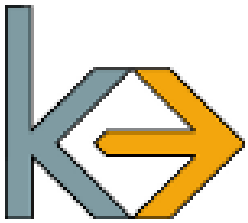
Global Optimization using Monte Carlo Tree Search in discrete State Lattices

Globale Optimierung mit Monte Carlo Tree Search in diskreten Zustandsräumen
Master-Thesis von Simon Schimmels
Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Markus Zopf

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Global Optimization using Monte Carlo Tree Search in discrete State Lattices
Globale Optimierung mit Monte Carlo Tree Search in diskreten Zustandsräumen

Vorgelegte Master-Thesis von Simon Schimmels

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Markus Zopf

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 2. Oktober 2017

(Simon Schimmels)

Danksagung

Zunächst möchte ich mich an dieser Stelle bei Herrn Prof. Dr. Johannes Fürnkranz für die Betreuung meiner Masterarbeit bedanken.

Daneben gilt mein Dank Matthias und Mario, die beide für mich meine Arbeit Korrektur gelesen haben.

Weiterhin danke ich meinen Eltern für die stetige Unterstützung während des gesamten Studiums.

Insbesondere gilt mein Dank meiner Freundin Hannah, die mich während dieser Arbeit und des kompletten Studiums immer motiviert und unterstützt hat.

Abstract

Global Optimization is a highly researched area with applications in different scientific fields. The goal to find the overall best solution to a problem has been tried to solve with many different approaches. Two of the more popular algorithms are the Genetic Algorithms and the Monte Carlo methods which both have the idea of using random numbers to find global optima. In this thesis, a new Global Optimization algorithm is proposed that is influenced to an extent by the two latter methods, among others.

This new method is at its core a modified variant of the Monte Carlo Tree Search, which is usually applied in the gaming domain. The idea of the algorithm is that it operates on a binary subset lattice. All elements of this subset lattice are encoded as binary strings that can be viewed as discrete states in the solution space. This form of states is based on the idea of the state representation in Genetic Algorithms. The name of the algorithm can be justified by means of this short description: *(binary) Subset Lattice Monte Carlo Tree Optimization*.
Furthermore, two enhancements of the base version of the algorithm are presented. Firstly, the UCD algorithm is integrated which provides a mathematical framework in order to make the most use of transpositions. Secondly, the RAVE algorithm is added. This AMAF-technique has the task to spread rewards in more regions of the lattice. After careful research, it can be assumed that the UCD algorithm is used for the first time in combination with RAVE.

The performance of the algorithm is first tested on artificial benchmark functions by comparing it with a Genetic Algorithm. Additionally, the performance is investigated on the more complex Lennard-Jones Atomic Clusters problem. The results of these experiments indicate that the Subset Lattice Monte Carlo Tree Optimization is able to find optima in particularly large solution spaces. In case of such solution spaces, it outperformed the Genetic Algorithm. In contrast to that, the newly proposed algorithm has difficulties to converge to global optima at even simple test functions. In these cases, the algorithm only improved its solution up to a certain point. These results lead to suggestions of how this first version of a new optimization algorithm can be enhanced in the future.

Zusammenfassung

Globale Optimierung ist ein stark erforschtes Gebiet mit Anwendungen in verschiedenen wissenschaftlichen Bereichen. In der Vergangenheit wurde versucht, die bestmöglichste Lösung für ein Problem durch viele verschiedenen Ansätze zu erreichen. Zwei der bekanntesten Algorithmen sind die Genetischen Algorithmen und die Monte-Carlo-Methoden, die beide die Idee haben, mit Zufallszahlen globale Optima zu finden. In dieser Thesis wird ein neuer Globaler Optimierungs-Algorithmus vorgestellt, der bis zu einem gewissen Maß unter anderem von den beiden letztgenannten Methoden beeinflusst wird.

Diese neue Methode ist im Kern eine modifizierte Variante der Monte-Carlo Tree Search, die üblicherweise im Gaming-Bereich eingesetzt wird. Die Idee des Algorithmus besteht darin, dass er auf einem binären Teilmengenverband arbeitet. Alle Elemente dieses Teilmengenverbandes sind als binäre Strings kodiert, die als diskrete Zustände im Lösungsraum betrachtet werden können. Diese Zustandsform basiert auf der Idee wie Zustände in Genetischen Algorithmen repräsentiert werden. Der Name des Algorithmus kann anhand dieser kurzen Beschreibung begründet werden: *(binäre) Subset Lattice Monte Carlo Tree Optimization*.
Darüber hinaus werden zwei Erweiterungen der Basisversion des Algorithmus vorgestellt. Um Transpositionen optimal zu nutzen, wird zum einen der UCD-Algorithmus integriert. Dieser liefert den benötigten mathematischen Rahmen. Zum anderen wird der RAVE-Algorithmus hinzugefügt, bei der es sich um eine AMAF-Technik handelt, die die Aufgabe hat Belohnungen in mehr Regionen des Verbandes zu verteilen. Nach sorgfältiger Recherche kann davon ausgegangen werden, dass der UCD-Algorithmus erstmals in Kombination mit RAVE verwendet wird.

Die Leistung des Algorithmus wird zunächst an künstlichen Benchmark-Funktionen getestet, indem er mit einem Genetischen Algorithmus verglichen wird. Darüber hinaus findet eine Untersuchung der Leistung auf dem komplexeren Lennard-Jones Atomic Clusters-Problem statt. Die Ergebnisse dieser Experimente zeigen, dass die Subset Lattice Monte Carlo Tree Optimization in besonders großen Lösungsräumen Optima finden kann. Bei solchen Lösungsräumen übertraf dieser den Genetischen Algorithmus. Im Gegensatz dazu, hat der neu vorgeschlagene Algorithmus Schwierigkeiten, selbst bei einfachen Testfunktionen zu einem globalen Optimum zu konvergieren. In solchen Fällen hat der Algorithmus eine Lösung nur bis zu einem gewissen Punkt verbessert. Die Ergebnisse führen zu Vorschlägen, wie in Zukunft die nächste Version des hier entwickelten Optimierungsalgorithmus verbessert werden kann.

Contents

List of Tables

List of Figures

List of Abbreviations

AMAF	All-Moves-As-First
DAG	Directed Acyclic Graph
GA	Genetic Algorithms
GO	Global Optimization
MCTS	Monte Carlo Tree Search
Poset	Partially ordered set
RAVE	Rapid Value Estimation
SLO	Subset Lattice (Monte Carlo Tree) Optimization
TSP	Traveling Salesman Problem
UCB	Upper confidence bound
UCD	Upper confidence Bound for Directed Acyclic Graphs
UCT	Upper Confidence Bound for Trees

1 Introduction

At the beginning of this chapter, an introduction to the topic is given which will be dealt with in the course of this thesis. In the second section, the problem is stated and the objectives for solving the problem are presented. The last section gives an overview of the structure of this work.

1.1 Motivation

Searching is one of the biggest aspects of artificial intelligence. Usually, this means finding a solution to a problem. In most cases the solution space, where the search is executed, can be abstracted as a graph of states. The search for the best solutions in any given problem has been the topic of research for many decades. Many scientific fields, like chemistry, mathematics or engineering, dedicate to create new and better methods to find an optimum. If the search tries to find the overall maximum or minimum in the whole search space, this type of method is called *Global Optimization*. An interesting example is the Lennard-Jones cluster problem [Fan02], where the coordinates of a cluster of atoms are searched for, so that the potential energy between the atoms is minimized. In the area of Global Optimization there exist many different methods. In the course of this thesis multiple types and sub categories of those will be covered.

Global Optimization methods often mimic natural behaviors. For instance, the Genetic Algorithms are based on the idea of evolution. Like in the biological field, the algorithm consists of chromosomes and genes that mutate and cross with other genes to find the optimal solution. Genetic Algorithms are particularly suitable for example on problems with many local optima or if the solution space is complex, large and hard to understand [SD07]. This leads to various applications of this type of algorithm, for instance in design problems like the optimization of airplane design in [Mar03].

Another important part of this work is the Monte Carlo Tree Search. This algorithm, which is related to the Monte Carlo optimization method, became popular in the gaming domain. In particular, the algorithm was very successful in the ancient Chinese game Go [Sil+16]. In addition to the quoted source, there are numerous other research works that have been conducted in this field. The most important one is the UCT algorithm which tries to solve the exploitation vs. exploration dilemma. The number of problems where MCTS has not been applied to games is limited.

If a Global Optimization method utilizes both Genetic Algorithms and Monte Carlo Tree Search, it will take advantages of both algorithms and unify them into a new approach. Such a system is proposed by this thesis.

1.2 Goal

The goal of this thesis is to create a new Global Optimization algorithm, that is based on Monte Carlo Tree Search and searches for the optimum in a state space of binary strings that are based on the chromosomes of Genetic Algorithms.

The algorithm should be based on well-defined mathematical explanations. First, the solution space, in which the algorithm operates, should be constructed as a lattice with the subset relation of an arbitrary powerset. With the help of a binary encoding, this leads to a state space that consists of chromosomes, such as the analogous elements of Genetic Algorithms. Secondly, the Monte Carlo Tree Search will be

adapted to work in this state space with the goal to find the global optimum, instead of suggesting the next best action in a game. Due to the binary lattice structure of the solution space, the algorithm should be improved in order to achieve better performance. Such enhancements will be investigated and eventually included into the algorithm.

In order to verify the performance of the proposed algorithm, it will be tested against a Genetic Algorithm. Although the algorithms are different, the problems they try to solve are the same. A Genetic Algorithm will be selected as a baseline algorithm, due to the fact that both algorithms share the same binary encoding, leading to a more significant comparison. These benchmarks will be executed in form of experiments on artificial test functions which allow conclusions under "laboratory conditions". Afterwards, the algorithm will be tested on a real-world problem in order to determine how it will perform in such scenarios.

1.3 Structure of the thesis

In chapter 2, fundamentals in Global Optimization and Genetic Algorithms are addressed that are necessary for further understanding of this work. Subsequently, the Monte Carlo Tree Search is presented in detail. The fourth chapter introduces the proposed Global Optimization algorithm of this thesis, from its base version to additional enhancements. In the next chapter, related work from previous research, that is related to this thesis, will be pointed out. To compare the proposed algorithm with the Genetic Algorithm, the experiments are carried out on the test functions in the following chapter. The seventh chapter analyzes and tests these algorithms on a real-world problem. Finally, the thesis will be completed with a conclusion and an outlook on future work.

2 Global Optimization and Genetic Algorithms

In this chapter, basic knowledge necessary for the further course of this thesis will be presented. First, an outline over Global Optimization is presented, the field in which a new approach is pursued. In addition to that, an introduction to Genetic Algorithms is given, a popular subfield of Global Optimization, which is used in this work.

2.1 Global Optimization

The goal of Global Optimization is to find the overall best solution to a given problem. In many cases, there is no structural information about the search space. With the combination of many local extrema, solving of problem is hard. This is the reason why most of the algorithms approximate to the best solution.

In general, Global Optimization tries to find the global extremum in a function $f : X \mapsto T$, where X is the solution space and T is the target set. In this function, called the *objective function*, X is any ordered set (usually a subset of $\mathbb{R}^n$), whereas T is a subset of $\mathbb{R}$ in most cases. The optimization process is defined as finding the minimum value for f:

$$\arg\min_x f(\boldsymbol{x})$$

where $\boldsymbol{x} = (x_1, x_2, ..., x_n)^T \in \mathbb{R}^n$ and $f(\boldsymbol{x}) \in \mathbb{R}$. The result of the optimization is the solution $\boldsymbol{x}^* = \arg\min_x f(\boldsymbol{x})$ for which the following equation holds:

$$\forall \boldsymbol{x} \in \mathbb{R}^n : f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$$

This shows that the optimization is here defined to search for the global minimum. If a maximum is to be found, the objective function can be negated. The search for the maximum can be seen as finding the minimum value for the negated objective function, or mathematically: $\arg\max_x f(\boldsymbol{x}) = \arg\min_x -f(\boldsymbol{x})$.

Besides the global extrema, a function can also have local extrema. Such a local optimum can be defined as the minimum or maximum within a neighborhood: Given a function $f : X \mapsto T$, $\boldsymbol{x}^*$ is a local minimum if $\exists \epsilon > 0, \forall \boldsymbol{x} \in X : f(\boldsymbol{x}^*) \leq f(\boldsymbol{x})$ and $|\boldsymbol{x}^* - \boldsymbol{x}| < \epsilon$. In this formula $|\boldsymbol{x}^* - \boldsymbol{x}|$ represents the distance between the point $\boldsymbol{x}$ and the optimum $\boldsymbol{x}^*$. A local maximum can be defined analogously. A function has the property multimodal if it has multiple (more than one) local extrema.

This formulation is specific for so called *unconstrained* optimization problems. The function can have general constraints, which modify it to a *constrained* problem. In this thesis, only bounds are considered, which are a very specific type of constraints. The topic of constraints itself is complex and would be outside the scope of this work. Given a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ and two bounds $\boldsymbol{x}_L$ and $\boldsymbol{x}_U \in \mathbb{R}^n$, the solution space is limited by the lower and upper bound, such that $\forall \boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{x}_L \leq \boldsymbol{x} \leq \boldsymbol{x}_U$.

Global Optimization can be clarified by means of Figure 2.1, illustrating the search for the global maximum. The x-axis represents the state space which is an example solution space. The value of the objective function to a given point in the solution space is mapped on the y-axis. In addition to the global maximum, two further local maxima are available in the figure. This simple one-dimensional example shows some of the hurdles when trying to find the global optimum.

