the bottom right image we can see that joining the populations after 40% ($\delta = 0.4$) of the generations have passed, results in much better values in the extremes, i.e., for each objective when considered individually, which is consistent with using specialized populations for too much time, but the single population have better results in the zone of the trade-off solutions, as no specialized population is used. As for the others, the decrease of good results using the single population is clear, as the percentage of generations using separated

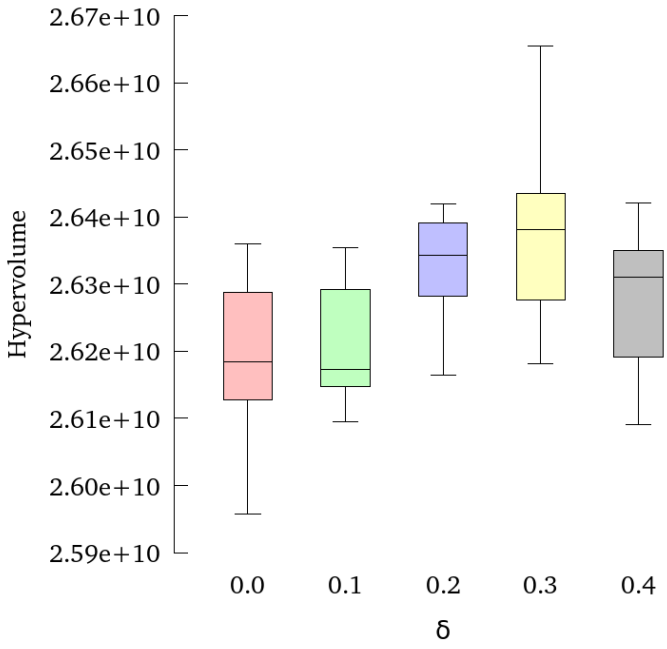


Figure 4.11: Hypervolume's boxplot for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using two separate sub-populations, one for each objective.

populations grows, until $\delta = 0.3$. But the one thing clear in the images is that the improvement in the results are mainly in the extremes, not in the trade-off area, where, supposedly, should lye the interesting solutions.

In figure 4.13 are the differences between the estimated attainment function as the percentage of generations used in separated solutions increases. In it we can see that in the extremes, the results always improve when increasing the number of generations with separated populations, but in the trade-off zone, after an initial improvement of results, is not clear that increasing the number of separated solutions results in better results, in fact, in the bottom right image is clear that raising δ from 0.3 to 0.4 with separated solutions, the results in the trade-off zone are much worse, meaning that this type of approach needs a number minimum of generations to get good trade-off solutions.

As, depending on the moment in which the populations were to be joined, this could lead to the algorithm not having enough time to reach those good trade-

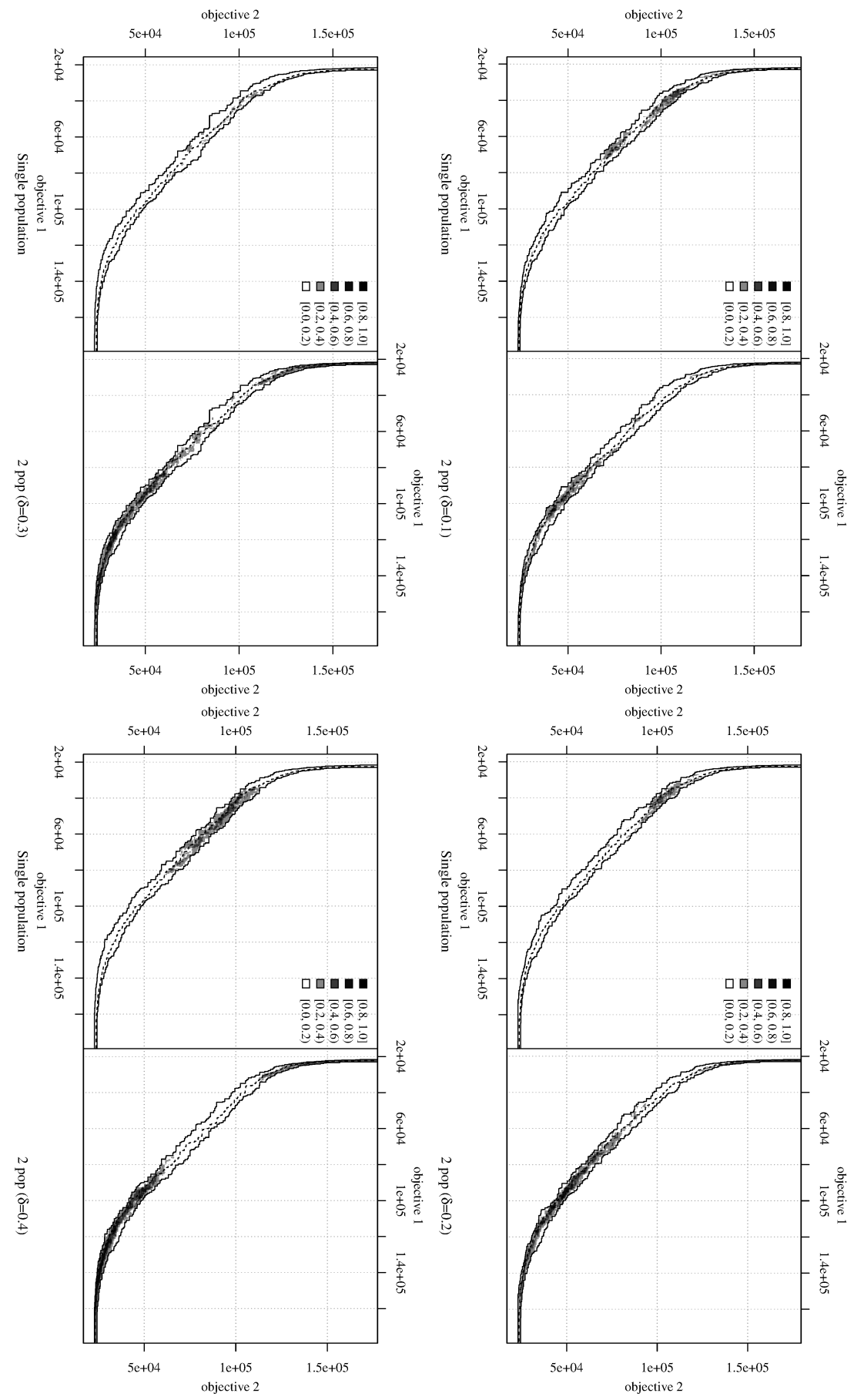


Figure 4.12: Differences in the estimated attainment function between using a single population or two sub-populations, with different option on when to merge back to one population.

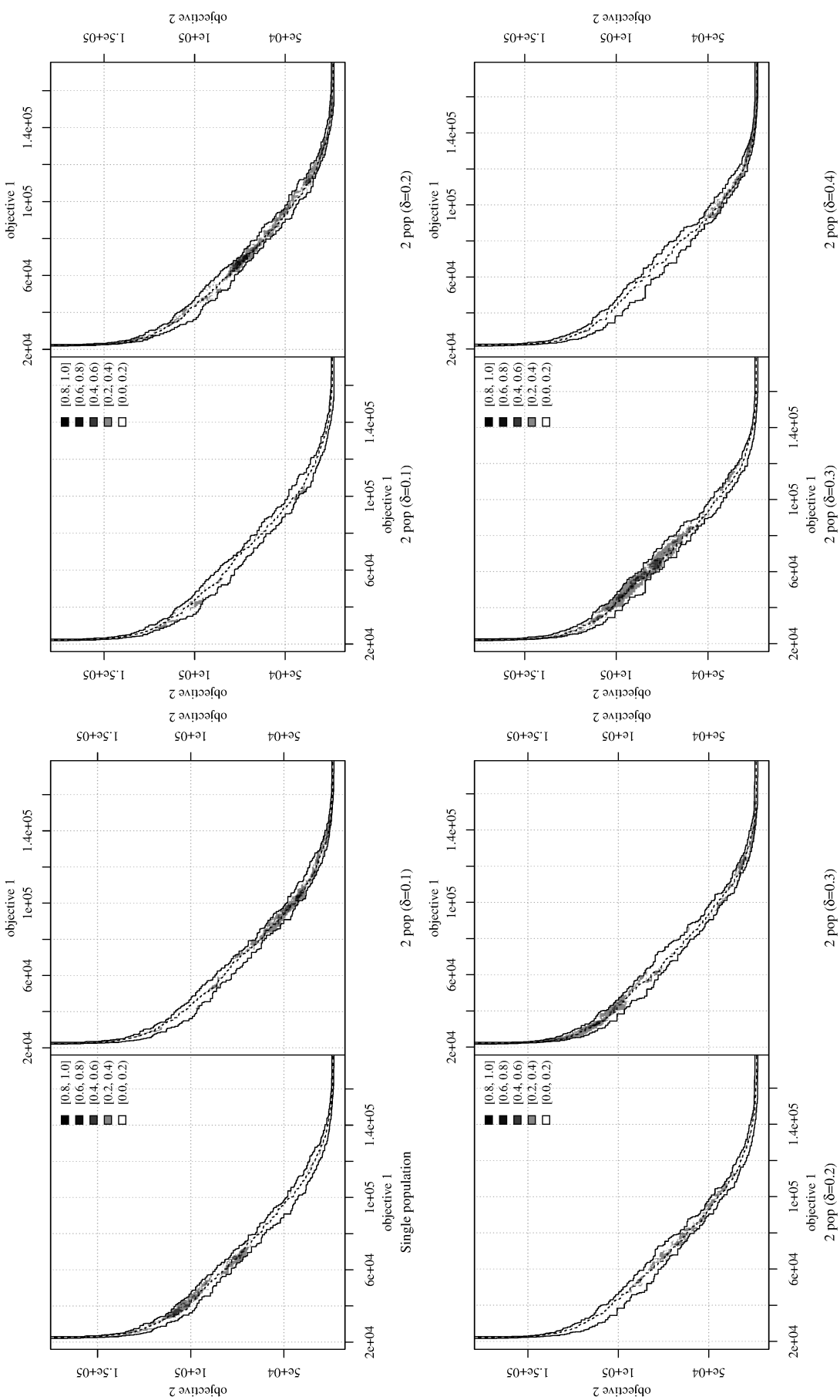


Figure 4.13: Differences in the estimated attainment function when raising the percentage of the number of solutions using separated solutions.

off solutions, as in the $\delta = 0.4$ above, an option to use one more sub-population, were introduced. This "extra" population is a general one, that would evolve without a defined objective, and the reasoning behind it is to have "specialized" sub-populations for each objective, and at the same time, a general population without objective. The algorithm would evolve all sub-populations separately, and join them at a certain moment in the evolution, as this would allow that good trade-off solutions were evolved along with the specialized ones, and when all solutions were to be joined into a single populations, there would be good solutions all around, not just for each objective. This extra population means that in an m-objectives problem, there would be $m + 1$ populations, where the first m populations would be assigned each their own objective, and the $(m + 1)$th population would evolve freely. Table 4.3 and figure 4.14 show the obtained results, for the same problem, with the same initial parameters, and the influence of this "extra" population is clear, as now there isn't the need for each sub-population to exist for so many generations, as the best value is reached is the sub-populations are all joined with just $\delta = 0.1$, and after this, the more generations the sub-populations are kept separated, the worse the results are. When using just "specialized" sub-populations, until all solutions are joined, there are no trade-off solutions, i.e., all solutions are good for either objective one or for objective two, but not for both. As there are no trade-off solutions, the algorithm needs to get very good solutions for either objective before joining all solutions and start to work in the trade-off solutions. With this extra population, those solutions exist from the start of the evolution, so the specialized sub-populations don't need to extend its work for so long, when the populations are joined, the lower quality "specialized" solutions, when mating with the already existing trade-off solutions, would allow to get better solutions, as there are generations to work on them.

In figure 4.16 are differences between the estimated attainment functions, when using a single population or using three populations with the different options on when to merge the populations back into one, and the results in the trade-

δ	Cardinality of Pareto Front	Hypervolume	
		μ	σ
0.0	206	2.6182E+10	1.2086E+8
0.1	215	**2.6367E+10**	1.4138E+8
0.2	212	2.6337E+10	1.2090E+8
0.3	217	2.6301E+10	1.4994E+8
0.4	217	2.6246e+10	1.7065E+8

Table 4.3: Statistical measures for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using three separate sub-populations: two specialized (one for each objective) and the third sub-population evolve freely.

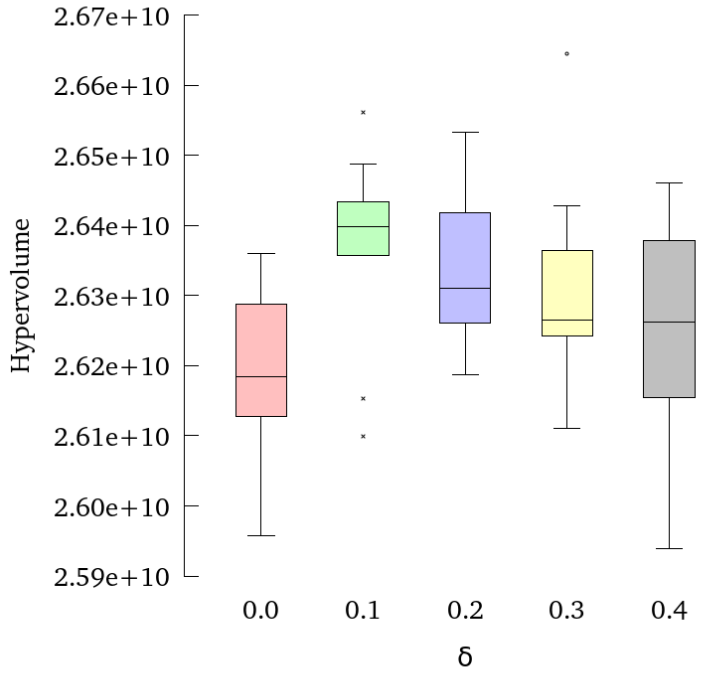


Figure 4.14: Hypervolume's boxplot for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using three separate sub-populations: two specialized (one for each objective) and the third sub-population evolve freely.

off zone, unlike those when using just two "specialized" solutions (see figure 4.12) are much better when using a third "global" population, although here we can see that as the number of generations using separated population increases, these results do not improve, in fact the results using a single population start to improve in these area than using separated populations.

When comparing the estimated attainment functions of increasing the number of generations using separated solutions, in figure 4.16, after the $\delta = 0.1$ mark, all others don't show any clear improvement, by the contrary, the results show that increasing the number of generations using separated populations decrease the quality of the overall solutions. Only when comparing the single population with the $\delta = 0.1$, do the results show a clear improvement, and not only in the extremes, but all around.

Whichever way to create the sub-populations and when to join them back together, the implementation needs to be consistent with the single-objective version. As in the differential evolution algorithm, the parameter Np is the number of solutions in "the" population, instead of creating each sub-population with that number of solutions, we split the value of Np between the number of sub-populations, i.e., if using p sub-populations, each population from $2, .., p$ would have $\lfloor Np/p \rceil$ solutions in it, and the first would have $Np - \lfloor Np/p \rceil \cdot (p-1)$, to make sure that altogether there are Np solutions. An high level description of the differential evolution multi-objective optimization algorithm is shown in algorithm 4.7.

Another idea is to use the already existing second population, the archive population, in the evolutionary process. Usually, the archive population is exactly that, an archive where solutions are stored, unchanged, until either the evolutionary process ends, or they are dominated by a new solution in the archive, and must be removed from it. But once they are in the archive, they are never changed, hence the usual name "archive" to designate this population. But nothing prevents this population to be evolved using the evolutionary operators, as the main popula-

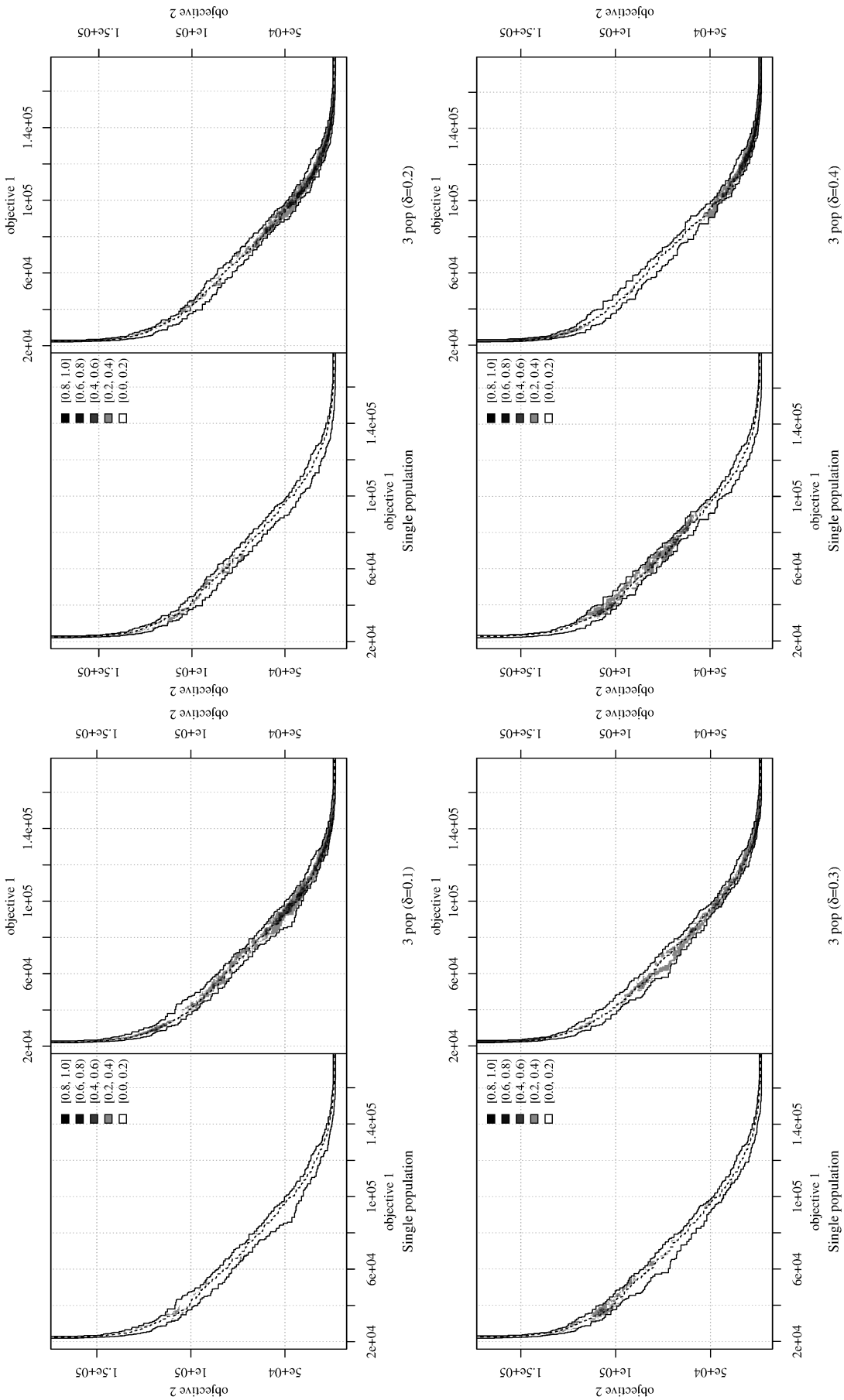


Figure 4.15: Differences in the estimated attainment function using a single population or three sub-populations, with different option on when to merge back into one population.

