Vrije Universiteit Brussel

Faculty of Science
Department of Computer Science
and Applied Computer Science

Algorithms for Multi-Objective χ-Armed Bandits

Dissertation for the degree of Computer Science

Robrecht Conjaerts

Promotor: Prof. Ann Nowé
Advisors: Madalina M. Drugan

June 15, 2015

Vrije Universiteit Brussel

Faculteit Wetenschappen
Departement Informatica
en Toegepaste Informatica

Algoritmes voor Multi-Objectieve χ-Armige Bandieten

Dissertation for the degree of Computer Science

Robrecht Conjaerts

Promotor:	Prof. Ann Nowé
Begeleiders:	Madalina M. Drugan

15 juni, 2015

Abstract

The multi-objective χ-armed bandit problem is a relatively new area of interest in the field of Artificial Intelligence. The field searches for algorithms that find solutions to multi-objective optimization problems that are susceptible to noise. The solution space to search in is a topological space, with an infinite set of arms to play, and due to the problems being multi-objective there is not one single optimal solution, but a set of optimal solutions. These problems occur a lot in our everyday lives, with a high interest for solutions in the fields of engineering, and finance.

This master's thesis proposes 3 algorithms to solve the χ-armed bandit problem. The first one uses a tree data structure to represent the solution space in which the algorithm searches for solutions. To rank solutions, and distinguish between good, and bad solutions the Pareto dominance relation was used.

The thesis introduced a naive version of a new algorithm. It decomposes the problem into smaller sub-problems and optimizes them simultaneously. Each sub-problem has a weight vector, and one solution associated to it. Using the weight vector, and scalarization techniques the algorithm is able to give a fitness value to the encountered solutions, and use this to find better solutions.

The third algorithm improved the previous one by combining the tree data structure with the decomposition strategy. Each sub-problem helps building a global tree where each node represents a solution. Creating children for a node depends on the fitness value of the solution, which is calculated using the weight vector associated to the sub-problem, and a scalarization technique.

Numerous experiments have been performed, and the results have shown the strengths and weaknesses of each algorithm. All algorithms have been compared with each other, with the non-dominated sorting genetic algorithm II (NSGA-II), and with the multi-objective evolutionary algorithm based on decomposition (MOEA/D). The algorithm using Pareto dominance, and the improved algorithm both seem to be able to handle a lot of different kind of problems, and return a large set of solutions, close to the optimal set. They outperform the NSGA-II, and the MOEA/D algorithm on problems with 2, and 3 objectives to optimize.

Abstract

Het multi-objectieve χ-armige bandieten probleem is een relatief nieuw onderzoeksgebied binnen het domein kunstmatige intelligentie. Er wordt gezocht naar algoritmes die oplossingen kunnen vinden voor multi-objectieve problemen die vatbaar zijn voor ruis. De verzameling van alle mogelijke oplossingen is topologisch, met een oneindig aantal armen die gespeeld kunnen worden. Doordat de problemen meerdere objectieven hebben is er niet 1 optimale oplossing, maar een set van optimale oplossingen. Dit soort problemen komen vaak voor in ons alledaagse leven. Er is veel interesse uit de financile, en technische wereld om dit soort problemen effectief op te lossen.

Deze masterproef stelt 3 algoritmes voor om the χ-armige bandieten probleem op te lossen. Het eerste algoritme maakt gebruik van een boomstructuur als representatie van de verzameling oplossingen waarin we zoeken. Om oplossingen te rangschikken gebruiken gebruiken we de Pareto efficintie regel, die ons de mogelijkheid geeft om een onderscheid te maken tussen slechte, en goede oplossingen.

De thesis stelt een naïve, en een verbeterde versie voor van een nieuw algoritme. The naïve versie splits het probleem op in sub-problemen die simultaan geoptimaliseerd worden. Elk sub-probleem heeft een vector van gewichten, en een oplossing verbonden aan zich. Gebruik makend van gewichten wordt er een scalarizatie techniek toegepast die het mogelijk maakt om de oplossing een score te geven. Die score kunnen we gebruiken om betere oplossingen te vinden, en ze te rangschikken.

Het derde, verbeterde, algoritme combineert het opsplitsen van het probleem, en de boomstructuur. Elk sub-probleem helpt bij het uitbreiden van de boom, waarin elke knoop een representatie is van een oplossing. Het aanmaken van kinderen voor knopen hangt af van de score van de knoop, die berekend wordt door gebruik te maken van de gewichten vector en scalarizatie techniek die geassocieerd is met elk sub-probleem.

In de masterproef zijn meerdere experimenten uitgevoerd, en de resultaten geanalyseerd die de sterktes en zwaktes aantoonden van elke algoritme. Alle voorgestelde algoritmes zijn vergeleken met elkaar en met het *non-dominated sorting genetic algorithm II* (NSGA-II), en met het *multi-objective evolutionary algorithm based on decomposition* (MOEA/D) algoritme. Het algoritme dat gebruik maakte van de Pareto efficintie regel, en het verbeterde (3de) algoritme bleken beide goed te zijn op verschillende types van problemen. Ze gaven beide veel oplossingen terug die tevens ook dicht bij de optimale set van oplossingen lag. Hun performantie is beter dan die van het NSGA-II, en MOEA/D algoritme op problemen met 2, en 3 objectieve die geoptimaliseerd moeten worden.

Acknowledgements

I want to thank Professor Ann Now for all the knowledge she has imparted upon me during all of my academic years. She sparked an interest in Artificial Intelligence in me that was not there before, which paved the way of my master years.

To my advisor Madalina M. Drugan, who was there for me from the beginning of the thesis. She pushed me to tread upon these new grounds of Artificial Intelligence, and directed me towards the best areas where I was able to gather more information, which gave me ideas to create new algorithms.

To my family, and Marie for all the Friday nights that helped me shake off the stress I gathered throughout the week, and made sure I was not able to work on Saturday morning.

A special thanks to my dear friend Tom, my partner during all assignments, but more importantly my friend during life.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A lot of problems in life have multiple objectives that can be conflicting towards each other. Therefore it is not easy to find one perfect solution, because some solutions might be better for one objective, and others are better for different objectives. Such problems occur a lot in finance, i.e. a portfolio can be evaluated on 2 points: the expected value which we want to be as high as possible, and the amount of risk which we want to be as low as possible. These two are clearly related to each other, because the higher the expected return, the higher the risk will be since this is calculated using the standard deviation of the portfolio return.

This thesis focuses on solving such multi-objective problems, by searching for algorithms that return better solutions faster. It starts by explaining reinforcement learning, it is one of the basic approaches of machine learning, and serves as an introduction to the multi-armed bandit problem. It is from there that we mention multi-objective optimization problems, and how this all comes together in the multi-objective χ-armed bandit problem. In the following chapters will discuss algorithms that can solve this problem, and discuss their characteristics.

1.1 Reinforcement Learning

Learning by interacting with an environment, and aiming to maximize the rewards we are getting from these interactions, is what we call reinforcement learning [50]. A simplification of reinforcement learning, would be to imagine a child in an environment he has not encountered before. Without an adult nearby, there is no one to teach him how he should act upon these new surroundings. It looks around, tries to understand how it all works, but it eventually comes down to act and learn by trial and error. This happens mostly to children, because they are dropped into a new habitat on a daily basis, but the same goes for adults. Participating in a new sport, learning how your body responds to the movements you make is practically the same thing. This is where reinforcement learning comes from. Making machines learn when they encounter new situations by receiving rewards when they act upon this environment. Rewards are values given to the machine to distinguish between good and bad actions, and it is the goal of the machine to maximize the accumulation of these rewards.

Formalization

The reinforcement learning problem is mostly defined as a Markov decision process, which is a 5-tuple (S, A, P, R, γ) where:

- S is a finite set that contains the states of the environment
- A is a continuous set with actions available in each state
- P is the probability distribution that assigns a probability to each state, action pair $(S \text{ x } A) \mapsto [0, 1]$
- R contains the expected immediate rewards (given) for performing an action in a state $(S \text{ x } A) \mapsto \mathbb{R}$
- γ is the discount factor

An agent can perceive from the environment the state $s_i \in S$ he finds itself in. The set of actions A_i, from which the agent can choose one to perform, are bounded to this state s_i.
e.g. imagine a cleaning robot (agent) that has to vacuum the room. It finds itself in a corner (state), and now has to choose in which direction he wants to start cleaning (action). This direction only spans about 90 degrees. While if he would find himself in the middle of the room (different state), he would be free to move in any direction he wanted, the full 360 degrees.

The agent has a policy π that decides which action to select form all possible actions possible in the state s_i. After every (s_i, a_j) combination, the agent will find himself in a new state and will be given a reward r for this combination. This reward will be used to update π, so that the agent can use this information to maximize the accumulated sum of rewards. It means that when our received reward r is positive, for our combination (s_i, a_j). The probability of choosing action a_j at states_i will increase. It also works the other way around, that when r is negative, the probability of choosing a_j will lower.

The main advantages of reinforcement learning is that it can learn online, it does not need to know beforehand which actions are good or bad. It learns this by experimenting in the environment, and receiving rewards based on these actions.

Figure 1.1 shows a diagram of the agent interaction with the environment. After receiving state s_t and reward r_t, it updates policy π and choses an action a_t, for which the environment returns s_{t+1} and r_{t+1}. This keeps happening until the agent has reached a goal state, or until a stopping criterion is met, since the problem to solve might be continuous.

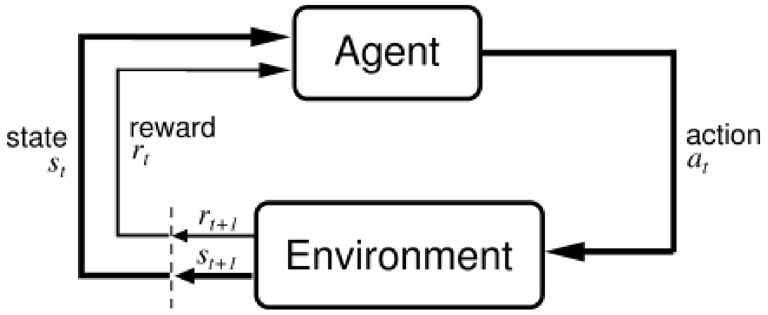

Figure 1.1: The agent-environment interaction in reinforcement learning, taken from [50]

Q-Learning

Reinforcement learning has numerous algorithms to solve its problem, but one of the most well-known is the Q-Learning algorithm [58]. The algorithm allows to find the optimal action in every possible state, and it works as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1.1}$$

Q-Learning learns an action-value function, so that for every state-action pair it will return an expected utility. When this function is learned, we can simply form the optimal policy by selecting the action with the highest value in each state. Equation 1.1 shows how these values are being updated, there are 3 influential parameters, that we will discuss.

α is the learning rate. $\alpha \in [0, 1]$, where a value of 0 indicates that we learn nothing, and a value of 1 means that we will only use the last information given, and that we "forget" the old one. This learning rate can be different for each state-action combination, and would have the optimal value of 1 in a fully deterministic environment.

γ is the discount factor. A value of 0 for γ would mean that our algorithm is short-sighted, only the current reward is taken into account, since we would get:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} - Q(s_t, a_t)] \tag{1.2}$$

A value of 1 would indicate that we look mostly in the long run. Where we would expect a high reward in the end, and not necessarily in the beginning. If our γ value is higher than or equal to 1, our action-values would diverge, which is why: $\gamma \in [0, 1[$.

r is the given reward for a state-action pair. The higher the reward, the better the action was compared to the other possible actions in that state. We want our agent to find the optimal policy, which means that we want the longterm reward to be as high as possible. It might be more interesting for the agent to chose an action with a lower current reward, when it knows that it gives him an opportunity for a higher reward in the next state.

In the next section we will discuss the multi-armed bandit problem. Here we have an agent that finds itself in a stateless environment, and has a finite number of actions to take. It is its objective to maximize the longterm reward, which is done by searching for a good tradeoff between exploring the search space, and exploiting it.

1.2 Multi-Armed Bandit Problem

1.2.1 Introduction

The slot machines from casino's are known for being called one-armed bandits, because the chances of winning are so low, that they are practically stealing from you. Picture a gambler in front of multiple slot machines, he has to determine for himself how many times he is going to play each machine, and in which order he is going to play them. This where the multi-armed bandit problem [45] comes from. When our agent, in this case the gambler, choses an arm, he will receive a reward drawn from a fixed probability distribution. The goal is to maximize the cumulative sum of these rewards. The agent does not know the probability distribution, which is why he has to find the middle ground between exploration (pulling arm randomly, and thus exploring the environment) and exploitation (pulling the best arm possible).

Formalization

The multi-armed bandit has a couple of parameters to take into account. The number of arms K, probability distributions ν_1, ν_2, ..., ν_K for each arm, and the number of rounds n. At each round t, the agent will choose an arm $j \in K$, and will receive a reward drawn from distribution $\nu_{t,j}$. $\mu_1, \mu_2, ..., \mu_K$ are the mean values (expected rewards) affiliated to the distributions. Knowing that the agent has to maximize the cumulative sum of the rewards, we call the regret p after t rounds, the difference between the expected rewards from the optimal strategy, and the sum of rewards given after the arm pulls.

$$p = t\mu^* - \sum_{l=1}^{t} r_l \tag{1.3}$$

Where μ^* is the maximum reward mean drawn from the probability distributions, and r_l the reward given at round l. It is obvious that we want to have a regret that is as low as possible.

1.2.2 Strategies

In the following paragraphs we will introduce some strategies to solve the multi-armed bandit problem. We will explain how these strategies operate, to find out more about their performance, we advice the reader to explore the following research: [4, 57, 33].

ϵ-greedy

The ϵ-greedy strategy vastly depends on the ϵ parameter. When having a choice out of K arms, the strategy will choose the arm with the highest sample mean reward $x^* \in K$, with a probability of 1 - ϵ. In the other case it will chose an arm randomly with probability ϵ. An ϵ-value between 0.1 - 0.2 is preferred in most cases.

Softmax

Softmax algorithms pick their arm based on the average returned reward. In this case we show the Boltzamn exploration, which choses an arm using the Boltzman distribution [50]. It works as follows:

$$p_i(t+1) = \frac{\mathrm{e}^{\hat{\mu}_i(t)/\tau}}{\sum_{j=1}^{k} \mathrm{e}^{\hat{\mu}_j(t)/\tau}}, i = 1...n \tag{1.4}$$

Where $\hat{\mu}_i(t)$ is the sampled mean reward from arm i at step t, and τ is called the *temperature* parameter. The lower this τ value is, the higher the probability of the arms with a high expected reward. A high $\tau \to \infty$, will give the same probability to all arms, and thus choses an arm randomly.

Upper Confidence Bounds

The upper confidence bounds (UCB) algorithms [4] are one of the best in the field for the multi-armed bandits problem. They attain logarithmic regret, while being very easy to implement. UCB1 plays the arm j for which: $\max \bar{r_j} + \sqrt{\frac{2 \ln n}{n_j}}$. Where $\bar{r_j}$ is the average reward gained from arm j, n the total number of pulls, and n_j the number of pulls for arm j.

1.2.3 Variations

Multi-Objective Multi-Armed Bandit Problem

Just like its single-objective counterpart, our agent solving the multi-objective multi-armed bandit problem has to maximize the sum of rewards. Only in this case the given rewards are multi-dimensional, and most of the time these objectives are conflicting. The returned reward is now a vector, and because of this characteristic, it is harder to classify and arrange them.
There are two ways to qualify and order these rewards. Using scalarization functions [23], we transform the reward vector into a single value. The set of single values can be totally ordered. Well-known scalarization functions are the weighted sum approach [38], and the Tchebycheff approach [38]. The other way to distinguish between rewards, is the pareto-dominance method [63], which will be explained in chapter 2 when we aim to optimize our multi-objective problem.

χ-Armed Bandit Problem

In the normal multi-armed bandit problem, there are a finite number of arms to pull. This changes when we focus on the χ-armed bandit problem, also called continuum-armed bandit problem [14, 5, 1]. The arms are part of a set, χ, which contains an infinite number or arms, and can be viewed as a topological space. Simple example; Our arms are $\in \mathbb{R}$, within a certain interval. The returned reward r, is given by a function $f(x)$, where x is our pulled arm. It is then the job of the algorithm to maximize the accumulated sum of the rewards, and find the best arm possible. Some work has been done towards solving this problem [5, 30], and we will discuss it in detail in chapter 3.

1.3 Multi-Objective Optimization

When making decisions, if it is in your personal or professional life, there are always a couple of criteria that will clash with each other. There is not one perfect choice, only a couple of good choices that are not better than each other. i.e. when buying a car, the higher the price, the more power, and comfort the car will have. Thus there is a tradeoff between money, and car performance, which will have an influence on your decision. This is called multiple-criteria decision analysis, from which multi-objective optimization is an area of interest, that optimizes multiple mathematical objective functions.

Formalization

$$Minimize f(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_M(\mathbf{x})] \tag{1.5}$$

Where $\mathbf{x} = (x_1, x_2, ..., x_N)$, is the vector with N decision variables. There are M number of objectives, and the decision variables are prone to constraints, which means that every variable has a lower- x_i^L, and upper bound x_i^R, and thus finds itself within a domain.
To order these solutions, we have to introduce the dominance relation. A solution x_i is said to be dominating another solution x_j when:

- $f_k(x_i) \leq f_k(x_j) \forall k \in [1, m]$, x_i is not bigger than x_j for any objective solution (in case of minimization)

- $f_l(x_i) < f_l(x_j) : l \in [1, m]$, solution x_i is smaller than solution x_j for at least one objective l

Solutions that are not dominated by any other solution is called non-dominated solution. The set of non-dominated solutions is what we call the Pareto Front. The job of our algorithms is to return a set of solutions that are as close as possible to the Pareto Front. This front is convex, only if all the objective functions are convex, else it is non-convex. This is important, because we are going to use scalarization techniques that will help us move faster to this Pareto Front.

We chose the Tchebycheff approach instead of the weighted sum approach, because the weighted sum approach can not find all the optimal points when our objective space is non-convex [48, 2].

1.4 Evolutionary Computation

Evolutionary computation is a research domain of artificial intelligence, that focusses on using evolutionary biology techniques in algorithms, to solve combinatorial- and continuous optimization problems. The research domain has been around for a long time (since 1960's), and a lot has been accomplished. The most important techniques out of this work are: genetic algorithms [27, 40], evolutionary algorithms [6, 7], genetic programming [32, 49] and swarm intelligence [9, 8]. We will be focussing mostly on evolutionary- and genetic algorithm techniques to help with multi-criteria optimization. In the following part we give a brief introduction to evolutionary algorithms, but we will go into more detail in chapter 2.

1.4.1 Evolutionary Algorithms

When given a function to be optimized, evolutionary algorithms start by selecting a number of potential solutions within the domain randomly, or by using a problem-specific method. These solutions form a population, and they are graded using a fitness measure. Out of this population we choose an amount of good candidates, depending on the fitness measure. Depending on the decision makers choice, two kind of techniques can be applied separately or in combination.The recombination technique takes two or more solutions from the set of good solutions to operate as parents. By recombining these parent we create new, better solutions which we call children. The mutation technique only takes one solution as input, and mutates the solution slightly.

The children are forming a new set, which we call the offspring, and based on the fitness measurement we create the population for the next iteration, also referred to as generation. This will keep happening until the algorithm has reached a computational limit, or when we find a solution whose fitness is large enough. The pseudo code for a standard evolutionary algorithm is given by Algorithm 1, while Figure 1.2 shows us an example of how an evolutionary algorithm could be formed.