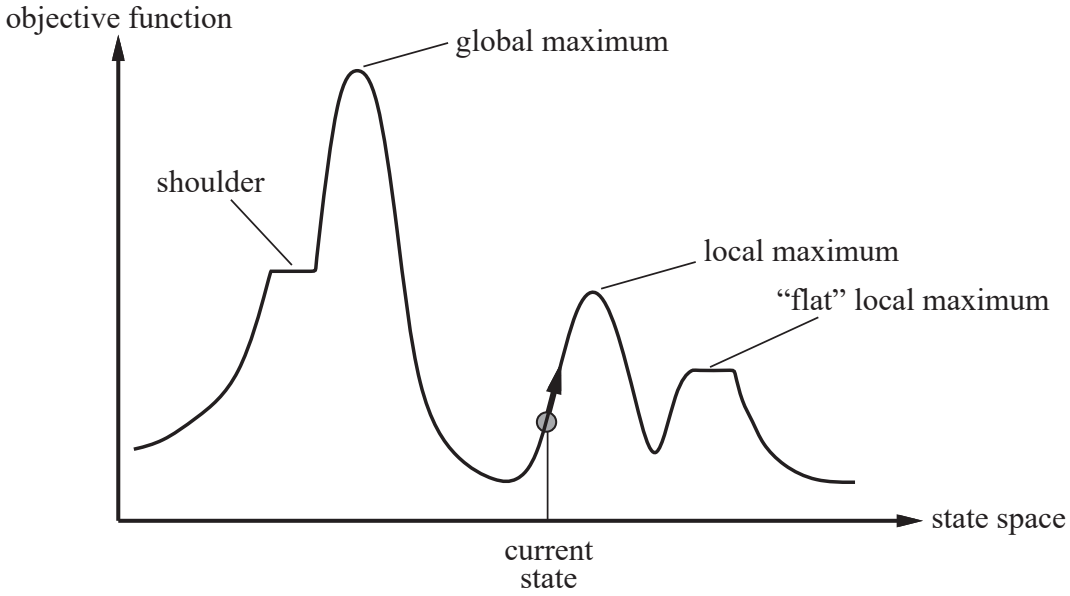


Figure 2.1: An example objective function in the one dimensional solution space (here called state space). (Source: [RN09, p. 121]).

In most cases, the functions are not as simple as in Figure 2.1. The search for the global optimum can be very hard. Among the difficulties according to [DP98] are:

- **Number of dimensions:** Many functions, either test functions or real-world problems, have more than one dimension, even more than ten or a hundred. The size of the solution space grows with the number of dimensions. An optimization algorithm has to be able to deal with that.
- **Multimodal functions:** When a function has many local optima, it is called multimodal. Finding the global optimum gets harder if more local optima exist. It may get even more difficult if there are many local optima in the neighborhood of the global optimum. In addition to that, the shape of the global optima can provide challenges, whether, for example, the neighborhood of a minimum is strongly or weakly descending.
- **Sensitivity:** Some algorithms may tend to optimize only one or a few of all the given dimensions. The difficulty is to find the global minimum across all given dimensions of the objective function without neglecting any.
- **Runtime and convergence:** The runtime of the optimization process is a crucial goal. Considering all challenges and difficulties, the algorithm has to find the global optimum in a reasonable amount of time. Furthermore, the algorithm has to converge to the optimum. It may not be found directly, but the faster the algorithm tends to get in the near of the global optimum, the better.

Since different problems have different challenges, the goal of an algorithm is to deal with different factors and find the global optimum as fast as possible and with a certainty as high as possible.

Because of the amount of different types of algorithms, Global Optimization techniques can be divided into several categories. The first distinction can be made between deterministic and probabilistic methods. Deterministic algorithms try to guarantee to find the optimal solution. On the downside, these methods only work on continuous functions that are twice differentiable. The probabilistic algorithms, on the other hand, use random numbers in some form to predict a feasible good solution within a predefined region of the search space. One advantage of these algorithms is that they work well with higher dimensional and more complicated functions [DP98]. Furthermore, the probabilistic algorithms can be distinguished in instance-based and model-based methods. The idea behind model-based methods is

that the candidate solutions are created by a model which is updated every iteration to direct the search into promising regions. In contrast to that, the instance-based algorithms generate candidate solutions only depending on previously generated solutions [Bar16].

Another important part of many optimization methods is the heuristic. The task of a heuristic is to suggest the next solutions an algorithm should investigate. Thus, a heuristic offers an intelligent way of approximating the path to good solutions [Wei09].

This thesis presents a new approach in the field of probabilistic instance-based Global Optimization algorithms. At this point, some of the most prominent algorithms in this category will be introduced.

- **Hill Climbing:** Also known as greedy local search, the hill climbing algorithm is one of the simpler Global Optimization techniques. At the beginning, a starting point is chosen randomly in the search space. Then, the algorithm iteratively selects the neighbor with the best objective value from the current position. The algorithm terminates if no improvements are possible from the current position. The behavior of this algorithm is analogous to a hill climber who always chooses the way which brings him the nearest to the top of the hill. Once there is no way going up, the hill climber stops, hopefully reaching the top of the hill. The algorithm doesn't work well on functions with many local extrema because it can get stuck on one of the local extrema. Thus, the algorithm is incomplete. This can be counteracted with random restarts, initializing the starting point at different locations [RN09].
- **Simulated Annealing:** The particularity of the simulated annealing algorithm is that it allows values to be selected in the iterative optimization process which have a negative impact on the candidate solution. At first, the algorithm selects the next solution from a random neighbor of the current position. If the neighbor improves the solution, it is always accepted. Otherwise, the neighbor is only accepted with a certain probability. This probability decreases over time, such that the algorithm can get out of local extrema and stay at the best solution. The decreasing of the mentioned probability is analogous to annealing of, for example, steel, hence the name of the algorithm [RN09].
- **Swarm Intelligence:** The idea of this class of algorithms came from the observation of biological swarms, for example, birds or ants. The most famous representatives are the Particle Swarm Optimization (PSO) and the Ant Colony Optimization (ACO). In the latter, the analogy is made with ants trying to find food and bring it back to the nest. This process can be seen as finding the extrema in the search space. Multiple ants spread out and leave behind a trail of pheromones which evaporate after a certain amount of time. New ants of the colony spread out to new directions or to already good routes with certain probabilities. As a result, short routes or more frequented routes are likely to lead to better results. In the meantime, routes which don't lead to food are probably no longer visited because the pheromone trail will be gone [NM06].
- **Monte Carlo methods:** The category of Monte Carlo methods contains a lot of different algorithms. In general, these methods try to find a good solution by executing random simulations [Rob14]. In this thesis, an individual version of Monte Carlo Tree Search is implemented which can be seen as a very specific sub-category of Monte Carlo methods. Basically, Monte Carlo Tree Search is mostly used in game playing because the simulations of Monte Carlo methods can help approximating the next game moves [Bro+12]. Because Monte Carlo Tree Search has a major role in the presented algorithm (see Chapter 4), it will be described in detail in Chapter 3.
- **Genetic Algorithms:** The last presented example in this list are Genetic Algorithms. Inspired by the genetic process of natural selection, these methods try to find a good solution by evolving a set of candidate solutions which is analogous to a natural population. Since Genetic Algorithms take a larger role in this thesis, they will be described in Section 2.2 with more detail.

In addition, there are several more types and algorithms that are not listed here.

Furthermore, there are many applications for Global Optimization in different scientific fields such as mathematics, chemistry or engineering. One famous application is the Traveling Salesman Problem (TSP). Given n cities, the TSP aims to find the shortest path in which every city is visited exactly one time. This problem can be used in various applications, for example, to optimize routes of trucks which have to empty mailboxes in a certain amount of time [LK75]. Next to other logistic and scheduling applications, the TSP is used in other situations where it is not obvious, for example in the DNA synthesis. Tang and Chilkoti show in [TC16] that finding the least repetitive gene sequence is analogous to the Traveling Salesman Problem.

Another example application is found in the field of civil engineering: In [CZN09] the authors optimize the size and topology of pile foundations with Genetic Algorithms. Pile foundations are important in the support of bridges, buildings or other structures.

2.2 Genetic Algorithms

Being a subfield of Evolutionary Algorithms, the idea of Genetic Algorithms is based on the theory of evolution. In 1859, Charles Darwin published his work *On the Origin of Species* [Dar59], marking the beginning of this field of science. Basically, the theory of evolution describes how species develop over generations. An organism, either a plant or an animal, can adapt to an ecosystem by altering its properties and behaviors. This process occurs by changing the genes of the children in relation to their parents. The children, which are most "fit" to an ecosystem survive, while some children with less survival probability die. This concept of natural selection can be summarized with the well-known term *survival of the fittest*, meaning that only the strongest individuals of a species or a population survive. Even if a species changes the ecosystem or the ecosystem itself changes, a species can adapt to that. Possibly a new species can arise from this change. Multiple species can share an ecosystem.

Many expressions in Evolutionary Algorithms are borrowed from the theory of evolution. In addition, principles and methods of the algorithms have strong analogies with the biological field. Evolutionary Algorithms include many different subcategories, from which a few are introduced below:

- **Genetic Algorithms:** The most popular type is the Genetic Algorithm which is used mainly for optimization tasks. In the context of evolution, chromosomes in the algorithm are traditionally represented as a bitstring. One of the first works, which shaped Genetic Algorithms, is Holland's *Adaption in Natural and Artificial Systems* [Hol92]. Genetic Algorithms are described in detail in the course of this section.

- **Evolution Strategy:** The next category of algorithms are the Evolution Strategy algorithms. This type is similar to Genetic Algorithms, but with the main difference that a list of real values is used and optimized instead of just binary values [Wei09].

- **Genetic Programming:** In contrast to the other mentioned algorithm types, the results of Genetic Programming are computer programs. Starting with simple expressions, the programs evolve and get better with increasing number of iterations. The representation of the programs is achieved with trees [Wei09].

This thesis only focuses on Genetic Algorithms. Before describing the algorithm itself, some basic concepts are described in Section 2.2.1. After that, some genetic operators will be introduced in Section 2.2.2, followed by the sketch of a simple Genetic Algorithm in Section 2.2.3.

2.2.1 Components of Genetic Algorithms

The first and the most important component of Genetic Algorithms is the way how states are handled and how variables are decoded. Each state in the solution space is represented as a so-called chromosome in analogy to the evolutionary theory. In the literature, the expression "*individual*" is also very common for representing one such a state. An individual can be seen as the carrier of one chromosome, thus, the two terms stand for the same. A chromosome can contain one or more segments, named genes. In the context of Global Optimization[1] a chromosome corresponds to one point in the solution space and the genes act as the different variables. If the solution space is $\mathbb{R}^n$, the genes represent one dimension each. In Genetic Algorithms, a chromosome is usually encoded as a binary string. Figure 2.2 shows how a chromosome containing two genes looks like in the context of Genetic Algorithms.

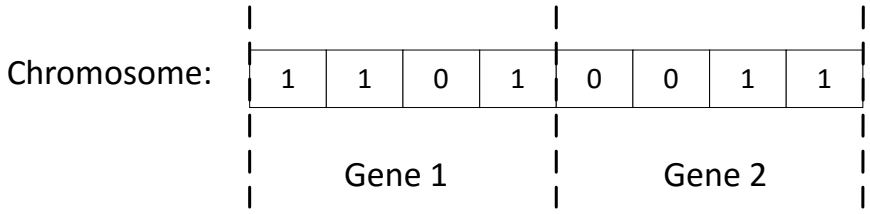


Figure 2.2: One example chromosome with two genes, representing one individual. (Source: Own representation based on: [SD07])

Mathematically, a chromosome with N_{var} genes can be written as a list of genes: chromosome $= [\text{gene}_1, \text{gene}_2, ...\text{gene}_{N_{\text{var}}}]$, where each gene can have different numbers of bits. A Gene itself can be expressed as

$$g = [g_{N_{\text{Gene}}}, g_{N_{\text{Gene}}-1}, ..., g_2, g_1]$$

where each g_i represents a bit of the gene. The bit string is only the encoded representation of a value that is used by the algorithm. The value can be of any type, for example, a real value, a decimal value or even an object like a color. In this thesis, the genes are mainly encoded from real values with a lower and an upper bound x_{lo} and x_{hi}, such that $\forall x \in \mathbb{R} : x_{\text{lo}} \leq x \leq x_{\text{hi}}$. These bounds are defined by the problem for each variable.
The decoding of a gene is different for each type of value. In the case of real values, the binary gene is converted to a decimal number: Given a gene g with length N_{Gene}, the decimal value of g is

$$x_{\text{dec}} = \sum_{i=1}^{N_{\text{Gene}}} 2^{i-1} g_i$$

The next step of decoding to real values is to normalize them to the given range, bounded by x_{lo} and x_{hi}:

$$x = x_{\text{dec}} \left(\frac{x_{\text{hi}} - x_{\text{lo}}}{2^{N_{\text{Gene}}} - 1} \right) + x_{\text{lo}}$$

This procedure is called quantization. The task of quantization is to map continuous numbers to discrete sub ranges. In Table 2.1 some example decoding values for a two bit binary gene (first column) are shown. The second column presents the decimal values of the binary representation. Next, the third

[1] See Section 2.1.

column shows the quantized values, assuming $x_{\text{lo}} = 2$ and $x_{\text{hi}} = 3.5$. A gene can also be decoded to quantitative types, for example, column 4 shows the representations of colors.