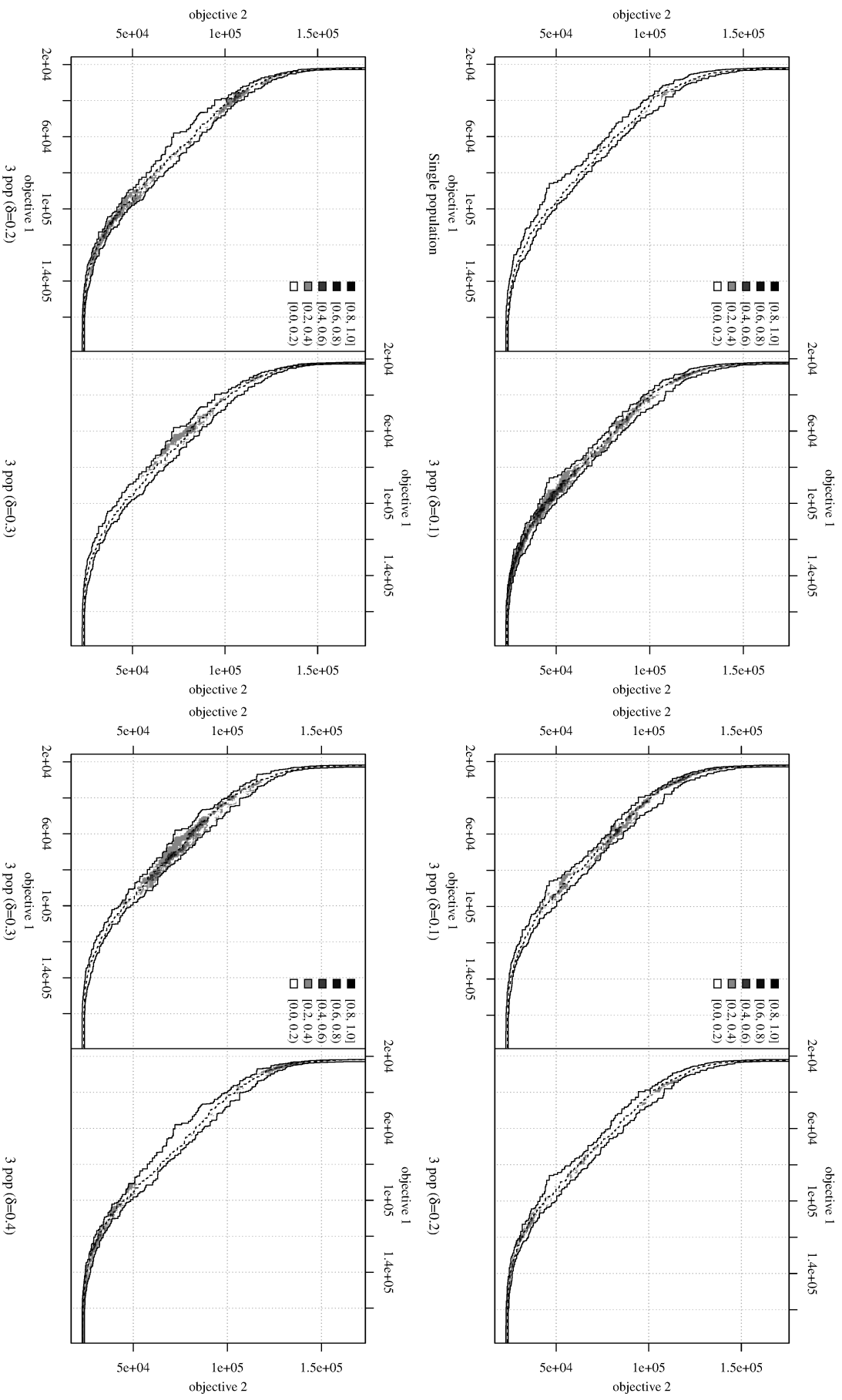


Figure 4.16: Differences in the estimated attainment function when raising the number of generations used by three separated populations.

Algorithm 4.7 High-level differential evolution multi-objective optimization algorithm, using multiple-populations.

```
 1: population ← empty list
 2: for i = 1..p do
 3:    sub-population ← initialization( )
 4:    insert sub-population in population
 5: end for
 6: save in the archive the non-dominated solutions in all sub-population
 7: while not end criteria do
 8:    for all sub-population in population do
 9:       mutants ← mutate( sub-population )
10:       repaired ← repair( mutants )
11:       trials ← crossover( repaired )
12:       sub-population ← replace( sub-population, trials )
13:    end for
14:    if in generation to merge populations then
15:       merge all solutions from each sub-population into one
16:    end if
17:    update archive of non-dominated solutions using all sub-population
18: end while
```

tion is, although this evolution cannot be made together, and has two differences: first, solutions in the archive are not going back to the main population, once they are in the archive, they can either help to improve the archive or not, but neither the original solutions in the archive nor the trial solutions, obtained after the crossover operator being used in the archive population, are send back to the main population. The second difference, is that the archive population is not replaced using the respective "trial archive", but rather updated, i.e., the archive population and the trial archive are joined and all dominated solutions in this merged population are removed. Evolving the archive population, as it contain only non-dominated solutions, should help to approximate the real Pareto front. Algorithm 4.8 represents an high level description of the process, and the difference from the base multi-objetive algorithm is after the populations being replaced for the next generation, but before updating the archive with the non-dominated solutions, the evolutionary operators are used in the archive population (except the replace, as explained), and the process of updating the archive uses not only the existing archive and the new population, but also the trial archive population.

Algorithm 4.8 High-level differential evolution multi-objective optimization algorithm, using the archive population in the evolutionary process.

```
1: population ← initialization( )
2: save in the archive all non-dominated solutions in the population
3: while not end criteria do
4:    // Evolve the main population
5:    mutants ← mutate( population )
6:    repaired ← repair( mutants )
7:    trials ← crossover( repaired )
8:    population ← replace( population, trials )
9:    // Evolve the archive population
10:   mutant_archive ← mutate( archive )
11:   repaired_archive ← repair( mutant_archive )
12:   trial_archive ← crossover( repaired_archive )
13:   update archive with solutions in population ∪ trial_archive
14: end while
```

	Cardinality of	Hypervolume	
	Pareto Front	μ	σ
Single population	206	2.6182E+10	1.2086E+8
Evolving the archive	215	**2.6288E+10**	1.2247+E8

Table 4.4: Statistical measures, for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using a single population and also evolving the archive population.

As can be seen in table 4.4 and figure 4.17, evolving the archive population along with the main population also improves the hypervolume quality metric, when compared with using only one population, and although not reaching the best average hypervolume value (see results of table 4.3), the results in the boxplot shows that the results are more consistent across different executions than any other multi-population approach tested.

When comparing the difference between the estimated attainment function, in figure 4.18, the results confirm that evolving the archive solutions produces a better outcome than using it just to store the non-dominated solutions, not just in the extremes, but also in the trade-off area, where the interesting solutions should be.

Overall, the results show that our evolutionary multi-objective optimization al-

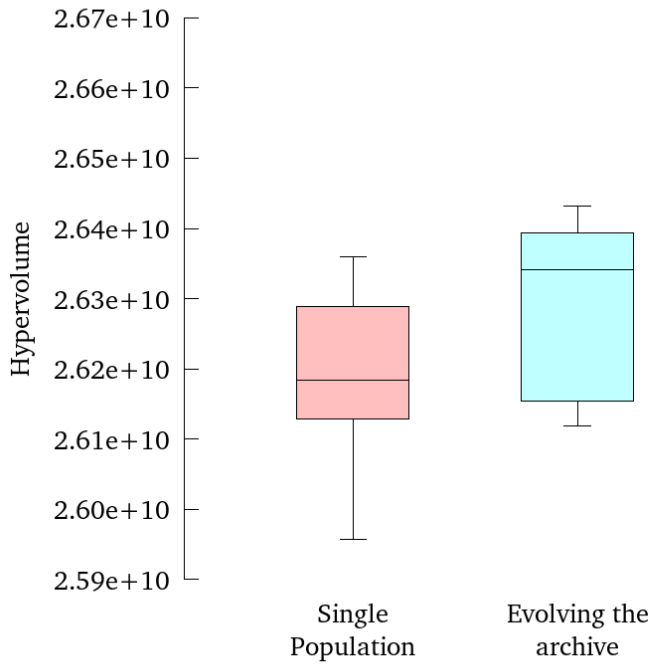


Figure 4.17: Hypervolume's boxplot for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using a single population and also evolving the archive population..

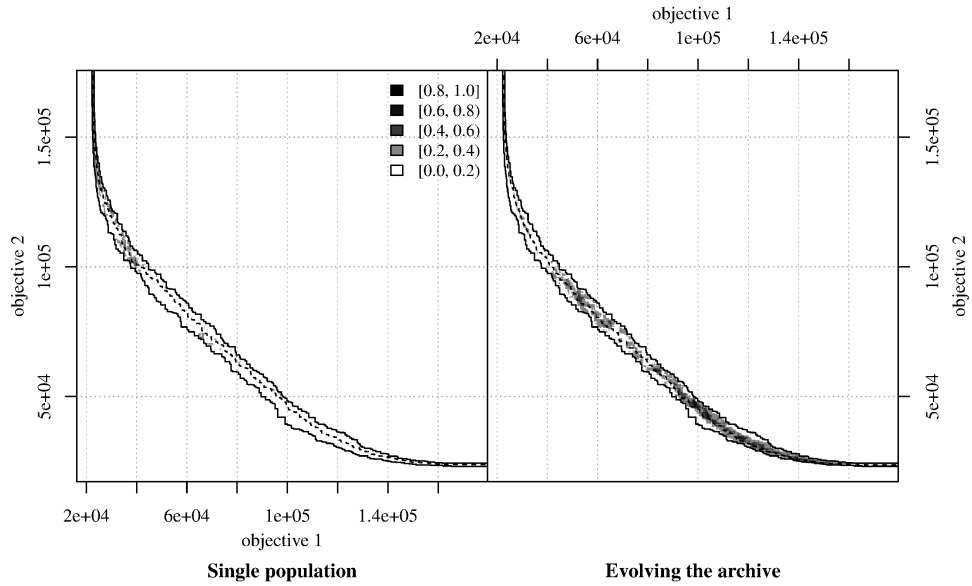


Figure 4.18: Differences in the estimated attainment function using the archive just to store the non-dominated solutions and evolving it.

gorithm produces better results using more than one population, even using our unorthodox method of not migrating elements between sub-populations, although using just two "specialized" sub-populations probably isn't worth it, as the good results are mainly due to existing good "single-objective" solutions, i.e., good solutions that only represent a single objective, not the entire multi-objective problem, and probably are uninteresting solutions from the decision maker perspective.

4.6 Self-adaptive parameters

Another technique used in differential evolution is to adapt the parameters during the evolutionary process, instead of using always the initial value. Although the differential evolution algorithm has very few parameters, and, according to one of the authors, is difficult not to get any result, regardless of the values for the parameters [45], several DE variations using adaptive and/or self-adaptive parameters were developed. From the three control parameters of the differential evolution algorithm, only the scale factor F and the crossover rate CR are usually subject to adaption, leaving the population size Np as the only free parameter. Two of the most used algorithms using adaptive parameters are the self-adaptive differential evolution (SaDE) by Qin and Suganthan [74, 73] and the jDE, by Brest et al. [10]. In the SaDE algorithm, both the learning strategy used to create the mutant individual (equations (2.1.4)-(2.1.8)) and the parameters F and CR are adapted during the evolutionary process. The first implementation of SaDE [74] used two strategies: DE/rand/1 (equation (2.1.4)) and DE/current-to-best/1 (equation (2.1.8)) for the mutation process, with a probability for each based on the success of each strategy in the last generations, i.e., a probability p_1 (with initial value of 0.5) is assigned to each solution in the population, meaning that strategy DE/rand/1 will be used in that solution with probability p_1, otherwise, the other strategy will be used). In each generation, the number of trial solutions that used each of the strategies and survived to the next generation is counted,

and after a certain number of generations, this value is used to recalculate p_1. A crossover rate is also assigned to each solution in the population, initially created using a normal distribution with center $CR_m = 0.5$ and variance 0.1, denoted $\mathcal{N}(CR_m, 0.1)$. In each generation, the value of CR for each trial solution that survived to the next generation is saved, and after a number of generations, the value of CR_m is recalculated using all saved CR values, and new values of CR for all solutions are recalculated using this new CR_m. The value of the scale factor F is the only one not self-adapted, as is randomly created using a normal distribution $\mathcal{N}(0.5, 0.3)$ for each solution in every generation. In [73] the authors presented another version of SaDE, in which they generalized for any number of strategies.

The jDE algorithm [10] is a self-adaptive algorithm, in which each solution is extended to include the respective parameters F and CR, and in each generation, the parameters are adapted according to

$$F_{i,g+1} = \begin{cases} F_l + \text{rand}() \cdot F_u & \text{if rand}() < \tau_1 \\ F_{i,g} & \text{otherwise} \end{cases}, \tag{4.6.1}$$

and

$$CR_{i,g+1} = \begin{cases} \text{rand}() & \text{if rand}() < \tau_2 \\ CR_{i,g} & \text{otherwise} \end{cases}, \tag{4.6.2}$$

where $\text{rand}()$ is a random uniform generator $\in [0, 1]$, F_l and F_u represent the lower and upper limit of the F parameter, and τ_1 and τ_2 represent the probability to adjust the parameters. Although this seems to be switching two parameters by four, in practice the authors suggested using always $F_l = 0.1$, $F_u = 0.9$, and $\tau_1 = \tau_2 = 0.1$.

However, setting apart the different strategies used by SaDE, in practice what both these self-adaptive algorithms do, is create new values for the parameters F and CR, using a somewhat controlled random generator.

Anyway, if for single-objective, using self-adaptive parameters is a way to ease the trial-and-error process of searching the best set of parameters for a certain problem, in multi-objective optimization problems this is even more useful, due to their final result not being one solution that can be easily compared with other from another set of parameters, but rather a set of non-dominated solutions. In evolutionary multi-objective optimization algorithms, the tuning of the parameters can be a crucial task in trying to get good trade-off solutions in the convergence vs. diversity objectives for the non-dominated set of solutions, and using a self-adaptive algorithm can ease this task, as the algorithm itself adapts the parameters according to its own results.

In the purposed self-adaptive mechanism, similar to the jDE algorithm, each solution need to have its own set of parameters, so each solution is extended by two values, to include both F and CR, and is represented by

$$X_i = (x_i, F, CR)\,, \tag{4.6.3}$$

where x_i is the actual problem solution, F and CR are the control parameters used to generate solution x_i, and X_i is our new high-level solution. In the initial population, random values are assigned to each parameter, using an uniform random generator $\in [0, 1]$.

To adapt the scale factor, we purpose to use the classic DE mutation to evolve the scale factor, i.e., before creating the mutant solution using three random solutions in the population, use those solutions' scale factor, and apply the same operation (in the real domain) to evolve the parameter, using a random value $\in [0, 1]$ instead of the scale factor in this mutation. The scale factor for the mutated solution V_i is defined by

$$V_{F,i} = \max\left(0,\ \min\left(1,\ X_{F,r1} + \mathrm{rand}() \cdot (X_{F,r2} - X_{F,r3})\right)\right) \tag{4.6.4}$$

and the crossover by

	Cardinality of Pareto Front	Hypervolume μ	σ
Fixed parameters	206	2.6182E+10	1.2086E+8
jDE	232	**2.6656E+10**	2.4565E+8
Evolving parameters	172	2.6478E+10	2.2827E+8

Table 4.5: Statistical measures for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using fixed parameters $(F = 0.9, CR = 0.7)$, the jDE algorithm and the evolving parameters idea.

$$V_{CR,i} = \max\left(0,\ \min\left(1,\ X_{CR,r1} + \text{rand}() \cdot (X_{CR,r2} - X_{CR,r3})\right)\right) \tag{4.6.5}$$

where $X_{F,k}$ and $X_{CR,k}$ are, respectively, the scale factor F and the crossover rate CR, of solution x_k, $k \in \{r1, r2, r3\}$, which are the three random selected solutions for the mutation operator, the $\max()$ and $\min()$ functions serve to bound the parameter in the $[0, 1]$ range, as both parameters need to be bounded by this values.

After the parameters are evolved, the usual operators are applied, but using $V_{F,i}$ instead of F in the mutation operator, and $V_{CR,i}$ instead of CR in the crossover. If the trial solution survived for the next generation, the values of the parameters used to create it are saved with it, and used to evolve the next solutions, otherwise they are discarded with the respective solution.

The results after 10 executions of the same multi-objective TSP with a maximum of 500 generations are shown in table 4.5 and figure 4.19. According to the hypervolume metric, both self-adaptive algorithms return better values that using fixed parameters, with the jDE algorithm giving the best overall result, and the results can be confirmed in the figure, where is shown that our approach produces some outliers, certainly due to the small number of experiences.