Fitness measurement

As mentioned before, the fitness measurement grades solutions so that they can be compared, and that the best ones can be selected. The fitness function can be decided by the designer, but because of the iteration, it has to be calculated numerous times. So the designer has to make sure that the function can be computed quickly, but at the same time give helpful information about the environment so that the algorithm finds better solutions more quickly. When

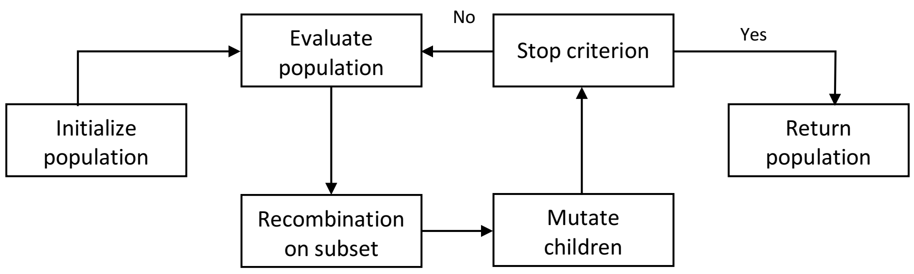

Figure 1.2: Example of a evolutionary algorithm scheme

we consider a multi-objective problem, we use scalarization in our algorithms. We will be using the Tchebycheff approach as our fitness function, because it gives a good indication of the value of our solution, and is calculated quickly.

Recombination

After selecting the best candidates from the population, we use them to create new, and even better solutions. This is based on evolutionary biology, where through natural selection, the best genes from the parents live forward in the child. There are multiple recombination techniques: one-point crossover, split and slice, two-point crossover, uniform crossover and simulated binary crossover [18]. Figure 4.2 shows how one-point crossover works, which is the kind of recombination we will be using in our evolutionary algorithms.

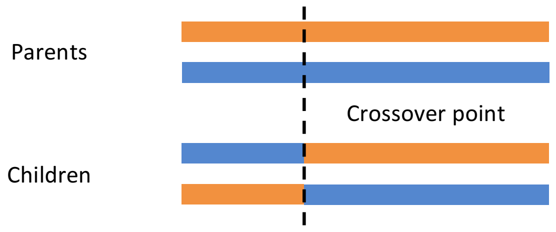

Figure 1.3: One-point crossover in action

Mutation

To make sure that we do not converge to one area of the solution space, and that we explore it, we mutate our solutions such that there is some randomness within our population. In most cases the solution will be changed by a user-defined probability. If the probability is too high, our algorithm would be doing a random search, which is why the mutation probability should be kept quite

Algorithm 1 Pseudo code for a standard evolutionary algorithm.

```
    Initialize population by selecting solutions randomly or using problem-
    specific method
    Evaluate population with a particular fitness measure
    while not stopping criterion met do
      Select solutions based on fitness, called parents
5:    Use recombination technique on the parents to create children
      Perform mutation on the children
      Evaluate children
      Select solutions from children and population based on fitness to create a
      new population
    end while
10: return population
```

low. There are a couple of mutation methods: bit string, boundary, uniform and gaussian mutation. We will be using gaussian mutation, where we add a random value to our solution. This random value has a user-specified upper and lower bound for the solution. i.e. solutions that are already close to the optimal solution or more generally the Pareto Front, should not be mutated as much as solutions that are far away from the Pareto Front.

1.5 Outline

The thesis is composed of nine chapters, and is structured as follows:
Chapter 1, started with a small introduction to get a broad view on the subject of the thesis.
Chapter 2 goes into more detail on the multi-objective optimization problem, and shows off two algorithms to solve it. These two will provide the basis for our algorithms that will solve the multi-objective χ-armed bandit problem.
The multi-objective χ-armed bandit problem is explained in chapter 3, we discuss the problem itself and the work that already has been done by others.
Chapter 4 takes two algorithms from chapter 2, and adapts them to solve the problem from chapter 3.
Chapter 5 introduces a new algorithms, based on simultaneous optimistic optimization. Some of these algorithms use scalarization to distinguish between solutions, others use Pareto-dominance.
We explain our experimental setup in chapter 6, and show, and discuss the results in chapter 7.
We will finish by drawing a conclusion and talk about future work that can be done in chapter 8.

1.6 Summary

We started by explaining the reinforcement learning problem. Receiving rewards for interacting with the environment, and learning using this information. Brute force methods or optimality criteria algorithms [34] could solve this problem too, but we choose to explain one of the best-known algorithms, the Q-Learning algorithm. It can find an optimal policy for any finite Markov decision process. We used the reinforcement learning as building blocks to explain the multi-armed bandit problem. Given a number of arms to pull, how many times, and in which order should these be pulled to maximize the sum of rewards, returned by these arm pulls. We showed some strategies to solve it, and went in a little more detail, because the topic of this thesis, the χ-armbed bandit problem, is a variation of the multi-armed bandit problem. We went on to describe multi-objective optimization, where we have to optimize a number of objectives simultaneously, while there are trade-offs to be made between these objectives. Evolutionary computation was illustrated, the evolutionary algorithms are inspired on evolution techniques to quickly solve optimization problems. We finished by explaining the outline of the thesis for clarification.

Chapter 2

Multi-Objective Evolutionary Algorithms

We introduced evolutionary algorithms in chapter 1. We explained the scheme of these algorithms, and gave some details on how every part works. In that same chapter we presented the multi-objective optimization problem. Optimizing multiple objective functions, when there is a tradeoff between these objectives. We will now see how we can use evolutionary algorithms, to solve the multi-objective optimization problem, and how this will be useful for solving the χ-armed bandit problem.

2.1 Introduction

A lot of problems in our life, consists out of multiple objectives that are conflicting with each other, which prevents us from optimizing all of them simultaneously. Evolutionary algorithms are a well-known approach to solving this problem, due to its characteristics to measure, and diversify solutions [31, 60, 61]. There are two methods to create a total order ranking between different solutions.
In the first approach, we use Pareto dominance ranking to find the set of Pareto optimal solutions, these are solutions that are non-dominated to each other.
The second approach combines each of the objective functions result into one single scalar value. Most of the times, this is done by giving weights to the objective functions, and then using a weighted sum-approach [38], or Tchebycheff approach [38], that will return a single value. This is much easier to work with, but has its drawbacks. i.e. the weighted sum method is not able to return solutions that find themselves in a non-convex part of the solution space [48, 2]. Not only that, the decision-maker has to make their preferences clear [35], in these weights, which is not an easy task. Small changes in the weights, can lead to completely different results [36].

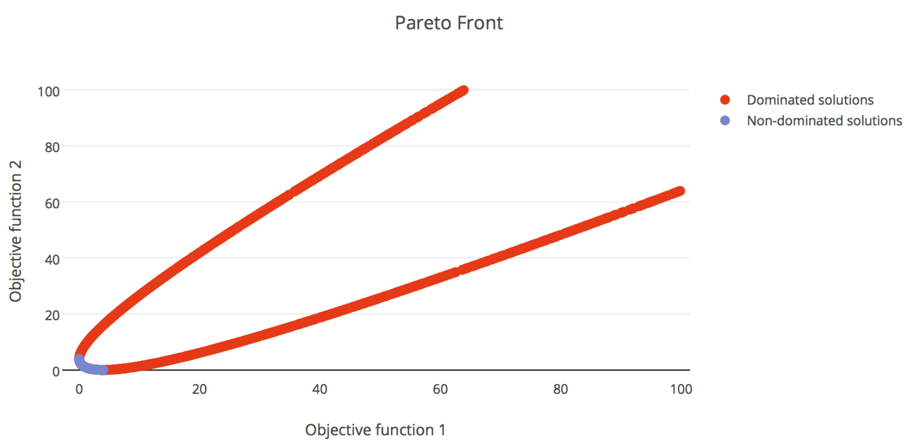

Figure 2.1: Objective space with distinction between the Pareto front, and the dominated solutions

Formalization

$$Minimize\ f(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_M(\mathbf{x})] \tag{2.1}$$

Where $\mathbf{x} = (x_1, x_2, ..., x_N)$, is the vector with N decision variables. There are M objective functions, and the decision variables are prone to constraints. Every variable has a lower- x_i^L, and upper bound x_i^R.
If we would optimize $\mathbf{x}$ for a single objective f_i, we might get unsatisfactory results for the other objective functions, because of the potential conflicts between them. This means that we will be practically unable to find the perfect solution that optimizes all objective functions at the same time. Which is why find a set of solutions that are non-dominated to each other. A set of non-dominated solutions consists out of solutions that are not better or worse than each other. The dominance relation is explained as follows:

$$\text{Solution } \mathbf{x} \text{ is said to dominate a solution } \mathbf{y}\ (\mathbf{x} \succ \mathbf{y}) \Leftrightarrow f_i(\mathbf{x}) \leq f_i(\mathbf{y})$$
$$\text{for } i = 1, ..., M, \text{ and } f_j(\mathbf{x}) < f_j(\mathbf{y}) \text{ for } j \neq i$$

A Pareto optimal solution, is the solution that is not dominated by any other solution in the complete solution space. This tells us that this solution cannot be improved in one objective, without weakening one of the objective functions. Most algorithms aim to find the Pareto optimal set, or a subset of it. The decision-maker does not know which solution he will find to be the best, which is why we find a set of solutions that are uniformly distributed over the complete Pareto optimal set, also called the Pareto front. The difficulty lies in finding solutions that are as close as possible to the Pareto front, and at the same time spread over the complete Pareto front, which is conflicting for the use of our resources [64]. Figure 2.1 shows an example of a Pareto front, for two objective functions.

The above formalized the multi-objective optimization problem, but before diving into the specifics of the evolutionary algorithms that will be used, we will first give an in-depth view of the characteristics of evolutionary algorithms. Expanding on the details shown in chapter 1.

2.1.1 Evolutionary Algorithms

Evolutionary algorithms are population-based, this means that you start, and end with a population of solutions. The population has a maximum size in most cases, and evolves per iteration based on the survival of the fittest principal [15]. When the algorithm starts, it has two choices, it creates the population by adding a number of randomly chosen solutions, or it uses a heuristic optimization method that is problem-specific.

When using the survival of the fittest approach, there needs to be an indicator of which solutions are the fittest. We calculate this using a fitness measure chosen by the decision-maker, it is important that this fitness measure works fast, but at the same time helps us converge to the optimal solution faster. At each iteration of the algorithm, a number of solutions will then be chosen, based on their fitness to help create new solutions for our population. There are a number of selection methods like tournament selection [39], and truncation selection [41], but we choose to use fitness proportionate selection in our algorithms, because the probability assigned to a solution for being chosen, is correlated to the fitness of the solution. Thus when the fitness of a solution increases, so will its chances of being chosen to serve as a parent.

Fitness proportionate selection

Fitness proportionate selection uses the fitness value of the solutions, to calculate the probability of going to be chosen as a parent for the evolution process.

Solution $\mathbf{x}$ has a probability $p_x = \frac{f_x}{\sum_{i=1}^{N} f_{x_i}}$, where N is the size of our *population*, and f_x the fitness value for solution $\mathbf{x}$

The fitness proportionate selection method is given in Algorithm 11.
Once we have a selection criteria, we can create new solutions using variation operators. There are two operators, that can be distinguished by the number of objects they take as input. These operators are called recombination, and mutation. They can be used both, or separately, this depends on the decision-maker. For example, genetic algorithms normally use both, while evolutionary computation algorithms do not use recombination.

Recombination

Recombination is based on evolution, where the stronger genes from the ancestors are reflected in their descendants. Depending on which recombination technique will be used, the procedure will receive two, or more solutions selected

Algorithm 2 Pseudo code for Fitness proportionate selection

```
    Calculate probabilities for each solution
    Generate a random number $r \in [0, 1]$
    foundSolution = **false**
    $i = 1$
5:  $probabilitySum = 0$
    **while not** foundSolution **do**
       probabilitySum += $p_{x_i}$
       **if** $r < probabilitySum$ **then**
          foundSolution = **true**
10:    **else**
          $i++$
       **end if**
    **end while**
    **return** $x_i$
```

from the population. These solutions are called the parents, and they will be used to create new solutions, called children, that hopefully have a higher fitness. We are using a recombination technique called one-point crossover in our algorithms, which accepts as input two parent solutions. We generate a random number r, and split our parents along that index. Like this we create two new solutions, by combining the split parts of both parents. Figure 4.2 shows how one-point crossover works from a high-level perspective.

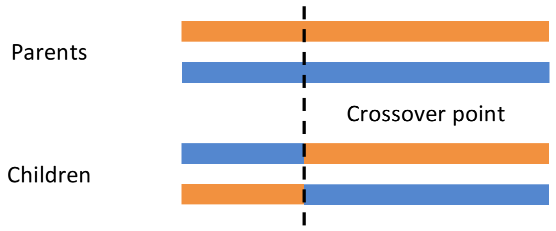

Figure 2.2: One-point crossover in action

Mutation

Mutation takes as input only one solution, and mutates this solution by slightly changing one of its values. This could be by changing one bit, or like in our case, by adding, or subtracting a uniformly random value that was chosen between an upper and lower bound. Algorithm 3 shows the pseudo code of our approach. Mutation is used to prevent our population of converging to one part. Thanks to the random elements in the mutation, our population should diverge. This depends mostly on the uniformly random value called the *mutationGrade*. If

Algorithm 3 Pseudo code for uniform mutation, that takes a *mutationGrade* as input. The *mutationGrade* $\in [0, 1]$

```
$\mathbf{x} = (x_1, x_2, ..., x_N)$
Generate a random number $r \in [1, N]$
Generate a random number $s \in [0, 1]$
if $s<0.5$ then
    $x_r = x_r + x_r * mutationGrade$
else
    $x_r = x_r - x_r * mutationGrade$
end if
return $\mathbf{x}$
```

this is a high number compared to the decision variables of our solution, we would be doing a randomized search. Then again, if the value is too low, we would barely mutate, and our population would converge prematurely.

We have seen how evolutionary algorithms work, and we will now discuss two scalarization techniques, and one based on Pareto dominance ranking. These are used as a fitness measurement for solutions from a multi-objective optimization problem.

Weighted sum method

Weighted sum method is one of the easiest and well-known scalarization techniques. The decision-maker assigns a weight w_i to each objective function $f_i(\mathbf{x})$. The multi-objective optimization problem transforms into a single-objective optimization problem as follows:

$$min\ g(\mathbf{x}) = w_1 f_1(\mathbf{x}) + ... + w_M f_M(\mathbf{x}), \text{where } w_i \geq 0, \text{ and } \sum_{i=1}^{M} w_i = 1 \tag{2.2}$$

Thanks to the simplicity of the weighted sum approach, it is easy to implement, and computationally efficient. On the other hand, as mentioned above, it can not find all Pareto optimal solutions if the Pareto front is non-convex. Using only one set of weights for the complete population would make our population converge to one solution, perfect for those weights. This is why different weights are assigned to each solution in the weight-based genetic algorithm for multi-objective optimization algorithm[26]. It makes sure that the population is diversified, and spans a bigger part of the Pareto front.

Tchebycheff approach

Just like the weighted sum method, the Tchebycheff approach uses a weight w_i for each objective function $f_i(\mathbf{x})$, but instead of making a summation, the single-objective function works as follows:

$$min\ g(\mathbf{x}) = \max_{1 \leq i \leq M} \{w_i |f_i(\mathbf{x}) - z_i^*|\} \tag{2.3}$$

Where $z_i^* = \min\{f_i(\mathbf{x})\}, \forall \mathbf{x} \in \mathit{population}$. The Tchebycheff approach is able to find all optimal solutions, thus also when our solution space is non-convex. One of the downsides of the Tchebycheff approach is that if $\mathbf{x}$ strictly dominates $\mathbf{y}$ ($\mathbf{x} \succ \mathbf{y}$), we could still have that $g(\mathbf{x}) = g(\mathbf{y})$.

Pareto-ranking approach

Using the previously explained Pareto dominance rule, we can rank the solutions from the population, and give a fitness value based on this rank [25]. In other words, we group non-dominated solutions, exclude them from the population, and give them a rank. We keep doing this until the population is empty.

We can clearly see that the lower the ranking, the better the fitness of the solution compared to the others. There are a number of other techniques using the Pareto-ranking approach, and the reader is advised to checkout Fonseca's approach [24], and the Strength Pareto Evolutionary Algorithm [62].

We have seen some evolutionary algorithm techniques in more detail, so that the remaining part of the chapter will be easier to understand. We will be explaining two evolutionary algorithms, namely the multi-objective evolutionary algorithm based on decomposition (MOEA/D) [59], and the non-dominated sorting genetic algorithm II [20]. These are two highly performing algorithms for multi-objective optimization problems. We chose to explain these two, because we will eventually expand upon these in chapter 4, to solve the χ-armed bandit problem.

Algorithm 4 Pseudo code for Pareto-ranking approach, which takes a *population* as input. It will assign a rank to each solution in this population, based on the non-dominated front it is in.

```
   $i = 1$
   $currPop = population$
   while not $currPop = \emptyset$ do
     $F_i \leftarrow ND(currPop)$ {Add non-dominated solutions $currPop$ to $F_i$}
5:   $\forall \mathbf{x} \in F_i, rank(\mathbf{x}) = i$
     $currPop = currPop \setminus F_i$, and $i{+}{+}$
   end while
```

2.2 Multi-Objective Evolutionary Algorithm Based on Decomposition [59]

2.2.1 Introduction

Multi-objective optimization problems can sometimes be very complex, and hard to tackle. To deal with this intricacy, there is a technique called decomposition, that is based on the divide-and-conquer approach. Using decomposition, the problem is divided into multiple sub-problems, that are easier to manage.

The multi-objective evolutionary algorithm based on decomposition uses this technique, and optimizes all these sub-problems at the same time.

2.2.2 The MOEA/D-Algorithm

The multi-objective evolutionary algorithm based on decomposition [59] is a modular algorithm, that gives the decision-maker a lot of freedom to adjust the methods and parameters of the algorithm. The problem is decomposed by using a scalar optimization technique, this can be the weighted-sum method, Tchebycheff approach, or an other method.

Initialization

The algorithm starts by creating a set of N evenly spread weight vectors $(\lambda^1, ..., \lambda^N)$, where each weight vector $\lambda^i = [\lambda^i_1, ..., \lambda^i_M]$, is part of a sub-problem that will be optimized simultaneously. The use of such a weight vector makes sure that we find solutions that are spread over the complete Pareto front. We calculate the euclidean distance between any two weight vectors, and keep track of the T closest ones, which we keep in a list $B(i)$ for each weight vector λ^i. We do this because a solution $\mathbf{x}$ that is optimal for a weight vector λ^i, will also be a good result for a weight vector λ^j, if they are close together. Which means that if we find good solutions for one sub-problem, we can use this information to solve the problems from its neighbors.

We said before that evolutionary algorithms use a population, in this case the population consists of the best solution for each sub-problem. This is not the same as the external population of the algorithm. The external population contains the Pareto optimal solutions we have found so far, in all the sub-problems throughout all iterations. Our population could contain solutions that are not good enough, and are dominated by solutions in the external population.

We generate an initial population x^1, ..., x^N randomly, and use these to initialize our variable z^* that is needed for the Tchebycheff approach. We are using the Tchebycheff approach as a scalarization technique in our algorithms, because it is known to find all solutions of the Pareto front, even when it is non-convex. The Tchebycheff approach is given in Equation 2.3. The initialization of z^* goes as follows:

$$z^*_i = \min\{f_i(\mathbf{x})\}, \forall \mathbf{x}, \text{ and } \forall i \in [1, M] \tag{2.4}$$

After initialization we perform the following loop on each sub-problem i until the stopping criterion is met.

Loop

We select two indexes k, and l from $B(i)$ using the fitness proportionate selection method. The fitness is given by calculating the Tchebycheff value for solution x^i. Using a recombination technique with solution x^k, and x^l as the parents, we create a new solution y. This solution is mutated, to create y', which we use to update z^*. For all the indexes $u \in B(i)$ of the neighbors of the sub-problem i, we do the following:

$$\mathbf{x^u} = y', \text{ if } g(y') \leq g(\mathbf{x^u}) \tag{2.5}$$

Where the function g() is our Tchebycheff approach, and thus calculates the fitness of the solution. The loop ends by updating the external population EP by removing all solutions from EP that are dominated by y', and only add y' if it does not get dominated by any other solution of EP.