Binary representation	Decimal Number	Quantized value	Color
00	0	2.0	red
01	1	2.5	green
10	2	3.0	yellow
11	3	3.5	blue

Table 2.1: Decoding examples of a two bit gene. (Source: Own representation based on: [HH98])

The maximum deviation between a continues value and the quantized value is called a quantization error. This is a flaw of this discretization that has to be considered before the optimization. In the example above, the quantization error is 0.25. If the number of bits in a gene is increased, the quantization error decreases because there are more bits the value can be mapped to.

In general, an *encoding* defines the method of how a gene is represented. Above, only binary encoding was presented. This is the most used procedure in the literature as well as in this thesis. If a different encoding is used other than the binary encoding, then the rest of the algorithm, especially the genetic operators [2], have to be adapted to that eventually. That's why the focus of this thesis is on binary encoding. However, there are many more methods in this regard. According to [SD07], some of them are:

- Binary encoding: A gene is represented as a binary string.
- Octal encoding: Each digit of the gene is an octal number (0-7).
- Hexadecimal encoding: The digits of this encoding are to base 16, supporting the values 0-9, A-F.
- Permutation encoding: Every gene in permutation encoding is a sequence of integer values in which each value exists exactly once. This encoding can be used in ordering problems, for example in the Traveling Salesman Problem.
- Value encoding: A gene can also be represented as a sequence of values. The values can be of any type, for example, real numbers, chars, quantitative values (e. g. colors) or even complex objects.
- Tree encoding: Each gene with this encoding is handled as a tree. This can be used for example for Genetic Programming.
- Object: Generally, a gene can be represented by any kind of object. This might be a list, an array or any other representation, specifically for a problem.

One of the major components of any Genetic Algorithm is the so-called *fitness function*. Each individual has a fitness, in analogy to the theory of evolution, which determines how good it fits in an ecosystem. To calculate the fitness, first, the chromosome of the individual has to be decoded. After that, the variables are evaluated with the fitness function. The algorithm prefers chromosomes that have a higher fitness evaluation, like in nature. In Global Optimization, in most cases, the objective function is evaluated as the fitness function. If the optimum to be found is a minimum, then the negated objective function will be used as the fitness function.

[2] See Section 2.2.2

In a Genetic Algorithm, multiple individuals are kept in a so-called population which is the last component mentioned in this section. The population will be maintained by the Genetic Algorithm throughout its iterations. At the start of the algorithm, an initial population with N_{Pop} chromosomes is selected randomly. Each iteration of the algorithm alters the population by changing, deleting and adding new chromosomes. Such an iteration is named a *generation* according to the analogy with the evolutionary theory. The goal of this iteration process is that only chromosomes with better fitness remain in the population. This mimics a natural selection.

2.2.2 Genetic Operators

This section describes the basic concepts of genetic operators. In the Genetic Algorithm, these operators are executed in the same order in each generation. They have the task to modify the population. Thus, the solution space will be explored and eventually, the best chromosomes of the population converge to the optimum. The algorithm starts with the first generation of N_{Pop} randomly selected chromosomes. In each generation, the following operators are performed.

- **Selection:** This operator determines which chromosomes are used for the reproduction and which are discarded. In addition, a selection mechanism can decide which specific chromosomes are used for a particular reproduction. The chromosomes, which produce the next generation, are called parents. After the selection, N_{keep} are kept and the other $N_{\text{Pop}} - N_{\text{keep}}$ are removed from the population. The degree, in which amount more fitter chromosomes are preferred by a selection method, is called selection pressure. After this definition, the higher the selection pressure, the faster the algorithm converges. However, if the pressure is too high, the algorithm may get stuck on local optima. This is why a selection method should select diverse chromosomes to best explore the solution space. The most used selection mechanisms according to [SD07] are:
  - Roulette Wheel Selection: In this traditional selection technique, the chromosomes are weighted with their fitness. From these weighted individuals, random parents are selected. This has the disadvantage that chromosomes with higher fitness do not necessarily make the cut to the next generation.
  - Rank selection: This selection method is similar to the Roulette Wheel selection. But first, the chromosomes are mapped with the rank of the ordered fitness values. In a population with N_{Pop} individuals, this means that the chromosome with the highest fitness gets rank N_{Pop} and the chromosome with the least fitness gets rank 1. In the last step, these ranks are used as weights for a random selection of the parents.
  - Tournament selection: Out of the presented selection methods, the tournament selection mimics natural behavior of an ecosystem the most. Out of all N_{Pop} individuals of the current population, a small subset is randomly selected. This subgroup performs a tournament. The winner is the individual with the highest fitness, becomes a parent for the next generation and stays in the population. This procedure is repeated until the number of parents N_{keep} is reached.
  - Random selection: In this last method, the parents, who are able to reproduce, are selected completely at random. This works disruptive for the algorithm because no improvement is guaranteed.
- **Reproduction:** After the selection, the reproduction is the next major genetic operator. The task is to create new individuals based on the parents which are chosen by a selection operator before. The new individuals are called children. Exactly $N_{\text{Pop}} - N_{\text{keep}}$ children are created to get the population back to the original size. The method, which performs the reproduction, is called *Crossover*. The idea behind this is to hopefully find better fitting chromosomes from parts of good chromosomes and to converge the algorithm to an optimum with these. The most common crossover method is

the so-called *single point crossover*. In this method, two parents create two children. The parents are separated at a random point and the children are each created by concatenating the first part of one parent and the second part of the other. An example of a single point crossover of 8-bit chromosomes is shown in Figure 2.3.

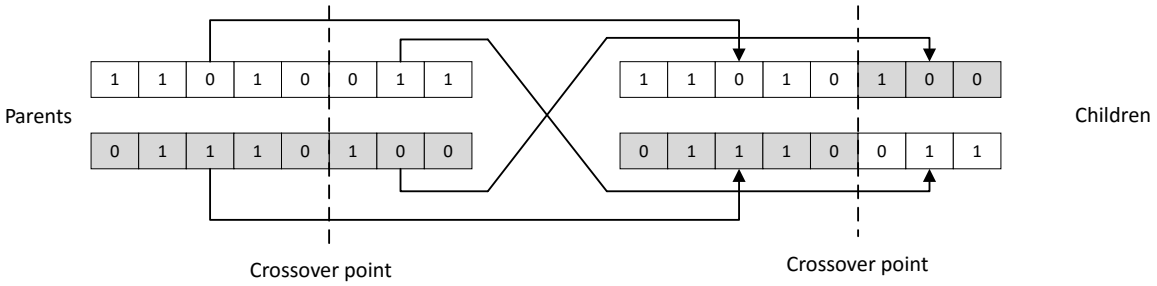


Figure 2.3: Single point crossover example (Source: Own representation based on: [HH98, p. 42])

There are many more crossover methods in the literature that are more complex and work well in some optimization cases. At this point some of the more known methods according to [SD07] are listed:

- Two-point crossover: It is possible to select two random crossover points. The children are created by putting together the three different parts of the parents' chromosomes. The number of crossover points is not restricted. Depending on the number of bits in a chromosome, the so-called N-point crossover can have an arbitrary number of points at which the parents are separated.
- Three parent crossover: In this method, the chromosome of the children are created by three parents.
- Uniform crossover: For uniform crossover, a random mask is created in the form of a binary string with the length of the chromosomes. The value of the bit at each position in the mask decides from which parent the corresponding child bits are created.

One important parameter of a Genetic Algorithm is the crossover probability p_C. The value determines how often a crossover should be performed from all the parents that came through the selection process. If a crossover is not performed, the chromosomes of the children are exact copies of their parents.

- **Mutation:** The mutation is the last operator in the sequence of modifying the population. In biology, the genes are not always copied one to one, some parts of the genes can randomly change. This process is called mutation and the corresponding genetic operator of the algorithm mimics this behavior. At the beginning, for each chromosome of the current population random points are determined and after that, the corresponding bits are changed. Depending on the selected mutation method, this change has different effects. The mutation probability p_M is one of the basic parameters of any Genetic Algorithm and specifies how much of a chromosome should be mutated. The reason mutations are used is to better explore some parts of the solution space. The algorithm does not get stuck on certain chromosomes, which means that with mutations it can get out of local optima. As with the other operators, there are many possibilities of how the operator can be executed. Sivanandam and Deep describe three basic mutation schemes in [SD07]:
  - Flipping: At the random points, the bit is flipped, meaning $0 \mapsto 1$ and $1 \mapsto 0$.
  - Interchanging: For this method, two random positions in the chromosome are necessary. The values of these points are switched.
  - Reversing: All the bits after a random point are flipped.

- **Elitism:** Technically, this is not an operator. Instead, the concept of elitism ensures that the fittest individuals always stay in the population. After executing selection, reproduction and mutation, in some cases the best individuals of a population do not make the cut to the next generation. Elitism protects these individuals with the highest fitness by the specification of an elite parameter p_E which expresses the number of chromosomes to stay in the generation. The consequence of using elitism is that the results after each generation are monotonically increasing [BC95].

2.2.3 Algorithm outline

At this point, all essential pieces of Genetic Algorithms to understand the algorithm were presented and are now put together. The functionality of a simple Genetic Algorithm will be explained by means of the flowchart in Figure 2.4. Before the beginning, the parameters need to be set and the respective genetic operators to be chosen. This selection should be made depending on the problem. In addition, the encoding for the chromosome needs to be specified, i.e. for the binary encoding how many bits the different genes occupy.

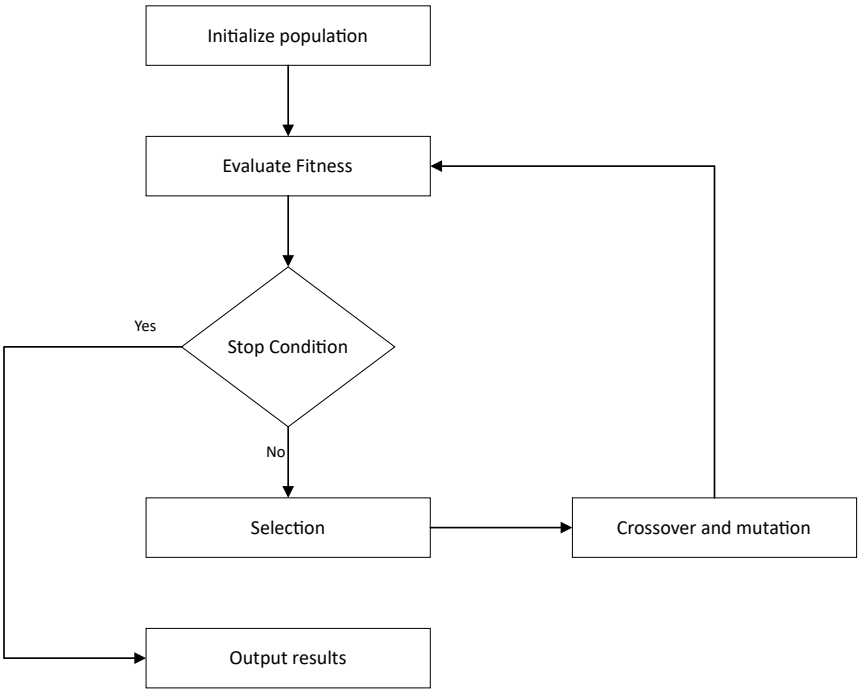


Figure 2.4: Simple Genetic Algorithm flow chart (Source: Own representation based on: [ABB14])

The first task of the algorithm is to create an initial population. For that, N_{Pop} chromosomes are created randomly with the goal to cover as much of the solution space as possible. Next, the algorithm loop starts. Each loop contains a new generation of individuals. At the start of the loop, the stop condition is checked. This may be one of or a combination of the following termination criteria based on [SD07]:

- Number of generations: If a specified number N_{Gen} of generations is reached, the stopping criterion is fulfilled.
- Time limit: Similar to the first point, this type of criterion is reached when a certain amount of time has passed.

- No fitness improvement: This stopping condition is true, if either after a specific amount of time or after a given number of generations, no improvement of the result has been made.
- Convergence criterion: A few techniques are summarized under this last termination criterion. The common property of these methods is, that the criterion will be fulfilled if the fitness of the solution reaches a specific value. This can be, for example, if the best individual of the current population has a specified minimum fitness. Another possible stopping condition can be that the fitness of the worst individual falls under a lower boundary.

If the stopping condition is met, the algorithm aborts and returns the current best chromosome of the population. If not, the algorithm loop starts. First, a certain number of chromosomes is chosen by the selection operator to get to the next step of the loop. These selected parents create new children with the crossover operator. At the last step of the algorithm loop randomly selected parts of the chromosomes are changed by the mutation method. The new generation is now evaluated by the fitness function and, after that, the stopping condition is checked again which finally closes the loop.

Genetic Algorithms belong to the class of anytime algorithms. This means that the algorithm can be interrupted at any given time and it is still able to return a valid solution [Zil96]. In the flowchart in Figure 2.4 it becomes apparent that this behavior can be implemented by allowing an interruption as a stopping condition. The currently best chromosome of the population is returned.

In the literature, there are many theories that try to prove or justify why Genetic Algorithms work. The number of theories shows that there is no consensus reached. The most famous theory is the *Building Block Hypothesis* which is based on the idea of how children can build large castles out of small building blocks. In the context of Genetic Algorithms, this hypothesis states that solutions with high fitness (building blocks) are put together by the algorithm to better solutions. In addition, it is assumed that the neighborhood of this good building blocks contains solutions of high fitness as well. With the reasoning that good solutions are next to better solutions, the convergence is justified by the author of the hypothesis [SD07].