To try and justify the differences between the two self-adaptive algorithm, in figure 4.20 are the average of each parameter in each solution, for every generation, and in it is clear the difference, as in the jDE algorithm the crossover rate

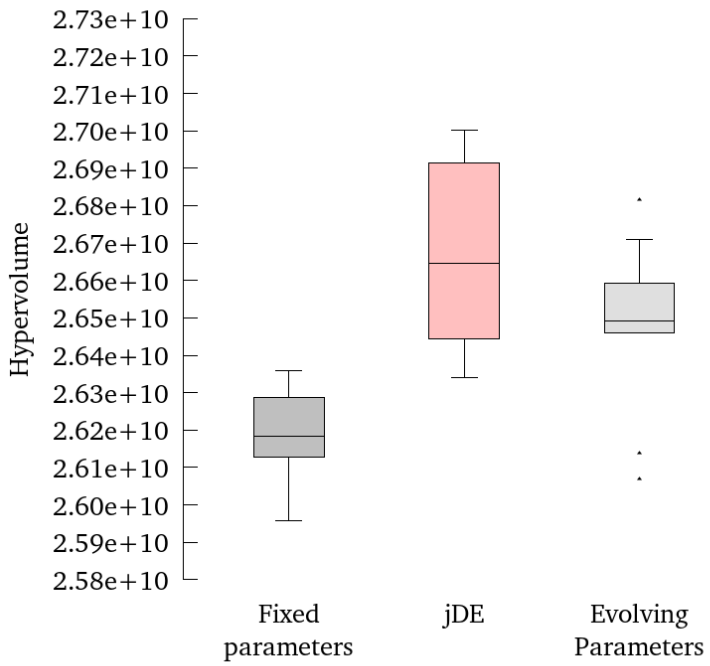
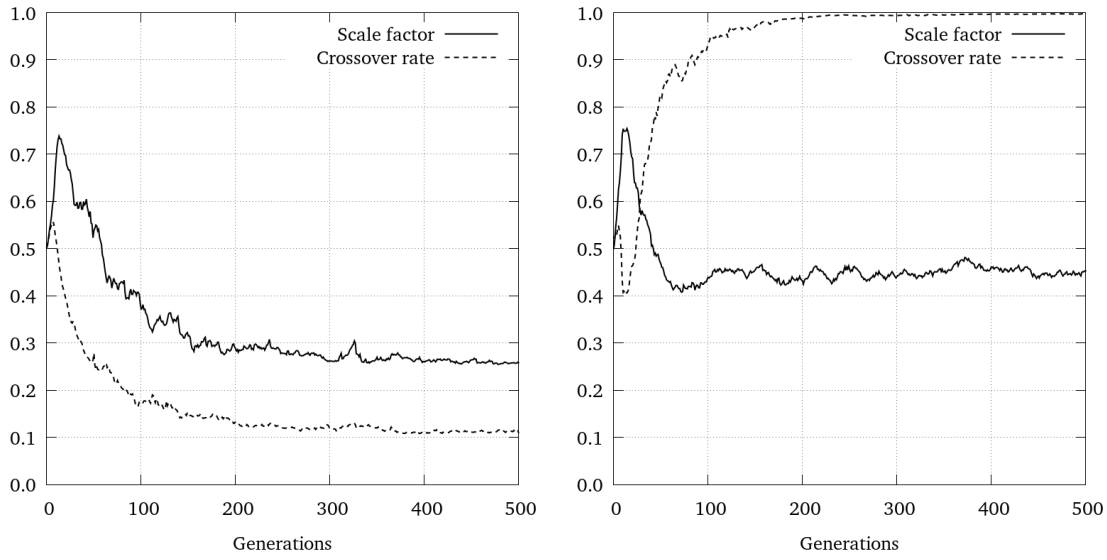


Figure 4.19: Hypervolume's boxplot for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using fixed parameters $(F = 0.9, CR = 0.7)$, the jDE algorithm and the evolving parameters idea.



(a) jDE algorithm (b) Evolving parameters

Figure 4.20: Average values of the scale factor F and crossover rate CR parameters, for each generation, in (a) for the jDE algorithm; in (b) for the algorithm evolving both parameters.

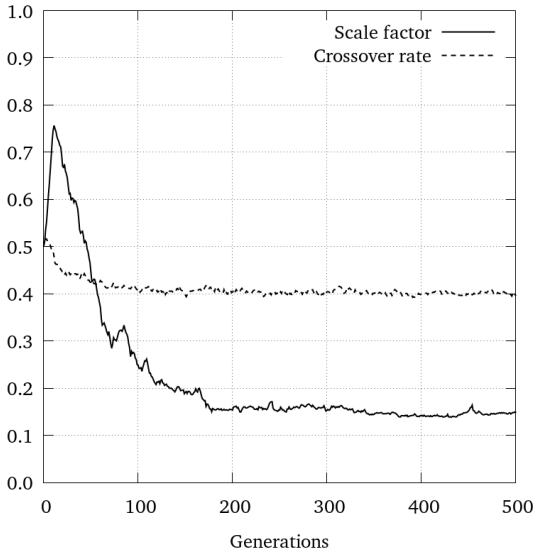
descends to a value around 0.1, and in our approach it raises to 1.0. This means that in our approach the crossover is almost never really used, as a value of 1 means that the trial solution would be, in fact, equal to the mutant solution. Also, probably due to the crossover rate value, the scale factor stabilizes around the 0.45, meaning that it never goes down enough to allow an exploitation of the domain space.

To try a workaround for the shortcomings of the previous approach, another formulation was tested for the crossover rate, and in it, it was just changed using a random value, as

$$V_{CR,i} = \mathrm{rand}(), \tag{4.6.6}$$

where $\mathrm{rand}()$ in a random uniform generator.

In figure 4.21 are the average of each parameter in each solution, for every generation, using a random crossover rate and evolving the scale factor. The results are quite different from evolving both parameters (see figure 4.20(b)), as



(a) Evolving F, random CR

Figure 4.21: Average values of the scale factor F and crossover rate CR parameters, for each generation, evolving the scale factor and a random crossover rate.

the crossover rate, although still higher that the scale factor, stabilizes around the 0.4 value. As for the scale factor, has a similar evolution, raising in the beginning of the process to allow the exploration of the search space, and then lowering to about 0.15, the lowest value of all self-adaptive algorithm tested, for a good exploitation of the search space.

The results after 10 executions of the same multi-objective TSP with a maximum of 500 generations for all self-adaptive algorithms are shown in table 4.6. According to the hypervolume metric, again all self-adaptive algorithms return better values that using fixed parameters, but now is this latter idea of evolving the scale factor along with a random crossover rate that gives the best overall result, with also the lowest standard deviation, meaning that the values are all very close, and this can be confirmed in the boxplot in figure 4.22, where are shown the more consistent results, and also that the worst results for this approach is only slightly worse than the best of the jDE algorithm.

Studying the differences between the different algorithms using the estimated attainment function, in figure 4.23, are the difference between using fixed parameters and all self-adaptive algorithms, and in it we can see that using fixed pa-

	Cardinality of Pareto Front	Hypervolume μ	σ
Fixed parameters	206	2.6182E+10	1.2086E+8
jDE	232	2.6656E+10	2.4565E+8
Evolving parameters	172	2.6478E+10	2.2827E+8
Evolving F, random CR	218	**2.7071E+10**	1.1387E+8

Table 4.6: Statistical measures for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using fixed parameters $(F = 0.9, CR = 0.7)$, the jDE algorithm and the two evolving parameters approaches.

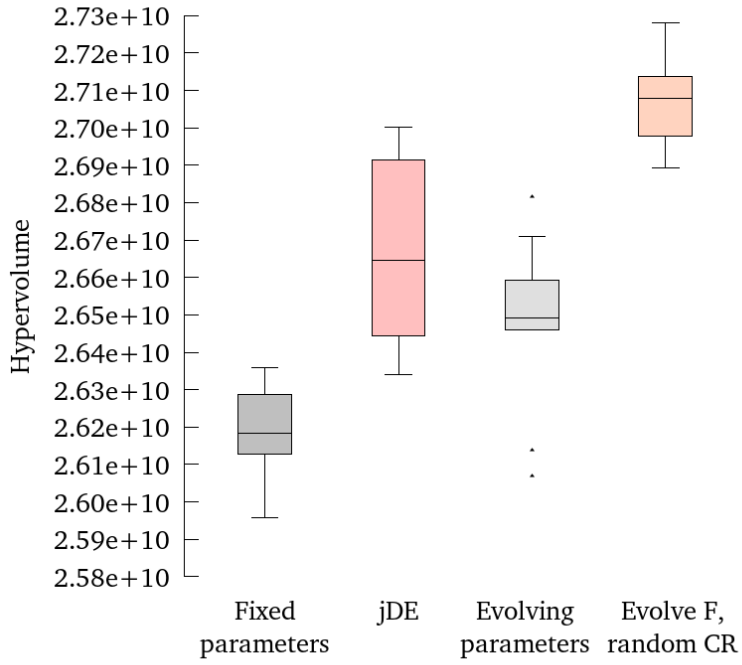


Figure 4.22: Hypervolume's boxplot for 10 executions of a multi-objective TSP, with a maximum of 500 generations, using fixed parameters $(F = 0.9, CR = 0.7)$, the jDE algorithm and the evolving parameters idea.

rameters results in better values in the extremes, especially when compared to the jDE algorithm, but all self-adaptive algorithms shows better results in the trade-off area. Also, the fixed parameters algorithm has the mean values of the estimated attainment function very close the the worse case scenario, when compared to all self-adaptive algorithms.

In figure 4.24 are the differences between the jDE algorithm and our two evolving parameters ideas. In it we can see that the jDE algorithm gives better results in the trade-off area when compared with the first proposal, although the latter has better results in the extremes. However, when comparing jDE to our second proposal, the results are quite different, has our approach provides better results throughout the entire Pareto front, with the advantage of not needing any extra parameters.

4.7 The MODECO algorithm

In the previous sections, all different approaches were compared against the base algorithm, using the Pareto dominance replacement operator with fixed parameters, using the best values found for the single-objective problem, from section 3.3.4 in the previous chapter. To find the best possible algorithm, the different options were combined when possible, using the best variations from the different options, giving a total of eighteen possible options, enumerated in table 4.7.

When using the concave hull as replacement operator, in algorithms modeco10 to modeco17, we use always $\alpha = 135^{\text{o}}$, as seen in section 4.4, and when using three sub-populations, we join all solutions in a single population using $\delta = 0.1$, as seen in section 4.5. When using fixed parameters, we use always $F = 0.9$ and $CR = 0.7$, as those values were the best ones for the single-objective problem.

In table 4.8 are the results after 10 executions of the same multi-objective TSP with a maximum of 500 generations for all different options of our multi-objective differential evolution combinatorial optimization algorithm, with the top three

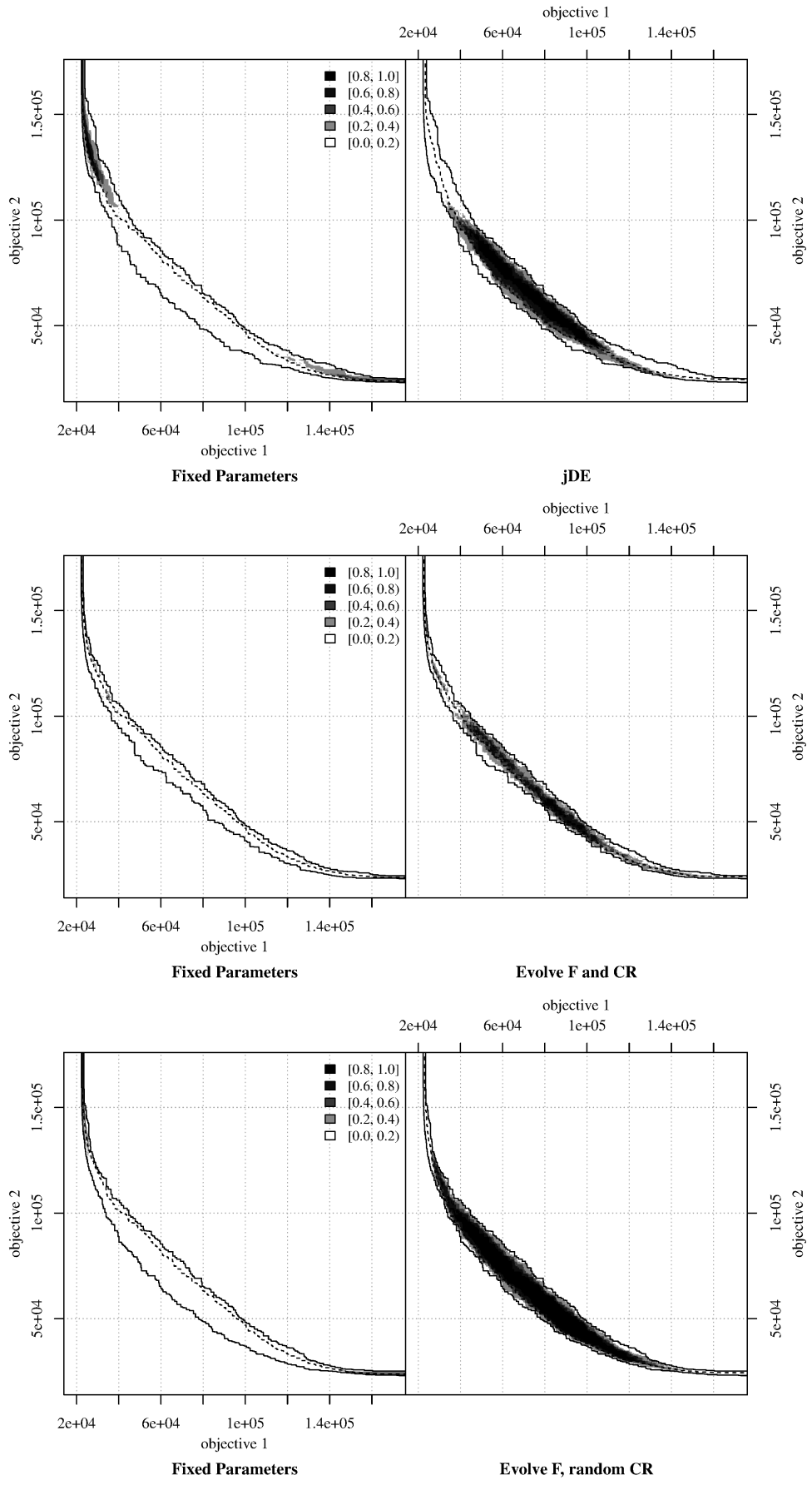


Figure 4.23: Differences in the estimated attainment function using fixed parameters ($F = 0.9, CR = 0.7$), the jDE algorithm and the evolving parameters idea.

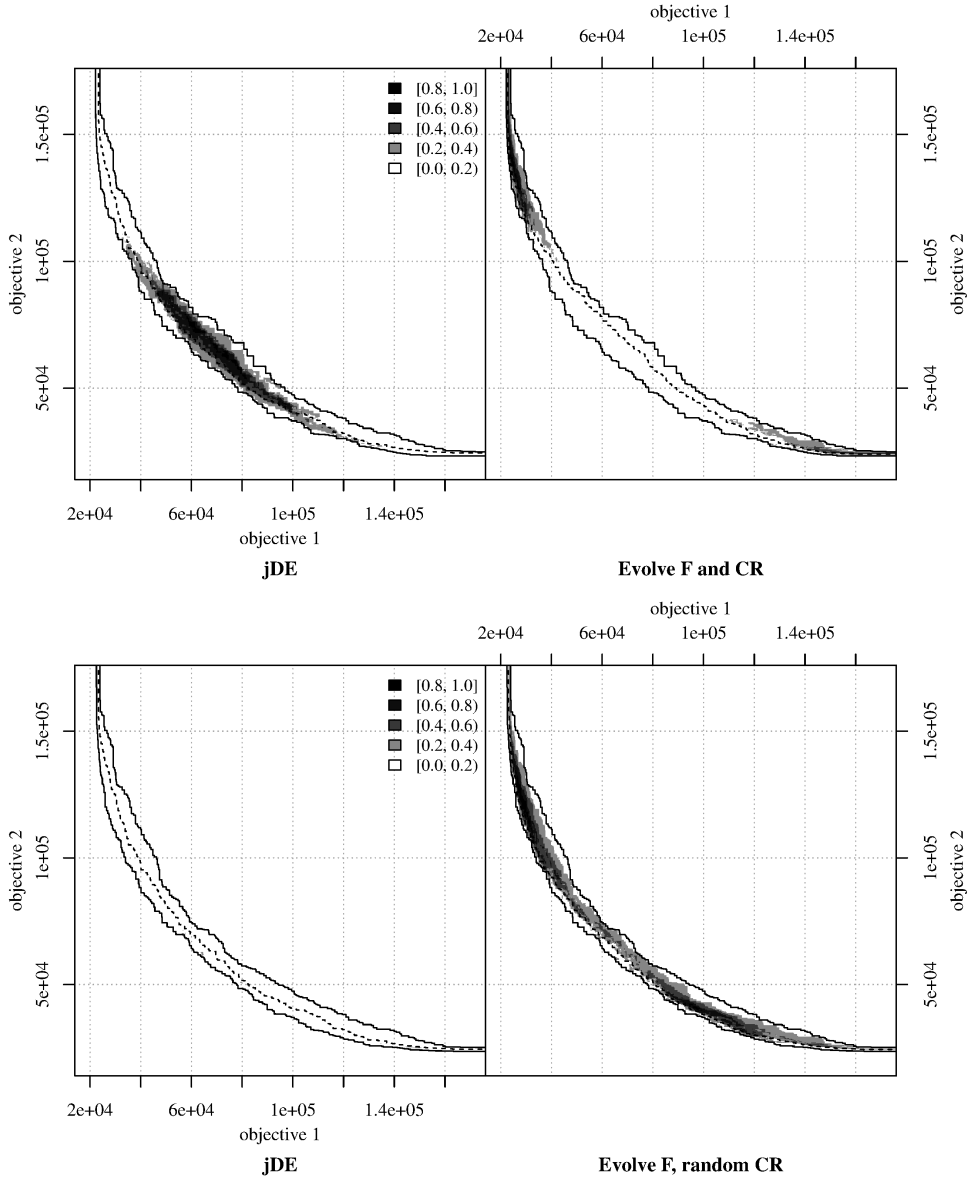


Figure 4.24: Differences in the estimated attainment function between the jDE algorithm and the evolving parameters approaches.

Algorithm	Replacement	Populations	Parameters
modeco1	Pareto dominance	One	Fixed
modeco2	Pareto dominance	One	jDE
modeco3	Pareto dominance	One	Evolving F, random CR
modeco4	Pareto dominance	Three sub-populations	Fixed
modeco5	Pareto dominance	Three sub-populations	jDE
modeco6	Pareto dominance	Three sub-populations	Evolving F, random CR
modeco7	Pareto dominance	Evolve archive	Fixed
modeco8	Pareto dominance	Evolve archive	jDE
modeco9	Pareto dominance	Evolve archive	Evolving F, random CR
modeco10	Concave hull	One	Fixed
modeco11	Concave hull	One	jDE
modeco12	Concave hull	One	Evolving F, random CR
modeco13	Concave hull	Three sub-populations	Fixed
modeco14	Concave hull	Three sub-populations	jDE
modeco15	Concave hull	Three sub-populations	Evolving F, random CR
modeco16	Concave hull	Evolve archive	Fixed
modeco17	Concave hull	Evolve archive	jDE
modeco18	Concave hull	Evolve archive	Evolving F, random CR

Table 4.7: Different variations of the algorithm to be tested.

results in bold. As we can see, the best results are all given using the Pareto dominance to replace the populations, and our self-adaptive algorithm, with the algorithm that creates three sub-populations (modeco6) being the one that has best average hypervolume value and also the higher cardinality of solutions on the Pareto front. The second best both in the hypervolume and in the cardinality is the one that evolves the archive population (modeco9), and the third one in the hypervolume metric is the single population version (modeco3).