Stop

We stop when the stopping criterion is met. This can occur when we found a EP that is close enough to the Pareto front for the decision maker, or when we performed a maximum number of evaluations. The pseudo code of the MOEA/D algorithms is given in Algorithm 12.

Algorithm 5 Pseudo code for the multi-objective evolutionary algorithm based on decomposition. It takes a *stopCriterion*, N sub-problems, and T, the number of neighbors, as input. Tchebycheff function used to calculate the fitness of a solution is given as $g()$.

```
    $externalPopulation = \emptyset$
    create $N$ evenly spread weight vectors
    for each weight vector $\lambda^i, i = 1, ..., N, B(i) = \{i_1, ..., i_T\}$ is the collection of
    the $T$ closest weight vectors for $\lambda^i$ based on the euclidean distance
    generate an initial population randomly
 5: initialize $\mathbf{z}^* = [z_1^*, ..., z_M^*]$, where $M$ is the number of function objectives
    while not $stopCriterion$ do
      for $i = 1, ..., N$ do
        use selection criteria to select two indexes $k$, and $l$ from $B(i)$
        perform a recombination technique on the solutions $\mathbf{x^k}$, and $\mathbf{x^l}$ to create
        new solution $y$
10:     mutate $y$ to create $y'$
        update $z^*$ using $y'$
        for $j = 1, ..., T$ do
          if $g(y') \leq g(\mathbf{x^j})$ then
            $\mathbf{x^j} = y'$
15:       end if
        end for
        remove all solutions from $externalPopulation$ if $y'$ dominates them
        add $y'$ to $externalPopulation$ if there does not exist a solution from
        $externalPopulation$ that dominates $y'$.
      end for
20: end while
    return $EP$
```

2.3 Non-Dominated Sorting Genetic Algorithm II

2.3.1 Introduction

Contrary to the MOEA/D algorithm, the non-dominated sorting genetic algorithm II (NSGA-II) does not convert the multi-objective problem in to a single-objective one using scalar optimization techniques. It is based on the Pareto-ranking approach we showed with Algorithm 4. NSGA-II is an improvement on NSGA [47], which had some drawbacks. NSGA had a high computational complexity (O(MN^3)), the need for specifying a sharing parameter, and it lacked the use of elitism [20]. All these downsides have been tackled by the enhanced second version of the algorithm. We have not explained elitism yet, so before examining NSGA-II, it is best we understand this subject matter.

Elitism is a straightforward concept when it comes to single-objective problems. It is based on the survival of the fittest approach, which means that at each iteration, the strongest solutions will always survive to the next generation. For multi-objective problems, the non-dominated solutions can be seen as the elite, that will always persevere. There are two important factors to using elitism in algorithms. Not only does it avoid removing good solutions from the population, it also has an increasing effect on the performance [46, 64].

2.3.2 NSGA-II

The first version of NSGA sorted each solution into non-dominated front, with a computational complexity of O(MN^3). Deb et al. came up with a solution called the fast non-dominated sorting approach, which would reduce the computational complexity to O(MN^2). Algorithm 6 shows the fast non-dominated sorting approach in pseudo code, but we first explain it in words.

The fast non-dominated sorting approach starts by calculating two new values for each solution $\mathbf{x}$. n_x is the number of solutions that dominate the solution $\mathbf{x}$, and S_x is a set of solutions that get dominated by solution $\mathbf{x}$. We can see that this requires O(MN^2) calculations, M for the number of function objectives of the multi-objective optimization problem, and N^2 because we have to compare each solution to every other solution. During the calculations of n_x, and S_x, we add all the solutions with a value of 0 for n_x to the first front. This is where the second part of the fast non-dominated sorting approach starts.For each solution in the front, it goes over the set of solutions it dominates, and decreases their n_x value each time it encounters this solution. If this value reaches 0, we know it belongs to the next front. We keep doing this until we reach an empty front. The pseudo code of the fast non-dominated sorting approach is given in Algorithm 6.

Like the MOEA/D algorithm, NSGA-II does not specify which kind of recombination or mutation technique should be used, it advices to use both though.

Algorithm 6 Pseudo code for the fast non-dominated sorting approach

```
for all $\mathbf{x} \in population$ do
    $S_x = \emptyset$
    $n_x = 0$
    for all $\mathbf{y} \in population$ do
        if $\mathbf{x} \succ \mathbf{y}$ then
            $S_x = S_x \cup \{\mathbf{y}\}$
        else if $\mathbf{y} \succ \mathbf{x}$ then
            $n_x + +$
        end if
    end for
    if $n_x = 0$ then
        $x_{rank} = 1$
        $\mathcal{F}_1 = \mathcal{F}_1 \cup \{\mathbf{y}\}$
    end if
end for
i = 1
while not $\mathcal{F}_i = \emptyset$ do
    $\mathcal{Q} = \emptyset$
    for all $\mathbf{x} \in \mathcal{F}_i$ do
        for all $\mathbf{y} \in S_x$ do
            $n_y - -$
            if $n_y = 0$ then
                $y_{rank} = i + 1$
                $\mathcal{Q} = \mathcal{Q} \cup \{\mathbf{y}\}$
            end if
        end for
    end for
    $i + +$
    $\mathcal{F}_i = \mathcal{Q}$
end while
```

Besides identifying each non-dominating front, the crowding distance of each solution plays an important role in the algorithm.

Crowding distance helps us in identifying groups of solutions that are spread evenly over the non-dominated fronts. We calculate this crowding distance and use it as a tie breaker between solutions when we have to add them to the next generation. Algorithm 13 shows us how we calculate this value.

NSGA-II uses the crowding distance as the fitness value for the binary tournament selection when we have to chose parent to create new solutions. Algorithm 14 shows us the pseudo code of the complete NSGA-II algorithm. We have seen its characteristics, and how it was an improvement compared to the original NSGA algorithm. If one wants to know how it performs compared to other algorithms on the multi-objective optimization problem, we advice to read the original paper [20].

Algorithm 7 Pseudo code for calculating the crowding distance of solutions, takes a non-dominted front $\mathcal{F}$ as input

```
$l = |population|$
$v = |\mathcal{F}|$
for $i = 1, ..., v$ do
    $\mathbf{x}_{i,distance} = 0$ {solution $\mathbf{x_i} \in \mathcal{F}$ with distance value = 0}
end for
for all $k \in objectiveFunctions$ do
    sort $\mathcal{F}$ based on the value of $k$ for each solution in $\mathcal{F}$
    $\mathbf{x}_{1,distance(k)} = \mathbf{x}_{v,distance(k)} = \infty$
    for $i = 2, ..., v-1$ do
        $\mathbf{x}_{i,distance(k)} = \mathbf{x}_{i,distance(k)} + (f_k(\mathbf{x}_{(i+1),distance(k)}) - f_k(\mathbf{x}_{(i-1),distance(k)}))/(f_k^{max} - f_k^{min})$
    end for
end for
for $i = 1, ..., v$ do
    $\mathbf{x}_{i,distance} = \sum_{i=1}^{|objectiveFunctions|} x_{i,distance(k)}$
end for
```

Algorithm 8 Pseudo code for the NSGA-II algorithm. It takes a maximum population size N, and *stopCriterion*, as input.

```
$t = 0$
create a random population $P_t$ of size $N$
perform recombination and mutation to create offspring $O_t$
while not $stopCriterion$ do
    $R_t = P_t \cup O_t$
    use fast non-dominated sorting approach on $R_t$ to create non-dominated fronts $\mathcal{F}_1, ..., \mathcal{F}_k$
    $P_{t+1} = \emptyset$
    for $i = 1, ..., k$ do
        calculate the crowding distance for all the solutions in $\mathcal{F}_i$
        if $|P_{t+1}| + |\mathcal{F}_i| \leq N$ then
            $P_{t+1} = P_{t+1} \cup \mathcal{F}_i$
        else
            add N - $|P_{t+1}|$ least crowded solutions from $\mathcal{F}_i$ to $P_{t+1}$
        end if
    end for
    use binary tournament selection based on the crowding distance to select solutions from $P_{t+1}$ to create new offspring $O_{t+1}$ using recombination and mutation
    $t = t + 1$
end while
```

Chapter 3

Multi-Objective χ-Armed Bandit Problem

We explained the multi-objective optimization problem, and the way multi-objective evolutionary algorithms work in chapter 2. We introduced two algorithms in detail that solve the problem swiftly as a basis for what is to come. In this chapter we will start by explaining the multi-armed bandit problem, and its multi-objective variation. This serves as a basis for the χ-armed bandit problem, and the multi-objective χ-armed bandit problem. Afterwards we will present an algorithm to solve the multi-objective version, called the multi-objective hierarchical optimistic optimization algorithm [54].

3.1 Introduction

The name "multi-armed bandit", comes from slot machines, which are called one-armed bandits. In the case of slot machines, a person has to pull a lever, called an arm, to make a gamble, and gets a reward as return. When he plays multiple slot machines simultaneously, he can find a machine that is bound to return higher rewards.

The multi-armed bandit problem is a lot alike, where we have an agent, that maximizes the cumulative sum of the rewards, drawn by a fixed probability distribution [45]. The agent does not know this probability distribution, so even if the returned rewards are high, there might be better arms to pull. This is why most algorithms pull an arm randomly on some occasions, and rely on exploring the search space, to shake things up. This is called the exploration vs. exploitation tradeoff in reinforcement learning.

Formalization

At each round t, an agent can choose an arm i out a finite number of K-arms to pull, and in return receives a reward r_i. The reward r_i is drawn from a probability distribution ν_i associated to each arm i, and μ_i is the mean value affiliated to this probability distribution. The agent wants to maximize the total sum of rewards, which means that we can introduce a value called the regret p of the agent. The regret is the difference between the rewards from the optimal strategy, and the sum of rewards returned after the arm pulls.

$$p = t\mu^* - \sum_{l=1}^{t} r_l \tag{3.1}$$

Where μ^* is the maximum expected reward, and r_l the reward given at round l. It is obvious that we want to have a regret that is as low as possible.

Multi-Objective Multi-Armed Bandit

The multi-objective multi-armed bandit is not that different from the multi-armed bandit problem. The goal is still to maximize the cumulative sum of rewards, but the distinction lies in the returned reward. With the multi-armed bandit problem, the reward was a single value, while in this case, the returned reward is a multi-dimensional vector.

In the previous chapters we introduced multi-objective optimization. As seen previously, there are two well-known methods for comparing multi-dimensional rewards: Scalarization functions [23], and Pareto partial order [63]. These two work very well, but there was a need for new regret metrics, to handle these methods for creating a order relationship for the reward vectors. Drugan, and Nowé introduced one regret metric for each method, and one to make sure each optimal arm will be played fairly [22].

In single objective optimization we would want to pull the best arm possible, but when it comes to multi-objective optimization problems there are a number of optimal arms. All these optimal arms should be pulled frequently, this is why an $unfairness$ indicator was introduced, it keeps track of the diversification of the pulled arms. The agent will be punished for not pulling the optimal arms equally.

χ-Armed Bandit

The basics of the multi-armed bandit problem mention that the number of arms are finite, and each arm has its own probability distribution from which rewards are being drawn. This changes when we discuss the χ-armed bandit problem [11], where χ can be a topological space, and thus the number of arms are infinite. This is also called the continuum-armed bandit problem [14, 5, 1], where we have a mean-payoff function f, known by the decision maker, and a domain associated to this function. This domain represents the set of arms χ.

3.2 Multi-Objective χ-Armed Bandit

In the previous section we briefly explained the multi-armed bandit problem, and some of the variations on this problem. These variations were chosen, because they lead up to the main problem of this master thesis, the multi-objective χ-armed bandit problem.

Like in the χ-armed bandit problem, our agent - the gambler - has to choose an arm out of a continuous set of arms, within a finite domain. This domain is connected to the probability distribution of the rewards, which is directly associated with each arm. The returned reward is a multi-dimensional vector, that consists out of a number of values, each given by a different objective function that takes the values associated to the arm pull as input.

In most cases these objective functions are conflicting with each other, which means that there is not one optimal arm to pull, but a set of optimal arms. This set is called the Pareto front of our objective space, it consists or arms that are non-dominated by other arms. The dominance relation is defined a follows:

$$\text{Arm } \mathbf{x} \text{ is said to dominate arm } \mathbf{y} \ (\mathbf{x} \succ \mathbf{y}) \Leftrightarrow r^i(\mathbf{x}) \geq r^i(\mathbf{y}) \\ \text{for } i = 1, ..., M, \text{ and } r^j(\mathbf{x}) > r^j(\mathbf{y}) \text{ for some } j \neq i$$

In most cases we want to pull arms that are as close as possible to the Pareto front, but at the same time we want our set of pulled arms to be diverse, thus spread out over the whole Pareto front. These two objectives might be conflicting towards each other [64], which has a direct influence on the way we pull arms, and the use of our resources.

Formalization

The multi-objective χ-armed bandit problem $\mathcal{B}$ is defined as a pair $\mathcal{B} = (\chi, M)$. χ represents the space of arms that can be pulled, while M is the probability distribution associated to these arms. At each round t, an agent has to decide which arm $k \in \chi$ it is going to pull, and will receive a reward r drawn from M for this. The reward r is a vector consisting out of at least two values. We can differentiate between these arms using the Pareto dominance ranking shown above. The goal of the agent is to find a set of arms whose reward is as close as possible to the Pareto front, while at the same time having a diverse set of arms, which means that those rewards span over the complete Pareto front.

3.3 Related Work

In the following section we will talk about the multi-objective hierarchical optimistic optimization algorithm (MO-HOO) [54]. This algorithm is an expansion on the hierarchical optimistic optimization algorithm [11], which was created to solve the χ-armed bandit problem.

3.3.1 Multi-Objective Hierarchical Optimistic Optimization Algorithm [54]

The multi-objective hierarchical optimistic optimization algorithm creates a binary tree, where each node $\mathcal{P}_{(h,i)}$ is the representation of a part of the action space χ. The pair (h, i) represents the node, where i is the index of the node at depth h. As in normal binary tree behavior, the children of a node $\mathcal{P}_{(h,i)}$ are given as $(h + 1, 2i - 1)$ and $(h + 1, 2i)$. Knowing that $\mathcal{P}_{(h,i)} \subset \chi$, we have two conditions:

1. $\mathcal{P}_{(0,1)} = \chi$, and is the root of our tree
2. $\mathcal{P}_{(h,i)} = \mathcal{P}_{(h+1,2i-1)} \cup \mathcal{P}_{(h+1,2i)}$, for $h \geq 0$ and $1 \leq i \leq 2^h$

The algorithm starts at the root of our binary tree, and keep traversing it until it reaches a leaf of the tree. The traversing is done by choosing a $\mathcal{B}$-vector randomly out of the parents $\mathcal{B}$-set, and going down to the child which contains this $\mathcal{B}$-vector. To clarify, a $\mathcal{B}$-set contains $\mathcal{B}$-vectors that contain optimistic estimates of each objective function. The $\mathcal{B}$-set of a node (h, i) is given as:

$$\mathcal{B}_{(h,i)} = ND(\bigcup_{\mathcal{B}}(\mathcal{B}_{(h+1,2i-1)}, \mathcal{B}_{(h+1,2i)})) \tag{3.2}$$

Where ND is a function that collects the non-dominated solutions into a new set. The pseudo code of the traversing through the tree is given in Algorithm 9.

Algorithm 9 Pseudo code of the MO-HOO method of traversing through a tree $\mathcal{T}$

```
    $(h, i) \leftarrow (0, 1)$
    while $(h, i) \in \mathcal{T}$ do
      $\mathbf{B}_{random} \in \mathbf{B}_{(h,i)}$ {Select B-vector randomly}
      if $\mathbf{B}_{random} \in \mathbf{B}_{(h+1,2i-1)}$ then
5:      $(h, i) \leftarrow (h + 1, 2i - 1)$
      else
        $(h, i) \leftarrow (h + 1, 2i)$
      end if
    end while
10: return leaf $(h, i)$
```

When we reach a leaf in our tree, we take a sample of the action space of our subtree, and are returned a reward vector $\mathbf{y}$. We use this reward to calculate another value, called the $\mathcal{U}$-value, it estimates the quality of our node optimistically.

$$\hat{\mu}_{(h,i)} = (1 - (1/T_{(h,i)}))\hat{\mu}_{(h,i)} + \mathbf{y}/T_{(h,i)} \tag{3.3}$$

$$\mathcal{U}_{(h,i)} = \hat{\mu}_{(h,i)} + \sqrt{(2ln(n))/T_{(h,i)}} + \nu_1 \rho^h \tag{3.4}$$

$\hat{\mu}_{(h,i)}$ is the mean vector of the newly added leaf. The second parameter is a basic exploration term that contains $T_{(h,i)}$ which is a counter that keeps track of the times node (h,i) has been sampled. The third and finale parameter $\nu_1\rho^h$ bears in mind the maximum variation of the payoff function. In case $\mathcal{P}_{(h,i)}$ is a leaf, and we just sampled its action space, the $\mathcal{B}$-set of the leaf, is this $\mathcal{U}$-value.

$$\mathcal{B}_{(h,i)} = \mathcal{U}_{(h,i)} \tag{3.5}$$

Once we pulled the arm, and updated these values, we traverse the tree in reverse order, and update all the $\mathcal{B}$-sets of all the nodes we have seen as given in Equation 3.2. It takes the set of non-dominated $\mathcal{B}$-vectors from the union of the $\mathcal{B}$-vectors of its children. An important notion is that when we create two children nodes, we divide the action space in two, and each child contains a part of the former action space. For example, the Schaffer 1 function:

$$maximize\ f(x) = \begin{cases} f_1(x) = -x^2 \\ f_2(x) = -(x-2)^2 \end{cases} \tag{3.6}$$

where $x \in [-10, 10]$. Thus if we would split our root node, the left child action space domain would be [-10, 0], and the action space domain of our right child would be [0, 10].
This concludes our explanation of the multi-objective hierarchical optimistic optimization algorithm. We have given the pseudo code in Algorithm 10. For more information, we refer the reader to the paper [54] concerning this subject.