In this thesis, the Genetic Algorithm Framework for .NET (GAF) [New] is used for the execution of this type of algorithm.

3 Monte Carlo Tree Search

The Global Optimization algorithm proposed in this thesis (see Chapter 4) is based on the Monte Carlo Tree Search (MCTS). This chapter describes fundamental knowledge of this algorithm. First, some background information is presented, followed by the description of the exploitation vs. exploration dilemma. In Section 3.3, the algorithm itself is illustrated. The explanation of the related Upper Confidence bound for Trees is given in the section after that. This chapter is concluded with an outlook which variations of the Monte Carlo Tree Search exist.

3.1 Background and Game theory

Historically, Monte Carlo methods were used for Global Optimization tasks since the 1940s, trying to approximate to an optimal solution with the help of random numbers, as mentioned in Section 2.1. Monte Carlo Tree Search emerged from this method with additional other influences. However, it became popular in the domain of artificial intelligence in games. In the last decade, MCTS became especially successful in the game of *Go* [1]. In 2016, the program Alpha Go defeated the current world champion in Go with the help of MCTS and deep reinforcement learning. Because MCTS is developed in the domain of games, some basic concepts of game theory and other relevant topics are presented in the further course of this section. The role of this algorithm in this thesis is very important and will be described in Chapter 4.

Game theory is a highly researched field of mathematics that exceeds its application of simple board games. Its basis are the Markov Decision Processes (MDP) which model a decision as *(state, action)* pairs. In game theory, a state $s \in S$ represents all current properties of the system and an action $a \in A$ is the move one player makes. S and A are the sets of all possible states and actions. MDP further describes, that from each state an action results in the next state s'. This process is modeled by a transition function $t : S \times A \mapsto S$:

$$t(s, a) = s'$$

The start of a game is modeled by an initial state s_0 and the game ends at so-called terminal states s_T. In addition, a utility function $R : S \mapsto \mathbb{R}^k$ interprets a state to a real value, that measures how good a state is for a player. Typically, every non-terminal state will be associated with the utility value 0, and terminal states with the values $+1$ or -1, whether or not a player wins in this state or loses [Bro+12]. The goal of artificial intelligence for games is to determine which next move to take by selecting the most promising action in the current state.

3.2 Exploitation vs. exploration

In addition to its game theoretical background, MCTS also belongs to the class of bandit-based methods. Given K one-armed bandits [2], the goal is here to maximize the reward by always finding the machine that wins. This scenario is often equated with finding the best arm of a multi-armed slot machine with K arms.

[1] Go is a traditional Chinese board game, where two players play against each other and try to occupy the most territory of the board.

[2] In this thesis, one-armed bandits are the slot machines in a casino.

To choose the right action is hard because it is not known which reward distribution is behind each arm. Because of that, the decision which arm to take next is purely based on experience. This results in the following problem: If one arm wins enough times, this information shall be either exploited by choosing the same arm again or the other actions shall be explored. This is called the *exploitation exploration dilemma*.

A possible control mechanism for this dilemma is the Upper Confidence Bound (UCB) of any arm j. This was proposed by Auer et al. in [ACF02] and became rather popular, especially since it is directly used in modern MCTS approaches. Given at arm j the total number of plays so far n, the number of plays at that arm n_j and the average reward of that arm $\overline{X_j}$, then the next action should select the arm that maximizes the UCB1 formula:

$$\text{UCB1} = \overline{X_j} + \sqrt{\frac{2 \ln n}{n_j}}$$

The advantage of this is that the two parts of the sum in the formula each favor one of exploitation and exploration of their dilemma. The average reward at the jth arm $\overline{X_j}$ is in favor of exploitation. The term $\sqrt{\frac{2 \ln n}{n_j}}$, on the other hand, stands for the exploration of other arms [Bro+12].

3.3 Algorithm description

At first, a general MCTS algorithm is presented that may be used in any gaming application. Throughout the time the algorithm is running, it builds a so-called game tree that helps to decide which next action to take. Game trees are a common tool in this domain and are used by several other algorithms, for example, the *Minimax* algorithm [RN09]. Furthermore, game trees are similar to the standard data structure named trees in which a node represents the state of the game and an edge stands for the move a player can make.
Many games can be very complex and, therefore, have many states and moves to take from each state. For example, the game of Go has more possible states than atoms in the universe. Thus, the construction of a complete game tree may be impossible. MCTS solves this issue by iteratively building the game tree with the help of random simulations. In an AI game, this algorithm would be re-executed each time the agent has a move to make.

The tree building process takes place in the algorithms main loop. In each iteration, four different steps are executed. Within these steps, two policies have to be distinguished. With the help of this six elements, the tree is built step by step. The different pieces are illustrated in Figure 3.1 below.

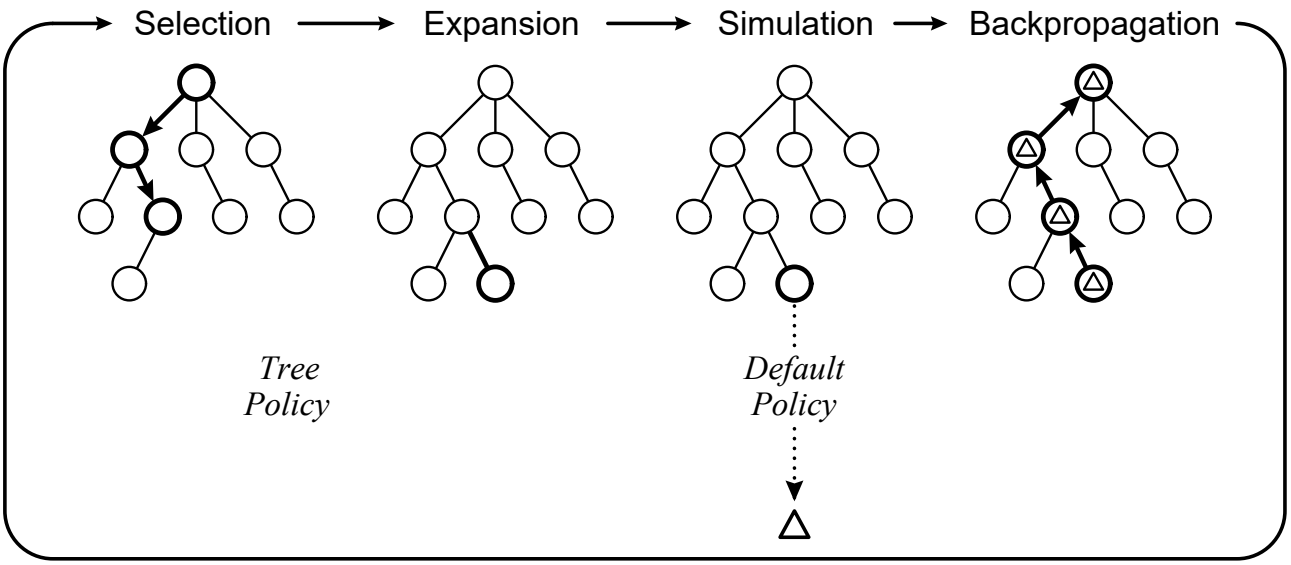


Figure 3.1: MCTS iteration example with all four steps. (Source: [Bro+12]).

The four steps of this figure are in detail:

- **Selection:** In the first step, a node of the tree is selected on which the focus lies upon in the remaining steps. This selection process follows the *tree policy* by traversing the tree from the root to the node that is selected eventually. Hereby, the policy defines which path is taken, i.e., which child node is selected in each step of the selection. Usually, the selection is halted if a node is not fully expanded yet, which means that not all possible actions of the current state are added to the node in the form of edges.
- **Expansion:** The selected node from before will be extended with a move that has not been taken so far from the selected state.
- **Simulation:** From the newly created node, a simulation of the remaining game is executed. The *default policy* defines the rules how the simulation is run. In the most simple and common way, in each state, a random action is taken until a terminal state is reached. It should be noted that these simulated states are not added to the tree. From the simulated terminal state, the utility function will be called and only the reward of the state is saved for later use.
- **Backpropagation:** In the last step of the main loop, the simulated reward is backpropagated to the root. Thereby, the same path is taken like in the selection step of this iteration, only backwards. At each node of this backup process, the number of visits is increased by one and the simulated reward is added to the total reward. This information is used in the next iteration by the selection step.

These steps are performed in a loop in which one node is added to the tree for each iteration. While building the tree, the results of the simulation are propagated to the correct nodes in order to suggest the best move to take next. Each node v of the tree stores four different fields: The corresponding state s, the total number of visits n_v, the total number of rewards q_v and a reference to the parent node a_v. It can be more practical to implement this by storing a list of children in each node to traverse the tree more easily.

Putting all the steps together, a general MCTS algorithm is summarized in Algorithm 1 in form of pseudo code. Starting with an initial state s_0, the root node of the tree is created with this state. After that, the previously described main loop is executed as long as the algorithm is within a computation budget. This budget is usually in form of a time limit or a specified number of iterations. At last, MCTS returns the action that leads from the root to the node with the best average reward.

Algorithm 1 General Monte Carlo Tree Search based on [Bro+12]

Input: The initial state s_0.
Output: The best action a_b.

```
1: function MCTS(s_0)
2:     v_0 ← create root node with s_0
3:     while in computation budget do
4:         v_l ← TreePolicy(n_0)
5:         Δ ← DefaultPolicy(n_l)
6:         Backup(v_l, Δ)
7:     end while
8:     return Action a that leads to the child of v_0 with the highest average reward.
9: end function
```

3.4 Upper Confidence bound for Trees

In the general MCTS algorithm of the previous section, the tree policy has to decide which path to take to select a node. At that, the policy faces the exploitation exploration dilemma mentioned in Section 3.2: Should a new node be explored or should nodes that were good before visited again? The solution to this is to maintain the upper confidence bound in this situation. This usage of UCB in MCTS forms the new algorithm *Upper Confidence bound for Trees* (UCT) which was introduced by Kocsis in [KS06]. The next child node of the node v in the path according to the tree policy with UCT is determined by

$$\pi(v) = \arg\max_{v' \in K(v)} \left(\frac{Q_{v'}}{n_{v'}} + C_p \sqrt{\frac{2 \ln n_v}{n_{v'}}} \right)$$

where $K(v)$ is the set of child nodes of v, $Q_{v'}$ is the total reward of child v', $n_{v'}$ is the total number of visits of child v' and n_v the total number of visits of parent v. In addition to that, $C_p \geq 0$ defines an exploration parameter that can tune how much exploration should be preferred over exploitation or the other way around. The first part of the sum, $\frac{Q_{v'}}{n_{v'}}$, expresses the average reward as in the original UCB formula. The higher C_p is set, the higher the second term of the sum is preferred and with that, the algorithm tends more to the exploration. If $C_p = 0$, the selection process is only based on the experience so far: The policy exploits the best nodes. If two or more children of a node share the same UCT value, one of these nodes will be selected randomly. Generally, a tree policy of MCTS in this thesis is denoted by π.

In [KS06], the authors show that each simulation in MCTS with UCT improves the probability of choosing the correct next move. Furthermore, the authors proof that with enough computing time and memory the algorithm eventually converges to an optimal game tree.
Like Genetic Algorithms, MCTS can be seen as an anytime algorithm because it can be interrupted at any possible time. Is that the case, the main loop is halted and the currently best child of the root node is returned.
As a result of the tree policy, the tree growth of UCT is asymmetric. It is expanded with a higher probability in regions where the simulations resulted in more promising states. This does not mean that the tree grows only in one direction. Depending on the value of the exploration parameter C_p, the tree can expand in different areas.

It should be noted, that the nomenclature and variable names of the algorithms in this chapter are only to demonstrate the functionality of MCTS and UCT. Although these algorithms are used in the proposed algorithm, these elements will be reintroduced there (see Chapter 4). This can be justified that they are better matched with other elements and not be confused with something else.

3.5 Outlook on Monte Carlo Tree Search

In the literature, there are many extensions and variations of MCTS. For example, the algorithm may be used in single and multiplayer games. There are enhancements of the tree and the default policy. Furthermore, in a parallel variation of the algorithm, many simulations are executed simultaneously to reduce runtime. Finally, different backup strategies were investigated. This thesis uses some of those enhancements which will be presented in Chapter 4. Further information of MCTS can be taken from [Bro+12] in which the authors give a good and widespread survey on the topic (as of 2012).

4 Subset Lattice Monte Carlo Tree Optimization

This significant chapter presents the approach of this thesis. It describes a new optimization technique based on Monte Carlo Tree Search and Genetic Algorithms. First, basic concepts are introduced which are put together and form the base version of the proposed algorithm. In the following section, enhancements and extensions are explained that try to improve the base version. The last section concludes the approach by offering an outline. Throughout this and the following chapters, the algorithm is also referred as Subset Lattice Optimization (SLO), omitting the expressions “Monte Carlo Tree”.

4.1 Base algorithm

At the beginning of this section, basic concepts and definitions are introduced. These topics are extended in the following subsections and ultimately lead to the proposal of how the new optimization algorithm works.