To see the differences between these three version of the algorithm, figure 4.25 present the differences between them using the estimated attainment function. In it, we can see that although modeco6 reached the best average hypervolume value, when compared with both modeco9 (a) and modeco3 (b) we can see that those results are due to its good results mainly in the extremes, as both the other two show better results in the trade-off area. This is probably due to the three sub-populations used in modeco6, where two of then are “specialized” populations, searching solutions only in the extremes. When comparing modeco3 with mod-

Algorithm	Cardinality of	Hypervolume	
	Pareto Front	μ	σ
modeco1	206	2,6182E+10	1,2086E+08
modeco2	232	2,6656E+10	2,4565E+08
modeco3	218	**2,7071E+10**	1,1387E+08
modeco4	215	2,6367E+10	1,4138E+08
modeco5	232	2,6696E+10	1,7004E+08
modeco6	243	**2,7099E+10**	1,7624E+08
modeco7	215	2,6288E+10	1,2247E+08
modeco8	232	2,6739E+10	1,0589E+08
modeco9	237	**2,7081E+10**	1,4858E+08
modeco10	199	2,6015E+10	1,1418E+08
modeco11	219	2,6486E+10	1,1702E+08
modeco12	203	2,6449E+10	1,9950E+08
modeco13	202	2,6062E+10	1,1217E+08
modeco14	211	2,6890E+10	2,6168E+08
modeco15	210	2,6517E+10	1,8277E+08
modeco16	205	2,6111E+10	1,0655E+08
modeco17	215	2,6856E+10	2,0464E+08
modeco18	214	2,6550E+10	2,2706E+08

Table 4.8: Statistical measures for 10 executions of a multi-objective TSP, with a maximum of 500 generations, for the different options of the multi-objective differential evolution for combinatorial optimization.

eco9 (c) we cannot see a clear winner, as both of then are better than the other in a portion of the trade-off area, however modeco9 seems to have a slight advantage as it has it looks to have better values over a wider area, and also has a better hypervolume and cardinality than the modeco3 version.

In figure 4.26 we can see the boxplot of the hypervolume metric for all versions tested, and here we can see that there are clear differences between the different version of the algorithm. First of, we can see that the fixed parameters versions have always the worst results, regardless of the other variations. There are two possible reasons for this: either this is a proof of the advantage of an exploration/exploitation configuration, as the algorithm shows a large scale factor in the beginning of the evolution to allow the exploration of the search space and then decreases the value for the exploitation phase (see section 4.6); or a simpler reason could be that the parameters are not the best for this problem, as we used the best parameters chosen for the single-objective algorithm, but no attempt was made to calibrate the parameters for this problem.

Our self-adaptive algorithm, which, as seen previously in table 4.8, give the best results when using the Pareto dominance replacement operator (modeco3, modeco6, modeco9), but those good results are not replicated when using the concave hull ($\alpha = 135º$) as the replacement operator (modeco12, modeco15, modeco18). This is probably due to the way the parameters evolve in this operator, as they don't show a similar evolution as in the Pareto dominance operator. Figure 4.27 shows the evolution of the parameters using the modeco9, figure 4.27(a), and the modeco18, figure 4.27(b), algorithms, and although the crossover shows similar values, albeit slightly higher in modeco18, the scale factor don't show the same exploration/exploitation variation, at least not in the same number of generations. As the scale factor never reaches a low value, the exploitation phase never really happens, and as such the results are worse. Maybe with a higher number of generations, the results could be better, as the scale factor shows a descendent evolution, but a very slow one, not comparable at all with the one in figure

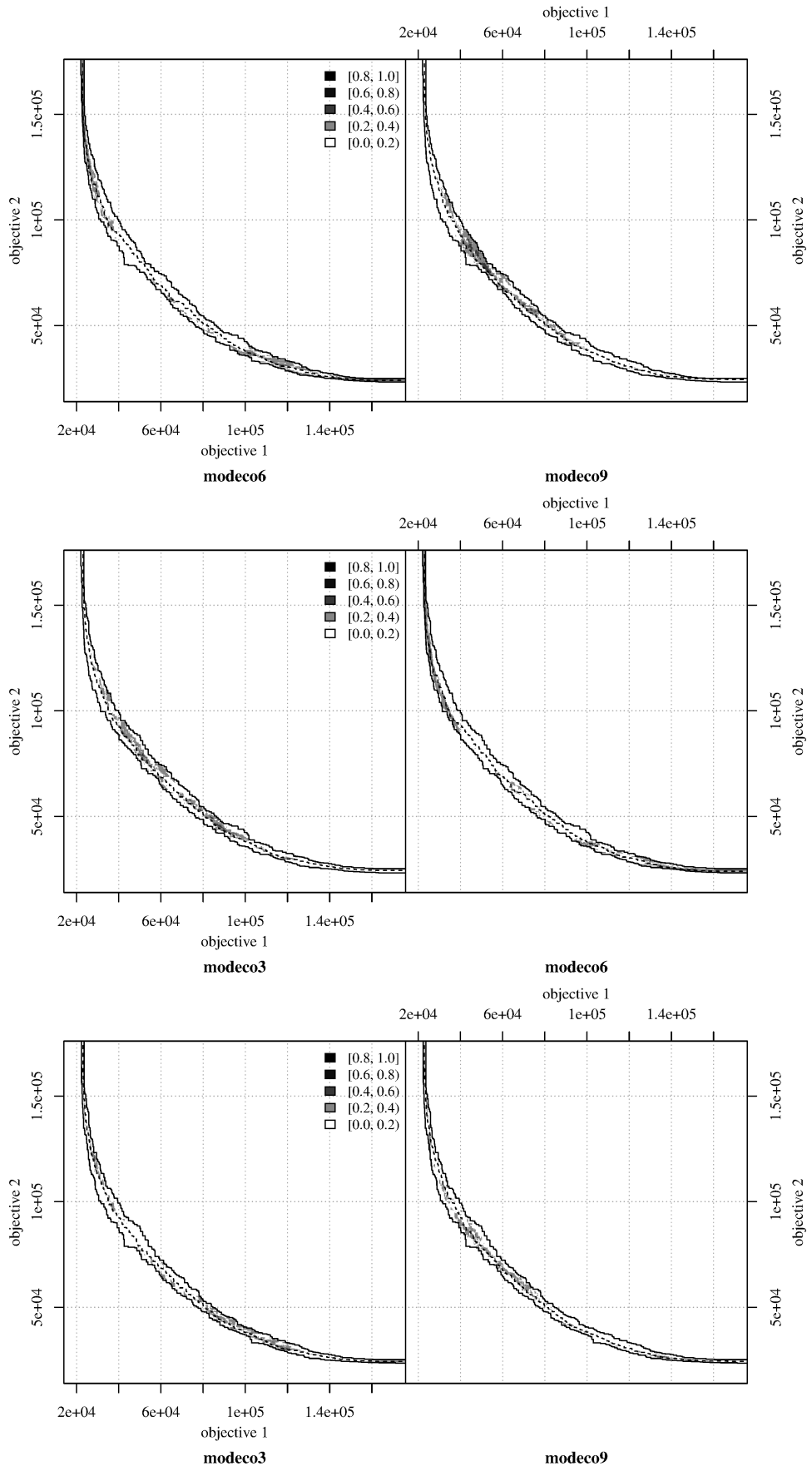


Figure 4.25: Differences in the estimated attainment function between the best three algorithms.

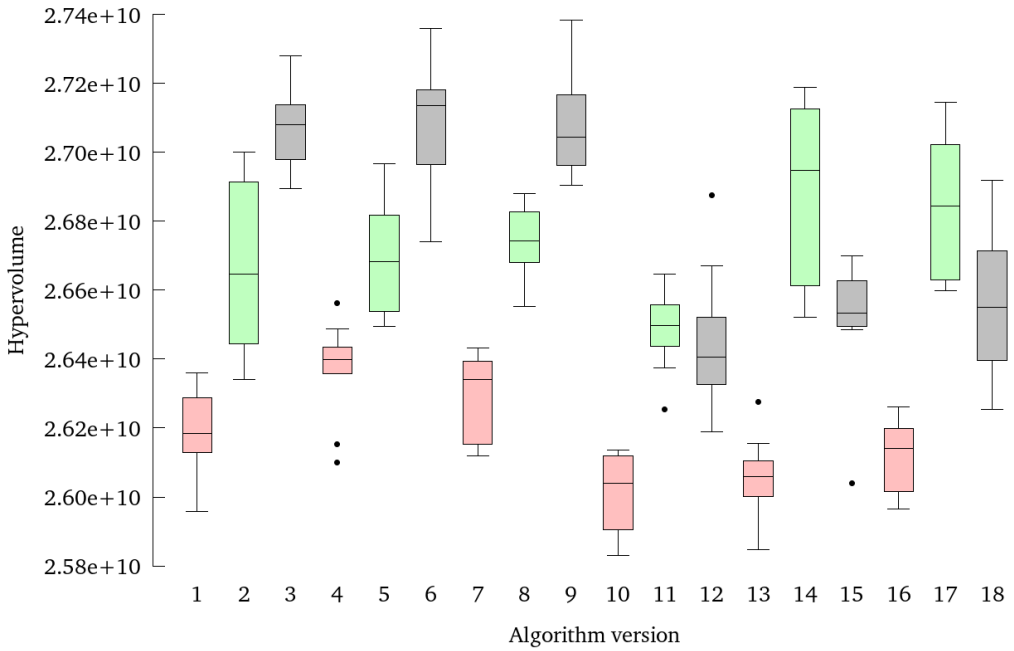


Figure 4.26: Hypervolume's boxplot for 10 executions of a multi-objective TSP, with a maximum of 500 generations for the different versions of the algorithm.

4.27(a).

The jDE algorithm, in figure 4.26, shows mid range values using the Pareto dominance replacement operator, but shows very acceptable results using the concave hull replacement with anything but a single population, in fact, modeco14, and modeco17, i.e., the concave hull replacement with the jDE algorithm, using either three sub-populations or evolving the archive, give the fourth and fifth better hypervolumes, respectively, although with a much higher standard deviation, meaning the results are not as consistent as the three best algorithms. When comparing the difference between the two using the estimated attainment function, in figure 4.28, we can see that the results are similar to first graphic in figure 4.25, i.e., those differences are consistent between using three sub-populations and evolving the archive, with the latter giving better results in the trade-off area.

From the previous analysis, we conclude that the best results are given by using the Pareto domination replacement operator, evolving also the archive population and using our self-adaptive algorithm, i.e., the modeco9 variation, and algorithm 4.9 shows a high-level description of the process.

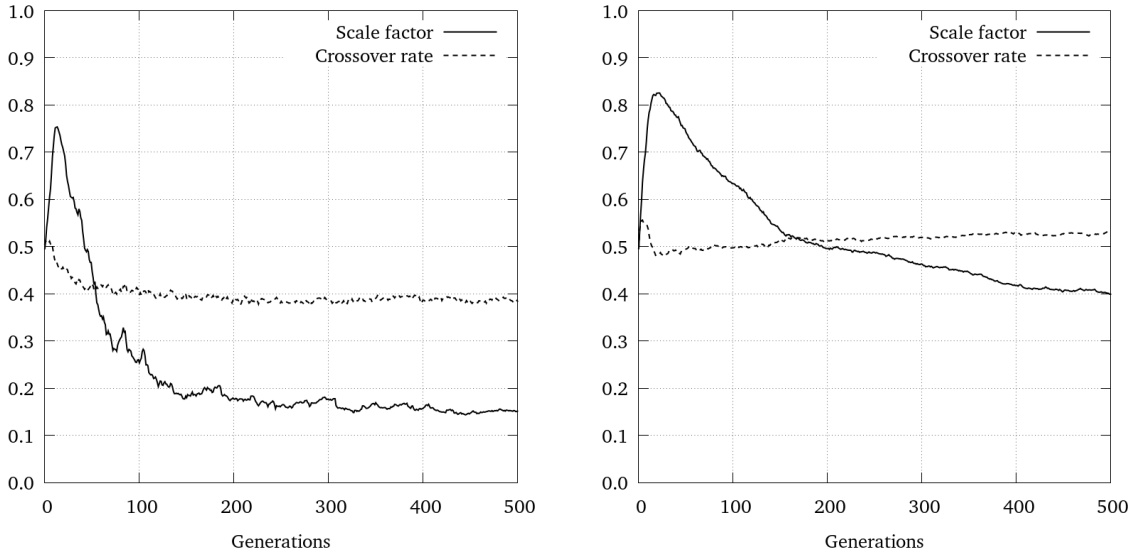


Figure 4.27: Average values of the scale factor F and crossover rate CR parameters, for each generation, using modeco9 and modeco18 algorithms.

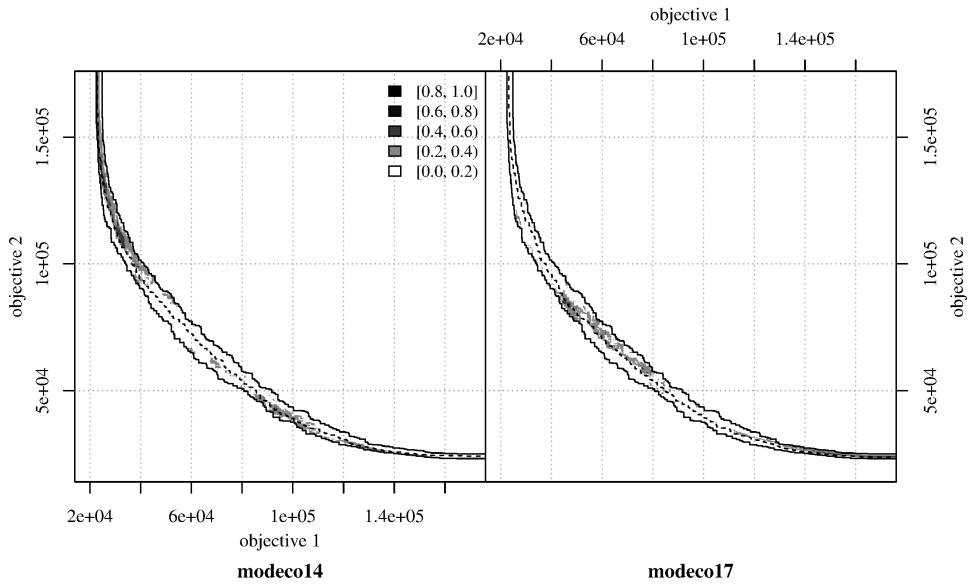


Figure 4.28: Differences in the estimated attainment function between modeco14 and modeco17.

Algorithm 4.9 High-level multi-objective optimization differential evolution for combinatorial optimization (MODECO) algorithm.

population$\leftarrow$ initialization()
archive $\leftarrow$ non-dominated solutions in *population*
while not end criteria **do**
  // mutation for population
  for all X_i in *population* **do**
    select three random solutions, X_{r1}, X_{r2}, X_{r3}
    $V_{F,i} \leftarrow \max(0,\ \min(1,\ X_{F,r1} + \text{rand}() \cdot (X_{F,r2} - X_{F,r3})))$
    $v_i \leftarrow x_{r1} \setminus V_{F,i} \otimes (x_{r2} \vartriangle x_{r3})$
    repair v_i
    $V_i \leftarrow (v_i, V_{F,i}, X_{CR,i})$
    insert V_i in *mutant population*
  end for
  // crossover for population
  for all X_i, V_i in *population*, *mutant population* **do**
    $U_{CR,i} \leftarrow \text{rand}()$
    $u_i \leftarrow$ calculate the trial solution using $U_{CR,i}$ as the crossover rate
    $U_i \leftarrow (u_i, V_{F,i}, U_{CR,i})$
    insert U_i in *trial population*
  end for
  use the NSGA-II replacement operator to replace the current *population* using the *trial population*
  // mutation for archive population
  for all A_i in *archive* **do**
    select three random solutions, A_{r1}, A_{r2}, A_{r3}
    $AV_{F,i} \leftarrow \max(0,\ \min(1,\ A_{F,r1} + \text{rand}() \cdot (A_{F,r2} - A_{F,r3})))$
    $av_i \leftarrow a_{r1} \setminus AV_{F,i} \otimes (a_{r2} \vartriangle a_{r3})$
    repair av_i
    $AV_i \leftarrow (av_i, AV_{F,i}, A_{CR,i})$
    insert AV_i in *mutant archive*
  end for
  // crossover for archive population
  for all A_i, AV_i in *archive*, *mutant archive* **do**
    $AU_{CR,i} \leftarrow \text{rand}()$
    $au_i \leftarrow$ calculate the trial solution using $AU_{CR,i}$ as the crossover rate
    $AU_i \leftarrow (au_i, AV_{F,i}, AU_{CR,i})$
    insert AU_i in *trial archive population*
  end for
  archive $\leftarrow$ non-dominated solutions in {*current archive* $\cup$ *trial archive* $\cup$ *trial population*}
end while

4.8 Results

The MODECO algorithm given in the previous section was implemented in Python, using the DEAP [27] and the SCOOP [37] libraries, as was the case for the single-objective algorithm. The SCOOP library allows for the actual calculations to be forked to a different core in the processor in the current machine, or to a different computer in the network if the setup includes this configuration, and as the results returns, another calculation is send to that "free" core/processor, until all computations are done, applying this way a parallel computation using a very simple implementation. In the MODECO algorithm, this is only used in the mutation, including the repair mechanism, and the crossover operators, as the other operators are not worth it, as their computation is fast enough without it.

Although our decision on the modeco9 variation, modeco3 and modeco6 were also implemented to compare the results across a wide range of problems.

The code was tested using several instances of the TSPLIB library, with different sizes, and using always a population five times the number of nodes in the problem. As a stopping criteria, was defined the maximum number of generations to be 500, and the code was executed 10 times for each instance.

Table 4.9 shows the cardinality of the Pareto front and the average and standard deviation for the hypervolume metric, and we can see that the modeco6 version, i.e, using three sub-populations, produces the best results for almost all instances, with modeco9 giving the best results on the other instances. In figure 4.29 are the boxplots of the hypervolume, where the results can be confirmed graphically, and where can be observed that the results of the modeco3 algorithm are almost always the worst ones.

However, when observing the differences between the algorithms using the estimated attainment function, in figures 4.30 to 4.41, we can see where do the results came from, i.e., if the good values are due to good trade-off solutions, or to good solutions in the extremes, which are not that interesting for the decision

	modeco3			modeco6			modeco9		
	#PF	Hypervolume		#PF	Hypervolume		#PF	Hypervolume	
		μ	σ		μ	σ		μ	σ
kroAB50	167	6,2562E+09	3,0649E+07	179	**6,2806E+09**	3,2826E+07	171	6,2587E+09	3,0103E+07
kroAB100	218	2,7071E+10	1,1387E+08	243	**2,7099E+10**	1,7624E+08	237	2,7081E+10	1,4858E+08
kroAC100	228	2,6753E+10	1,7890E+08	249	**2,6857E+10**	2,3187E+08	236	2,6849E+10	2,5932E+08
kroAD100	213	2,7059E+10	1,7583E+08	236	2,7205E+10	1,6706E+08	217	**2,7246E+10**	1,4042E+08
kroAE100	213	2,7186E+10	1,4232E+08	222	**2,7389E+10**	1,5379E+08	213	2,7182E+10	1,3125E+08
kroBC100	215	2,7112E+10	1,8524E+08	236	2,7249E+10	1,0967E+08	239	**2,7251E+10**	1,4188E+08
kroBD100	219	2,6895E+10	2,1903E+08	241	**2,7056E+10**	1,8588E+08	241	2,6912E+10	8,4873E+07
kroBE100	229	2,6596E+10	1,7426E+08	238	**2,6621E+10**	2,3391E+08	231	2,6597E+10	1,4184E+08
kroCD100	206	2,7539E+10	1,3189E+08	216	**2,7676E+10**	1,3301E+08	208	2,7636E+10	1,3082E+08
kroCE100	224	2,6829E+10	1,1308E+08	232	**2,7134E+10**	9,4700E+07	235	2,6927E+10	1,0281E+08
kroDE100	230	2,6495E+10	1,9998E+08	244	2,6609E+10	1,7852E+08	233	**2,6721E+10**	1,3226E+08
kroAB150	316	4,8963E+10	5,5193E+08	330	**4,9222E+10**	3,9663E+08	324	4,8984E+10	4,8333E+08

Table 4.9: Statistical results for different instances of multi-objective traveling salesman problems, using three different version of MODECO.