Algorithm 10 Pseudo code of the MO-HOO algorithm for a multi-objective χ-armed bandit problem $\mathcal{B} = (\chi, M)$. It takes as input $\nu_1 > 0$, $\rho \in [0, 1]$, and *stopCriterion*. It uses a function $LEAF$, to get the leaf nodes out of a set.

```
    $\mathcal{T} = \{(0,1)\}$
    $\mathcal{B}_{2,1} = [+\infty, ..., -\infty]$, and $\mathcal{B}_{2,2} = [-\infty, ..., +\infty]$
    while not $stopCriterion$ do
       $(h,i) \leftarrow (0,1)$ {We start at the root}
 5:    $\mathcal{P} \leftarrow \{(h,i)\}$ {$\mathcal{P}$ stores the path of the tree we traverse}
       while $(h,i) \in \mathcal{T}$ do
          $\mathbf{B}_{random} \in \mathbf{B}_{h,i}$ {Select B-vector randomly}
          if $\mathbf{B}_{random} \in \mathbf{B}_{h+1,2i-1}$ then
             $(h,i) \leftarrow (h+1, 2i-1)$
10:       else
             $(h,i) \leftarrow (h+1, 2i)$
          end if
          $\mathcal{P} \leftarrow \mathcal{P} \cup \{(h,i)\}$
       end while
15:    $(H,I)/gets(h,i)$
       Choose arm X in $\mathcal{P}_{H,I}$ pull it, and receive reward $\mathbf{y}$
       $\mathcal{T} \leftarrow \mathcal{T} \cup \{(H,I)\}$
       $\hat{\mu}_{(h,i)} = (1 - (1/T_{(h,i)}))\hat{\mu}_{(h,i)} + \mathbf{y}/T_{(h,i)}$
       for all $(h,i) \in \mathcal{T}$ do
20:       $\mathcal{U}_{(h,i)} = \hat{\mu}_{(h,i)} + \sqrt{(2ln(n))/T_{(h,i)}} + \nu_1\rho^h$
       end for
       $\mathcal{B}_{H+1,2I-1} = [+\infty, ..., -\infty]$, and $\mathcal{B}_{H+1,2I} = [-\infty, ..., +\infty]$
       $\mathcal{T}' \leftarrow \mathcal{T}$
       while $\mathcal{T}' \neq \{(0,1)\}$ do
25:       $(h,i) \leftarrow LEAF(\mathcal{T}')$
          if $\mathbf{B}_{h,i}$ is $LEAF$ then
             $\mathcal{B}_{(h,i)} = \mathcal{U}_{(h,i)}$
          else
             $\mathcal{B}_{(h,i)} = ND(\bigcup_{\mathcal{B}}(\mathcal{B}_{(h+1,2i-1)}, \mathcal{B}_{(h+1,2i)}))$
30:       end if
          $\mathcal{T}' \leftarrow \mathcal{T}' \setminus \{(h,i)\}$
       end while
    end while
    return leaf $(h,i)$
```

Chapter 4

Multi-objective Evolutionary Computation

In the previous chapter we explained the foundations of the χ-armed bandit problem [11], and its multi-objective variation. It started by discussing the multi-armed bandit [45] problem where an agent has a finite number of arms to pull, and he has to learn the probability distribution associated to each arm such that it can maximize the cumulative sum of rewards, returned by these arms.

We expanded on this in more detail, and described the multi-objective multi-armed bandits problem where our returned reward is a vector-based, and it is thus harder to differentiate between pulled arms. We went on to describe the χ-armed bandit problem where our arms are part of a compact set within a certain domain, which means that the number of arms are not finite. The chapter ended by explaining the multi-objective χ-armed bandit problem, which is the main problem we will tackle in this thesis.

In the following chapter we will introduce the first two algorithms that solve the multi-objective χ-armed bandit problem. They are based on already existing algorithms created for multi-objective optimization, and only adjusted slightly to work with the χ-armed bandit problem.

4.1 Multi-Objective Evolutionary Algorithm Based on Decomposition (MOEA/D)

The MOEA/D algorithm [59] was created to combine evolutionary algorithms with the decomposition technique to solve multi-objective optimization problems. Using decomposition a problem is broken down into smaller, more manageable, and easier to understand sub-problems. MOEA/D optimizes all these sub-problems simultaneously by combining evolutionary algorithms with scalarization techniques, and sharing gathered information of the environment between the neighbors of a sub-problem.

There are not many algorithms that use decomposition, and most algorithms tackle a multi-objective optimization problem as a whole [19, 12, 51, 62]. Scalarization techniques convert the vector-based solutions into a scalar value which is easier to handle, and makes it more straightforward to create total order ranking in the solution space. In the previous chapters we have seen some scalarization techniques like the weighted sum approach [38], Tchebycheff approach [38], but there are many more techniques, e.g. boundary intersection [16, 37].

The power of MOEA/D lies in the simplicity of the approach, low computational complexity, and that its such a high-level algorithm that leaves a lot of room for input from the decision maker on which evolutionary algorithm techniques the decision maker wants to use. In the following section we give our version of the MOEA/D algorithm, to solve the multi-objective χ-armed bandit problem.

4.1.1 MOEA/D for Multi-Objective χ-Armed Bandit Problem (MOEA/D-χ)

Given a multi-objective χ-armed bandit problem $\mathcal{B} = (\chi, M)$, where χ is the set of arms that can be pulled, and M is the probability distribution associated to these arms. An arm $\mathbf{x} = (x_1, ..., x_l)$ is a vector of l elements, with a domain D_l being present for each element. After an arm pull we get a reward vector r consisting of m elements returned by M.

The MOEA/D-χ algorithm only has a few parameters that have to be decided before it can run. There is need for a stopping criterion, which in the case of the multi-objective χ-armed bandit problem is a maximum number of arm pulls the algorithm can do, $pull_{max}$. A population size, pop_{size} is necessary to tell the algorithm in how many sub-problems it should decompose, our population consists out of pulled arms. T tells us the number of neighbors each sub-problem has, and the external population EP holds all the non-dominated solutions we have encountered.

Initialization

The algorithm starts by creating a list of evenly spread weight vectors $\boldsymbol{\lambda} = (\lambda^1, ..., \lambda^{pop_{size}})$, each weight vector $\lambda^i = (\lambda^i_1, ..., \lambda^i_m)$ is related to an sub-problem i, with m objective functions element of M. For each sub-problem i we calculate the T closest neighbors, which are the T closest weight vectors to λ^i based on the euclidean distance, and keep these in a list $B(i)$.

$$d(\lambda^i, \lambda^j) = \sqrt{\sum_{k=0}^{m} (\lambda^i_k - \lambda^j_k)^2} \tag{4.1}$$

Where λ^i_k is the weight of an objective function k for weight vector i. We generate the initial population randomly, this means that for each sub-problem

i, we have pulled an arm x^i, and are given a reward $r^i = M(x^i)$. We will be using the Tchebycheff approach [38] as scalarization technique, thus we need to have a parameter $\mathbf{z}^* = (z_1^*, ..., z_m^*)$, where $z_j^* = r_j^* = max\{M(x)_j | x \in population\}$, which means the best returned reward by the probability distribution for a specific objective function m, of M. The following loop is done on each subproblem i.

Loop

We calculate the fitness measurement for each pulled arm associated to the weight vectors in $B(i)$ using the Tchebycheff approach [38].

$$x^i_{fitness} = \max_{1 \le s \le m} \{\lambda^i_s |r^i_s - z^*_s|\} \tag{4.2}$$

Fitness proportionate selection, also known as roulette wheel selection, is used to select two indexes u, and v from $B(i)$ to serve as the parents when we create new solutions using evolutionary algorithm techniques. The better the result for the Tchebycheff approach, the closer it will be to the Pareto front, thus the higher the probability of the arm will be for being chosen as a parent. Make sure you take in regard that due to the Tchebycheff approach, the lower the fitness measurement, the better its result is, because we want our returned reward from an arm pull to be as close as possible to z^*.

$$x^i_{prob} = 2/T - (x^i_{fitness} / \sum_{j=0}^{T} x^j_{fitness}), j \in B(i) \tag{4.3}$$

The pseudo code for fitness proportionate selection is given in Algorithm 11, and Figure 4.1 shows an example of the fitness proportionate selection.

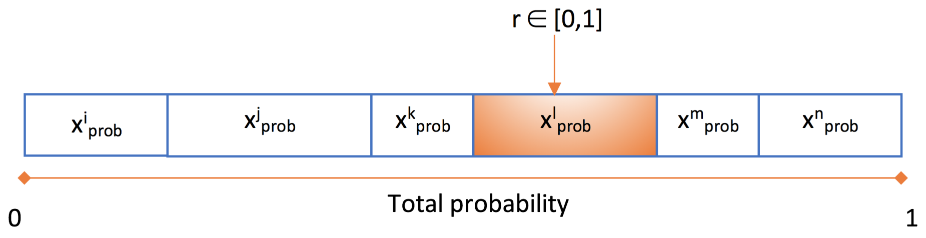

Figure 4.1: Fitness proportionate selection example

When two indexes u, and v have been picked by the fitness proportionate selection method we use the arms x^u, and x^v for recombination. As a recombination technique we use one-point crossover. We pick an index randomly between $]0, l[$, and use it as the crossover point. As mentioned before l was the number of elements in our vector based arm. As a return we receive two new arms, namely y^u, and y^v, created by swapping the left part from the index of x^u, with the part left from the index of x^v. Figure 4.2 clarifies this to make it easier to

Algorithm 11 Pseudo code for Fitness proportionate selection

```
    Generate probabilities for each arm using their fitness measurement
    Generate a random number $rand \in [0, 1]$
    foundSolution = **false**
    $i = 1$
5:  $probabilitySum = 0$
    **while not** foundSolution **do**
        $probabilitySum += x^i_{prob}$
        **if** $rand < probabilitySum$ **then**
            foundSolution = **true**
10:     **else**
            $i++$
        **end if**
    **end while**
    **return** $x^i$
```

comprehend. Before pulling these arms, and getting a reward as return, we adjust each arm by performing a uniform mutation on one of its elements. This element is chosen randomly, and the mutation has to stay between a lower-, and upper-bound, which is the domain in which the element must find itself. We have given an example of uniform mutation in Figure 4.3. After mutating the arms y^u, and y^v into new arms $y^{u'}$, and $y^{v'}$, we pull them, and compare their rewards. The best arm y' is kept, this is based on their fitness measurement. Last thing to do is updating some variables, and checking if we have reached the stopping criterion.

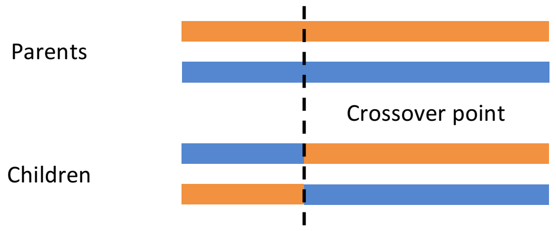

Figure 4.2: One-point crossover in action

Once we have a new arm, and reward for our sub-problem, we start by updating z^*. This is done by comparing each element of the vector based reward with the element of z^*, and keeping the best one.

For each $s = 1, ..., m$, if $r'_s > z^*_s$ then $z^*_s = r'_s$, in the case we want to maximize the cumulative sum of rewards.

We do something similar to each neighbor of sub-problem i, if the reward r' is better than the reward of a neighboring sub-problem, we replace its arm by y'.

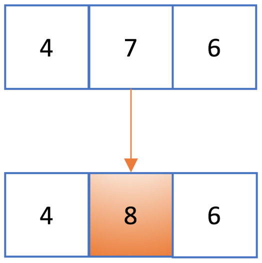

Figure 4.3: Example of a uniform mutation on a vector with three elements

For each $j \in B(i)$, if $r^{'} > r^{j}$ then $x^{j} = y^{'}$, in the case we want to maximize the cumulative sum of rewards.

We end by updating the external population EP, by removing all arms that are dominated by $y^{'}$, and we only add $y^{'}$ if it is not dominated by any other arm in EP. We have given the complete pseudo code in Algorithm 12. We have seen that decomposing one problem, into smaller sub-problems and optimizing each one separately using evolutionary algorithm techniques has a low computational cost, and is easy to implement. The MOEA/D algorithm for a multi-objective χ-armed bandit problem could show to be fruitful, which we will see in the results.

4.2 Non-Dominated Sorting Genetic Algorithm II (NSGA-II)

While the MOEA/D algorithm decomposes the problem, and uses scalarization techniques to work with scalar values, NSGA-II [20] uses a Pareto ranking approach to assign a fitness measure to possible solutions. NSGA-II is an improvement on the first version of the NSGA algorithm [47], because it had some downsides to it.

NSGA used a sharing parameter which had to be tuned for each problem, its computational complexity was $O(MN^3)$, and it did not use elitism [20]. All these problems were solved in the second version, by introducing elitism into the algorithm, and using the fast non-dominated sorting approach to get the computational complexity down to $O(MN^2)$.

We introduced elitism in chapter 2, which came down to letting the best solutions always survive in the population, because this makes sure that we do not remove good solutions out of the population, and it has an increasing effect on the performance [46, 64]. In the following section we will present the NSGA-II algorithm for the χ-armed bandit problem.

Algorithm 12 Pseudo code for the MOEA/D for a multi-objective χ-armed bandit problem $\mathcal{B} = (\chi, M)$. It takes as input a max number of arm pulls $pull_{max}$, pop_{size} as number of sub-problems, number of neighbors T

```
    $EP = \emptyset$
    create $pop_{size}$ evenly spread weight vectors $\boldsymbol{\lambda}$
    compute euclidean distance $d(\lambda^i, \lambda^j)$ between weight vectors
    for each $\lambda^i, i = 1, ..., pop_{size}, B(i) = \{i_1, ..., i_T\}$ is the collection of the $T$
    closest weight vectors for $\lambda^i$
5:  generate initial population randomly
    initialize $\mathbf{z}^* = [z_1^*, ..., z_m^*]$, where $m$ is the number of function objectives
    returned by $M$
    **while not** $pull_{max}$ **do**
      **for** $i = 1, ..., pop_{size}$ **do**
        use fitness proportionate selection to select two indexes $u$, and $v$ from
        $B(i)$ {We use the Tchebycheff approach to calculate the fitness of an
        arm}
10:     perform one-point crossover on $x^u$, and $x^v$ to receive two new arms $y^u$,
        and $y^v$
        mutate $y^u$, and $y^v$, to create $y^{u'}$, and $y^{v'}$
        pull each arm, and keep the best one $y'$ {based on their fitness mea-
        surement}
        update $z^*$, neighboring solutions, and $EP$
      **end for**
15: **end while**
    **return** $EP$
```

4.2.1 NSGA-II for χ-Armed Bandit Problem

After receiving a multi-objective χ-armed bandit problem $\mathcal{B} = (\chi, M)$, the only parameter needed for NSGA-II to work, is the size of the population pop_{size}. When tuning this parameter the decision maker mostly has to look at the computational time, because this increases the higher the size of the population gets.

Initialization

The algorithm starts by generating a parent population P_0 randomly, and thus pulling pop_{size} arms randomly, and keep track of the returned rewards from M. This first round we use as fitness measurement the non-domination level of the arm, thus if an arm is non-dominated it has level 1, the following group that is only dominated by the first one has level 2, and so on. We then use binary tournament selection to select parents from P_0, and use them for recombination and mutation.

With binary tournament selection, we select two arms out of the population, and the one that had the best returned reward, based on Pareto ranking, will serve as a parent. In the case of the rewards both being non-dominated towards each other, we select one randomly between the two. We use the same recombination (one-point crossover), and mutation (uniform mutation) techniques as for the MOEA/D algorithm for the χ-armed bandit problem. The difference is that for NSGA-II we use both new arms after mutation, and put them in the offspring population O_0. Once the population O_0 reaches a size of pop_{size}, the initialization ends, and we combine P_0, and O_0, into a new population $R_1 = P_0 \cup O_0$.

Loop

We start the loop by identifying the non-dominated fronts $F_1, ..., F_k$ of R_t, using the fast non-dominated sorting algorithm, whose pseudo code is given in chapter 2. Here t is just the number of the round we are in, given for clarification. Once these fronts are identified, we loop over them to fill the population P_t for the next round.

This is done by starting to calculate the crowding distance (Algorithm 13) for each arm within that front, and adding all the arms from the complete front to P_t, as long as its size does not reach pop_{size}. In the case of the front being too large to be added to P_t we only add the arms that were the least crowded until our population is full.

Algorithm 13 Pseudo code of calculating the crowding distance for each arm within a non-dominated front $\mathcal{F}$

```
$v = |\mathcal{F}|$
for $i = 1, ..., v$ do
    $\mathbf{x}_{i,distance} = 0$ {arm $\mathbf{x_i} \in \mathcal{F}$ with distance value = 0}
end for
for all $k \in rewardElements$ do
    sort $\mathcal{F}$ based on the value of $k$ for each arm in $\mathcal{F}$
    $\mathbf{x}_{1,distance(k)} = \mathbf{x}_{v,distance(k)} = \infty$
    for $i = 2, ..., v-1$ do
        $\mathbf{x}_{i,distance(k)} = \mathbf{x}_{i,distance(k)} + (f_k(\mathbf{x}_{(i+1),distance(k)}) - f_k(\mathbf{x}_{(i-1),distance(k)}))/(f_k^{max} - f_k^{min})$
    end for
end for
for $i = 1, ..., v$ do
    $\mathbf{x}_{i,distance} = \sum_{i=1}^{|rewardElements|} x_{i,distance(k)}$
end for
```

Like in the initialization phase we use binary tournament selection to select parents, and perform recombination, and mutation on these arms. The algorithm returns a set of offsprings, O_t, that together with the already known population P_t forms $R_{t+1} = P_t \cup O_t$. We check if we have reached the maximum number of arm pulls, if not we start the loop again, else we return R_{t+1}.

As you can see the NSGA-II algorithm is not adjusted, and we only customized the recombination, mutation technique. It will be used as a tool for comparison, and a performance indicator towards the other algorithms. The complete pseudo code of NSGA-II is given in Algorithm 14.

Algorithm 14 Pseudo code for the NSGA-II algorithm. It takes a maximum population size pop_{size}, and a maximum number of arm pulls $pull_{max}$, as input.

```
    $t = 0$
    create a random population $P_0$ of size $pop_{size}$
    perform recombination and mutation to create offspring $O_0$
    $R_1 = P_0 \cup O_0$
 5: while not $pull_{max}$ do
      use fast non-dominated sorting approach on $R_t$ to create non-dominated
      fronts $\mathcal{F}_1, ..., \mathcal{F}_k$
      $P_{t+1} = \emptyset$
      for $i = 1, ..., k$ do
        calculate the crowding distance for all the arms in $\mathcal{F}_i$
10:     if $|P_{t+1}| + |\mathcal{F}_i| \leq pop_{size}$ then
          $P_{t+1} = P_{t+1} \cup \mathcal{F}_i$
        else
          add N - $|P_{t+1}|$ least crowded arms from $\mathcal{F}_i$ to $P_{t+1}$
        end if
15:   end for
      use binary tournament selection based on the crowding distance to select
      solutions from $P_{t+1}$ to create new offspring $O_{t+1}$ using recombination and
      mutation
      $R_{t+1} = P_t \cup O_t$
      $t = t + 1$
    end while
20: return $R_t$
```

Chapter 5

Multi-Objective Simultaneous Optimistic Optimization

In the previous chapter we introduced two algorithms based on evolutionary techniques to solve the multi-objective χ-armed bandit problem. We start this chapter by showing the Simultaneous Optimistic Optimization algorithm (SOO) [44], an algorithm created to optimize single-objective functions. Based on this, we introduce a naive version using scalarization, and a version using Pareto-dominance to solve the multi-objective χ-armed bandit problem. We improve the naive algorithm by extending the functionalities, and thus have 5 algorithms to start the experiments with. These include the multi-objective evolutionary algorithm based on decomposition, the non-dominated sorting genetic algorithm, and the 3 algorithms we introduce in this chapter.

5.1 Simultaneous Optimistic Optimization

5.1.1 Introduction

The SOO-algorithm [42] was first introduced by Munos to optimize deterministic functions without knowing smoothness that is intended to be found near the local optima. It was shown [44] that the SOO-algorithm can be used to solve the best arm identification in multi-armed bandits problem [3, 10]. This is a pure exploration problem where we do not take the cumulative regret that was introduced in Chapter 3 into account, but only look at the simple regret.

After a number of arm pulls, we will return an arm which the algorithm found to be the best one. Which is different from our approach where we return a set of arms which all returned a non-dominated solution as close as possible to the Pareto Front. Therefore we need to adjust the SOO-algorithm to be able to handle multi-objective functions.

5.1.2 The SOO-Algorithm

The most important characteristic of the SOO-algorithm is that its solution space will be represented as a tree. In the beginning, our tree will only exist out of a root. This root contains the whole domain of a function, and is represented as one big n-dimensional cell where each dimension of the cell, corresponds to a dimension of our domain. The solution that resembles this cell, are the center points of each dimension. The nodes of our tree are thus represented by cells.