4.1.1 Partially ordered set

In the order theory of mathematics, according to [BC02], a partially ordered set (or *poset*) is a set on which a partial order is defined. This means that not all elements of the set must relate to each other. Given a set S and a binary relation $\leq \in P \times P$ which orders elements of P, $\leq$ is a partial order if the following three axioms hold ($\leq$ does not necessarily stand for the *less or equal* relation on numbers, but rather any binary relation):

- **Reflexivity:** $\forall x \in P : x \leq x$
- **Antisymmetry:** $\forall x, y \in P : x \leq y$ and $y \leq x \implies x = y$
- **Transitivity:** $\forall x, y, z \in P : x \leq y$ and $y \leq z \implies x \leq z$

This partial order is also called non-strict, because of its reflexivity property. An order is called a strict partial order if the irreflexivity axiom holds instead, which means that no element is related to itself. One example of a partially ordered set are the natural numbers $\mathbb{N}$ with the divisibility operator $|$ as an order relation. All three listed axioms hold for this relation that expresses if number $a \in \mathbb{N}$ can be divided by $b \in \mathbb{N}$. If this is the case, it is written as $b|a$, which implies that there exists an integer k such that $bk = a$. In addition, this example is not a totally ordered relation. The reason for this is that there are natural numbers that have no relation to each other. For example, 5 cannot be divided by 2 or vice versa.

4.1.2 Lattice

The definition of lattices is built on posets. However, before describing them, a few further terms need to be presented. The following derivation of the terms in this section is based on Davey’s and Priestley’s work in [DP02]. Let P be a partially ordered set with an order relation $\leq$ and $S \subseteq P$, then a lower bound is defined as x_{lo}, if $\forall s \in S : x_{\text{lo}} \leq s$. Similarly, an upper bound can be specified as x_{hi} if $\forall s \in S : s \leq x_{\text{hi}}$. All upper and lower bounds of S are condensed in the two sets

$$S_{\text{up}} = \{p \in P | \forall s \in S : s \leq p\}$$
$$S_{\text{lo}} = \{p \in P | \forall s \in S : p \leq s\}$$

Following that, a supremum is the least upper bound and an infimum is the greatest lower bound of the two sets. Those two elements are denoted as $\sup S$ for the supremum of S and $\inf S$ for the infimum respectively. With these explanations, a general definition of a lattice can be provided. A partially ordered set P with an ordering relation $\leq$ is called a lattice if each two element subset $\{a, b\} \subseteq P$ has a supremum and an infimum:

$$\forall a, b \in P : \exists \inf\{a, b\} \text{ and } \exists \sup\{a, b\}$$

An example for a lattice is any subset of $\mathbb{N}$ that contain only divisors of a specific number x with the divisibility as an order relation. The lattice is lower limited by 1 because natural numbers can be divided by 1. In addition, every natural number can be divided by itself, thus, the lattice has the upper limit of x. For instance the divisors of 30 are considered: $P_1 = \{1, 2, 3, 5, 6, 10, 15, 30\}$. P_1 is a poset under the axioms of Section 4.1.1 and also every two element subset of P_1 has an infimum and supremum.

4.1.3 Hasse diagram

A Hasse diagram is a visual representation of a lattice or a poset [BP11]. It was first made popular by the German mathematician Helmut Hasse. The diagram visualization consists of a graph. Within that, all elements of the poset (or lattice) are represented as nodes and all relations between these elements are displayed as edges. Technically, these edges are directed, indicating the order of the relation. For example, at the relation $a \leq b$, an arrow is drawn from the node a to the node b (and not the other way around). These arrows are often replaced by edges without arrows. This can have multiple reasons, for instance, that the focus of the diagram should be on the relation itself and not the order.

A further rule for creating Hasse diagrams out of posets is, that no transitive relations are visualized. When $a \leq b$ and $b \leq c$, the transitivity axiom of partially ordered sets states that $a \leq c$. But in the diagram, only the edges of a to b and b to c are drawn.

If a Hasse diagram is visualized from a poset that is not a lattice, i.e. has no upper or lower limit, then parts of the lattice can be presented and indicate the order relations. Some examples of Hasse diagrams can be seen in the Figures 4.1, 4.2 and 4.3. Because of its limits, Hasse diagrams are often diamond shaped: The upper and lower ends are tapered, while towards the middle the graph becomes wider. In addition to that, the nodes of the diagram are arranged in levels, where each level corresponds to one step of the order relation. Usually, elements of the same level do not have any edges because of the nature of the underlying order relation.

Finally, a Hasse diagram can be interpreted as a directed acyclic graph (DAG). The graph is called directed because of its underlying order relation of the poset. In addition, the graph does not contain any cycles, since this would break the antisymmetry axiom of partially ordered sets.

4.1.4 Subset Lattices

In this section, a special lattice is considered, on which the proposed algorithm builds its base on. Given any finite set S, the power set $\mathcal{P}(S)$ gives all possible subsets of S. With the subset operator $\subseteq$ defined on $\mathcal{P}(S)$, this structure forms a lattice. First, $\subseteq$ is a partial order relation on $\mathcal{P}(S)$ [1]. Basically, assuming $\subseteq$ is an ordering, there exist elements in $\mathcal{P}(S)$ that are not related to each other: Let $a, b \in S$, then $\{a\} \in \mathcal{P}(S)$ and $\{b\} \in \mathcal{P}(S)$. However, because $\{a\} \nsubseteq \{b\}$ and $\{b\} \nsubseteq \{a\}$ this is partial order. In addition, this structure is a lattice because it has a lower and an upper bound ($\emptyset \in \mathcal{P}(S)$ and $S \in \mathcal{P}(S)$). Specifically, each

[1] This statement, that $\subseteq$ is an ordering relation is assumed true at this point in the thesis. It can be shown, that all axioms for a partial order hold, but this would extend the scope of this work. For a detailed proof, see for example `https://proofwiki.org/wiki/Subset_Relation_on_Power_Set_is_Partial_Ordering`, last visited on August 25th, 2017.

two-element subset has an infimum and a supremum, thus confirming the definition of a lattice.

The subset relation on a powerset will now be shown at an example with a set that contains three elements. Let $S_1 = \{a, b, c\}$, the powerset of S_1 is $\mathcal{P}(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Figure 4.1 illustrates all these elements and their $\subseteq$ relation in a Hasse diagram.

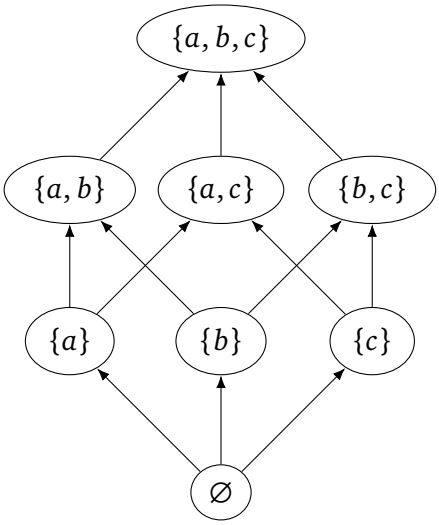


Figure 4.1: Hasse diagram of the lattice of $\mathcal{P}(\{a, b, c\})$ with the subset operator $\subseteq$ as an order relation. (Source: Own representation based on [BC02])

These subsets can be encoded by a specific scheme which provides a more intuitive readability and a better scalability for larger sets. Each subset of $\mathcal{P}(S)$ is represented by a bitstring that has the length $|S|$. If one element of S is in a subset, it will be represented by a `'1'` in the bitstring and if a subset does not contain this element, the representation is a `'0'`. It is important that the positions of the elements are the same in each bitstring representation of the subsets. For the example above with $\mathcal{P}(\{a, b, c\})$, the corresponding encoding and Hasse diagram are shown in Figure 4.2. The first position of each bitstring indicates whether or not an 'a' is in the subset, the second position stands for the presence of a 'b' and the third position stands for a 'c'.

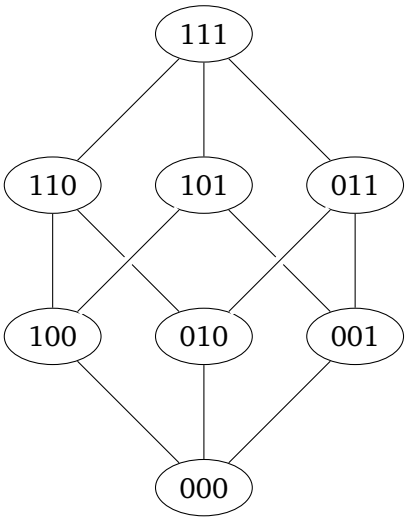


Figure 4.2: Hasse diagram of the binary encoded subset lattice of $\mathcal{P}(\{a, b, c\})$

In the further course of this thesis, this encoding of the subsets will be used. This lattice of subsets of a power set with the $\subseteq$-relation is called *subset lattice* in short. The subset lattices can be used in a general way, independent of the size of the set. In these cases the content of the original set is irrelevant, only the size of it, $|S|$, is necessary to build a binary encoded subset lattice. Henceforth, this size is called lattice base size and denoted by the variable $b = |S|$. The character b in the variable is short for *bits* or the number of bits, which is due to the fact that the base size is equal to the number of bits of each node. The size of the lattice grows exponentially. Figure 4.3 shows the Hasse diagram of the subset lattice for $b = 4$ to demonstrate this growth.

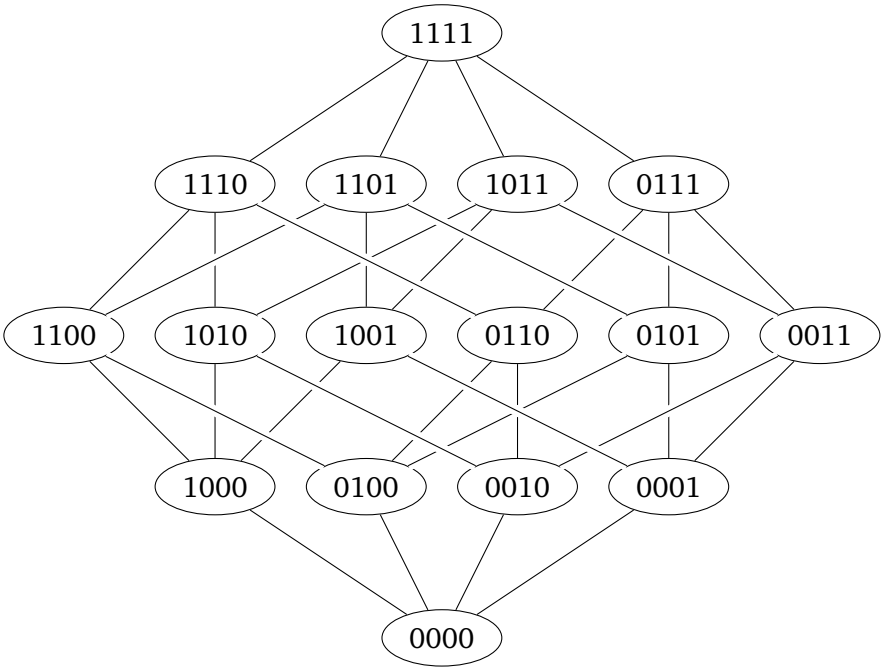


Figure 4.3: Hasse diagram of the binary encoded subset lattice with $b = 4$.

When visually compared to the Hasse diagram of a subset lattice with $b = 3$, this lattice is much bigger in the sense that it has more nodes, edges and is more interlaced. The number of nodes and edges can be determined in dependency of b. A Hasse diagram of subset lattice with base size b has exactly $b + 1$ levels. When numbering the levels, starting from 0 at the bottom and finishing with $b + 1$ at the top, each node of a level $l_i \in \{0, 1, ...b, b + 1\}$ contains (d_i) 1's and $(b - l_i)$ 0's. Consequently, each level has as many nodes as combinations of the 1's and 0's without repetition. This number is expressed with the binomial coefficient. The total number of nodes is denoted by

$$n = \sum_{i=0}^{b} \binom{b}{i} = 2^b$$

Similarly, the number of edges can be calculated. Each node has the same number of outgoing edges, depending on the level where it is located. This is because in a level d_i, all nodes only have as many successors as number of 0's: $(b - l_i)$. Thus, the number of edges is determined by

$$e = \sum_{i=0}^{b} \binom{b}{i} i$$

The size of the lattice grows exponentially with increasing number of bits per node, as it is illustrated in Table 4.1.

b	nodes	edges
1	2	1
2	4	4
3	8	12
4	16	32
5	32	80
10	1024	5120
20	1.05E+06	1.05E+07
30	1.07E+09	1.61E+10
50	1.13E+15	2.81E+16
100	1.27E+30	6.34E+31

Table 4.1: The size of the lattice depending on the lattice base size b.

4.1.5 Optimization on Subset lattices

The previously introduced concepts are brought together in this section to present the approach of the proposed optimization algorithm. The idea is to perform a Monte Carlo Tree Search on a binary encoded subset lattice. The subsets are treated like the chromosomes of genetic algorithms (see Section 2.2): In this case, the binary string represents a state in the solution space. The optimization algorithm tries to find the best state in the lattice. As in Genetic Algorithms, the binary string can consist of multiple genes which are decoded at specific points of the algorithm to evaluate the fitness. In the domain of Monte Carlo Tree Search, the fitness of chromosomes can be equated with the reward of states. Both expressions mean the same and are used equally in the following.

Subset lattices of $\mathcal{P}(S)$ have a unique lowest element $\emptyset$ and a unique greatest element S. Therefore, this can be viewed as a tree. The root of the tree can either be represented by this lowest or greatest element. The proposed algorithm creates a tree where each node contains a state that stands for a binary encoded subset. In this thesis, the proposed algorithm starts by creating a root node out of the state that contains the binary encoding of the empty set. The lattice base size b also defines the number of `0`'s in the root node, and can also be called chromosome length.