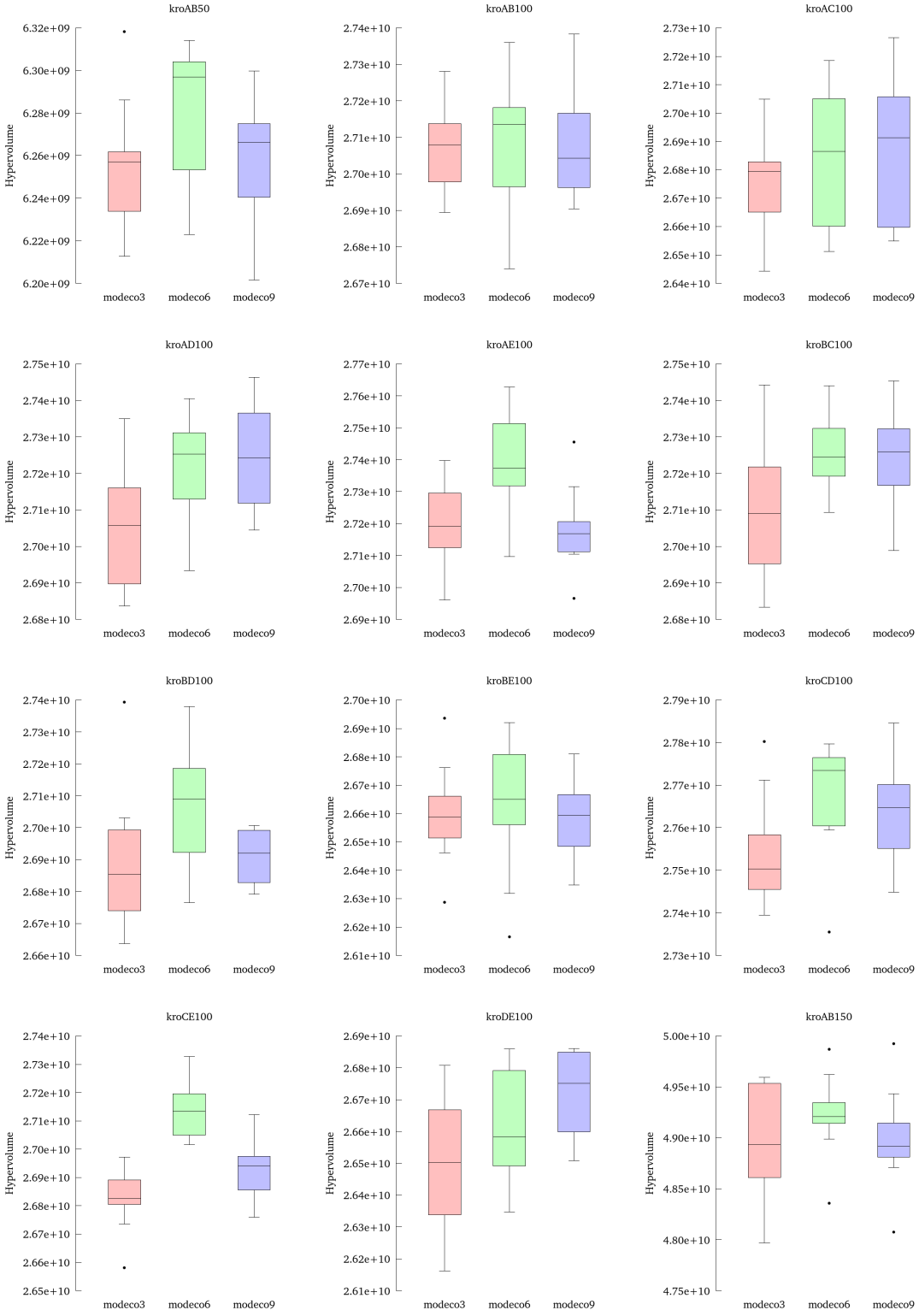


Figure 4.29: Boxplots of the hypervolume for 10 executions of different instances of multi-objective traveling salesman problems, using a maximum of 500 generations, for three different version of MODECO.

maker. Observing those figures, we can see that in the kroBE100 instance, figure 4.37, the modeco3 algorithm appears to give the best trade-off solutions, even though is has the worse hypervolume metric, albeit very close to the modeco9 version. In the kroAE100 (figure 4.34), kroBD100 (figure 4.36), kroCE100 (figure 4.39) and kroAB150 (figure 4.41) instances, the better solutions are reached using the modeco6 algorithm, and not only in the trade-off area, but across the entire Pareto front. In all other instances, the best trade-off solutions are reached using the modeco9 algorithm, even if it is not the algorithm that gives the best value for the hypervolume.

This apparently conflicting conclusions are natural when analyzing and comparing multi-objective algorithms, as different quality indicators measure different aspects, and although one algorithm may give better results when compared using a certain indicator, using another can produce a contrary conclusion, and that is the reason that when saying that one algorithm is better than another, is necessary to say which quality indicator has been used to reach that conclusion.

When weighing all indicators across the different instances, we maintain the previous decision that the best overall algorithm is the modeco9 variation, particularly because the modeco6 variation, although reaching a better hypervolume, it usually does so by obtaining better solutions in the extremes, and those solutions are probably not that interesting for the decision maker. On the contrary, the modeco9 variation, although not reaching always the best hypervolume, it does so some times, and even when it doesn't, the hypervolume results are due to good trade-off solutions.

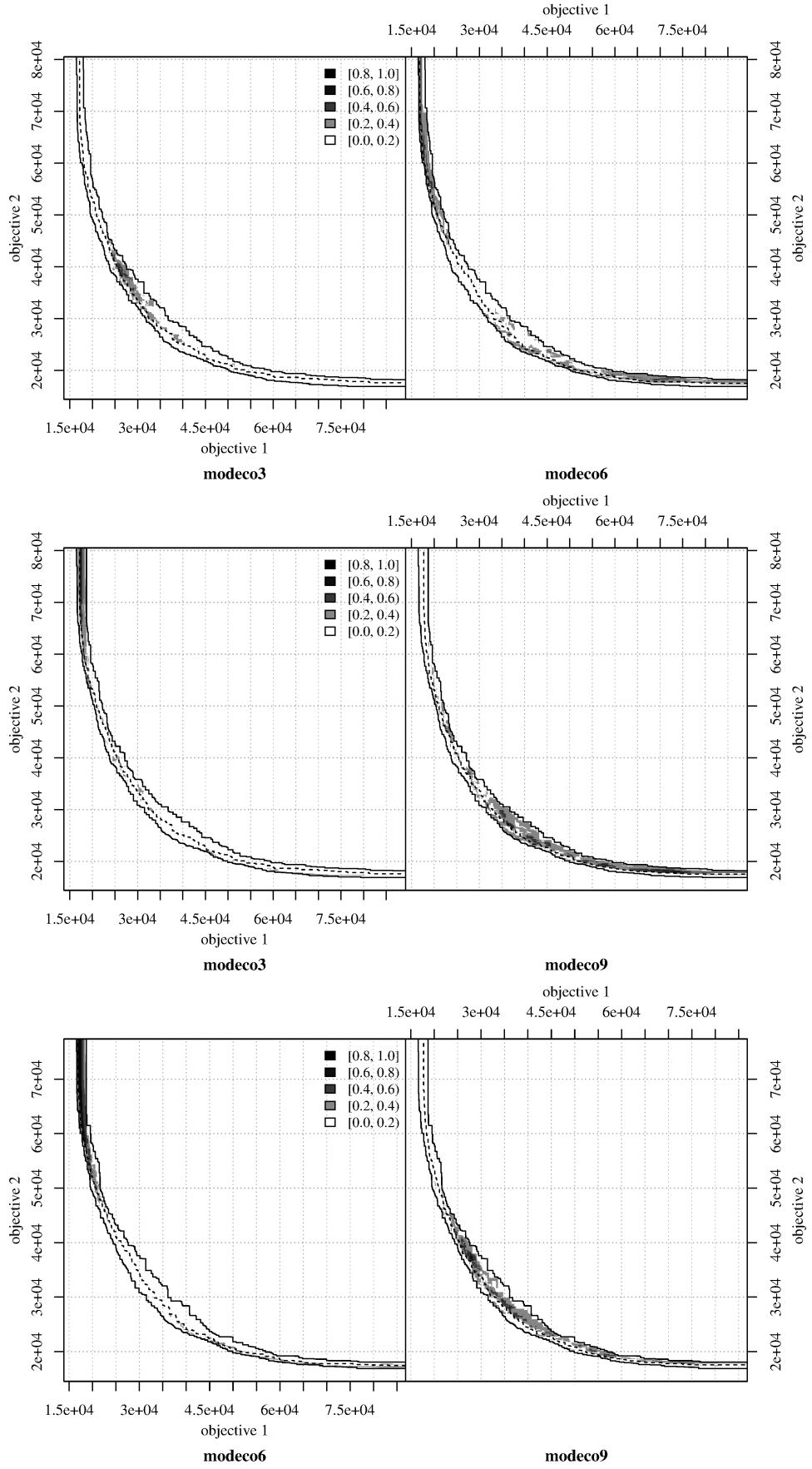


Figure 4.30: Differences in the estimated attainment function between the best three algorithms for the kroAB50 instance.

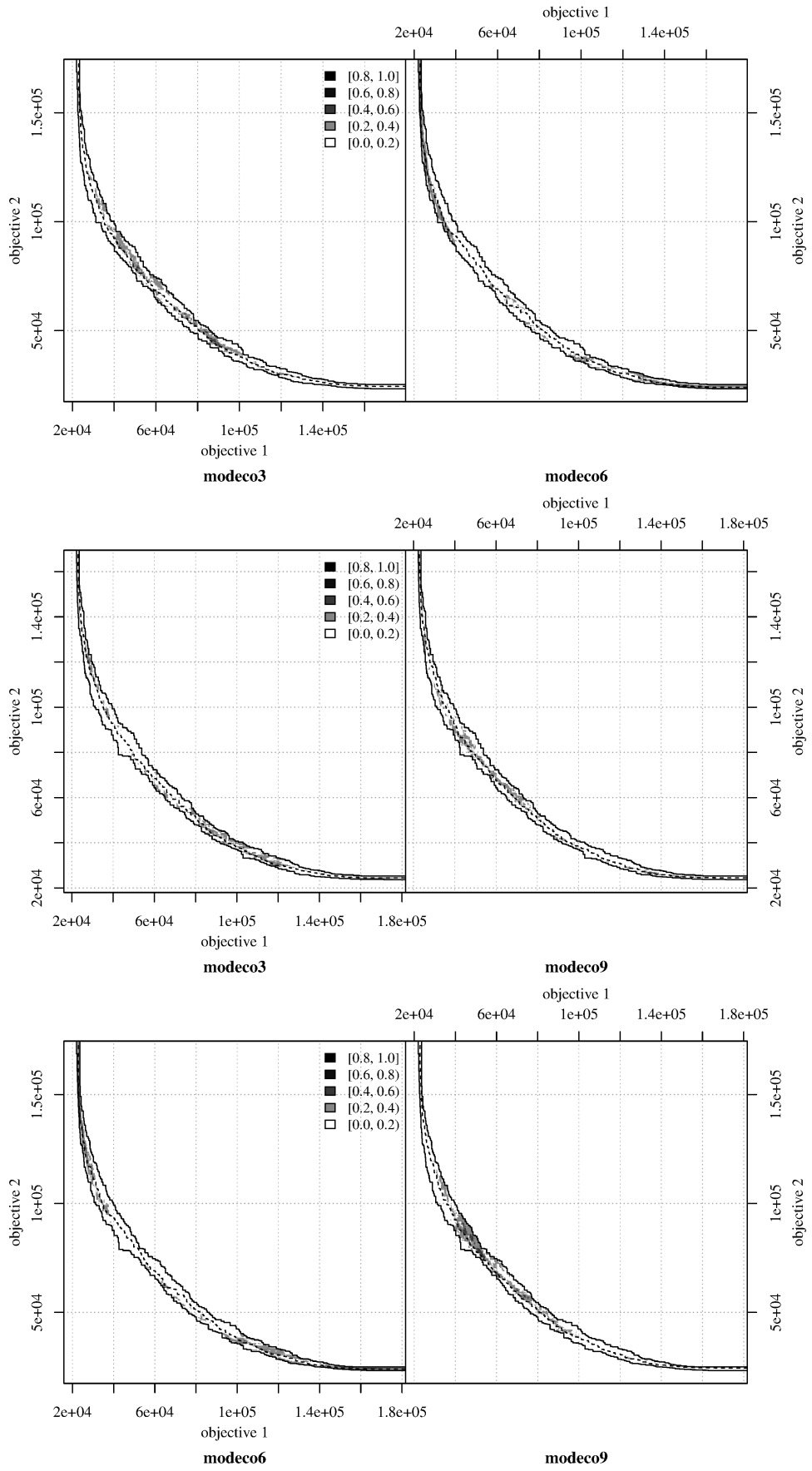


Figure 4.31: Differences in the estimated attainment function between the best three algorithms for the kroAB100 instance.

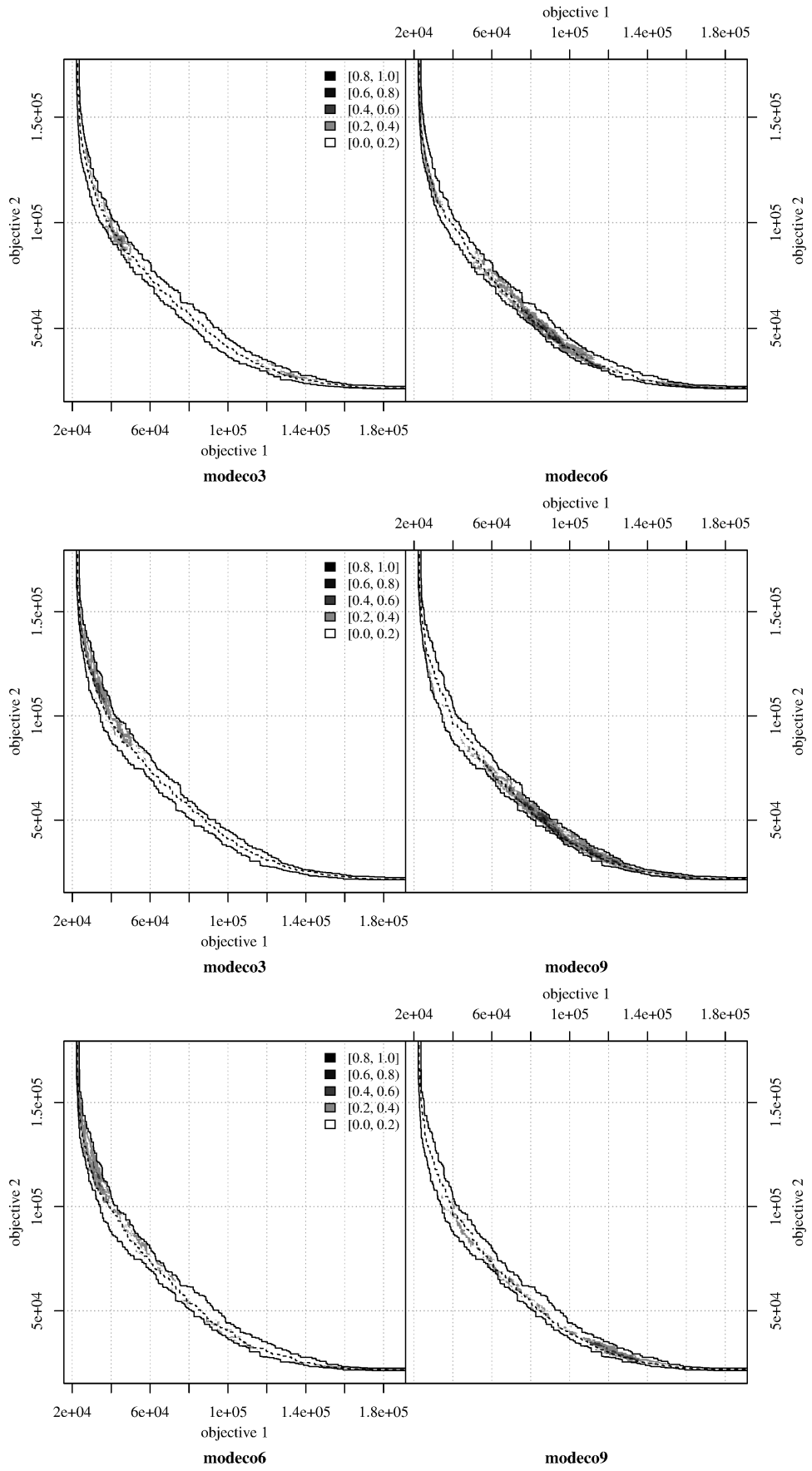


Figure 4.32: Differences in the estimated attainment function between the best three algorithms for the kroAC100 instance.