Initialization

We start by evaluating the root of the tree, which is the center of the domain from all the dimensions. e.g. If we have a two-dimensional input, and the first element has a domain of [0,5], and the second element has a domain of [-2,4], then the input for the root of our tree would be [2.5,1].

Loop

At each iteration, we start at the root of the tree and work our way down, level by level, until we reach the highest depth of the tree, or until we reach a depth that is higher than a parameter $depth_{max}$ given by the decision maker. At each depth h, we solely look at the cells without children, because one of these cells will be split into k new sub-cells along one dimension. The algorithm does not have a way of deciding along which dimension the split should happen, it alternates between them. The cell that will be split has to satisfy two conditions based on its value:

1. It must be the lowest value of all leafs at the depth of the cell
2. We cannot have encountered lower values from previous selected leafs in lower depths

This is in the case of minimization, for maximization we look at the highest values. These iterations keep happening until a stopping criterion is met, then the solution with the lowest encountered value is returned. This solution resembles

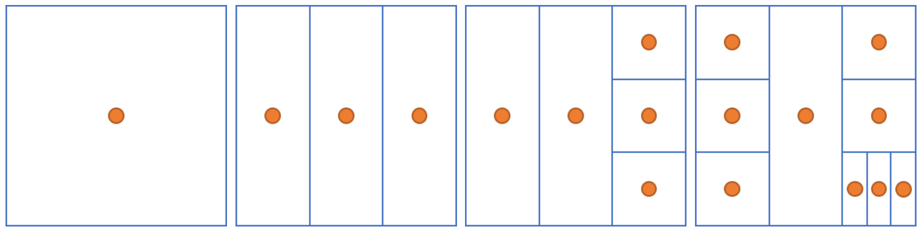

Figure 5.1: SOO-algorithm at work for three iterations, for an input of two dimensions, and for $k = 3$. Points indicate the value within the domain.

the best arm in our pure exploration problem which we mentioned above. Figure 5.1 shows the SOO-algorithm at work for three iterations, while algorithm 15 shows the pseudo code of the algorithm. It is clear that the SOO-algorithm is quite easy to understand and implement, but has some downsides.

Choosing a domain to splits happens purely based on whose turn it is, which is not a proper way to decide which domain should be split, because if domain sizes vary by a big margin, it would be more logical to split the domain that still has a lot of room to be narrowed down. SOO has the tendency to converge asymptotically to the global optimum, and prior knowledge about the objective function that has to be optimized is unnecessary [44]. In the following sections we will expand the algorithm to be able to work with multi-objective rewards that are prone to noise.

Algorithm 15 Pseudo code SOO-algorithm where we *minimize* an objective function f. We require maximum number of evaluations $eval_{max}$, maximum depth of our tree $depth_{max}$, and split factor k as an input before starting the algorithm.

```
    $T \leftarrow (wholeDomain, 0)$ {Our complete domain that will be represented as a tree, at depth 0}
    $evals \leftarrow 1$ {Number of evaluations, we evaluated the one point already, namely the root of the tree}
    $currDepth \leftarrow 0$ {Keeps track of the current depth of our tree}
    while $evals < eval_{max}$ do
5:    $cellsToSplit \leftarrow \emptyset$ {Set of cells that are going to be split}
      $bestResult \leftarrow +\infty$ {Best result encountered during this iteration}
      for $h = 0$ to $\min(currDepth, depth_{max})$ do
        $x^* \leftarrow x : f(x) < f(y), \forall x, y \in LEAFS(T^h)$ {$T^h$, cells at depth $h$}
        if $f(x^*) \leq bestResult$ then
10:         $cellsToSplit \leftarrow cellsToSplit \cup \{x^*\}$
          $bestResult \leftarrow f(x^*)$
        end if
      end for
      split each cell in $cellsToSplit$ into $k$ sub-cells. Evaluate their center point, update $currDepth$ and $evals$
15: end while
    return $x \in T : f(x) < f(y), \forall y \in T$
```

5.2 Multi-Objective Simultaneous Optimistic Optimization using Pareto-Domination

The original SOO algorithm can easily make a distinction between cells, because it works with scalar values, which can be compared, and ordered in a straightforward way. Due to the fact that our returned rewards are multi-objective, it is much harder to distinguish them. In this first algorithm we choose to use the Pareto dominance relation to pick a cell (arm) that we are going to split, or pull.

$$\text{Cell } \mathbf{x} \text{ is said to dominate a cell } \mathbf{y} \ (\mathbf{x} \succ \mathbf{y}) \Leftrightarrow r^i(\mathbf{x}) \leq r^i(\mathbf{y})$$
$$\text{for } i = 1, ..., M, \text{ and } r^j(\mathbf{x}) < r^j(\mathbf{y}) \text{ for } j \neq i$$

At each depth h we look at the returned reward of all the leafs, and select the leaf that dominates the other leafs based on the Pareto dominance relation given above. If there is a set of non-dominated rewards among the leafs, we choose the cell randomly out of that set, and see if it does not get dominated by the current best result we have encountered in previous depths.

Once this cell is chosen we do not immediately split it into k new sub-cells, because of the problem the algorithm solving, namely the multi-objective χ-armed bandit problem. As mentioned in Chapter 3, our multi-objective reward is drawn from a probability distribution M, which means that we might not receive the same reward if we pull the arm twice.

For the sake of argument, imagine two possible arms to pull. The reward of the first arm is very susceptible to noise, while the reward of the second arm is barely influenced by noise, and the average mean reward of the second arm is higher than the one from the first arm. It could be the case that after one pull, the first reward of the arm is actually higher than that of the second arm, while in the long run the average will be higher for the second arm.

In the original SOO algorithm a cell only gets evaluated one time, which would not work for a stochastic environment like the one from the multi-objective χ-armed bandit problem due to the noise. This is why we introduce a new variable $pulls_{min}$ that requires a cell to be evaluated at least $pull_{min}$, before being split into k new sub-cells. It is shown that for $pull_{min} = \frac{eval_{max}}{log^3(eval_{max})}$, and $depth_{max} = \sqrt{\frac{eval_{max}}{pull_{min}}}$ the expected regret is bound to $O(\frac{log^2(eval_{max})}{\sqrt{eval_{max}}})$ [53]. The rest of the algorithm stays the same, and you can find the pseudo code in Algorithm 16.

5.3 Multi-Objective Simultaneous Optimistic Optimization using Scalarization

The multi-objective simultaneous optimistic optimization algorithm explained in the previous section was using Pareto-domination at each depth to choose a cell that we wanted to evaluate, or split. This means that if we encounter a number of non-dominated cells, we have to pick one randomly. Choosing a cell randomly is not preferred, which is why we solve this by using the Tchebycheff approach [38], such that we can work with scalar values. The Tchebycheff approach uses a weight vector $\boldsymbol{\lambda}$ consisting of λ_i weights, for each element i in our reward vector. Equation 5.1 shows us how we receive a scalar value g using our reward vector $\mathbf{r}$, and a weight vector $\boldsymbol{\lambda}$.

$$g^x = \max_{1 \leq i \leq m} \{\lambda_i |r_i^x - z_i^*|\} \qquad (5.1)$$

Algorithm 16 Pseudo code MO-SOO using Pareto-domination to solve a multi-objective χ-armed bandit problem $\mathcal{B} = (\chi, M)$. Requires max number of evaluations $eval_{max}$, max depth of tree $depth_{max}$, and split factor k as input.

```
    $T \leftarrow (wholeDomain, 0)$ {Complete domain represented as tree, depth 0}
    $evals \leftarrow 1$ {Number of evaluations, we evaluated the root of the tree}
    $currDepth \leftarrow 0$ {Keeps track of the current depth of our tree}
    while $evals < eval_{max}$ do
5:    $cellsToSplit \leftarrow \emptyset$ {Set of cells that are going to be split}
      $bestResult \leftarrow +\infty$ {Best result for the given problem}
      for $h = 0$ to $\min(currDepth, depth_{max})$ do
        $ndCells \leftarrow ND(LEAFS(T^h))$ {Non-dominated leafs at depth $h$}
        $x^* \leftarrow random(ndCells)$ {Random arm out of non-dominated leafs}
10:     if not $bestResult$ $DOMINATES$ $r(x^*)$ then
          if $pull_{min} < x^*_{pulls}$ then
            $cellsToSplit = cellsToSplit \cup \{x^*\}$
            $bestResult \leftarrow bestResult \cup r(x^*)$
          else
15:         pull $x^*$, and update average reward value, and $x^*_{pulls}$
          end if
        end if
      end for
      split cells in $cellsToSplit$ into $k$ sub-cells, update $currDepth$ and $evals$
20: end while
    return $ND(T)$ {Return set of non-dominated solutions of the tree}
```

Where $z_i^* = \max\{r_i\}, \forall r \in T$, m is the number of elements in the reward vector $\mathbf{r^x}$ from cell x, and we want to maximize the reward vector. Even when our solution space is non-convex, the Tchebycheff approach is able to find all optimal solutions. One of the drawbacks of the Tchebycheff approach is that if a cell $\mathbf{x}$ dominates a cell $\mathbf{y}$ ($\mathbf{x} \succ \mathbf{y}$), we could still have that $g^x = g^y$.

Weight vector

To properly comprehend decisions that will be made later on for this algorithm, it is vital that the reader understands the use of weight vectors. A weight vector $\boldsymbol{\lambda}$ consists of a number of weights λ_i, where each weight is associated to an objective function that has to be optimized, or in our case to an element of our reward vector $\mathbf{r}$. It has the following constraint:

$$\sum_{i=0}^{m} \lambda_i = 1 \qquad (5.2)$$

Where $\lambda_i \in [0, 1]$, although it is preferred that a weight does not equal 0, because the found solution could be weakly Pareto optimal [36]. The weights of a weight vector can be decided by the decision maker, if he knows where on the Pareto front he wants to find a arm to pull. For example, if the decision maker would choose a weight vector $\boldsymbol{\lambda} = [0.5, 0.5]$, the optimal solution would be found exactly in the middle of the Pareto front associated to the bi-objective space to optimize. Deciding the values of a weight vector is not easy, and there are two important factors one should keep in mind [36]:

1. Value of the weight is significant to other weights
2. Value of the weight is significant towards the objective function it is associated with

The first item tells us that setting a value for a weight, is actually telling how important it is compared to the others. The lower the weight, the less important it is, and this gives you the opportunity to rank all the weights from a weight vector, to get a better grasp of what values you are actually assigning.

The second item is best explained with an example. Imagine a bi-objective optimization problem where the range of the first objective is [0,10], and the range of the second objective is [0,10000]. It is clear that when our first objective would change by a value of 1, it has a much bigger effect using weights than when the second objective changes by a value of 1. This is why it is best to normalize all these functions such that the weights represent your preferences correctly.

For our problem we have no clue what the objective function to optimize is, and are just returned a reward for an arm this, so we can not normalize these values properly. Since we want to find a set of optimal arms, that preferably is spread over the complete Pareto front, we choose to divide our problem into sub-problems for a number of weight vectors.

5.3.1 Naive Multi-Objective Simultaneous Optimistic Optimization using Scalarization

The naive version of our multi-objective simultaneous optimistic optimization algorithm using scalarization does not keep track of a complete tree. It starts at the root, and splits a cell at each iteration, but immediately goes deeper into the tree, and never returns. Because of this we use an external population EP, to keep track of the encountered non-dominated solutions. Deciding which cell to split, depends on its fitness, which we calculate using the Tchebycheff approach.

Initialization

We divide our multi-objective χ-armed bandit problem into num_{prob} number of sub-problems, where each sub-problem is assigned a weight vector $\boldsymbol{\lambda}$. These weight vectors are created evenly-spread. Each sub-problem has a current cell it has to split during each iteration, which is the root of the tree at the initialization phase for all the sub-problems. For each sub-problem we perform the following loop until a stopping criterion is met.

Loop

The current cell is split in to k new sub-cells, called the children of the current cell. For each child we pull the arm that corresponds to the numerical values of this cell, and the returned reward is being saved. We assign a fitness value to these cells by using the Tchebycheff approach.

$$cell_{fitness} = \max_{1 \leq i \leq m} \{\lambda_i |r_i - z_i^*|\} \tag{5.3}$$

Where r is the returned reward vector after pulling the arm, z^* is the best returned reward for each objective function, and λ is the weight vector. Once assigned a fitness value, we use fitness proportionate selection to select which child will be the new current cell. After assigning a new current cell, we see if it dominates any arm in EP, or if it is dominated by any other arm in EP. The pseudo code is given in Algorithm 17.

Algorithm 17 Pseudo code for the naive MO-SOO using scalarization to solve a multi-objective χ-armed bandit problem $\mathcal{B} = (\chi, M)$. Requires max number of evaluations $eval_{max}$, split factor k, and num_{prob} as input

```
    create $num_{prob}$ evenly spread weight vectors $\boldsymbol{\lambda}$
    Set current cell of each sub-problem $i$ to root: $cell^i_{curr} \leftarrow root$
    $EP = \{root\}$ {External population}
    $evals \leftarrow 1$ {Number of evaluations, we evaluated the root of the tree}
5:  while $evals < eval_{max}$ do
      for $i = 0$ to $num_{prob}$ do
        $children = SPLIT(cell^i_{curr})$ {Split current cell of sub-problem $i$}
        Evaluate $children$, and calculate fitness using $\boldsymbol{\lambda^i}$
        $cell^i_{curr} = FPS(children)$ {Use fitness proportionate selection}
10:     Update $z^*$, and $EP$, and $evals$
      end for
    end while
    return $EP$
```

Properties

For the most part, the previous version is called naive due to two big factors:

1. No backtracking
2. Not optimistic

First item is a very important one. We keep splitting the cells, and go deeper, and deeper in the tree, but we do not backtrack. This means that if we would have a split factor $k = 3$, that after the first split, we would not evaluate around 2/3 of a complete objective function domain, and we will not see this part of the domain again. That is a huge chunk of the objective space that gets ignored, which is very unfortunate, and will not give us good arms to pull.

If the algorithm would be optimistic, than we would choose the best evaluated cell, this is not the case now. As you can see in Algorithm 17, we select a cell to split based on its fitness, using the fitness proportionate selection method, where the fitness is based on the Tchebycheff approach.

Both problems will be addressed in the improved version, and will give us much better results.

5.3.2 Improved Multi-Objective Simultaneous Optimistic Optimization using Scalarization

Improving the naive algorithm does not show to be difficult, but as we will see, it does have a higher computational complexity compared to the naive version. One other small difference is that we will reintroduce a minimum number of arm pulls before we split a cell, to assure that we have a better view on the average returned reward of the arm.

One of the major motivations of the improved multi-objective simultaneous optimistic optimization algorithm using scalarization is that we will keep track of one global tree, where a number weight vectors will iterate from top to bottom to decide if we will evaluate, or split a cell. Think of it as if each weight vector would have its own tree, and we would perform the same cell split behavior as the original SOO-algorithm did. Thus starting at the root, evaluating leafs, and only split them if we have not encountered a better one before.

The reason for one global tree, is that it would not be necessary to split a cell again, if it was already split in a tree of a different weight vector. We will have a list of weight vectors, and evaluate each cell without children using this list. Using this evaluation we choose a cell, and based on the number of times the arm associated to the cell has been pulled, we decide to split the cell, or pull the arm again. It might occur that we do not use the complete list of weight vectors to evaluate a cell, because if the fitness of the cell was not good enough for a weight vector, we are not going to evaluate its children, using that weight vector. This will reduce the computational time needed to create this tree, and evaluating cells.

The algorithm takes a multi-objective χ-armed bandit problem $\mathcal{B} = (\chi, M)$, a stopping criterion $eval_{max}$ for the maximum number of evaluations, and a value num_{weight} for the number of evenly spread weight vectors $\boldsymbol{\lambda}$ we have to create.

Initialization

Since we are going to have a minimum number of arm pulls, and thus a minimum number of cell evaluations before we split it, we introduce the value $pull_{min} = \frac{eval_{max}}{log^3(eval_{max})}$. The maximum depth of the tree can be altered, but we choose to go for $depth_{max} = \sqrt{\frac{eval_{max}}{pull_{min}}}$, since it was shown that using these two values we would have an expected regret that was bound to $O(\frac{log^2(eval_{max})}{\sqrt{eval_{max}}})$ [53].

We start by creating a list containing num_{weight} of evenly spread weight vectors, and evaluate the root of our global tree for $pull_{min}$ number of times. The root of the tree is the arm that represents the center values of each domain of the elements of our input. e.g. when our input consists of two elements, and the domain of both elements is [0,10], then the root of the tree will be the arm associated to the vector [5, 5].

We give the root a fitness based on the Tchebycheff approach, the same as we use for the naive version, for each weight vector. Each cell, and thus also the root, contains a set with weight vectors as members, and its fitness using these weight vectors. It keeps track of which weight vector is searching for better solutions in that part of the tree. We start a loop over the list of weight vectors, and iterate over the complete tree. The following loop keeps going until we have met the stopping criterion $eval_{max}$.

Loop

For each weight vector λ^i, we start at the root of the tree, thus at depth $h = 0$. At each depth h, we look at all the cells that are leafs in our tree for the weight vector λ^i, which means that the cell has λ^i in its set of members, and if the cell has children, than its children do not have λ^i as a member. We look at the associated fitness value which we calculated, and choose the leaf with the best value, as long as this value is better than previously encountered fitness values of leafs at a lower depth. There are two possible steps to take, depending on the number of times the cell has been evaluated (the arm has been pulled).

If the cell has been evaluated enough times, then we see if it already has children in our global tree. In this case we will add the weight vector to the set of members of the children, else we split the cell, and evaluate the children one time.

If the arm has not been pulled enough, we pull it again, and update its evaluation by taking the average for each element of the returned reward, and recalculate its fitness. This iteration of the loop ends when we reach the highest depth, and we start over for the next weight vector until we reach the stopping criterion. We return all the non-dominated cells from our tree as the selection of possible arms that we recommend to be pulled. We have given the pseudo code in Algorithm 18.

Properties

In the naive version of the multi-objective simultaneous optimistic optimization algorithm using scalarization occured two big problems. We did not do any backtracking in the tree, and the algorithm was not optimistic. Both problems were solved in the improved version of the algorithm. We were able to minimize the computational complexity by using one global tree, instead of a tree for each possible weight vector. This makes sure that we do not have to do the same evaluations, and fitness calculations over, and over again for the same cells. This concludes the explanation of methods we will be using for the multi-objective χ-armed bandit problem, and in the following chapter we will discuss the problems on which we will test our algorithms.

Algorithm 18 Pseudo code for the improved MO-SOO using scalarization to solve a multi-objective χ-armed bandit problem $\mathcal{B} = (\chi, M)$. Requires max number of evaluations $eval_{max}$, split factor k, weight vector list size of num_{weight}, minimum $pull_{min}$ of pulls before splitting a cell, and maximum depth $depth_{max}$ as input. Tchebycheff approach is given

```
    create $num_{weight}$ evenly spread weight vectors $\boldsymbol{\lambda}$
    $T \leftarrow root$ {Root is evaluated}
    $evals \leftarrow pull_{min}$ {Number of evaluations, we evaluated the root of the tree}
    while $evals < eval_{max}$ do
5:    for $i = 0$ to $num_{prob}$ do
        $bestResult = 0$
        for $h = 0$ to $\min\{depth_{max}, depth_{highest}\}$ do
          $bestLeaf = cell^x | cell^x_{fitness} > cell^y_{fitness} \forall x, y \in LEAFS(T^h, \lambda^i)$
          {Get leaf with best fitness at depth h for $\lambda^i$}
          if $bestLeaf > bestResult$ then
10:         $bestResult = bestLeaf$
            if $bestLeaf_{pulls} \geq pull_{min}$ then
              if $bestLeaf_{children} = 0$ then
                Split $bestLeaf$
              end if
15:           Add $\lambda^i$ to $members$ of children
            else
              Evaluate $bestLeaf$, and update its fitness
            end if
          end if
20:     end for
      end for
    end while
    return Non-dominated solutions of $T$
```

Chapter 6

Experimental Setup

In previous chapters we have seen a number of algorithms that are able to solve the multi-objective χ-armed bandit problem. We discussed their functionalities, and features, but did not examine their performance on such problems. In the following chapter we will explain the problems we will test our algorithms on, the parameters that will be used in our functions. All algorithms are written in Java, and experiments run independently on a Macbook Pro (late 2013), with the following specs:

- **Processor**: 2,4 GHz Intel Core i5
- **Memory**: 8 GB 1600 MHz DDR3
- **GPU**: Intel Iris 1536 MB

6.1 Quality Indicators

When we evaluate our algorithms, we will focus on 3 different kinds of evaluations.