Like in the classic MCTS algorithm as described in Chapter 3, the subset lattice optimization iteratively adds new nodes to the tree. For that, successor nodes have to be chosen from a list of possible actions. Each action has the purpose of flipping a bit from $0 \mapsto 1$ at a specific position. This means that every action contains an index of the position in the binary string that needs to be changed. If the tree was built completely, it would contain all possible binary combinations, like the subset lattice. The consequence of only flipping in one direction is that the number of possible actions reduces with increasing depth. The complete tree has the same properties as mentioned for subset lattices in Section 4.1.4. In this thesis, only one direction of building the lattice is described and used. But the other way around is also theoretically possible: Starting with a node that represents the full set, as a binary encoded string consisting of only 1's, the actions then represent only transitions from $1 \mapsto 0$.

The tree policy uses UCT to select which path it should take at each iteration. That's why the Subset Lattice Optimization uses the same exploration parameter C_p as the standard UCT algorithm. The default policy of this algorithm chooses a random action until a terminal state is reached. In the following, a special feature of the tree is introduced. Until this point, the tree could be regarded as a binary encoded subset lattice, but with no obvious terminal nodes, yet. This problem is solved by adding a **stopping condition** to the list of possible successor actions of each node. This extra action leads to a node that has the same binary representation as its parent node. However, it has a flag that indicates that it acts as a

terminal node. This concept doubles the number of nodes. If a random action is picked, the probability that a terminal node is reached next depends on the current level d_i and the chromosome length b:

$$p_{\text{terminal}} = \frac{1}{b - l_i + 1}$$

At its core, the base version of the Subset Lattice Optimization is very similar to UCT, but it has a major difference: The optimization does not search for the best action, the edge from the root that leads to the best successor. Instead, the entire state space is searched for the best node. That is why the best state together with its reward is maintained throughout the runtime of the algorithm. If this were not the case, a good state could be found, but not returned by the algorithm. This is justified by the asymmetric tree growth of UCT: The algorithm prefers to build the tree in promising regions. With the exploration part of the formula, it is possible to find other states that might be better. However, the algorithm prefers to select the successor nodes with the best average reward. If a good state is found that can have bad neighbors, this might not necessarily increase the average reward of its parent node. Thus, the other region from before will be exploited. Optimization needs to find these best states. As a result of this situation, the resulting state of each simulation will be compared to the current best state and replaced if the reward is better.

The described behavior of the binary encoded subset optimization is summarized in form of pseudo code in Algorithm 2 below.

Algorithm 2 Basic optimization on a subset lattice

Output: The best found state s_{best}.

```
1:  function BasicSubsetLatticeOptimization(s_0)
2:      s_best ← null
3:      Δ_best ← −∞
4:      v_0 ← create root node from a state with size b, containing only 0's
5:      while in computation budget do
6:          v_l ← TreePolicy(n_0)
7:          Δ, s ← DefaultPolicy(n_l)
8:          Backup(v_l, Δ)
9:          if Δ > Δ_best then
10:             s_best ← s
11:             Δ_best ← Δ
12:         end if
13:     end while
14:     return s_best
15: end function
```

4.2 Enhancements

In this section, two different enhancements of the basic Subset Lattice Optimization algorithm are introduced. These two improvements are integrated into the base version in a modular way. That means that the final algorithm has parameters that can control these modules and even deactivate them to a certain extent.

4.2.1 Transpositions

The first extension solves a problem that the basic variant has: the handling of transpositions. If a node of a tree is reached from multiple paths, it is called a transposition. This expression originates from the

game chess, where it means that different moves can result in the same game state. Generally considered in the domain of trees, a transposition is a node with multiple incoming edges, meaning that it could have more than one parent node. In subset lattices this is a very common circumstance, as it can be seen for example in Figure 4.2. The number of incoming edges of a node in a subset lattice can be calculated by the level it is in. Given the lattice base size b, a node in level l has l incoming and $(b-l)$ outgoing edges, because each of the l 1's in a node can have a different origin. Consequently, each node that is at a level ≥ 2 is a transposition. Only the nodes at the first two levels and the terminal nodes are not regarded. Thus, in a subset lattice with base size b, the total amount of transpositions is 2^b-b-1.

The basic version of the proposed Subset Lattice Optimization does not support transpositions at all. This is because a tree is generated the same way as MCTS: Each node is generated by randomly selecting an unused action that will lead to the child node that is created at this point. With that, multiple nodes could exist that have the same state (here: binary string representation). The states of other nodes are not considered. Usually, transpositions are implemented with the help of transposition tables which are basically like the hash map data structure. The transposition tables have the task to map each state to a node.

The major challenge with transpositions is the adaption of the selection process. The plain UCT formula, which is responsible for selecting the next node in the tree policy, does not consider transpositions. That means that the rewards of a node (possibly a transposition) are not available for nodes outside of the path the tree policy took so far in an iteration. The UCT formula has to be adapted to use additional information, which is available in the tree, but not used so far. For example, it could be an advantage for a node to know that one of its children or even grandchildren is a transposition and which information it collected so far. This could be used in order to improve the selection of the next child in the tree policy. The work of two papers ([CBK08] and [SCM12]) from different authors dealing with this problem is used in this thesis and is briefly presented below.

Both papers view the tree as a rooted directed acyclic graph (DAG). This assumption also holds for the subset lattices which are used in the proposed optimization algorithm. A DAG consists of a set of nodes and directed edges. The root of the graph is, like in a tree, the first element or starting node which has no incoming and only outgoing edges. By definition, a DAG does not contain any cycles. All those properties are also shared with the tree-like data structure that is used by the Subset Lattice Optimization.

The first solution of handling transpositions offered by both papers is called the *simple way* where no change of the UCT selection is made. Nevertheless, transpositions are detected and managed with the transposition tables mentioned above. However, both authors came to the conclusion that the simple way is better than ignoring transpositions completely.

Childs et al. [CBK08] suggest three variants based on each other that try to improve the basic UCT approach. The notation of the formulas differs from the basic UCT formula in Section 3.4 because of the assumption that with transpositions a node can have multiple parents. The focus is on which edges are selected in each node instead of looking directly for a successor node. Let $A(v)$ be the set of outgoing edges of a node v, $Q_{v,a}$ be the average reward of v when the edge a was selected and $N_{v,a}$ the total number of times a has been selected in v. Additionally, N_v stands for the total number of times v has been visited and $g(v,a)$ is the node that is reached after the edge a was selected in v. Given these definitions, the three suggestions of the authors for transposition friendly formulas are:

- **UCT1:** At the first approach, the selection is made as the simple way. The difference at UCT1 is that its value is calculated independently of where the node is in the tree. This is in contrast to

the simple way where the calculation depends on the path the selection policy takes. The selection policy is given by

$$\pi_{\text{UCT1}}(v) = \arg\max_{a \in A(v)} \left(Q_{v,a} + C_p \sqrt{\frac{2 \ln N_v}{N_{v,a}}} \right)$$

- **UCT2:** The next UCT formula uses more information with $Q_{g(v,a)}$ which considers the node that is reached from the current node through an edge a. This provides at least as much information as UCT1. If $g(v,a)$ is a transposition that was visited before, this will result in a more accurate estimate. The formula is denoted by

$$\pi_{\text{UCT2}}(v) = \arg\max_{a \in A(v)} \left(Q_{g(v,a)} + C_p \sqrt{\frac{2 \ln N_v}{N_{v,a}}} \right)$$

- **UCT3:** Finally, UCT3 goes even one step further in the calculation of a UCT variant. Where UCT2 takes child nodes into account, UCT3 integrates the information of all descendants into the calculation. The formula is defined with the help of a recursive definition of the average reward Q^*_s:

$$\pi_{\text{UCT3}}(v) = \arg\max_{a \in A(v)} \left(Q^*_{g(v,a)} + C_p \sqrt{\frac{2 \ln N_v}{N_{v,a}}} \right)$$

$$Q^*_s = \sum_{a \in A(s)} \frac{N_{s,a}}{N_s} Q^*_{g(s,a)}$$

The next paper, which deals with transpositions and is presented in this work, was written by Saffidine et al. [SCM12]. In this, the authors describe a more general and intuitive approach that can be configured individually. It is called *Upper Confidence bound for rooted Direct acyclic graphs* (UCD) and acts as a mathematical framework for handling transpositions. The UCD algorithm is used in the Subset Lattice Optimization of this thesis.

At this point, the notation which is used in the UCD-paper [SCM12] is introduced. First of all, x is an object that can either be an edge or a node of a rooted DAG. The set of children of an object is defined by $c(x)$. If x is a node, then this is the set of all outgoing edges. If x is an edge going to the node y, then $c(x) = c(y)$ are the outgoing edges of y. Furthermore, $b(x)$ denotes the "*siblings*" of an object: If an edge x has its origin in the node y, then $b(x) = c(y)$ is the set of outgoing edges of y. In addition to these graph traversing notations, there are a few values attached to an object: $\mu(x)$ stands for the mean reward at an object x, $n(x)$ is the number of times x has been visited so far and $p(x)$ denotes the total number of visits of the children of x. If an object has been visited before it is created in the graph, $\mu'(x)$ stands for the mean reward before it is added as a child. Similarly, $n'(x)$ denotes the number of visits the node has before it is added as a child. These scenarios seem unusual but are the case if transpositions occur.

These general notations result in the possibility that values can be either stored in edges or in nodes which is an important feature of the framework. This is also used in the algorithm where the given formulas below can only be possible if the values are stored in the edges. The overall approach of UCD is that the different components of the original UCT-formula are separated and adapted to make better use of transpositions. This is achieved by calculating each part of the tree with the help of information up to a certain depth of the tree. The depths are denoted by d and can be configured individually for

each component of the formula. First, the average reward is replaced with an *adapted score* $\mu_d(e)$ which uses the number of visits and the average reward of children up to a given depth d. The next component is named *move exploration* $n_d(e)$ by the authors and has the task to determine the number of visits at an edge and its descendants up to a certain depth d. The last part, which is adapted, is the parametric *origin exploration* $p_d(e)$ which counts the total number of visits of the algorithm up to a given depth d. In addition, the values $n'(e)$ and $\mu'(e)$ are necessary for the framework because the formulas can take advantage with this further information. Finally, the components are recursively defined by the following equations:

$$
\begin{aligned}
\mu_0(e) &= \mu(e) \\
\mu_d(e) &= \frac{\mu'(e)\, n'(e) + \sum_{f \in c(e)} \mu_{d-1}(f)\, n(f)}{n'(e) + \sum_{f \in c(e)} n(f)} \\
n_0(e) &= n(e) \\
n_d(e) &= n'(e) + \sum_{f \in c(e)} n_{d-1}(f) \\
p_d(e) &= \sum_{f \in b(e)} n_d(f)
\end{aligned}
$$

The framework also supports the ability to remove the boundary depth for the adapted score. Thereupon, the corresponding d can be set to ∞, thus, all successors are considered in the calculation. The formula for the adapted score has to be changed in this case to

$$
\mu_\infty(e) = \frac{\mu'(e)\, n'(e) + \sum_{f \in c(e)} \mu_\infty(f)\, n(f)}{n'(e) + \sum_{f \in c(e)} n(f)}
$$

All components put together result in the UCD-formula which is used as the tree policy of the Subset Lattice Optimization:

$$
\pi_{\mathrm{UCD}}(x) = \operatorname*{arg\,max}_{e \in c(x)} \left(\mu_{d_1}(e) + C_p \sqrt{\frac{\log p_{d_2}(e)}{n_{d_3}(e)}} \right)
$$

A simpler notation for UCD is given by $(d_1, d_2, d_3) \in \mathbb{N}^3$ which represents the three depth parameters. According to Saffidine et al. [SCM12], this framework can emulate the UCT policies from [CBK08]. The values for the depth parameters to achieve the corresponding policies are:

- $(1,1,1) \mapsto$ Simple way
- $(0,0,0) \mapsto$ UCT1
- $(1,0,0) \mapsto$ UCT2
- $(\infty,0,0) \mapsto$ UCT3

This short notation, which is the result of the modular equations defined above, is only possible because the values are stored in the edges, according to the authors. In both presented papers, the authors came to the conclusion that considering transpositions improves the algorithm significantly. They only tested the algorithms in games, primarily in Go. In [CBK08], the UCT2 value is considered to be the best choice, while UCT3 performs better, but has the drawback of a much longer computing time. Which values for the UCD framework are best for the optimization domain will be investigated in experiments (see Chapter 6).

4.2.2 Rapid Value estimation

The second addition to the basic Subset Lattice Optimization is the so-called Rapid Value Estimation (RAVE), first described by Gelly and Silver in [GS07]. RAVE belongs to the All-Moves-As-First (AMAF) heuristics of MCTS algorithms, specifically developed for the game of Go. In the standard MCTS algorithm, only the nodes of the path of the tree policy are updated with the simulated reward. The main contribution of AMAF algorithms is that many other nodes are also updated in the course of the backup step. This has the goal to increase the confidence of choosing the correct next action. Originally, this technique was used in the gaming domain as a *"warm up"*-phase to quickly fill the tree with good values at the beginning of the algorithm.

In [HP09], Helmbold and Parker-Wood give an overview of the most common AMAF-methods as of 2009. In addition, the authors compare the performance of these different methods in the game of Go and conclude that RAVE is one of the methods with the most improvements compared to the basic UCT algorithm. Only the permutation-AMAF heuristic, which updates more nodes than the other AMAF methods in the paper, came to slightly better results. Next to the basic RAVE algorithm, other variants of this heuristic exist as well, for example *KillerRAVE* or *PoolRAVE*.