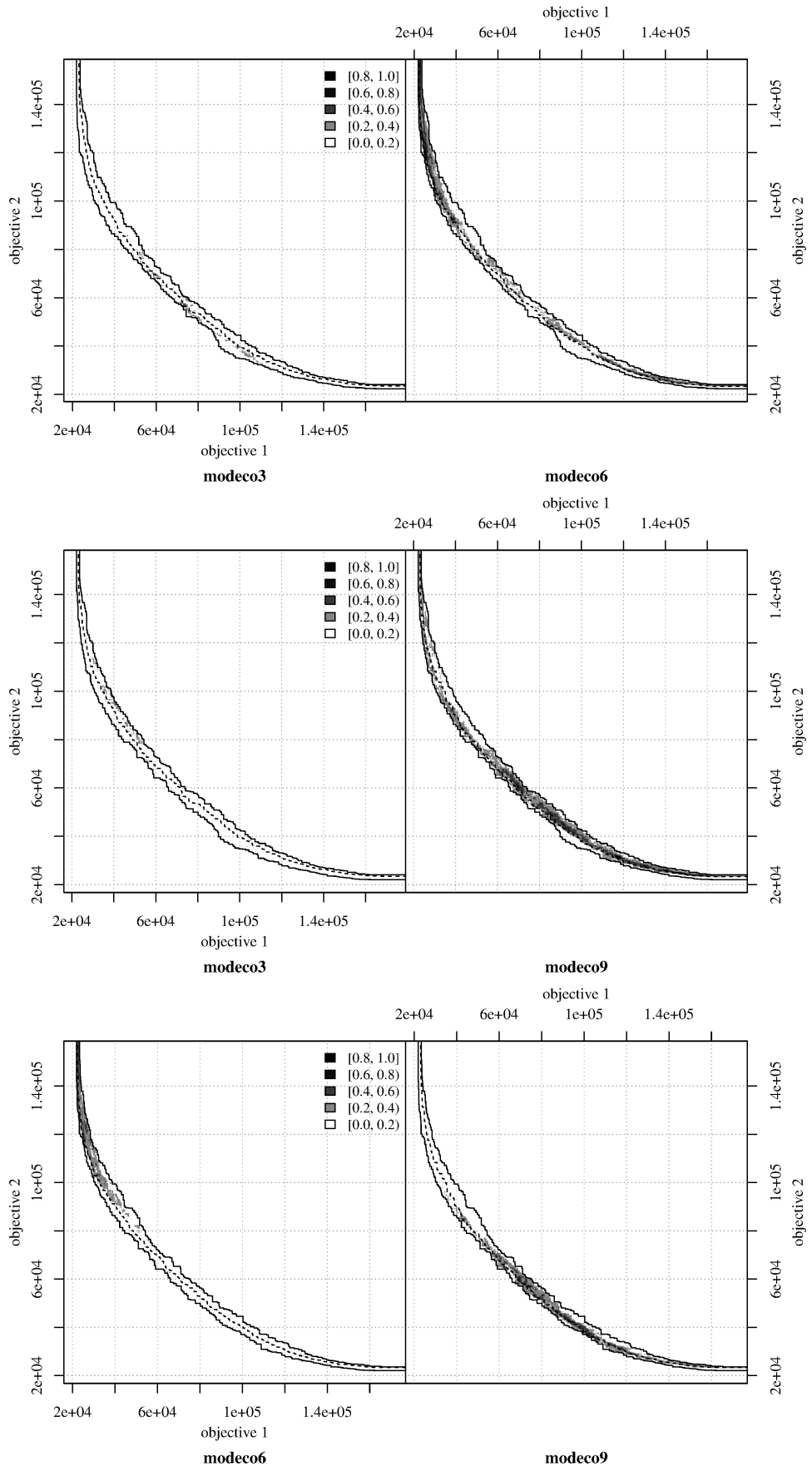


Figure 4.33: Differences in the estimated attainment function between the best three algorithms for the kroAD100 instance.

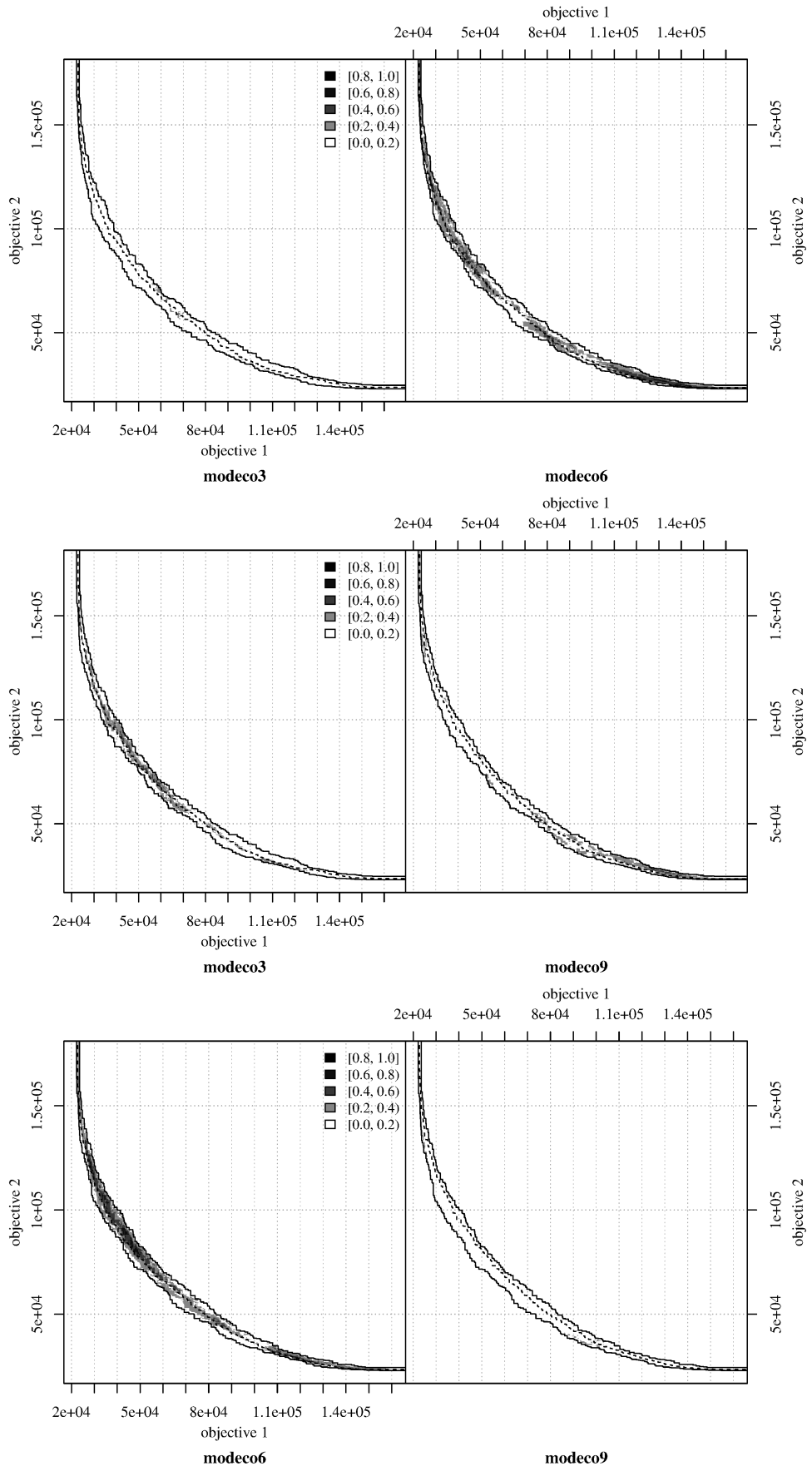


Figure 4.34: Differences in the estimated attainment function between the best three algorithms for the kroAE100 instance.

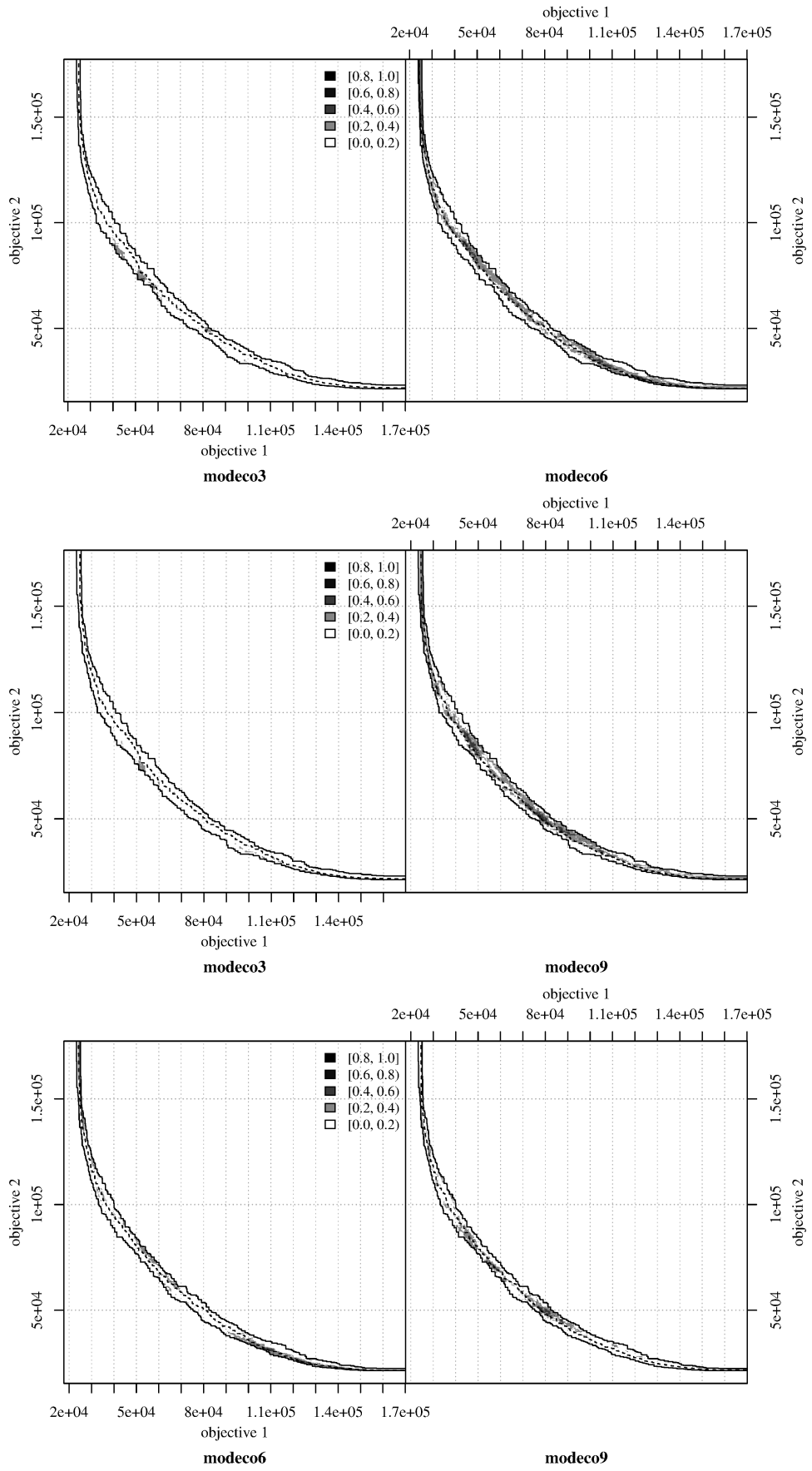


Figure 4.35: Differences in the estimated attainment function between the best three algorithms for the kroBC100 instance.

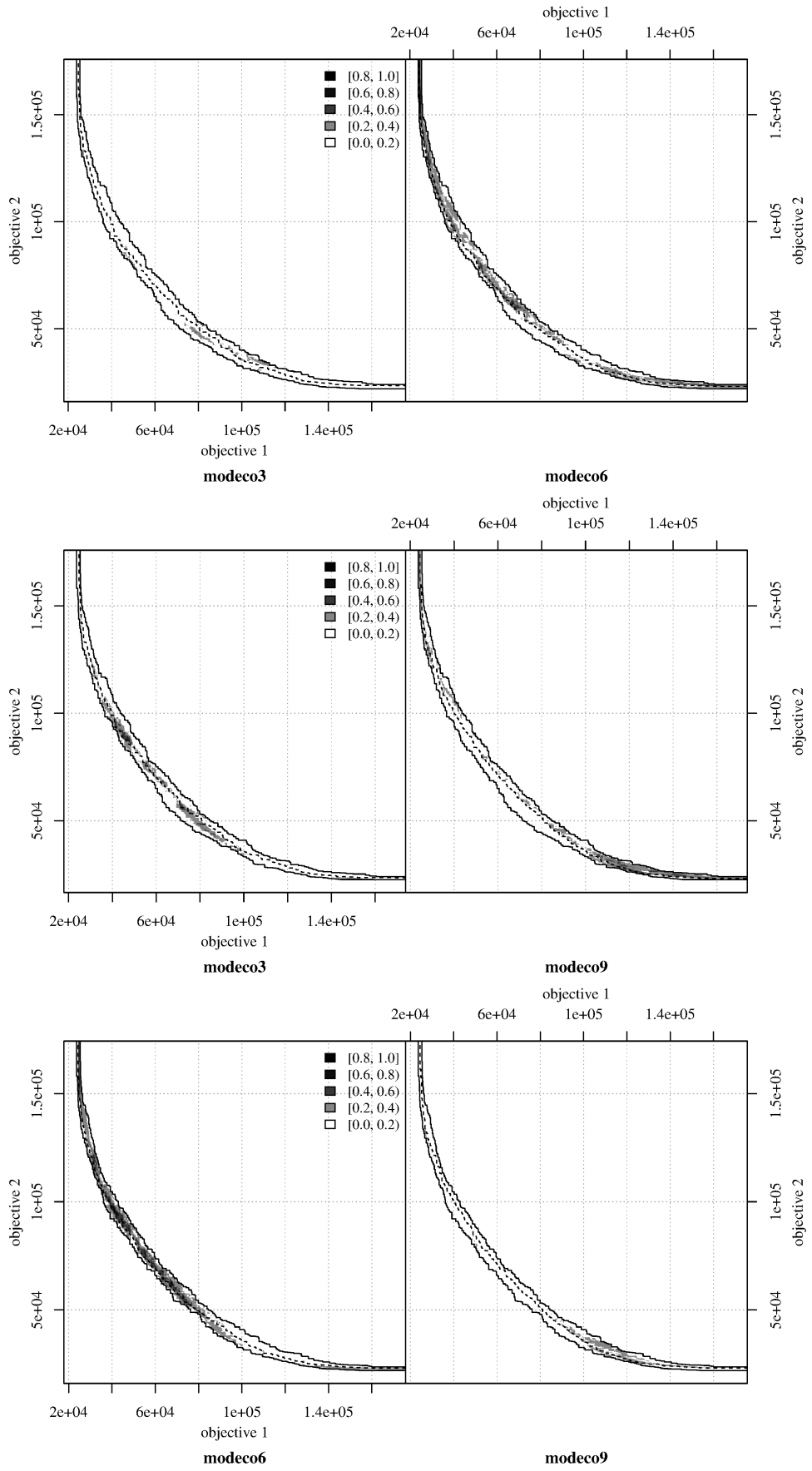


Figure 4.36: Differences in the estimated attainment function between the best three algorithms for the kroBD100 instance.

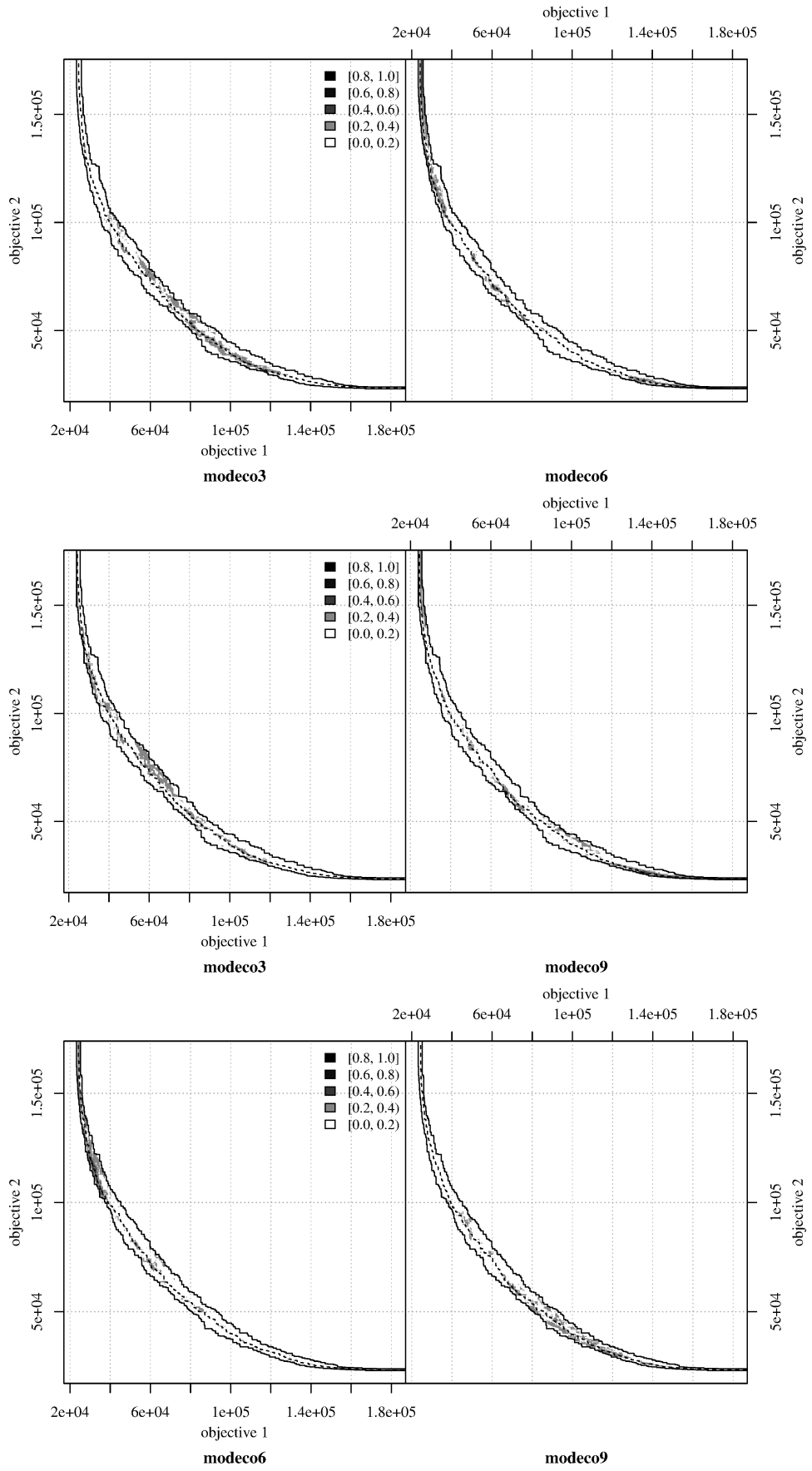


Figure 4.37: Differences in the estimated attainment function between the best three algorithms for the kroBE100 instance.