- Cardinality, which is the size of the non-dominated set of arms
- Distance from the non-dominated set of arms to the Pareto front
- The spread of the set non-dominated arms

The hardest part here is to divide the resources such that no quality indicator will be neglected. It would be easy to find 1 arm on the Pareto front, but it would be much better if we would find 50 different arms close to the Pareto front.

Cardinality

The algorithms introduced in previous chapters will always returns a set of non-dominated arms sol_{ND}. This can be the external population EP from the multi-objective evolutionary algorithm based on decomposition [59] algorithm, the non-dominated arms from population R_t from the NSGA-II [20] algorithm, or the set non-dominated arms created after traversing our complete tree T in the MO-SOO algorithms. To find out what the $Cardinality$ is of these sets, we only have to find out what its size is.

$$Cardinality(sol_{ND}) = |sol_{ND}| \tag{6.1}$$

The higher the $Cardinality$ the more arms we have in our non-dominated set, and thus the more options we give the decision maker to choose from.

Distance, and Spread

To find out the distance from our set of non-dominated arms to the Pareto front, we will calculate the average euclidean distance from a set of uniformly distributed points PF of the Pareto front, to our set of non-dominated arms sol_{ND}. This is called the Inverted Generational Distance IGD [55], and tells us something about the distance from the set of non-dominated arms, and the spread of this set.

$$IGD(sol_{ND}, PF) = \frac{\sum_{s \in PF} d(s, sol_{ND})}{|PF|} \tag{6.2}$$

Where $|PF|$ is the total number of uniformly distributed points taken from the Pareto front, and $d(s, sol_{ND})$ is the smallest euclidean distance between point $s \in PF$, and the set of non-dominated solutions sol_{ND}.

$$d(s, sol_{ND}) = min\{\sqrt{\sum_{i=0}^{m}(s^i - r_x^i)^2}|x \in sol_{ND}\} \tag{6.3}$$

Where m is the number of objectives that need to be optimized, and r_x is the reward vector associated to the arm x. If the IGD metric is small of a set of non-dominated arms it would indicate that not only are our arms close to the Pareto front, but also spread properly. This has to do with calculating the distance from points of the Pareto front, to our set, and not the other way around. This way we check at the same time if the spread of our set is good.

6.2 Problems

We want to test the algorithms on a wide range of problems that have different characteristics, so that we can understand the strengths and weaknesses of the algorithms. A lot of test suites have been presented to test multi-objective evolutionary algorithms on [17, 56, 21, 64, 43, 28, 29], and when we are going to perform our experiments we will be doing this on the ZDT test suite [64]. The ZDT test suite is one of the most used test suites available, and a lot of research papers have presented their results based on this test suite. The ZDT test suite consists solely out of bi-objective problems, which is why we will also be testing the algorithms on the DTLZ test suite [21]. The problems from the DTLZ test suite are more difficult, and need 3 objectives to optimize. Since this is not just a multi-objective optimization problem, but an multi-objective χ-armed bandit problem, we need to have some noise in our data.

Uniform Noise Distribution

We have chosen to add Uniform noise $\mu = 0.1$ to the result x of each objective of the problem. The probability of a degree of noise being added to the result x of an objective is equally probable for the complete interval. In case of $\mu = 0.1$, this means that the result will find itself in the range of $[x \times 0.9, x \times 1.1]$, which is also shown in Figure 6.1.

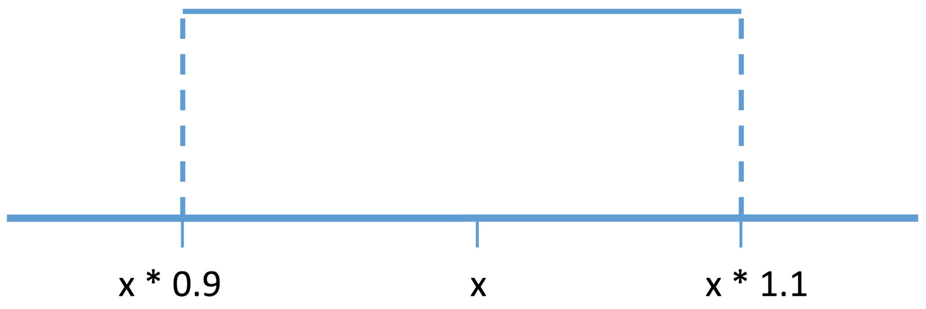

Figure 6.1: Uniform noise $\mu = 0.1$ on the result x of an objective.

6.2.1 ZDT Test Suite [64]

All ZDT problems are minimization problems, but the algorithms presented in previous chapters want to maximize the returned rewards. This is why we will multiply the rewards with -1 such that the algorithms will still be able to work with these problems. We have presented which problems from the ZDT test suite we will be using below, they are all bi-objective optimization problems.

ZDT2

The ZDT2 is a bi-objective problem, with a non-convex Pareto front given in Figure 6.2, the formulation of the problem is described in Equations (6.4) and (6.5).

$$f_1(x) = x_1 \tag{6.4}$$

$$f_2(x) = g(x)[1 - (\frac{f_1(x)}{g(x)})^2] \tag{6.5}$$

$$g(x) = 1 + \frac{9(\sum_{i=1}^{n} x_i)}{(n-1)} \tag{6.6}$$

Where $x = (x_1, ..., x_n)^T \in [0, 1]^n$. We will be using $n = 30$ in our experiments, since the problem will be quite easy to solve.

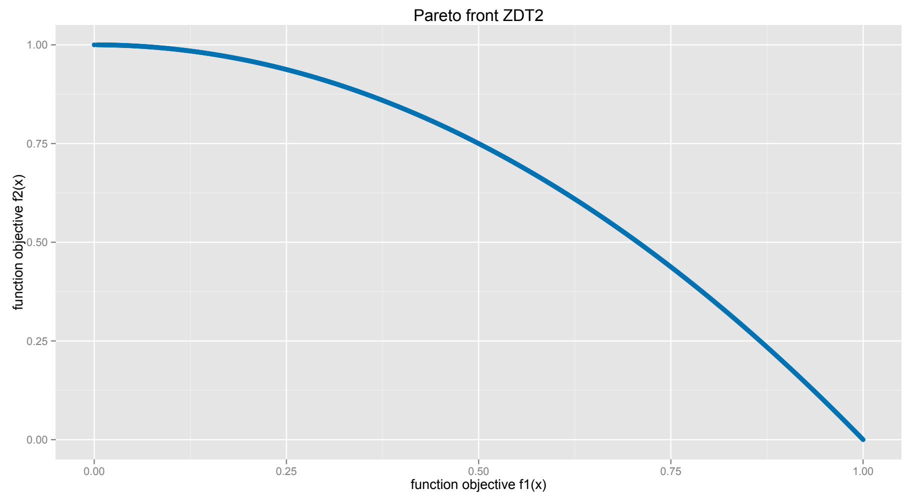

Figure 6.2: 1000 uniformly distributed points on the Pareto front of the ZDT2 problem, points taken from [52]

ZDT3

Figure 6.3 shows us that the Pareto front of the ZDT3 problem is disconnected, and consists of convex, and concave parts. The bi-objective ZDT3 problem is described as follows:

$$f_1(x) = x_1 \tag{6.7}$$

$$f_2(x) = g(x)[1 - \sqrt{\frac{f_1(x)}{g(x)}} - \frac{f_1(x)}{g(x)} sin(10\pi x_1)] \tag{6.8}$$

Where $g(x)$ is given in Equation (6.6), and the dimensionality, and range of x is the same as in the description of the ZDT2 problem.

ZDT4

The ZDT4 problem tests the algorithms how they can handle multimodality since it contains many local Pareto fronts. The Pareto front of the ZDT4 prob-

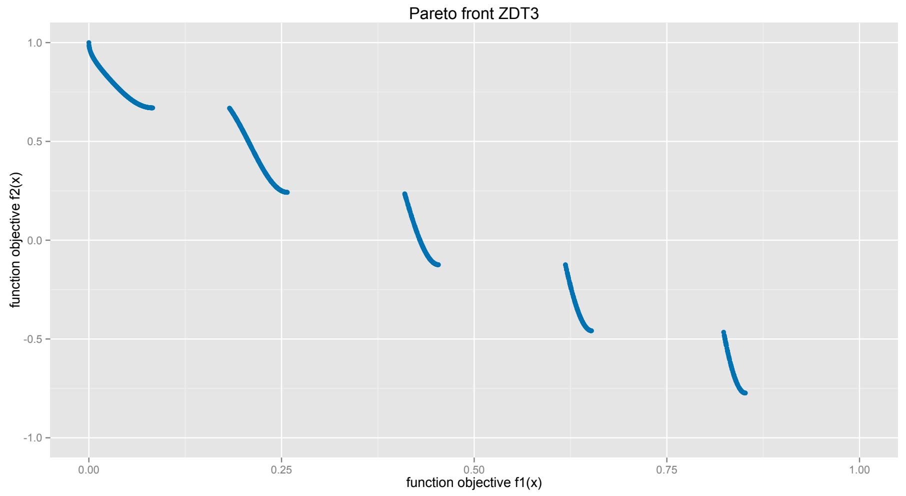

Figure 6.3: 1000 uniformly distributed points on the Pareto front of the ZDT3 problem, points taken from [52]

lem is given in Figure 6.4, and the formulation in Equations (6.9) and (6.10).

$$f_1(x) = x_1 \tag{6.9}$$

$$f_2(x) = g(x)[1 - \sqrt{\frac{f_1(x)}{g(x)}}] \tag{6.10}$$

$$g(x) = 1 + 10(n-1) + \sum_{i=2}^{n}[x_i^2 - 10cos(4\pi x_i)] \tag{6.11}$$

Where $x = (x_1, ..., x_n)^T \in [0,1]x[-5,5]^{n-1}$. We will be using $n = 10$ in our experiments.

ZDT6

The Pareto front of the ZDT6 problem is non-convex, and the points on the Pareto front are distributed very nonuniform. The formulation of the problem is given in Equations (6.12) and (6.13).

$$f_1(x) = 1 - exp(-4x_1)sin^6(6\pi x_1) \tag{6.12}$$

$$f_2(x) = g(x)[1 - (\frac{f_1(x)}{g(x)})^2] \tag{6.13}$$

$$g(x) = 1 + 9[\frac{\sum_{i=2}^{n} x_i}{(n-1)}]^{0.25} \tag{6.14}$$

Where $x = (x_1, ..., x_n)^T \in [0,1]^n$. We will be using $n = 10$ in our experiments.

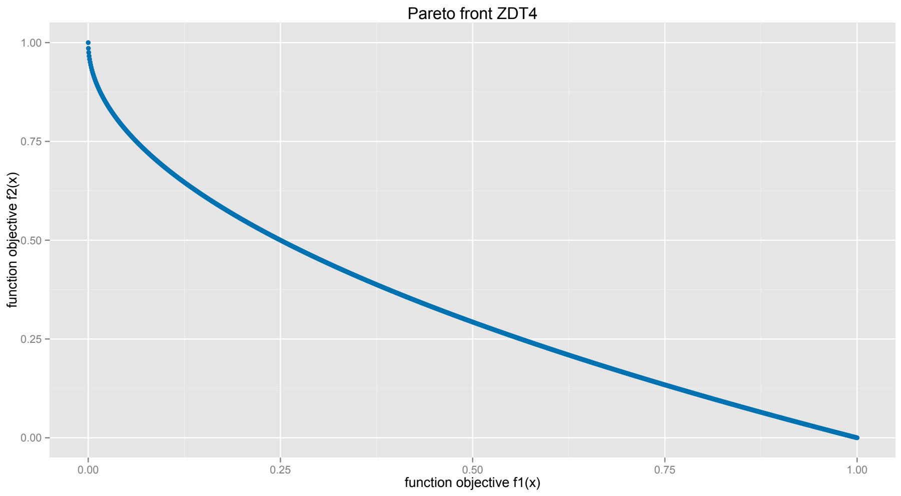

Figure 6.4: 1000 uniformly distributed points on the Pareto front of the ZDT4 problem, points taken from [52]

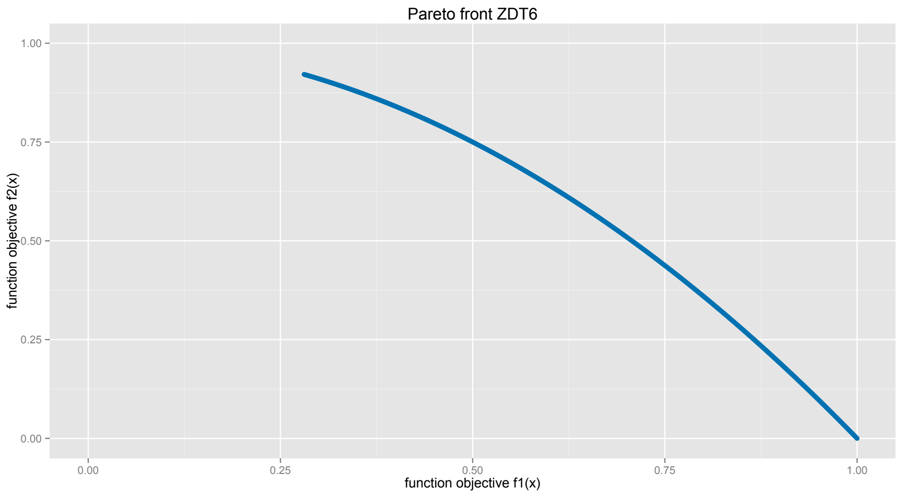

Figure 6.5: 1000 uniformly distributed points on the Pareto front of the ZDT6 problem, points taken from [52]

6.2.2 DTLZ Test Suite [21]

The DTLZ test suite consists out of 9 problems where 3 objectives have to be optimized, we choose to optimize 2 of these problems. As in the ZDT test suite we will multiply the returned reward with -1 so the algorithms can maximize the problem, instead of minimize them.

DTLZ1

The Pareto front of the DTLZ1 problem is clearly non-convex, as can be seen in Figure 6.6. Equations (6.15) to (6.17) shows that these objectives are separable, which means that it is possible to optimize each parameter separately, and find an arm on the Pareto front.

$$f_1(x) = (1 + g(x))x_1x_2 \tag{6.15}$$

$$f_2(x) = (1 + g(x))x_1(1 - x_2) \tag{6.16}$$

$$f_3(x) = (1 + g(x))(1 - x_1) \tag{6.17}$$

$$g(x) = 100(n - 2) + 100\sum_{i=3}^{n}\{(x_i - 0.5)^2 - cos[20\pi(x_i - 0.5)]\} \tag{6.18}$$

Where $x = (x_1, ..., x_n) \in [0, 1]^n$, and we can see that $\sum_{i=1}^{3} f_i = 1$. We will use $n = 10$ in our experiments.

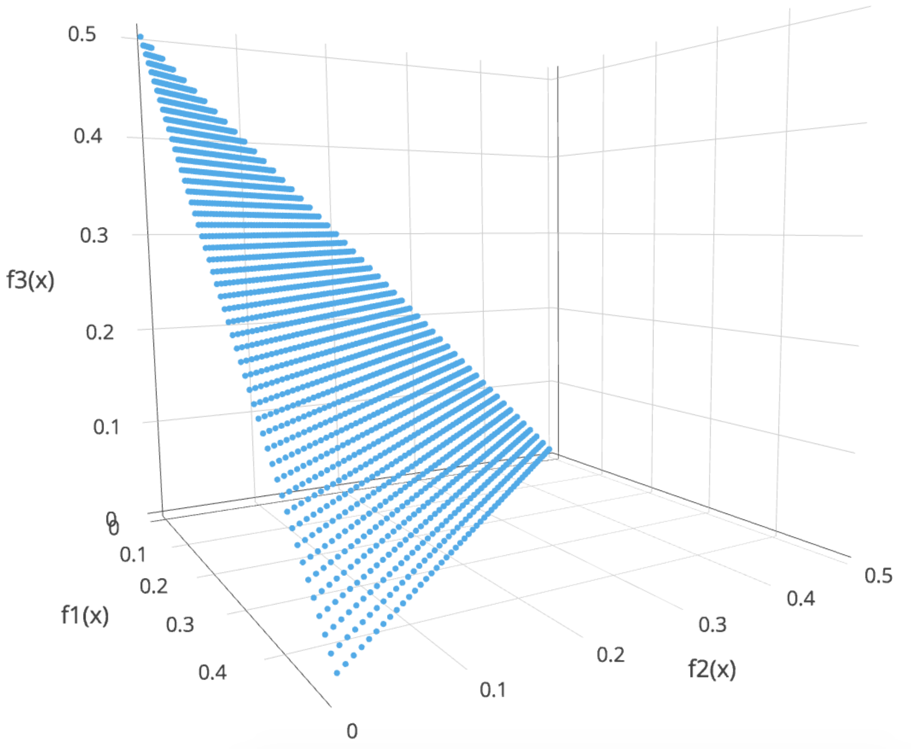

Figure 6.6: 2500 uniformly distributed points on the Pareto front of the DTLZ1 problem, points taken from [13]

DTLZ7

Figure 6.7 shows us that the Pareto front of the DTLZ7 problem is disconnected, and that each part has convex, and non-convex area's in it. Equations (6.19) to (6.21) shows us how we formulate the DTLZ7 problem.

$$f_1(x) = x_1 \tag{6.19}$$

$$f_2(x) = x_2 \tag{6.20}$$

$$f_3(x) = (1 + g(x))h(f_1, f_2, g(x)) \tag{6.21}$$

$$g(x) = 1 + \frac{9}{22}\sum_{i=1}^{n}(x_i) \tag{6.22}$$

$$h(f_1, f_2, g(x)) = 3 - \sum_{i=1}^{2}(\frac{f_i}{(1 + g(x))}(1 + sin(3\pi f_i)) \tag{6.23}$$

Where $x = (x_1, ..., x_n) \in [0, 1]^n$. We will use $n = 10$ in our experiments.