The reason why RAVE was used out of all AMAF methods is that it was also successfully used in the FUSE algorithm in [GS10] [2]. Most AMAF methods have been usually used in the gaming domain. The FUSE algorithm, on the other hand, tends to belong to the domain of optimization. In addition, after careful research, no works are known that use AMAF heuristics in trees with considering transpositions. In [SCM12], the authors suggest in an outlook on possible future work that using RAVE in an UCD algorithm could increase its performance. Because the lattice contains many transpositions, aggressive techniques that update more nodes, such as the permutation-AMAF, are not considered. It has to be investigated first, which influences the different AMAF heuristics have at trees with different amounts of transpositions. Until then, the basic RAVE algorithm is chosen as AMAF technique in the Subset Lattice Optimization.

The functionality of RAVE and most other AMAF heuristics in general can be divided into two subtasks. Firstly, the heuristic has to define how the backup step is altered to update more nodes and, secondly, how this possibly additional information is used in the tree policy:

- **Backup step:** The additional nodes, that are updated in the backpropagation step of a general MCTS, are defined by the path the tree policy has taken. Every action, that is in the path to the newly created node, is considered and updated even if the action does not belong to the path itself. All actions of the tree are taken into account if they occur in any subsequent subtree. Thus, the name All-moves-as-first was given to the heuristic. To achieve this backup step in RAVE, two additional fields are attached to each edge: $\mu_{\text{RAVE}}(e)$ contains the current average reward and $m(e)$ stands for the number of visits of edge e, with both variables being propagated using the RAVE algorithm only.
  In the context of the Subset Lattice Optimization, an action corresponds to flipping a bit at a specific position in the binary string that represents the current state. With RAVE, each edge of the current tree is updated where a 1 was chosen at the same positions as in the path of the tree policy.

- **Tree policy:** The special feature of RAVE is that its influence on the tree policy is decaying with an increasing number of iterations. This is achieved by a linear combination of the RAVE policy and the tree policy from before. The notations of the RAVE algorithm are based on the UCD notations introduced in Section 4.2.1. The tree policy before integrating an AMAF technique

[2] The FUSE algorithm is focused in Section 5.2.

is denoted by π, which chooses an edge after applying a function u on each edge, such that $\pi(x) = \arg\max_{e \in c(x)}(u(e))$. In the case of the Subset Lattice Optimization with transpositions, this corresponds to the UCD formula mentioned above and is denoted here by u_{UCD}. Similarly, the RAVE formula u_{RAVE} is based on the UCT-formula, but only uses the RAVE specific values mentioned above. Furthermore, $M(e) = \sum_{f \in b(e)} m(f)$ is the sum of all the visits the parent node of e has passed through. The value β is calculated for every edge and depends on the *equivalence parameter* k which denotes after how many visits RAVE and UCD have equal weight. Let the final tree policy be π_{RAVE}, then this value is calculated with the following equations:

$$u_{\text{UCD}}(e) = \mu_{d_1}(e) + C_p \sqrt{\frac{\log p_{d_2}(e)}{n_{d_3}(e)}}$$

$$u_{\text{RAVE}}(e) = \mu_{\text{RAVE}} + C_p \sqrt{\frac{\ln M(e)}{m(e)}}$$

$$\beta(e) = \sqrt{\frac{k}{3n(e) + k}}$$

$$\pi_{\text{RAVE}}(x) = \arg\max_{e \in c(x)} \Big(\beta(e) u_{\text{RAVE}}(e) + (1 - \beta(e)) u_{\text{UCD}}(e) \Big)$$

These definitions imply that the Subset Lattice Optimization will be complemented with the equivalence parameter k. The nature of the linear combination in the RAVE heuristic indicates that, if $k = 0$, RAVE is not used at all.

4.3 Outline

This last section gives a summary and an overview of the Subset Lattice Optimization which is the central approach of this thesis. The introduced algorithm combines concepts and algorithms from different works that are based in different domains. The chapter describes the algorithm in all its details, starting with the concept of binary string representations of states that originates in Genetic Algorithms. Next, the basic algorithm was introduced which is essentially the execution of a Monte Carlo Tree Search that ultimately builds a subset lattice. In this search, each state is represented by a binary string that stands for a subset.

In Section 4.2 two enhancements are presented that eventually lead to the tree policy that is used in the final version of the algorithm. Mathematically, this is denoted by π_{RAVE} which includes the UCD policy π_{UCD} of Section 4.2.1 to an extent. The functionality of the final algorithm is based on the notation introduced in regard of the UCD-paper [SCM12]. From this source, the approach has also been adopted that all information is stored in the edges. The final tree policy is designed in a way that the RAVE part of the algorithm can be turned off completely and both, the UCD and RAVE part, can be configured individually.

Finally, an overview of all parameters of the algorithm is given in Table 4.2. The first column shows the variable that represents the parameter. The second column lists the type of the parameter, noting that all values must be ≥ 0. In the third column, the given name of each parameter is presented and the last column shows from which algorithm the parameter originates.

Parameter	Type	Name	Origin
I	Int	Number of iterations	MCTS
C_p	Float	Exploration parameter	UCT
d_1	Int	Adapted score depth	UCD
d_2	Int	Move exploration depth	UCD
d_3	Int	Origin exploration depth	UCD
k	Int	RAVE equivalence parameter	RAVE

Table 4.2: Parameter overview of the subset lattice optimization.

5 Related Work

The following chapter deals with previous works that address aspects related to this thesis. First of all, an attempt is made to give a general classification of the topics that the proposed Subset Lattice Optimization relies on. Subsequently, the conclusions of the most related papers are presented in separate sections.

5.1 General Classification

In Chapter 4 the Subset Lattice Optimization was introduced. Theoretically, it uses concepts from different fields. However, at its core, it is still a Monte Carlo Tree Search. It uses a specific state representation based on Genetic Algorithms. Furthermore, the solution space itself is not a tree like in the standard MCTS algorithm, instead, it is a lattice that is mathematically defined with the help of subsets. One of the other main differences is that the Subset Lattice Optimization operates in the domain of Global Optimization. Apart from that, Monte Carlo Tree Search is usually executed in the gaming domain. An introduction in all the related fields was given in the individual Section 2.1, 2.2 and 3. At these points, the previous works on the respective topics were also presented.

Because of its core algorithm concept, the Subset Lattice Optimization can be classified as a Monte Carlo Tree Search, with the peculiarity that it is supposed to be executed on Global Optimization problems. The number of works in this area is manageable. In [Bro+12], Browne et al. give a short outline of non-gaming usages of the Monte Carlo Tree Search. From these presented works, the ones that can be classified as optimization methods were categorized as follows:

- *Combinatorial Optimization:* This area is a broad and general area of optimization. Often, the aim is to search for a combination of objects. The solutions and the solution candidates are then expressed using combinatorics, for example, with permutations of objects. Such is the case of the popular Traveling Salesman Problem, which has been mentioned in Section 2.1. The authors of [RTC11] successfully solved this problem for small numbers of objects with the help of the modified *Nested* Monte Carlo Tree Search [1].

- *Scheduling problems:* In scheduling problems, different tasks are usually assigned to multiple individuals or objects that process these tasks. The goal of algorithms that deal with this kind of problems is to minimize the total time after all tasks are fulfilled. Cazenave et al. have developed a Nested Monte Carlo Tree Search algorithm, that minimizes the waiting times on bus stops by regulating bus tours [CBP09].

- *Feature Selection:* The authors of [GS10] created a variation of the Monte Carlo Tree Search that has the task to find the optimal set of features. Their work will be presented in the next section 5.2, because of its similarities with this thesis.

In addition to these mentioned papers, there are a few other papers that have dealt with the topic after the survey from Browne et al. [Bro+12] was released. For instance in [Mun14], the author proposes a hierarchical Monte Carlo Tree Search as an optimization algorithm. The hierarchy which is described in this algorithm stands for several instances of a Monte Carlo Tree Search that depend on each other in several levels of the tree.

[1] A Nested Monte Carlo Tree Search has the special feature of saving the best sequence of moves of any level of the search tree. Further information can be found for example in [RTC11].

Two enhancements were introduced for the Subset Lattice Optimization. Firstly, the improved handling of transpositions with the UCD algorithm [SCM12] that is based on the work of Childs et al. in [CBK08]. Secondly, the RAVE method was integrated, which was developed by Gelly and Silver in [GS07].

The main contribution of this thesis is the proposed Subset Lattice Optimization itself with its Monte Carlo Tree Search in a discrete state lattice. Additionally, after thorough research, it can be claimed that the UCD algorithm is first being tested in an optimization domain. Furthermore, the UCD algorithm is extended with the RAVE method, which was proposed to be investigated in future works by Saffidine et al. in [SCM12] and is first developed and tested in this thesis.

5.2 Feature UCT Selection

Gaudel and Sebag introduce the Feature UCT Selection (FUSE) in [GS10]. As the name suggests, this algorithm was specifically designed for the domain of Feature Selection. This is typically used in the area of machine learning. Firstly, a model is generated from training data that describe it with a set of features. The Feature Selection tries to find the best set of features so that the error that the model predicts new data is minimized. Thus, Feature Selection can be seen as an optimization task.

The state space on which the FUSE algorithm operates consists of all feature combinations. This is achieved by formalizing a lattice of all subsets of the given features. FUSE is at its base a UCT algorithm, which leads to an iterative creation of the lattice based on random simulations. For each subset, a *stopping action* is added, which results in a terminal node representing the same subset. The algorithm is more complex than this short sketch, but the relevant properties are shown. Additionally, the algorithm has integrated RAVE scores to find the best features.

This description shows the similarities with the Subset Lattice Optimization. The idea of creating a lattice of subsets is also given in this thesis. Furthermore, the execution of a Monte Carlo Tree Search algorithm on this lattice is also a common feature of both algorithms. In addition to that, the two algorithms are executed in an optimization domain because the Feature Selection can be seen as an optimization problem.

Despite the mentioned similarities, there are still fundamental differences between the two algorithms. For instance, although the FUSE algorithm tries to solve an optimization problem, it was primarily developed to find the best feature subset. The Subset Lattice Optimization, on the other hand, is developed with the goal to solve *any* optimization problem. It is irrelevant for this algorithm which content the subsets have because it utilizes only the binary encoding that represents the individual subsets in the lattice. This binary encoding, which is based on ideas of Genetic Algorithms, is what makes the proposed algorithm unique.

5.3 Evolutionary Algorithms and Monte Carlo Tree Search

In the literature exist very little research that combines Monte Carlo Tree Search and Evolutionary Algorithms. At this point, two works are presented that share this property, although they are very different to the approach of this thesis.

The authors of [BS13] introduce a variation of Monte Carlo Tree Search that was enhanced with Genetic Programming. This algorithm was designed to work in the gaming domain, specifically, it was tested on the game of Reversi. In contrast to the standard Monte Carlo Tree Search, this variation of the

algorithm has a unique simulation strategy. These random play-outs are replaced with learned functions from Genetic Programming. This sub-category of Evolutionary Algorithms creates function trees that indicate under what circumstances a player is most likely to win. In their paper, the authors propose own genetic operators that help to improve the algorithm.

Havránek creates in his master thesis [Hav15] a Genetic Algorithm that is enhanced with a Monte Carlo Tree Search. In contrast to all other mentioned works in this chapter, this algorithm is at its core a Genetic Algorithm. The Monte Carlo Tree Search is executed multiple times within specially developed genetic operators. Thus, the Genetic Algorithm is highly modified while the Monte Carlo Tree Search stays in its basic version with the UCT policy. The algorithm was mainly tested on the Traveling Salesman Problem.

6 Experiments on Test Functions

In this chapter, the performance of the Subset Lattice Optimization, proposed with this thesis, is investigated. For this purpose, the algorithm will be compared with a simple Genetic Algorithm on different test functions. The first section of this chapter describes the general design of the experiments carried out and the second section gives an overview of all test functions that were used. After that, the experimental setup is described. The following section contains the experiment results which are summarized in a conclusion in the last section of this chapter.

6.1 Experiment design

The goal of the experiments is to determine how good the proposed binary Subset Lattice Optimization works in artificial scenarios. In the field of Global Optimization, the algorithms are often executed on test functions (in the literature also called benchmark functions). All used test functions of the experiments will be explained in Section 6.2 in detail.

In order to evaluate the performance of the Subset Lattice Optimization, it is compared with a simple Genetic Algorithm. The reason for using a Genetic Algorithm as a Global Optimization method reference is that the SLO has its core concept based on Genetic Algorithms. This means that the binary string representation of states is identical in both algorithms. Each test function is considered separately. However, the strategy on how the experiments are carried out is the same in each case. The performance of an algorithm can be defined by its efficiency, meaning how fast an algorithm can find a solution. Alternatively, if this solution wasn't found within a given period of time, then the efficiency would be determined by how close the algorithm came to the solution. For this to measure, the expression of an evaluation is introduced: Each time the objective function is called with a candidate solution, this execution is called an *evaluation*. An optimization method tries to improve its candidate solution with increasing number of evaluations. [1] According to [Cha+10], there are three major influences that decrease the run time of an optimization algorithm:

- smaller number of evaluations
- shorter evaluation
- parallelism of multiple evaluations

In this thesis, only the number of evaluations is considered as a measure of effectiveness because this measure is independent of the algorithm, the system or the implementation.

The Genetic Algorithm which is used in the experiments is a basic one with selected operators. The functionality of this algorithm was described in Section 2.2 in detail. It was mentioned at this point, that many different genetic operators exist in the literature. It would exceed the course of this work to find the best operators for any given test function. This is why the operators are fixed at this point. In [CBM11], the authors study and compare a few operators. The concluding operators of this work are used and fixed in the Genetic Algorithm of this thesis:

- Selection: Tournament
- Reproduction: Single point Crossover

[1] This was a short introduction of evaluations. More on this topic of Global Optimization was given in Section 2.1.