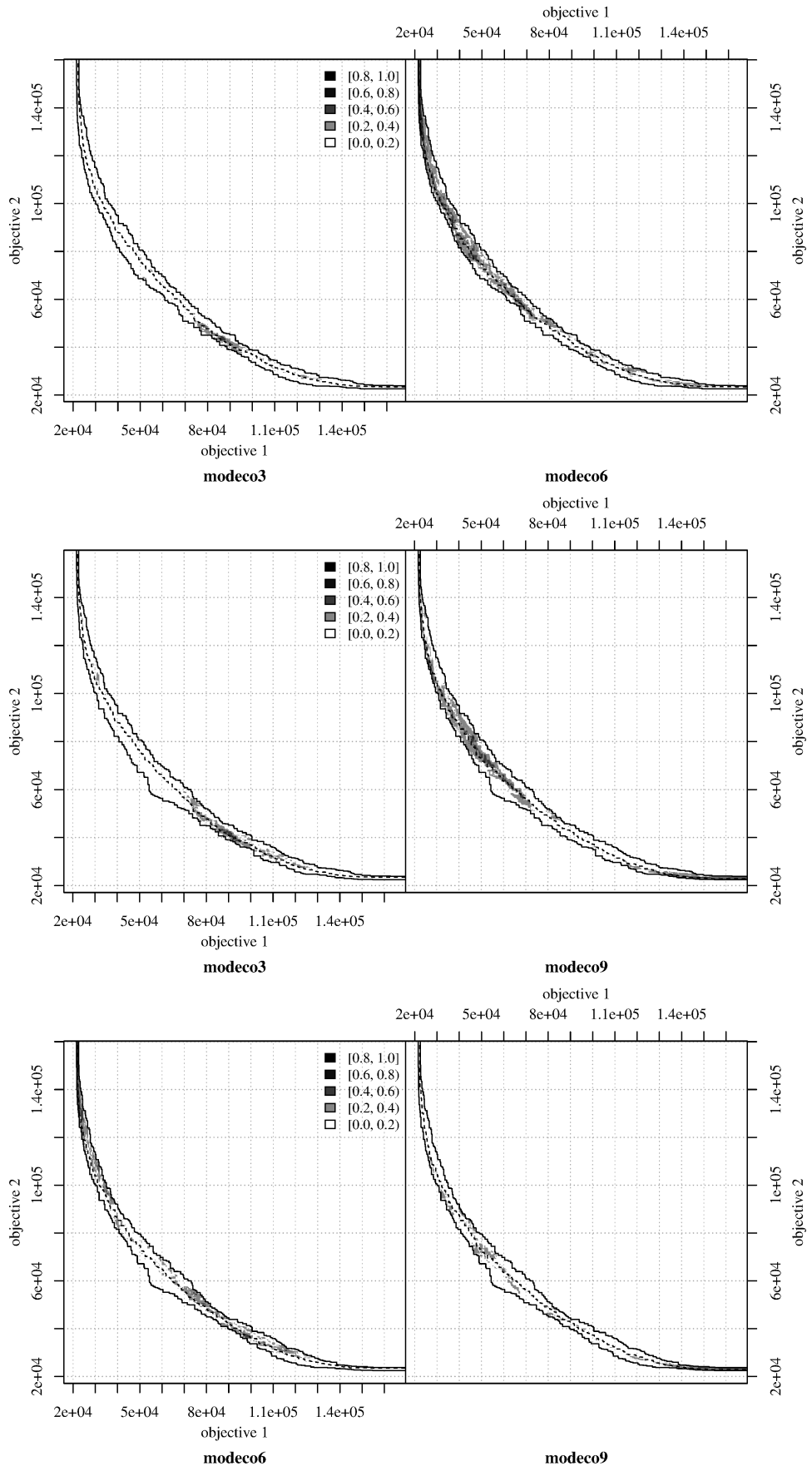


Figure 4.38: Differences in the estimated attainment function between the best three algorithms for the kroCD100 instance.

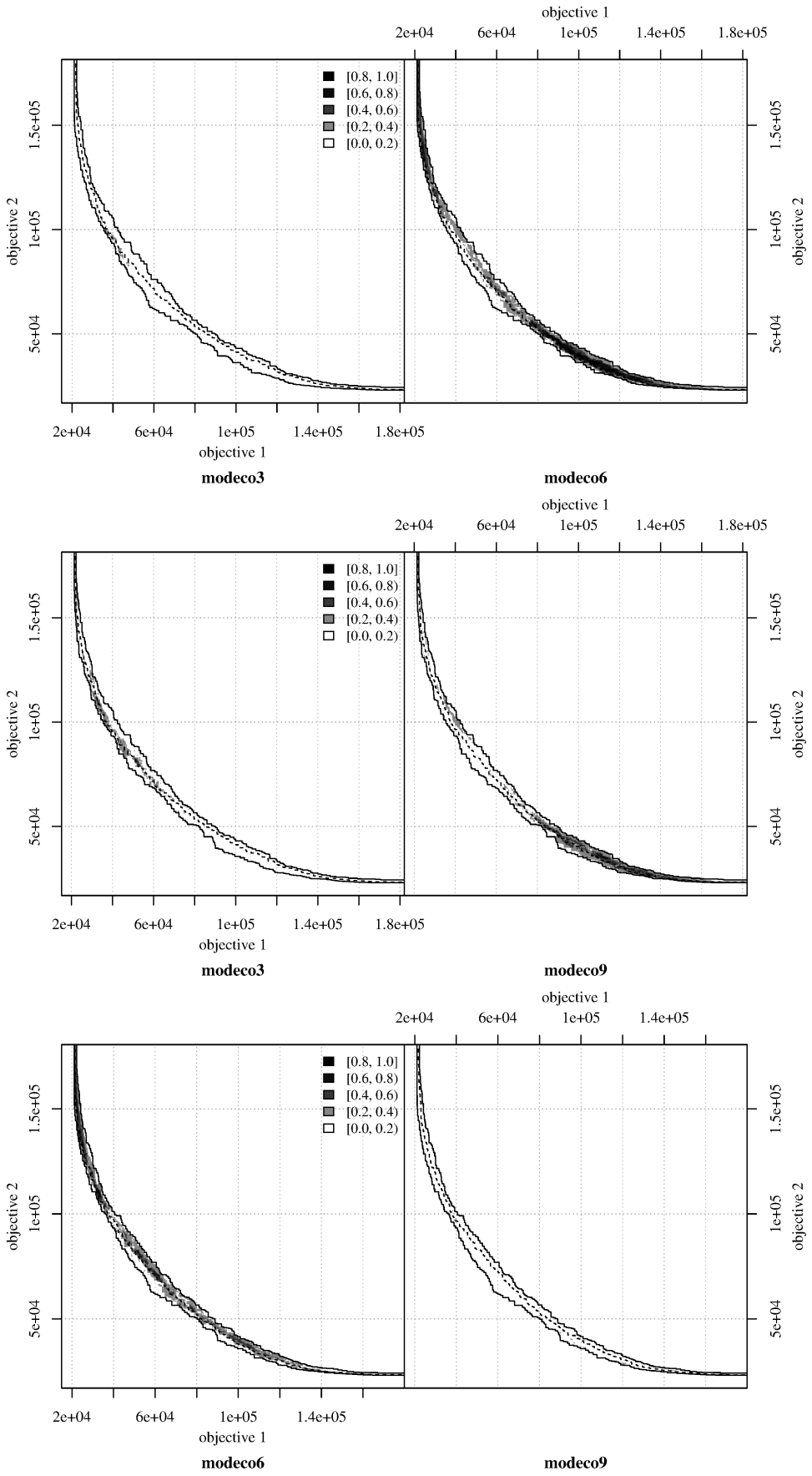


Figure 4.39: Differences in the estimated attainment function between the best three algorithms for the kroCE100 instance.

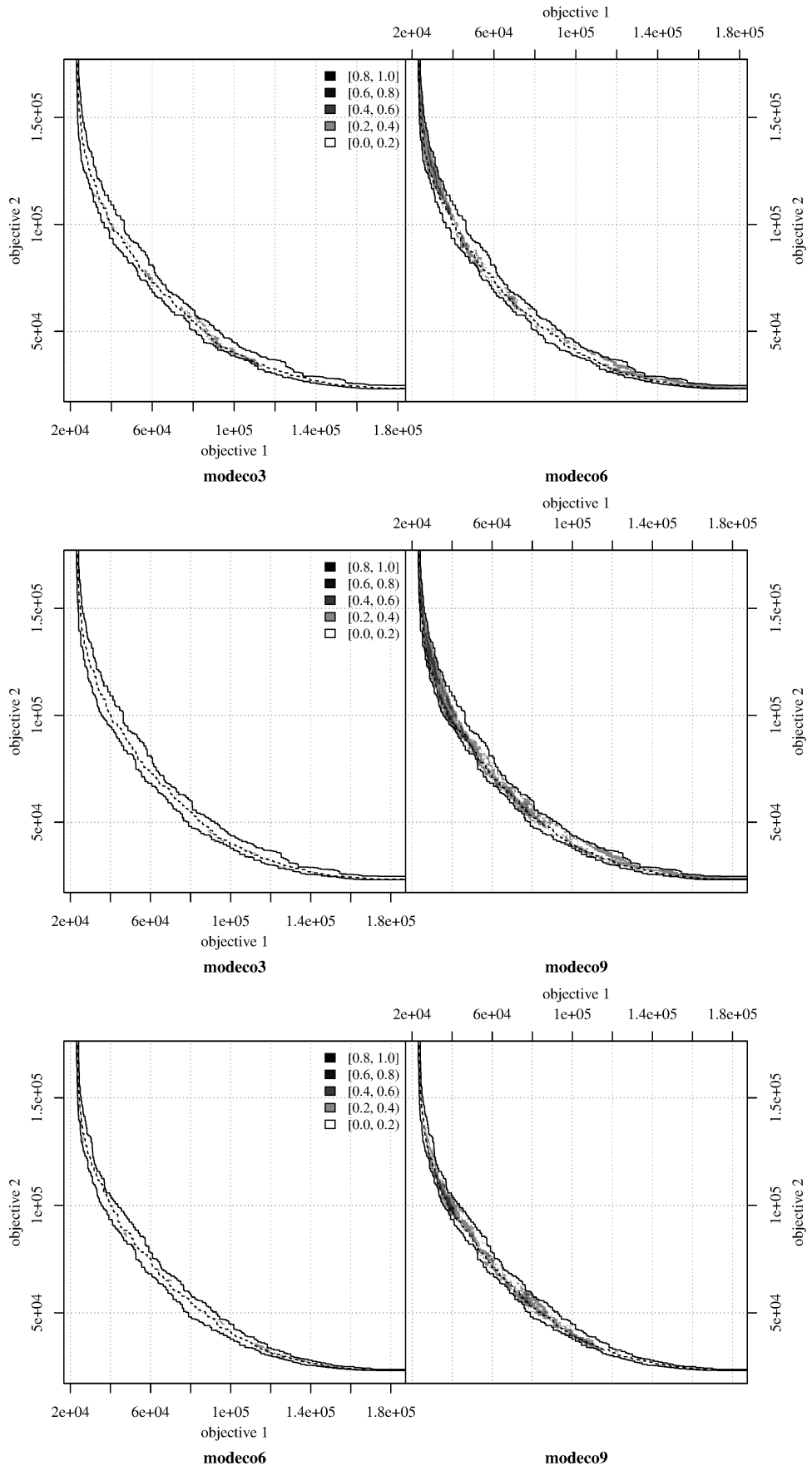


Figure 4.40: Differences in the estimated attainment function between the best three algorithms for the kroDE100 instance.

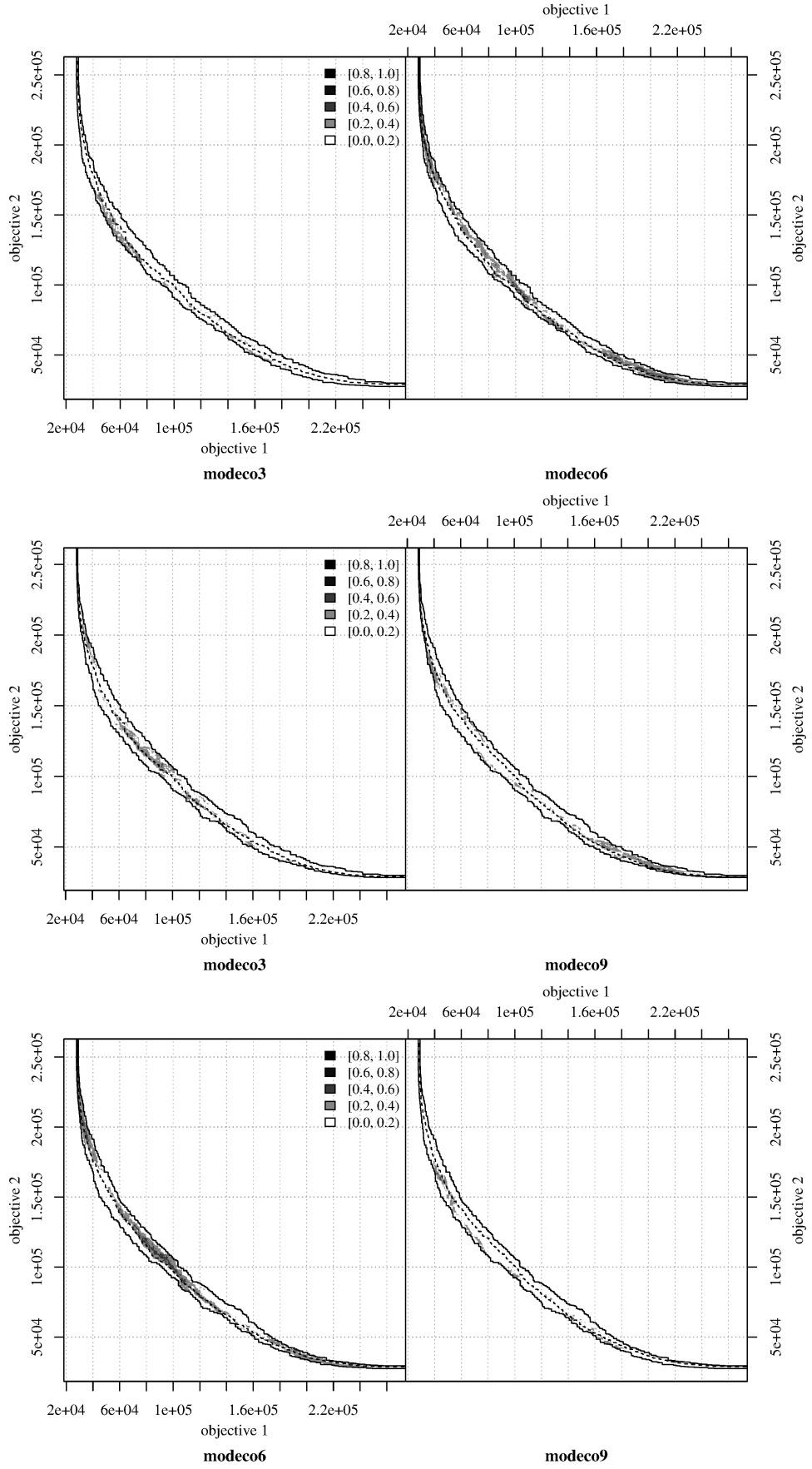


Figure 4.41: Differences in the estimated attainment function between the best three algorithms for the kroAB150 instance.

Chapter 5

Conclusions and future work

The main focus of this thesis is the introduction of a new approach to be used in the differential evolution algorithm when applying it to combinatorial optimization problems, as this evolutionary algorithm were not developed for this type of problems.

The previous approaches to circumvent the problem of using the differential evolution algorithm in combinatorial optimization problems were discussed, and was shown that most of them have the problem of not using the differential evolution in the combinatorial process, but rather translate the problem to the differential evolution domain and only then solve the continuous problem. As this as the disadvantage of not letting the algorithm take advantage of the intricacies of combinatorial oprimization problems, our idea was to change the differential evolution algorithm, or rather, its operators, to function in the discrete problem. We introduced the idea of using the concept of sets as the representation for the solutions, allowing this way the usage of set-base operations instead of arithmetic ones in the genetic operators of the algorithm.

We discussed different formulations for the mutation operator, the crucial operator in the differential evolution algorithm, explaining the pros and cons of each of them, before setting with the one that, in our opinion, presents the best characteristics to be used in this type of problems. As the set operations, basically, insert

and remove elements from a set, this means that using these operations would change the cardinality of the solutions, during the execution of the algorithm. As this, usually, is not the way for a solution to behave in an evolutionary algorithm, we had to introduce a repair mechanism in the process, to keep the cardinality of the solutions from generation to generation. However, one main consideration in the decision for the best formulation for the mutation operator, was to keep this version the "truest" to the original as possible, meaning that the repair would be used as little as possible, and our final formulation has this characteristic, of, although needing to repair the solutions, its interference in the process is minimum. We also conclude about the best values for the differential evolution parameters to be used for this type of problems, as these values are problem dependent, they are clearly depended of the *type* of problem, i.e., if different real-domain problems can behave differently using different parameters, clearly, problems from a different domain cannot be expected to simply follow the parameters guidelines given for the real-domain problems, and expect to obtain the best possible results.

The algorithm was used for different instances of the most know and studied single-objective combinatorial optimization problem: the traveling salesman problem. The results show that when compared with some of the previous approaches, our set-based mutation shows quite good results, using a fraction of the number of generations used by the other approaches.

After proving the methodology for the single-objective problem, we started focusing on adapting the algorithm to multi-objective problems, as this type of problems constitute the all day, every day type of problems we encounter in real live, from buying groceries to choosing an holiday destination, almost every (if not all) decision in life is made using some multi-criteria decision, so is more than natural that multi-objective algorithms are developed to help our decisions.

Unlike the single-objective optimization problems, the result of the multi-objective ones are not one best solution, but the Pareto set of solutions, with the best trade-off solutions, considering all the objectives in the problem. As such, the

algorithm needs to find this solutions, save them, and in each generation decide which of those solutions are to maintain because they are still good solutions, and which are to discard, as better ones supersede them. We discuss the adaptation needed to be made to the algorithm for multi-objective optimization problems, as adding another population to save the Pareto set of solutions, adapting the repair mechanism and replacing the population from one generation to the next.

Adapting the repair mechanism has resulted in a inherent and underlying structure of "single-objective-minded" solutions, with a dynamically changing number of solutions for each objective, that when mixed together in the genetic operators tend to produce good multi-objective solutions.

Other ideas were added to the algorithm, as using more than one population in the evolutionary process and self-adapting parameters, and these ideas proved their value, as they improve the Pareto set of solutions obtained. A final multi-objective differential evolution for combinatorial optimization algorithm was suggested, and its results compared against other variations of the algorithm, using different quality indicators for evaluating different characteristics of the results. The results shown that our chosen algorithm, although not giving the best overall results, give the best results where it matter the most, i.e., in the trade-off area of the Pareto front, where, expectantly, are the solution the decision maker will have a closer look to make its final decision.

Although the obtained results have been good, further research is needed, particularly a deeper analysis of our self-adaptive algorithm regarding the different results obtained using the two replacement operators tested, the Pareto dominance and the concave hull. The discrepancy of the results were not expected, and an analysis of the underlying reasons for it is necessary.

Another aspect is expanding the usage of our algorithm, both the single and the multi-objective one, for other types of practical combinatorial optimization problems. We proved the validity of the algorithm for this type of problems, but applying it to other problems is necessary, for instance, multi-modal shortest path

problems, which is a practical problem in modern day society.

A third aspect is using our self-adaptive algorithm in single-objective optimization problems, and comparing it against other established self-adaptive algorithms. The results in the multi-objective case were good, but using problems where the actual optimal solution is known should prove invaluable to compare and benchmark our self-adaptive mechanism to the mainstream ones.

Bibliography

[1] H.A. Abbass, R. Sarker, and C. Newton. PDE: a Pareto-frontier differential evolution approach for multi-objective optimization problems. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, volume 2, pages 971–978. IEEE, 2001.

[2] Mostafa Z. Ali, Noor H. Awad, and Ponnuthurai N. Suganthan. Multi-population differential evolution with balanced ensemble of mutation strategies for large-scale global optimization. *Applied Soft Computing*, 33:304–327, aug 2015.

[3] Javier Apolloni, Guillermo Leguizamón, José García-Nieto, and Enrique Alba. Island Based Distributed Differential Evolution: An Experimental Study on Hybrid Testbeds. In *Hybrid Intelligent Systems, 2008. HIS '08. Eighth International Conference on*, pages 696–701. IEEE, 2008.