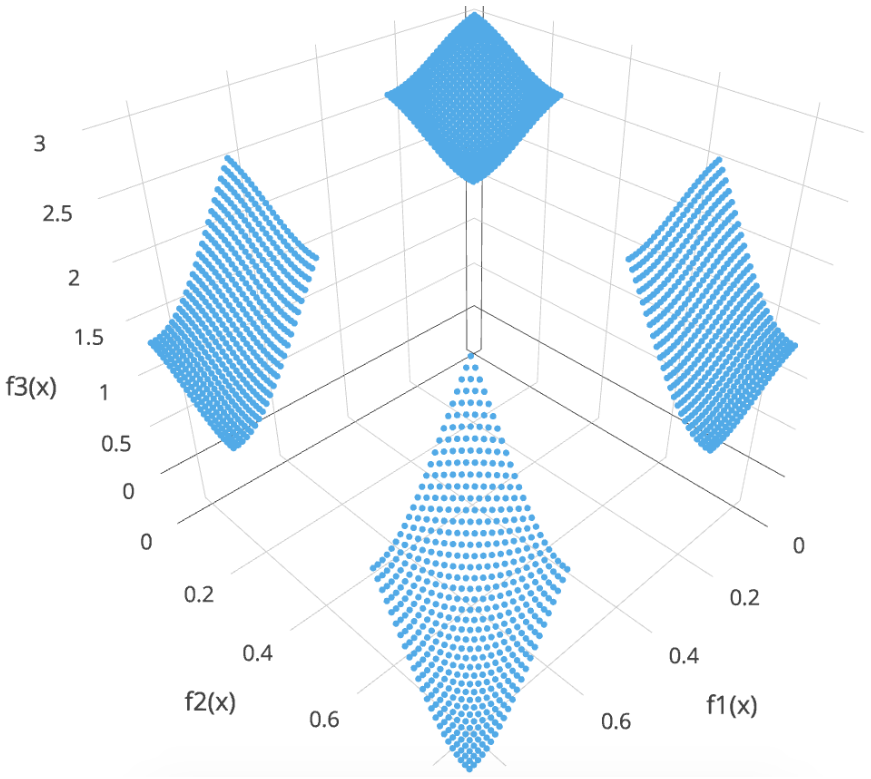

Figure 6.7: 2500 uniformly distributed points on the Pareto front of the DTLZ1 problem, points taken from [13]

6.3 Parameter Tuning

Most of our algorithms do not need any parameters to be tuned, and the ones that do are explained here. We are going in some detail what the exact functionality is of the parameter, and discuss the possible different values for it.

6.3.1 Multi-Objective Evolutionary Algorithm Based on Decomposition for a Multi-Objective χ-Armed Bandit Problem

The MOEA/D algorithm decomposed a problem in to sub-problems and optimized these simultaneously using scalarization techniques. The number of sub-problems that have to be created is chosen by the decision maker. Each sub-problem is associated to a specific weight vector, that reflects the area of the Pareto front the decision maker wants to investigate. The number of sub-problems is not really a parameter that has to be tuned, but the number of neighbors T for each sub-problem is.

Parameter T

The size of the neighborhood is decided by the parameter T, and has a direct influence on the recombination method we use. When searching for a new solution in a sub-problem i, we only use a solution from neighboring sub-problems for recombination. To decide the neighborhood of a sub-problem we solely look at the euclidean distance between their weight vectors. The neighborhood will than consist of the T closest sub-problems, based on that euclidean distance.

Too small of a value for T would have an influence on the exploration of our algorithm, because if the solutions are already close to each other, so will be their child which means that we are not exploring new area's. On the other hand if the value of T is too big, not only would this increase the computational complexity of the algorithm, but it also holds back the search for a wide spread of arms since we replace the arm of a sub-problem in our neighborhood if we found a better one in an other.

We used a population size $pop_{size} = 100$ for the following tests, a maximum number evaluations $eval_{max} = 250$, and vary T in the range of [10, 50], with steps of 10. We tested this on the bi-objective ZDT3 problem explained in Equations (6.7) and (6.8), without any variance. The results given in Table 6.1 are the average results over 100 experiments for each neighborhood size. To grade our results we use the *Cardinality* metric explained in Equation 6.1, and the *IGD* metric explained in Equation 6.2.

Our *PF* has a size of 1000, and is taken from [52]. Table 6.1 contains the results of the average distance to the Pareto front for different neighborhood sizes, while Figure 6.8 shows where they find themselves compared to others in a graph for just one experiment.

Table 6.1 shows us that the computational time increases when our neighborhood size increases. This is was to be expected since we have to calculate the fitness of each solution at each iteration for all sub-problems in our neighborhood. We can see that smaller the neighborhood size, the bigger the set of arms is that we find, and that the distance to the Pareto front is smaller. Figure 6.8 shows us the same results, that smaller neighborhood sizes perform better than bigger ones. We decide to chose a neighborhood size of $T = 20$, since the difference with $T = 10$ is so minor, and we want a bit more exploration.

T	$IGD(sol_{ND}, PF)$ (Std. Dev.)	$Cardinality$	computational time (in ms)
10	0.7863 (0.2071)	10.15	3.92
20	1.0754 (0.7148)	10.4	4.49
30	1.5822 (0.7728)	8.53	5.32
40	1.9007 (1.1157)	8.13	5.44
50	1.9867 (1.0502)	7.87	6.38

Table 6.1: Results for tuning neighborhood size T given in column 1. Column 2 contains the IGD metric from the sol_{ND} to PF, Column 4 is the $Cardinality$ of our set of arms, and column 3 is the computational time.

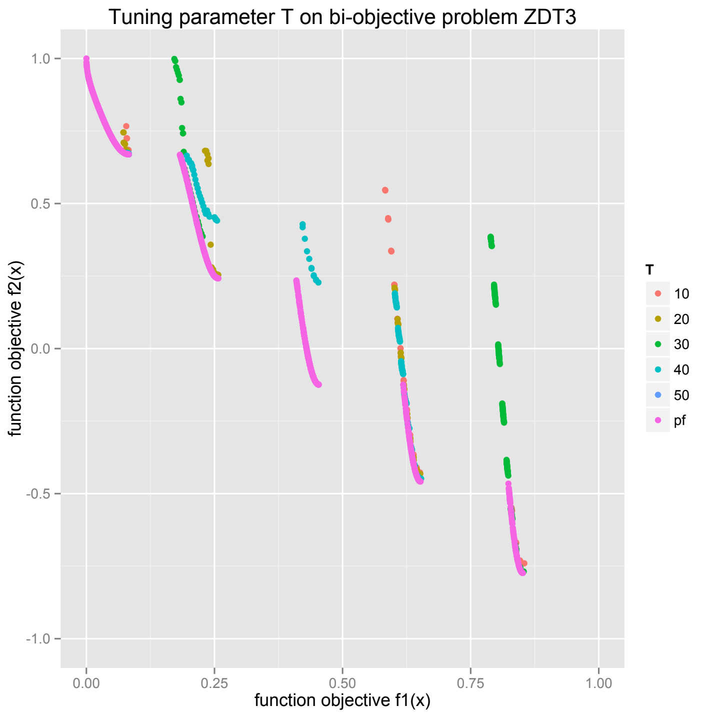

Figure 6.8: Results for the MOEA/D algorithm run on the ZDT3 problem, with a maximum of 250 evaluations, a population size of 100, and varying neighborhood size T.

6.3.2 Multi-Objective Simultaneous Optimistic Optimization (MO-SOO) for a Multi-Objective χ-Armed Bandit Problem

In the multiple MO-SOO algorithms we presented in chapter 5 the solution space is represented as tree, where the root of the tree contains the complete domain. To search for new solutions we split a node along 1 dimension into k new nodes. We will evaluate the influence of k on the performance of our algorithm.

Parameter k

When we decide to split a domain into smaller parts to investigate, the amount of parts is decided by parameter k. It has quite a big influence on the performance of the algorithm, thus it is very important to find a decent value for it.
Default value is $k = 3$, but since we are working with multi-objective reward vectors it might be interesting to see how the performance is for higher values of k. We will perform 100 experiments, on the bi-objective ZDT3 problem with $eval_{max} = 250$, $depth_{max} = 10 * \sqrt{log(eval_{max})^3} = 37$ as advised by [53], and $k \in [2, 10]$. We are going to test the algorithm on a multi-objective problem, but without noise, so we can ignore the $pull_{min}$ parameter. We are not adding noise since this does not have an influence on the performance of the parameter k, and thus would only increase the computational time. The MO-SOO algorithm using Pareto dominance will be used to tune the parameter, and its results will be used for all the MO-SOO algorithms, since we believe that the method of differentiating nodes of the tree does not influence the performance of parameter k.

Table 6.2 shows us that we are more likely to chose a lower value of k since the distance to the Pareto front increases when k increases. k does not seem to have an influence on the computational time, since it does not matter if a

k	$IGD(sol_{ND}, PF)$ (Std. Dev.)	$Cardinality$	computational time (in ms)
2	0.4011 (0.0051)	6.16	0.17
3	0.2912 (0.0032)	7.28	0.2
4	0.3229 (0.0029)	7.65	0.15
5	0.3516 (0.0109)	7.78	0.06
6	0.3591 (0.0155)	10.31	0.25
7	0.3581 (0.011)	6.15	0.11
8	0.3626 (0.0074)	8.0	0.11
9	0.3689 (0.011)	7.0	0.15
10	0.3886 (0.0157)	8.81	0.1

Table 6.2: Results for tuning split size k given in column 1. Column 2 contains the IGD metric from the sol_{ND} to PF, Column 4 is the $Cardinality$ of our set of arms, and column 3 is the computational time.

node will be split 4 time at 1 depth, or 2 times at 2 different depths in the tree. Figure 6.9 shows the results of one experiment, from which we can deduct the same analysis, that the lower the k parameter, the better our results are. We will use the default value for k, in our experiments.

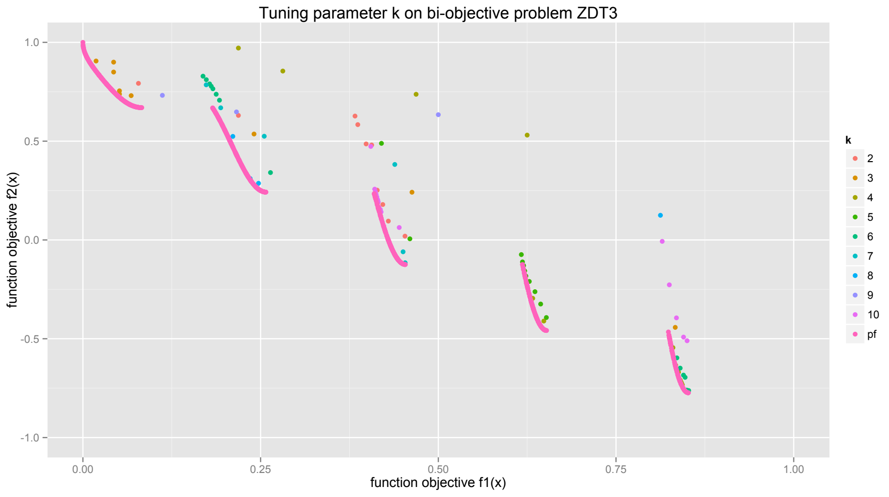

Figure 6.9: Results for the MO-SOO algorithm run on the ZDT3 problem, with a maximum of 250 evaluations, $depth_{max} = 37$, and varying split size k.

6.4 Overview

In this section we will giving all the parameters of the algorithms, and tell how many arm pulls they can perform. Each algorithm will be run for 100 times, and has a maximum of $pulls_{max} = 30.000$ arms it can pull.

The non-dominated sorting genetic algorithm (NSGA-II) [20] will have a population size of 100, which is the only variable we have to initialize before being able to run the algorithm.

The multi-objective evolutionary algorithm based on decomposition (MOEA/D) [59] will have a neighborhood size of 20, and will decompose the problem into 100 sub-problems in case the problem has 2 objectives to optimize, and decompose the problem into 300 sub-problems if the problem has 3 objectives to optimize.

The multi-objective simultaneous optimistic optimization algorithm using Pareto-domination (MO-SOO-PD) will use a split factor $k = 3$, and a maximum depth depending on the number of arm pulls it can perform maximally.

Like the MOEA/D algorithm, the naive multi-objective simultaneous optimistic optimization using scalarization (N-MO-SOOS) algorithm will decompose the problem into 100, or 300 sub-problems, depending on the number of objectives it has to optimize. It will also use $k = 3$ as a split factor.

The improved multi-objective simultaneous optimistic optimization using scalarization (I-MO-SOOS) algorithm has a split factor $k = 3$. The algorithm will keep 100 weight vectors, and a maximum depth for the tree depending on the maximum number of arm pulls it can perform. The number of arm pulls it has to do before splitting a cell also depends on the maximum number of arm it can play. The formula is given as follows:

$$pulls_{min} = min\{30, \frac{pulls_{max}}{log^3(pulls_{max})}\} \tag{6.24}$$

We believe that if we would pull an arm 30 times, we would have a decent approximation to the average expected mean of that arm.

Chapter 7

Results & Discussion

In the previous chapter we explained the experimental setup, and tuned the parameters of the algorithms we will be testing. In this chapter we will be showing the results we have gathered, and discuss them in detail. For reasons of clarification we make a summation of the algorithms we tested.

The non-dominated sorting genetic algorithm (NSGA-II) [20] discussed in Chapter 4 assigns a fitness measure to solutions by using a Pareto ranking approach. We used one-point crossover, and uniform mutation as our recombination, and mutation techniques respectively.

In Chapter 4 we introduced the multi-objective evolutionary algorithm based on decomposition (MOEA/D) [59]. The algorithm decomposes the problem in to sub-problems, and optimizes them simultaneously. We changed the algorithm slightly to be able to handle an multi-objective χ-armed bandit problem.

In Chapter 5 we discussed the simultaneous optimistic optimization algorithm (SOO) [44]. We expanded the algorithm to be able to work with multi-objective optimization problems, and created 3 versions. The first version, the multi-objective simultaneous optimistic optimization algorithm using Pareto-domination (MO-SOO-PD), used Pareto dominance to rank different solutions.

The other two versions use scalarization techniques to give a fitness value to solutions. From these two the first one is called the naive multi-objective simultaneous optimistic optimization using scalarization (N-MO-SOOS) algorithm. It divides the problem into sub-problems, and associates a unique weight vector to each problem. We call it naive because it does not keep track of a complete tree of solutions, like the original SOO, and the MO-SOO algorithm do.

We made the naive version better, and called it the improved multi-objective simultaneous optimistic optimization using scalarization (I-MO-SOOS) algorithm. The improved version still divides the problem into sub-problems, but each weight vector iterates over the same global tree.

For each problem we will be calculating the average *Cardinality*, Inverted Generational Distance (*IGD*) [55], and computational time taken over 100 experiments. From these 100 experiments we kept the results of the experiment with the best *IGD* score, and used it to create a graph as tool for clarification to show how close they are to the actual Pareto front, and how the non-dominated set of solutions given by the algorithms can be compared to each other. All problems have been explained in Chapter 6. To make it easier to analyze the data in the tables, we will be giving bad results a **red** color, and good results a **blue** color.

ZDT2 Problem

Table 7.1 shows us the performance of the algorithms on the ZDT2 problem. Even though the MO-SOO-PD, and I-MO-SOOS algorithm are performing the best, the differences are not that big. The arms returned by NSGA-II might be farther away from the Pareto front compared to those from MOEA/D, but MOEA/D only returns 2.52 solutions on average, which is very bad even though its computational time is one of the best. The algorithms that have the best result, also took the longest, which has to do with the computational complexity of both algorithms. At each iteration, they have to traverse, and expand a tree which takes a while after a couple of iterations. Figure 7.1 shows us why the *IGD* result of I-MO-SOOS is so good. It is not necessarily closer to the Pareto front than MO-SOO-PD, but it has much better spread over the front.

Algorithm	$IGD(sol_{ND}, PF)$ (Std. Dev.)	*Cardinality*	computational time (in ms)
NSGA-II	**2.728 (0.0584)**	5.01	429.9
MOEA/D	1.3938 (0.0208)	**2.52**	**97.8199**
MO-SOO-PD	**0.4211 (0.0062)**	10.86	1644.51
N-MO-SOOS	1.1007 (0.0305)	14.72	**129.78**
I-MO-SOOS	**0.459 (0.0)**	10.0	**3295.07**

Table 7.1: Results on the ZDT2 problem with the Algorithms in Column 1.

ZDT3 Problem

We can see in Table 7.2 the MOEA/D algorithm is outperforming all other algorithms. The returned rewards from the pulled arms might be finding themselves a bit further away from the Pareto front, or not as spread, but MOEA/D makes it up by returning more arms to pull, and calculating this faster than the other algorithms. NSGA-II is again one of the worst, together with the N-MO-SOOS

Algorithm	$IGD(sol_{ND}, PF)$ (St dev)	*Cardinality*	computational time (in ms)
NSGA-II	1.3498 (0.1255)	13.3	410.39
MOEA/D	**0.581 (0.0073)**	**21.57**	**111.97**
MO-SOO-PD	**0.4848 (0.0158)**	7.75	**2030.86**
N-MO-SOOS	1.4932 (0.0254)	14.41	**113.7**
I-MO-SOOS	**0.5388 (0.0)**	9.0	**2076.42**

Table 7.2: Results on the ZDT3 problem with the Algorithms in Column 1.

algorithm. These algorithms seem to have a problem with non-convex Pareto fronts, out of which some parts of the ZDT3 Pareto front consists. The MO-SOO-PD, and I-MO-SOOS algorithm are performing well, but again we see that the computational time is much higher. Figure 7.2 shows us that MO-SOO-PD has some arms that have rewards very close to the Pareto front, but only in one position, compared to the MOEA/D algorithm that seem to have found solutions farther away but more widely spread.

ZDT4 Problem

The performance of NSGA-II, and MOEA/D are horrific, they are clearly not able to handle multimodality, which means in this case searching for multiple local Pareto fronts. MO-SOO-PD, and I-MO-SOOS on the other hand seem to be handling this very well. They found a lot of possible arms to pull, and very close to the Pareto front. The only downside is that these algorithms are a bit slower. Figure 7.3 shows the results in a graph, we have chosen not to show MOEA/D, and NSGA-II since they are performing so bad, which would have a bad effect on the representation of the graph. We have also not added N-MO-SOOS so we can show how good the solutions are from MO-SOO-PD, and I-MO-SOOS. They beautifully align the Pareto front, with the solutions given by I-MO-SOOS spread over the complete Pareto front.

Algorithm	$IGD(sol_{ND}, PF)$ (St dev)	*Cardinality*	computational time (in ms)
NSGA-II	**44.6402 (160.991)**	10.63	405.56
MOEA/D	**33.9177 (139.1093)**	79.48	126.33
MO-SOO-PD	**0.0046 (0.0)**	**285.17**	1095.45
N-MO-SOOS	1.3676 (0.0316)	14.63	115.47
I-MO-SOOS	**0.0045 (0.0)**	81.0	**2202.6**

Table 7.3: Results on the ZDT4 problem with the Algorithms in Column 1.

ZDT6 Problem

The ZDT6 problem had a non-convex Pareto front, with the solutions distributed very non-uniform, which can be seen when we look at the Pareto front give in Chapter 6. The results are given in Table 7.4. The MO-SOO-PD, and I-MO-SOOS algorithm are clearly outperforming the other algorithms, with I-MO-SOOS returning a lot of arms where the decision maker can choose from. The computational time of I-MO-SOOS is still a problem, but it seems like a decent trade-off in this case. NSGA-II, and MOEA/D are performing the worst again, which is confirmed by Figure 7.4.

DTLZ1 Problem

The DTLZ1 is the first multi-objective problem with 3 objectives, Table 7.5 shows us the results. We can see that NSGA-II, and MOEA/D are performing

Algorithm	$IGD(sol_{ND}, PF)$ (St dev)	*Cardinality*	computational time (in ms)
NSGA-II	**5.5871 (0.139)**	**9.21**	470.85
MOEA/D	**3.1636 (0.0626)**	10.94	104.0699
MO-SOO-PD	**0.3685 (0.0035)**	34.93	1051.6
N-MO-SOOS	1.1131 (0.0329)	14.84	**115.64**
I-MO-SOOS	**0.4851 (0.0)**	**302.02**	**1806.39**

Table 7.4: Results on the ZDT6 problem with the Algorithms in Column 1.

by far the worst, probably due to the Pareto front being non-convex. Even though MOEA/D returns the most solutions, this does not matter since the solutions are all far away from the Pareto front. The best algorithm for this problem would be the MO-SOO-PD algorithm, it has a great result on the *IGD*, and the *Cardinality* metric. I-MO-SOOS also has a great *IGD* result, but the computational time clearly increases when there are more objectives. This is a problem that we might want to look at in the future. We did not include figures for the problems with 3 objectives because these were very cluttered, and unclear.

Algorithm	$IGD(sol_{ND}, PF)$ (St dev)	*Cardinality*	computational time (in ms)
NSGA-II	**59.5493 (714.9938)**	31.16	336.7
MOEA/D	**53.6748 (416.5787)**	**303.52**	279.43
MO-SOO-PD	**0.3214 (0.0)**	**272.98**	1363.3
N-MO-SOOS	1.5188 (0.0086)	**6.87**	140.44
I-MO-SOOS	**0.3115 (0.0)**	59.0	**5995.86**

Table 7.5: Results on the DTLZ1 problem with the Algorithms in Column 1.

DTLZ7 Problem

Table 7.6 shows us that I-MO-SOOS is by far the worst algorithm, with an immensely high computational time. The same goes for MO-SOO-PD, it seems that the algorithms that keep track of a tree are performing much worse. The MOEA/D algorithm has a good performance, just as in the ZDT3 problem, which was also a problem with a disconnected Pareto front. The best *IGD*

Algorithm	$IGD(sol_{ND}, PF)$ (St dev)	*Cardinality*	computational time (in ms)
NSGA-II	1.9119 (0.3064)	**1.48**	356.4
MOEA/D	1.8352 (0.1392)	**2254.52**	1042.02
MO-SOO-PD	**2.5254 (0.0589)**	88.19	863.84
N-MO-SOOS	**1.1698 (0.0102)**	7.0	**121.63**
I-MO-SOOS	**2.799 (0.0)**	**2.0**	**18365.41**

Table 7.6: Results on the DTLZ7 problem with the Algorithms in Column 1.

score is given by N-MO-SOOS, but unfortunately it barely has any solutions to return to the decision maker, which is why we would not advice to use it. We

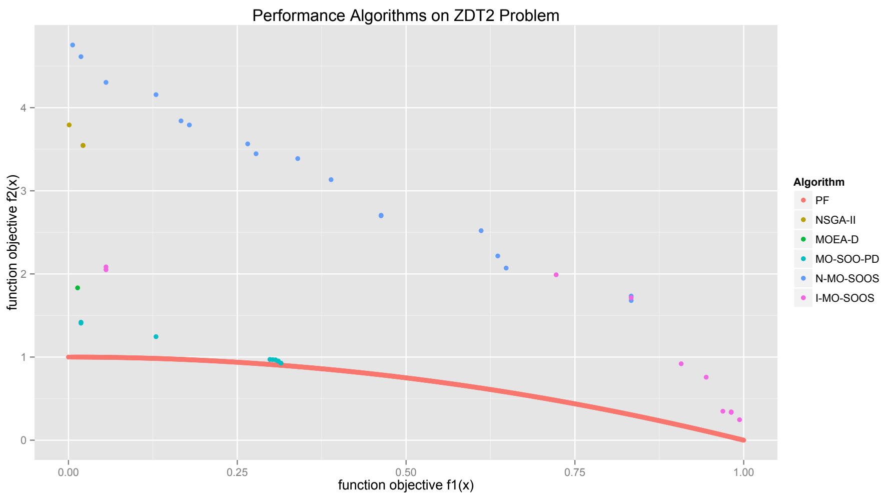

Figure 7.1: Performance of the best non-dominated set of solutions from all algorithms on the ZDT2 problem.

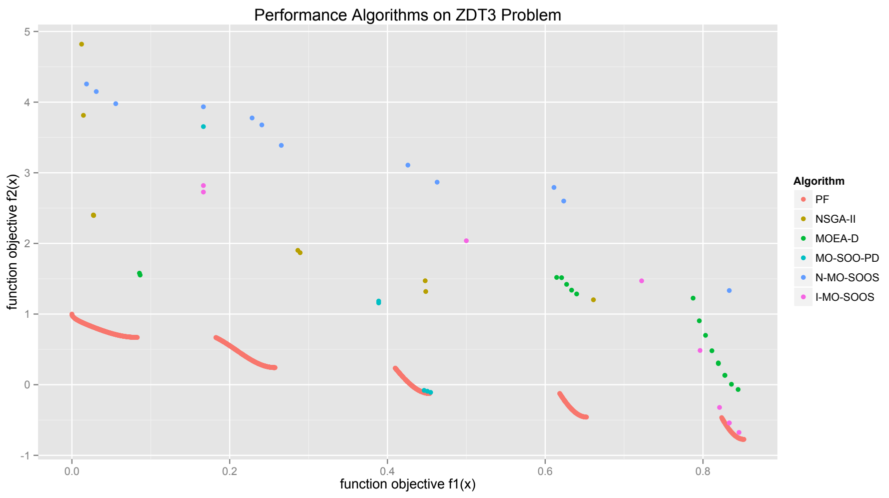

Figure 7.2: Performance of the best non-dominated set of solutions from all algorithms on the ZDT3 problem.

have seen the performance of 5 algorithms, and were able to draw conclusions from the results about their characteristics. The MO-SOO-PD, and I-MO-SOOS algorithm seem to perform very well on problems where the Pareto front is non-convex, or when the Pareto front consists out of multiple local Pareto fronts.

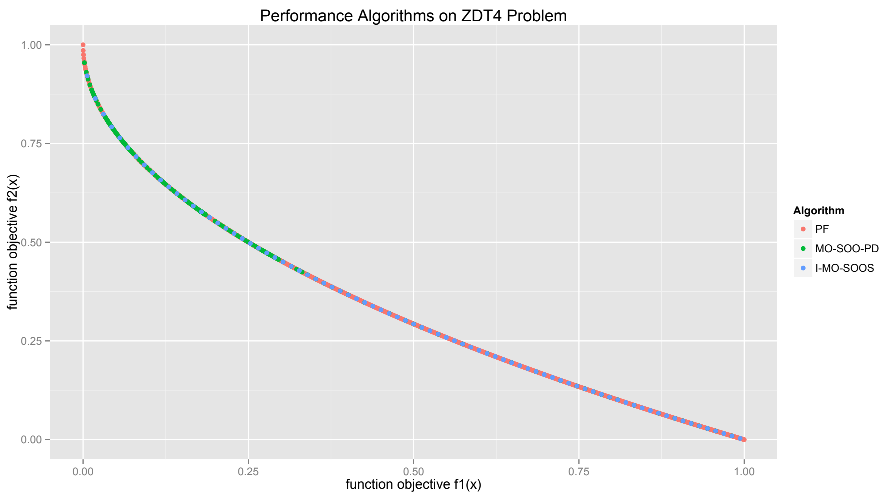

Figure 7.3: Performance of the best non-dominated set of solutions from all algorithms on the ZDT4 problem.

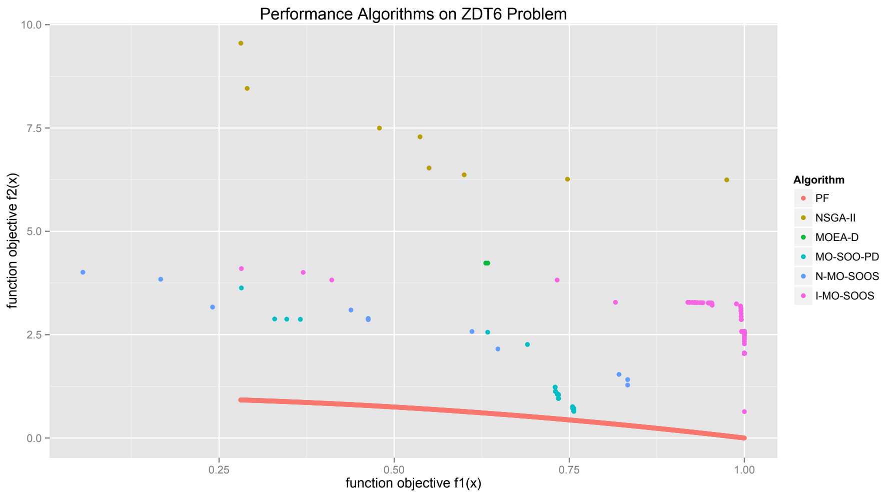

Figure 7.4: Performance of the best non-dominated set of solutions from all algorithms on the ZDT4 problem.

The NSGA-II, and MOEA/D algorithms are performing very bad on problems with a non-convex Pareto front. The MOEA/D does have the benefit that it performs rather well on problems with a disconnected, non-continuous Pareto front. When the number of objectives increase so does the computational time for the algorithms that keep track of a tree, which they have to traverse from top to bottom at each iteration. The reader might have noticed that the I-

MO-SOOS algorithm seems to have a standard deviation of 0, while the results are susceptible to noise. This has to do with the algorithm playing each arm 30 times, before going to the next solution, which gives us a result that is a good approximation to the average expected mean reward of that arm. The N-MO-SOOS algorithm never excels in a problem, but is also never the worst performer.

Chapter 8

Conclusion & Future Work

This master's thesis started by explaining the basics of reinforcement learning, the multi-armed bandit problem, and evolutionary algorithms. Evolutionary algorithms were discussed in detail in Chapter 2 because it served as the building blocks for the multi-objective evolutionary algorithm based on decomposition (MOEA/D) [59], and the non-dominated sorting genetic algorithm II (NSGA-II) [20] that were introduced in Chapter 4. The main problem of this thesis, the multi-objective χ-armed bandit problem was explained, and formalized in Chapter 3. We introduced the multi-objective simultaneous optimistic optimization algorithm using Pareto-domination (MO-SOO-PD), the naive multi-objective simultaneous optimistic optimization algorithm using scalarization (N-MO-SOOS), and the improved multi-objective simultaneous optimistic optimization algorithm using scalarization (I-MO-SOOS) in Chapter 5.

In the previous chapter the performance of these algorithms were discussed, and it demonstrated that some did a better job than others. The conclusions drawn from their performance were based on the experimental setup we explained in Chapter 6. It was shown that the MOEA/D, and NSGA-II algorithm did not work well with non-convex Pareto fronts when the returned rewards are susceptible to noise. It might be interesting to see how they would perform if they take in regard that there is noise in the data they are handling, which is not the case at the moment. These algorithms were created for multi-objective optimization, which does not have any noise in the problem setup.

The N-MO-SOOS algorithm never excelled in any problem, but always showed to be one of the algorithms with the lowest computational time. Unfortunately the set of arms the algorithm returned were always too far away, and not properly spread over the Pareto front to take serious, thus had IGD score that was too high. Its average performance was very consistent though. This was to be expected which was why the I-MO-SOOS algorithm was created. Its performance was much better, and except for the DTLZ7 problem [21] it never encountered any difficulties solving the problem, and was one of the algorithms that performed best. The computational complexity did seem to be an issue, even more so when the number of objectives increased. This had to do with the algorithm having to traverse a tree from top to bottom at each iteration, so it might be interesting to overcome this problem by making some alterations to the algorithm. For example, instead of checking at each depth, which nodes do not have any children, keep track of the depths where there are still nodes without children. This will increase the space required for the algorithm to work, but it will decrease the computational time. The same goes for the MO-SOO-PD algorithm. It was shown that it performed very well, most of the times even one of the best algorithms to use, but unfortunately the time it took to perform the arm pulls was too high.

In the MOEA/D, N-MO-SOOS, and I-MO-SOOS algorithm we used the Tchebycheff approach [38] as a scalarization technique, to translate the reward vector to a single scalar value. It might be worth looking at the performance of other known methods, like normal-boundary intersection [16], to see if the algorithm would return a better set of non-dominated arms to play.

Bibliography

[1] Rajeev Agrawal. The continuum-armed bandit problem. *SIAM J. Control Optim.*, 33(6):1926–1951, November 1995.

[2] Timothy W. Athan and Panos Y. Papalambros. A note on weighted criteria methods for compromise solutions in multi-objective optimization. *Engineering Optimization*, 27(2):155–176, September 1996.

[3] Jean-Yves Audibert and Sébastien Bubeck. Best Arm Identification in Multi-Armed Bandits. In *COLT - 23th Conference on Learning Theory - 2010*, page 13 p., Haifa, Israel, June 2010.

[4] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, May 2002.

[5] Peter Auer, Ronald Ortner, and Csaba Szepesvári. Improved rates for the stochastic continuum-armed bandit problem. In *In 20th Conference on Learning Theory (COLT*, pages 454–468, 2007.

[6] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, Oxford, UK, 1996.

[7] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation.* IOP Publishing Ltd., Bristol, UK, UK, 1st edition, 1997.

[8] Gerardo Beni. From swarm intelligence to swarm robotics. In *Proceedings of the 2004 International Conference on Swarm Robotics*, SAB'04, pages 1–9, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] Gerardo Beni and Jing Wang. Swarm intelligence in cellular robotic systems. In Paolo Dario, Giulio Sandini, and Patrick Aebischer, editors, *Robots and Biological Systems: Towards a New Bionics?*, volume 102 of *NATO ASI Series*, pages 703–712. Springer Berlin Heidelberg, 1993.

[10] Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in multi-armed bandits problems. In Ricard Gavaldà, Gábor Lugosi, Thomas

Zeugmann, and Sandra Zilles, editors, *Algorithmic Learning Theory*, volume 5809 of *Lecture Notes in Computer Science*, pages 23–37. Springer Berlin Heidelberg, 2009.

[11] Sébastien Bubeck, Gilles Stoltz, Csaba Szepesvári, and Rémi Munos. Online optimization in x-armed bandits. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 201–208. Curran Associates, Inc., 2009.

[12] Carlos A. Coello Coello, Gary B. Lamont, and David A. Van Veldhuizen. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[13] Carlos Artemio Coello. Sample pareto fronts. Digital, June 2015.

[14] E. Cope. Regret and convergence bounds for immediate-reward reinforcement learning with continuous action spaces. *IEEE Transactions on Automatic Control*, 54(6):1243–1253, 2009.

[15] Charles Darwin. *On the Origin of Species by Means of Natural Selection.* Murray, London, 1859. or the Preservation of Favored Races in the Struggle for Life.

[16] Indraneel Das and J. E. Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM J. on Optimization*, 8(3):631–657, March 1998.

[17] Kalyanmoy Deb. Multi-objective genetic algorithms: Problem difficulties and construction of test problems. *Evolutionary Computation*, 7:205–230, 1999.

[18] Kalyanmoy Deb and Mayank Goyal. A combined genetic adaptive search (geneas) for engineering design. *Computer Science and Informatics*, 26:30–45, 1996.

[19] Kalyanmoy Deb and Deb Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[20] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.

[21] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. Scalable test problems for evolutionary multi-objective optimization. Technical report, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH, 2001.

[22] M.M. Drugan and A. Nowe. Designing multi-objective multi-armed bandits algorithms: A study. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–8, Aug 2013.

[23] G. Eichfelder. *Adaptive Scalarization Methods in Multiobjective Optimization.* Vector Optimization. Springer, 2008.

[24] Carlos M. Fonseca and Peter J. Fleming. Genetic algorithms for multiobjective optimization: Formulationdiscussion and generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416–423, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[25] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[26] P. Hajela and C.-Y. Lin. Genetic search strategies in multicriterion optimal design. *Structural optimization*, 4(2):99–107, 1992.

[27] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, MA, USA, 1992.

[28] Simon Hub, Luigi Barone, Lyndon While, and Phil Hingston. A scalable multi-objective test problem toolkit, 2005.

[29] S. Huband, P. Hingston, L. Barone, and L. While. A review of multiobjective test problems and a scalable test problem toolkit. *Trans. Evol. Comp*, 10(5):477–506, October 2006.

[30] Robert Kleinberg. Nearly tight bounds for the continuum-armed bandit problem. In *Advances in Neural Information Processing Systems 17*, pages 697–704. MIT Press, 2005.

[31] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, September 2006.

[32] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[33] Volodymyr Kuleshov and Doina Precup. Algorithms for multi-armed bandit problems. *arXiv preprint arXiv:1402.6028*, 2014.

[34] Sridhar Mahadevan. Optimality criteria in reinforcement learning. In *In AAAI Fall Symposium on Learning Complex Behaviors for Intelligent Adaptive Systems*, 1996.

[35] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, April 2004.

[36] R.Timothy Marler and JasbirS. Arora. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, 41(6):853–862, 2010.

[37] A. Messac, A. Ismail-Yahaya, and C. A. Mattson. The normalized normal constraint method for generating the Pareto frontier. *Structural and Multidisciplinary Optimization*, 25(2):86–98, July 2003.

[38] K. Miettinen. *Nonlinear multiobjective optimization.* Kluwer Academic Publishers, Boston, 1999.

[39] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[40] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1998.

[41] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm – i. continuous parameter optimization. *EVOLUTIONARY COMPUTATION*, 1:25–49, 1993.

[42] Rémi Munos. Optimistic optimization of deterministic functions without the knowledge of its smoothness. In *Advances in Neural Information Processing Systems*, page ?, Spain, 2011.

[43] Tatsuya Okabe, Yaochu Jin, Markus Olhofer, and Bernhard Sendhoff. On test functions for evolutionary multi-objective optimization. In *Parallel Problem Solving from Nature - PPSN VIII, 8th International Conference, Birmingham, UK, September 18-22, 2004, Proceedings*, pages 792–802, 2004.

[44] P. Preux, R. Munos, and M. Valko. Bandits attack function optimization. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 2245–2252, July 2014.

[45] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematics Society*, 58:527–535, 1952.

[46] Günter Rudolph. Evolutionary search under partially ordered fitness sets. In *IN PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON INFORMATION SCIENCE INNOVATIONS IN ENGINEERING OF NATURAL AND ARTIFICIAL INTELLIGENT SYSTEMS (ISI 2001*, pages 818–822. ICSC Academic Press, 2001.

[47] N. Srinivas and Kalyanmoy Deb. Multi-objective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, September 1994.

[48] W. Stadler. Caveats and boons of multicriteria optimization. *Computer-Aided Civil and Infrastructure Engineering*, 10(4):291–299, 1995.

[49] Tim Stefanini. The genetic coding of behavioral attributes in cellular automata. In John R. Koza, editor, *Artificial Life at Stanford 1994*, pages 172–180. Stanford Bookstore, Stanford, California, 94305-3079 USA, June 1994.

[50] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[51] Kay Chen Tan, Eik Fun Khor, and Tong Heng Lee. *Multiobjective Evolutionary Algorithms and Applications.* Springer-Verlag London, 2005.

[52] Tik.ee.ethz.ch. Eth - sop - downloads/material - supplementary material - testproblems, 2015.

[53] Michal Valko, Alexandra Carpentier, and Rémi Munos. Stochastic simultaneous optimistic optimization. In Sanjoy Dasgupta and David Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 19–27. JMLR Workshop and Conference Proceedings, May 2013.

[54] K. Van Moffaert, K. Van Vaerenbergh, P. Vrancx, and A. Nowe. Multi-objective x-armed bandits. In *Neural Networks (IJCNN), 2014 International Joint Conference on*, pages 2331–2338, July 2014.

[55] David A. Van Veldhuizen and Gary B. Lamont. Multiobjective evolutionary algorithm research: A history and analysis, 1998.

[56] David A. Van Veldhuizen and Gary B. Lamont. Multiobjective evolutionary algorithm test suites, 1999.

[57] Joannès Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *In European Conference on Machine Learning*, pages 437–448. Springer, 2005.

[58] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, UK, May 1989.

[59] Qingfu Zhang and Hui Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *Evolutionary Computation, IEEE Transactions on*, 11(6):712–731, Dec 2007.

[60] 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.

[61] E. Zitzler, M. Laumanns, and S. Bleuler. A Tutorial on Evolutionary Multiobjective Optimization. In X. Gandibleux et al., editors, *Metaheuristics for Multiobjective Optimisation*, Lecture Notes in Economics and Mathematical Systems. Springer, 2004.

[62] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *Evolutionary Computation, IEEE Transactions on*, 3(4):257–271, Nov 1999.

[63] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. Grunert da Fonseca. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.

[64] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evol. Comput.*, 8(2):173–195, June 2000.