[4] Masoud Asadzadeh, Bryan A. Tolson, and Donald H. Burn. A new selection metric for multiobjective hydrologic model calibration. *Water Resources Research*, 50(9):7082–7099, sep 2014.

[5] Norbert Ascheuer, Martin Grötschel, and Atef Abdel-Aziz Abdel-Hamid. Order picking in an automatic warehouse: Solving online asymmetric TSPs. *Mathematical Methods of Operations Research (ZOR)*, 49(3):501–515, jul 1999.

[6] Giorgio Ausiello, Alberto Marchetti-Spaccamela, Pierluigi Crescenzi, Giorgio Gambosi, Marco Protasi, and Viggo Kann. *Complexity and approximation: combinatorial optimization problems and their approximability properties*. Springer Berlin Heidelberg, 1999.

[7] Christopher Bailey, Timothy McLain, and Randal Beard. Fuel saving strategies for separated spacecraft interferometry. In *Journal of the Astronautical Sciences*, volume 49, pages 469–488, Reston, Virigina, aug 2000. American Institute of Aeronautics and Astronautics.

[8] James C. Bean. Genetic Algorithms and Random Keys for Sequencing and Optimization. *ORSA Journal on Computing*, 6(2):154–160, 1994.

[9] Robert G Bland and David F Shallcross. Large travelling salesman problems arising from experiments in X-ray crystallography: A preliminary report on computation. *Operations Research Letters*, 8(3):125–128, jun 1989.

[10] Janez Brest, Sao Greiner, Borko Boskovic, Marjan Mernik, and Viljem Zumer. Self-Adapting Control Parameters in Differential Evolution: A Comparative Study on Numerical Benchmark Problems. *IEEE Transactions on Evolutionary Computation*, 10(6):646–657, dec 2006.

[11] Shelvin Chand and Markus Wagner. Evolutionary many-objective optimization: A quick-start guide. *Surveys in Operations Research and Management Science*, 20(2):35–42, 2015.

[12] Nicos Christofides. Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem. Technical Report 388, Graduate School of Industrial Administration, CMU, 1976.

[13] Vašek Chvátal, William Cook, George B. Dantzig, Delbert R. Fulkerson, and Selmer M. Johnson. Solution of a Large-Scale Traveling-Salesman Problem. In *50 Years of Integer Programming 1958-2008*, pages 7–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] Carlos A. Coello Coello and Margarita Reyes Sierra. A Study of the Parallelization of a Coevolutionary Multi-objective Evolutionary Algorithm. In Raúl Monroy, Gustavo Arroyo-Figueroa, Luis Enrique Sucar, and Humberto Sossa, editors, *MICAI 2004: Advances in Artificial Intelligence*, volume 2972, pages 688–697. Springer, Berlin, Heidelberg, 2004.

[15] G. A. Croes. A Method for Solving Traveling-Salesman Problems. *Operations Research*, 6(6):791–812, dec 1958.

[16] Laizhong Cui, Genghui Li, Qiuzhen Lin, Jianyong Chen, and Nan Lu. Adaptive differential evolution algorithm with novel mutation strategies in multiple sub-populations. *Computers & Operations Research*, 67:155–173, mar 2016.

[17] Swagatam Das, Sankha Subhra Mullick, and P. N. Suganthan. Recent advances in differential evolution - An updated survey. *Swarm and Evolutionary Computation*, 27:1–30, 2016.

[18] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: A survey of the state-of-the-art. *IEEE Transactions on Evolutionary Computation*, 15(1):4–31, 2011.

[19] Lawrence Davis. Applying Adaptive Algorithms to Epistatic Domains. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, volume 1, pages 162–164. Morgan Kaufmann Publishers Inc., 1985.

[20] Ivanoe De Falco, Antonio Della Cioppa, Domenico Maisto, Umberto Scafuri, and Ernesto Tarantino. Satellite image registration by distributed differential evolution. In Mario Giacobini, editor, *Applications of Evolutionary Computing*, volume 4448 of *Lecture Notes in Computer Science*, pages 251–260. Springer Berlin / Heidelberg, 2007.

[21] Ivanoe De Falco, Domenico Maisto, Umberto Scafuri, Ernesto Tarantino, and Antonio Della Cioppa. Distributed differential evolution for the registration of remotely sensed images. In *Proceedings - 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing, 1PDP 2007*, pages 358–362, 2007.

[22] K Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, 2001.

[23] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, apr 2002.

[24] Feng Xue, A.C. Sanderson, and R.J. Graves. Pareto-based multi-objective differential evolution. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 2, pages 862–869. IEEE, 2003.

[25] Carlos M. Fonseca, Viviane G. da Fonseca, and Luís Paquete. Exploring the Performance of Stochastic Multiobjective Optimisers with the Second-Order Attainment Function. In Carlos A. Coello Coello, Arturo Hernández Aguirre, and Eckart Zitzler, editors, *Evolutionary Multi-Criterion Optimization. EMO 2005*, volume 3410, LNCS, pages 250–264. Springer, Berlin, Heidelberg, 2005.

[26] Carlos M. Fonseca and Peter J Fleming. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *Proceedings of the Fifth Inter2national Conference on Genetic Algorithms*, pages 416–423. Morgan Kaufmann, 1993.

[27] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. {DEAP}: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[28] Michael R Garey and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W.H. Freeman, 1979.

[29] Ioannis Giagkiozis, Robin C. Purshouse, and Peter J. Fleming. An overview of population-based algorithms for multi-objective optimisation. *International Journal of Systems Science*, 46(9):1572–1599, 2015.

[30] Martin Grötschel, M. Jünger, and Gerhard Reinelt. Optimal control of plotting and drilling machines: a case study. *Mathematical Methods of Operations Research (ZOR)*, 35(1):61–84, 1991.

[31] Viviane Grunert da Fonseca, Carlos M. Fonseca, and Andreia O. Hall. Inferential Performance Assessment of Stochastic Optimisers and the Attainment Function. In Eckart Zitzler, Lothar Thiele, Kalyanmoy Deb, Carlos Artemio Coello Coello, and David Corne, editors, *Evolutionary Multi-Criterion Optimization. EMO 2001*, volume 1993, LNCS, pages 213–225. Springer Berlin Heidelberg, 2001.

[32] Pedro Guerreiro, Mário Jesus, and Alberto Márquez. A new set-based mutation operator for Differential Evolution. In Martín Cera Lopez, Pedro García Vázquez, Rocío Moreno Casablanca, and Juan Carlos Valenzuela Tripodoro, editors, *Avances en Matemática Discreta en Andalucía*, volume 3, pages 175–182, 2013.

[33] Pedro Guerreiro, Mário Jesus, and Alberto Márquez. A comparison of set-based mutation operators for Differential Evolution. In *Avances en Matemática Discreta en Andalucía*, volume 4, pages 141–148, 2015.

[34] Michael Pilegaard Hansen. Metaheuristics for Multiple Objective Combinatorial Optimization. 1998.

[35] Michael Pilegaard Hansen and Andrzej Jaszkiewicz. Evaluating the Quality of Approximations to the Non-Dominated Set. Technical Report IMM-REP-1998-7, Technical University of Denmark, 1998.

[36] Keld Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, oct 2000.

[37] Yannick Hold-Geoffroy, Olivier Gagnon, and Marc Parizeau. Once you SCOOP, no need to fork. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment - XSEDE '14*, pages 1–8, 2014.

[38] John Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, 1975.

[39] Hisao Ishibuchi, Hiroyuki Masuda, Yuki Tanigaki, and Yusuke Nojima. Modified Distance Calculation in Generational Distance and Inverted Generational Distance. In António Gaspar-Cunha, Carlos Henggeler Antunes, and Carlos A. Coello Coello, editors, *Evolutionary Multi-Criterion Optimization. EMO 2015*, volume 9019, LNCS, pages 110–125. Springer, Cham, 2015.

[40] Andrzej Jaszkiewicz. *Multiple Objective Metaheuristic Algorithms for Combinatorial Optimization*. Phd thesis, Poznan University of Technology, 2001.

[41] Kenneth V. Price. An introduction to differential evolution. In David Corne, Marco. Dorigo, Fred Glover, Dipankar Dasgupta, Pablo Moscato, Riccardo Poli, and Kenneth V. Price, editors, *New ideas in optimization*, pages 79–108. McGraw-Hill, 1999.

[42] Joshua Knowles, Lothar Thiele, and Eckart Zitzler. A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. Technical Report TIK-214, Computer Engineering and Networks Laboratory (TIK), 2006.

[43] S. Kukkonen and J. Lampinen. GDE3: The third Evolution Step of Generalized Differential Evolution. In *2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 443–450. IEEE, 2005.

[44] Saku Kukkonen and Jouni Lampinen. An Extension of Generalized Differential Evolution for Multi-objective Optimization with Constraints. In Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A. Bullinaria, Jonathan E. Rowe, Peter Tino, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII. PPSN 2004*, pages 752–761. Springer, Berlin, Heidelberg, 2004.

[45] Jouni Lampinen and Rainer Storn. Differential Evolution. In *New Optimization Techniques in Engineering*, pages 123–166. Springer Berlin Heidelberg, 2004.

[46] Jouni A Lampinen and Ivan Zelinka. Mixed Integer-Discrete-Continuous Optimization, by Differential Evolution, Part 1: the optimization method. In *Proceedings of MENDEL'99, 5th International Mendel Conference on Soft Computing*, pages 71–76, Brno, Czech Republic, 1999. University of Technology, Faculty of Mechanical Engineering, Institute of Automation and Computer Science.

[47] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, jun 1992.

[48] Daniel Lichtblau. Discrete Optimization using Mathematica. In *2002 World Multiconference on Systemics, Cybernetics, and Informatics*, pages 1–6, 2002.

[49] Daniel Lichtblau. Relative Position Indexing Approach. In Godfrey C. Onwubolu and Donald Davendra, editors, *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*, pages 81–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[50] S. Lin and B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*, 21(2):498–516, apr 1973.

[51] Tao Liu and Michiharu Maeda. Set-Based Differential Evolution for Traveling Salesman Problem. In *2013 6th International Conference on Intelligent Networks and Intelligent Systems*, pages 107–110, 2013.

[52] Tao Liu and Michiharu Maeda. An algorithm of set-based differential evolution for traveling salesman problem. In *2014 Joint 7th International Conference on Soft Computing and Intelligent Systems, SCIS 2014 and 15th International Symposium on Advanced Intelligent Systems, ISIS 2014*, pages 81–86. IEEE, 2014.

[53] Yu Liu, Wei Neng Chen, Zhi Hui Zhan, Ying Lin, Yue Jiao Gong, and Jun Zhang. A set-based discrete differential evolution algorithm. In *Proceedings - 2013 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2013*, pages 1347–1352. IEEE, 2013.

[54] Manuel López-Ibáñez, Luís Paquete, and Thomas Stützle. Exploratory Analysis of Stochastic Local Search Algorithms in Biobjective Optimization. In Thomas Bartz-Beielstein, Marco Chiarandini, Luís Paquete, and Mike Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 209–222. Springer Berlin Heidelberg, 2010.

[55] Andre L. Maravilha, Jaime A. Ramirez, and Felipe Campelo. A New Algorithm Based on Differential Evolution for Combinatorial Optimization. In *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*, pages 60–66. IEEE, sep 2013.

[56] André L. Maravilha, Jaime A. Ramírez, and Felipe Campelo. Combinatorial optimization with differential evolution. In *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion (GECCO '14)*, pages 69–70. ACM Press, 2014.

[57] Rajesh Matai, Surya Singh, and Murari Lal. Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. In Donald Davendra, editor, *Traveling Salesman Problem, Theory and Applications*, pages 1–24. InTech, nov 2010.

[58] Efrén Mezura-Montes, Margarita Reyes-Sierra, and Carlos A. Coello Coello. Multi-objective Optimization Using Differential Evolution: A Survey of the State-of-the-Art. In Uday K. Chakraborty, editor, *Advances in Differential Evolution*, pages 173–196. Springer Berlin / Heidelberg, 2008.

[59] Zbigniew Michalewicz and David B Fogel. *How to Solve It: Modern Heuristics*. Springer Berlin Heidelberg, Berlin, Heidelberg, second edi edition, 2004.

[60] M. Davoodi Monfared, A. Mohades, and J. Rezaei. Convex hull ranking algorithm for multi-objective evolutionary algorithms. *Scientia Iranica*, 18(6):1435–1442, 2011.

[61] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[62] Yuichi Nagata and Shigenobu Kobayashi. Edge Assembly Crossover: A high-power Genetic Algorithm for the Traveling Salesman Problem. In *Proceedings of the 7th International Conference in Genetic Algorithms*, pages 450–457, 1997.

[63] Andreas C. Nearchou. Meta-heuristics from nature for the loop layout design problem. *International Journal of Production Economics*, 101(2):312–328, 2006.

[64] Andreas C. Nearchou and Sotiris L. Omirou. Differential evolution for sequencing and scheduling optimization. *Journal of Heuristics*, 12(6):395–411, dec 2006.

[65] Temel Öncan, I. Kuban Altinel, and Gilbert Laporte. A comparative analysis of several asymmetric traveling salesman problem formulations. *Computers & Operations Research*, 36(3):637–654, 2009.

[66] Godfrey Onwubolu. Optimization using Differential Evolution Algorithm. Technical Report TR-2001-05, IAS, 2001.

[67] Godfrey Onwubolu and Donald Davendra. Scheduling flow shops using differential evolution algorithm. *European Journal of Operational Research*, 171(2):674–692, 2006.

[68] Godfrey C. Onwubolu and B. V. Babu. *New Optimization Techniques in Engineering*, volume 141 of *Studies in Fuzziness and Soft Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[69] Godfrey C Onwubolu and Donald Davendra, editors. *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*, volume 175 of *Studies in Computational Intelligence*. CRC Press, feb 2009.

[70] A.J. Orman and H.P. Williams. A Survey of Different Integer Programming Formulations of the Travelling Salesman Problem. In *Optimisation, Econometric and Financial Analysis*, pages 91–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[71] Ricardo S. Prado, Rodrigo C. P. Silva, Frederico G. Guimarães, and Oriane M. Neto. Using differential evolution for combinatorial optimization: A general approach. In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, pages 11–18. IEEE, 2010.

[72] Kenneth Price, Rainer Storn, and Jouni A Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Natural Computing Series. Springer, 2005.

[73] A. K. Qin, V. L. Huang, and P. N. Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, 13(2):398–417, 2009.

[74] A. K. Qin and P. N. Suganthan. Self-adaptive Differential Evolution Algorithm for Numerical Optimization. In *2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1785–1791. IEEE, 2005.

[75] Qingfu Zhang and Hui Li. MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, dec 2007.

[76] Gerhard Reinelt. TSPLIB - A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3(4):376–384, nov 1991.

[77] Nery Riquelme, Christian Von Lücken, and Benjamin Baran. Performance metrics in multi-objective optimization. In *2015 Latin American Computing Conference (CLEI)*, pages 1–11. IEEE, oct 2015.

[78] Tea Robič and Bogdan Filipič. DEMO: Differential Evolution for Multiobjective Optimization. In Carlos A. Coello Coello, Arturo Hernández Aguirre, and Eckart Zitzler, editors, *Evolutionary Multi-Criterion Optimization*, volume

3410 of *Lecture Notes in Computer Science*, pages 520–533. Springer Berlin Heidelberg, 2005.

[79] Ji Shan-Fan, Sheng Wu-Xiong, and Jing Zhuo-Wang. The Multi-objective Differential Evolution Algorithm Based on Quick Convex Hull Algorithms. In *2009 Fifth International Conference on Natural Computation*, pages 469–473. IEEE, 2009.

[80] W. Stadler. A survey of multicriteria optimization or the vector maximum problem, part I: 1776-1960. *Journal of Optimization Theory and Applications*, 29(1):1–52, sep 1979.

[81] Rainer Storn. On the usage of differential evolution for function optimization. In *Proceedings of North American Fuzzy Information Processing*, pages 519–523. IEEE, 1996.

[82] Rainer Storn and Kenneth Price. Differential evolution - A simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, International Computer Science Institute, 1995.

[83] 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.

[84] D.K. Tasoulis, N. G. Pavlidis, V.P. Plagianakos, and M.N. Vrahatis. Parallel Differential Evolution. In *Congress on Evolutionary Computation, 2004. CEC2004*, number 6, pages 2023–2029, 2004.

[85] David A. van Veldhuizen. *Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations*. Phd thesis, Graduate School of Engineering of the Air Force Institute of Technology, 1999.

[86] David A. van Veldhuizen and Gary B. Lamont. Multiobjective evolutionary algorithm test suites. In *Proceedings of the 1999 ACM symposium on Applied computing - SAC '99*, pages 351–357, New York, New York, USA, 1999. ACM Press.

[87] Christian von Lücken, Benjamín Barán, and Carlos Brizuela. A survey on multi-objective evolutionary algorithms for many-objective problems. *Computational Optimization and Applications*, 58(3):707–756, 2014.

[88] Matthieu Weber, Ferrante Neri, and Ville Tirronen. Distributed differential evolution with explorative-exploitative population families. *Genetic Programming and Evolvable Machines*, 10(4):343–371, 2009.

[89] Darrell Whitley, Doug Hains, and Adele Howe. A Hybrid Genetic Algorithm for the Traveling Salesman Problem Using Generalized Partition Crossover. In Robert Schaefer, Carlos Cotta, Joanna Koodziej, and Günter Rudolph, editors, *Parallel Problem Solving from Nature - PPSN XI*, pages 566–575. Springer Berlin Heidelberg, 2010.

[90] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, apr 1997.

[91] Saúl Zapotecas Martínez and Carlos A. Coello Coello. A novel diversification strategy for multi-objective evolutionary algorithms. In *Proceedings of the 12th annual conference comp on Genetic and evolutionary computation - GECCO '10*, pages 2031–2034, New York, New York, USA, 2010. ACM Press.

[92] Ivan Zelinka. Discrete Set Handling. In Godfrey C. Onwubolu and Donald Davendra, editors, *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*, chapter 7, pages 163–205. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[93] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. Multiobjective evolutionary algorithms: A survey of the state of the art. *Swarm and Evolutionary Computation*, 1(1):32–49, 2011.

[94] Eckart Zitzler, Joshua Knowles, and Lothar Thiele. Quality assessment of pareto set approximations. In Jürgen Branke, Kalyanmoy Deb, Kaisa Miettinen, and Roman Słowiński, editors, *Multiobjective Optimization*, volume 5252 LNCS, chapter 14, pages 373–404. Springer Berlin Heidelberg, 2008.

[95] Eckart Zitzler and Simon Künzli. Indicator-Based Selection in Multiobjective Search. In Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A. Bullinaria, Jonathan E. Rowe, Peter Tino, Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, pages 832–842. Springer Berlin Heidelberg, 2004.

[96] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, 2001.

[97] Eckart Zitzler and Lothar Thiele. Multiobjective optimization using evolutionary algorithms - A comparative case study. In Agoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN V*, pages 292–301. Springer Berlin Heidelberg, 1998.

[98] Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

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