In addition, the Genetic Algorithm uses an elite operator and a binary flipping mutation operator, both of which are common in the literature. The mutation and reproduction rates are always kept constant throughout an execution. A so-called Generational Replacement Model (GRM) is used in the reproduction operator, which has the task of replacing the whole population in each generation. In [CBM11], the authors also use a GRM. The parameters which are used in the Genetic Algorithm of this thesis are presented in Table 6.1. The task of the experiments is to find the best values for the listed parameters for each test function, that results in the best efficiency of the algorithm in each case.

Variable	Type	Name
N_{Pop}	Int	Population size
N_{Gen}	Int	Number of generations
p_C	Float	Crossover probability
p_M	Float	Mutation probability
p_E	Int	Number of elites per generation

Table 6.1: Parameter overview of the Genetic Algorithm

The parameters of the Subset Lattice Optimization are similarly outlined in Table 4.2. In addition to these, both algorithms have a parameter N_{Gene} which stands for the number of bits each gene of a chromosome is encoded. Because this encoding is the same for both algorithms, it is fixed for each test function and, thus, not relevant for finding the best algorithm configuration. Furthermore, each test function can be defined over different numbers of dimensions. This *dimensionality* is denoted by the parameter D. The higher this dimensionality is set, the harder it gets for an algorithm to find the optimal solution. For instance, at the two algorithms to be investigated, this parameter directly increases the state size b by adding a gene of size N_{Gene} to the chromosome.

The goal of each experiment in this chapter is to find the best possible parameters for both algorithms. A number of maximum evaluations I is given which defines an upper limit of evaluations of an algorithm. For the Genetic Algorithm, this parameter was defined as N_{eval}. However, both parameters mean the same in this context: If this limit is reached, the proximity to the optimal solution is decisive for the effectiveness of the currently tested parameter configuration. The parameters to investigate are given in the Tables 4.2 and 6.1. Given a set limit maximum evaluations $I = N_{\text{eval}}$ for each test function, then all experiments follow the same pattern:

- Firstly, the parameter configuration of the Genetic Algorithm is optimized. This is carried out at the beginning because the used framework allows only to retrieve results after each generation. Each generation has multiple evaluations which depend on the population size. Thus, it is not guaranteed to reach the exact limits of maximum evaluations. But, the Subset Lattice Optimization can adapt to these limits in order to better compare the both algorithms.
- Secondly, the best values for the parameters of the Subset Lattice Optimization are searched.
- Finally, the results of the first two sub-experiments are compared directly. After that, the results are interpreted and reasons for how the respective algorithms performed are searched for.

6.2 Test functions

If a new optimization method is introduced or an already existing method is modified, then it is common to test this algorithm on a set of benchmark functions. Typically, this algorithm is compared to a reference algorithm in such a way that conclusions on its performance can be deduced. After an algorithm

has successfully completed benchmark tests, it will be applied to a real-world problem. There are many advantages of why test functions are used to make statements about an algorithm first. For instance, the benchmark functions are typically much faster to evaluate, which opens the possibility to run multiple experiments in order to optimize the algorithm. Additionally, the optimal solution, as well as the function landscape, are typically known for a test function. With this information, the algorithm can be analyzed and the parameters optimized which is not necessarily possible in real-world scenarios.

Test functions can have several different properties that describe them or the solution landscape. Among the most popular of these properties is modality. A function is called multimodal if it has many local optima. This can be difficult for an algorithm because it can converge to a local optimum instead of exploring the whole solution space in order to find the global solution. Further attributes, for describing the solution landscape, are basins (for minimization) or plateaus (for maximization) which are characterized by relatively large flat areas surrounded by steep ascends or descends. An optimization algorithm might not know in which direction it should look for better solutions. Similarly, a further popular example is valleys which are also encircled by steep areas. However, they have a general direction instead of being flat. These property names are analogies and expressions of mountain regions. Beyond that, many test functions consist of more than two dimensions. A function is called *scalable* if it can be expressed with an arbitrary number of dimensions. In experiments, scalable functions have the advantage that it could be examined how the algorithms can handle the more complex functions. It is possible that an optimization method can perform better than another algorithm on a function with few dimensions, but performs worse on the same function with more dimensions. Test functions can also be classified into single and multi-objective functions which depicts if a function returns one or many values [JY13]. In this thesis, only single objective functions are considered because the Subset Lattice Optimization was designed to only support this kind of problems.

There are many different test functions available for experiments. In [JY13], the authors present 175 benchmark functions, which is a complete list of all available functions in the literature as of 2013, according to the authors. There is no fixed set of test functions where experts and scientists agree upon. That's why the number of different test functions varies from work to work, from a few (2-3) up to a few dozen (20-60). Jamil and Yang suggest in their survey that the goal should be to select a set of test functions that is diverse and covers as many different properties as possible of which the best known are described above [JY13].

Each test function has a limited search space, which is the only constraint on the functions. This is denoted by boundaries for all given dimensions. The test functions that are used in the experiments of this thesis are the following:

- f_1: **Sphere** function. This simple function is often used in optimization benchmarks (for example in [DM01], [SSM15], [MSB91] or [CBM11]). It is bowl-shaped, scalable and symmetric. An optimization algorithm should have no problem in finding the only global optimum. Furthermore, each algorithm should converge relatively fast compared to the other test functions. The function is defined by:

$$f_1(\boldsymbol{x}) = \sum_{i=1}^{D} x_i^2$$

  The solution space of this convex function is limited by the constraint $\forall x_i \in \boldsymbol{x} : -5.12 \leq x_i \leq 5.12$. These boundaries are based on the benchmarks of [SSM15], [MSB91] and [CBM11]. The function has no local optima and the only global optimum is at $\boldsymbol{x}_{\min} = (0, 0, ..., 0)$ with $f(\boldsymbol{x}_{\min}) = 0$. Figure 6.1 shows a plot of a section of the Sphere function in a two dimensional solution space.

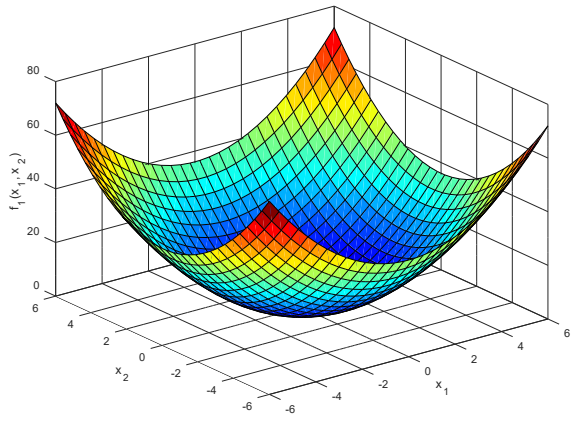


Figure 6.1: Sphere function in a two dimensional solution space ($D = 2$).

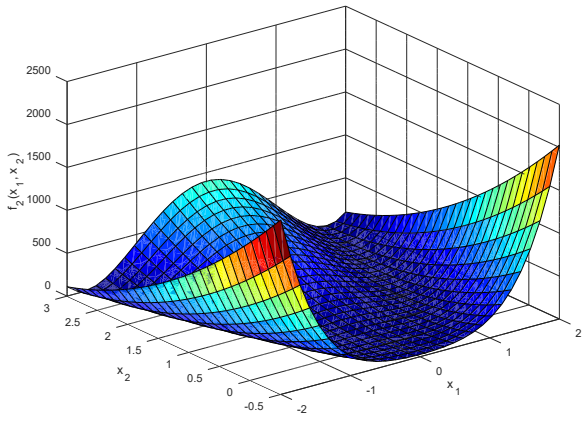


Figure 6.2: Rosenbrock function in a two dimensional solution space ($D = 2$).

- f_2: **Rosenbrock** function. The rosenbrock function, also called banana function, is one of the most used test functions (for example in [SSM15], [DM01], [CBM11] and [Cha+10]). The scalable function is denoted by:

$$f_2(\boldsymbol{x}) = \sum_{i=1}^{D-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right]$$

  The boundaries of the search area are $\forall x_i \in \boldsymbol{x} : -2.048 \leq x_i \leq 2.048$. The function has no local optima, but a single global minimum that is located in a narrow valley which is has the form of a parabolic curve (hence the name banana function). Algorithms should easily find this valley, but the challenge is to navigate in the valley towards the optimum. That is why many algorithms do not converge fast and why this test function is one of the more difficult to solve problems. The optimum is at $\boldsymbol{x}_{\min} = (1, 1, ..., 1)$ with $f(\boldsymbol{x}_{\min}) = 0$. Figure 6.2 shows the visualization of the Rosenbrock function for $D = 2$.

- $f_3(\boldsymbol{x})$: **Rastrigin** function. The next function to be investigated in the experiments is also very known. It is used in the experiments in [SSM15], [MSB91], [CBM11], [DH12] and [DM01]. The function is bounded on each dimension with $\forall x_i \in \boldsymbol{x} : -5.12 \leq x_i \leq 5.12$ and is defined by:

$$f_3(\boldsymbol{x}) = 10\,D \sum_{i=1}^{D} \left[x_i^2 - 10\cos(2\pi x_i) \right]$$

  This scalable definition was introduced by Mühlenbein et al. in [MSB91]. The special feature of the Rastrigin function, which makes it hard to solve, is that it has many evenly distributed local minima. The single global minimum is given at $\boldsymbol{x}_{\min} = (0, 0, ..., 0)$ with $f(\boldsymbol{x}_{\min}) = 0$. A part of the benchmark function for the case $D = 2$ is plotted in Figure 6.3.

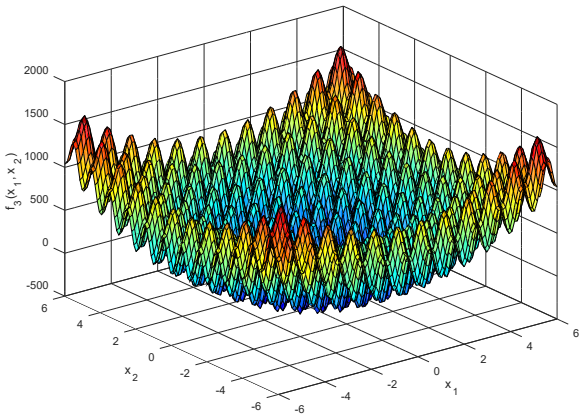


Figure 6.3: Plot of a part of the Rastrigin function in a two dimensional solution space ($D = 2$)

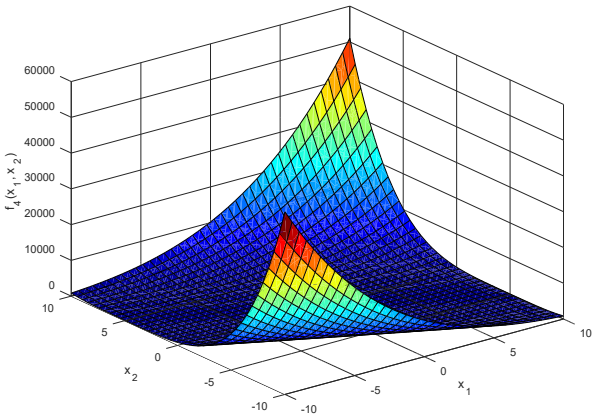


Figure 6.4: Plot of a part of the Zakharov function in a two dimensional solution space ($D = 2$)

- $f_4(\boldsymbol{x})$: **Zakharov** function. This benchmark function is scalable and has only one global minimum but no further local optima. It is used for example by Sidorov et al. in [SSM15]. The landscape of the function can be described as plate shape because of its large flat area around the coordinate origin that can be viewed as a plateau. The boundaries are denoted by $\forall x_i \in \boldsymbol{x} : -5 \leq x_i \leq 10$ and the Zakharov function itself is given by

$$f_4(\boldsymbol{x}) = \sum_{i=1}^{D} x_i^2 + \left(\frac{1}{2}\sum_{i=1}^{D} i\, x_i\right)^2 + \left(\frac{1}{2}\sum_{i=1}^{D} i\, x_i\right)^4$$

  The minimum is located at $\boldsymbol{x}_{\min} = (0, 0, ..., 0)$ with $f(\boldsymbol{x}_{\min}) = 0$. In Figure 6.4 the visualization of a part of the function is shown in its two dimensional form.

- $f_5(\boldsymbol{x})$: **Easom** function. Out of all used test functions in the experiments, the Easom function is the only non-scalable test function. With $D = 2$ being constant, the function is only it defined in two dimensional space. It is denoted by:

$$f_5(x_1, x_2) = -\cos(x_1)\cos(x_2)\exp\left[-(x_1 - \pi)^2 - (x_2 - \pi)^2\right]$$

  In the literature, the Easom function is used for example in [SSM15]. The limits are greater in comparison to the other test functions, they are $\forall x_i \in \boldsymbol{x} : -100 \leq x_i \leq 100$. This area is large in comparison to the region where the only global optimum is located, which is what makes the landscape of the function special. The global minimum is at $\boldsymbol{x}_{\min} = (\pi, \pi)$ with $f(\boldsymbol{x}_{\min}) = -1$ which is in a small neighborhood in the form of a steep drop, surrounded by a large flat area. Figure 6.5 reveals a part of the function within -10 and 10 for each dimension. The parts outside this area are not visualized, but can also be characterized as a flat area without any local optima.

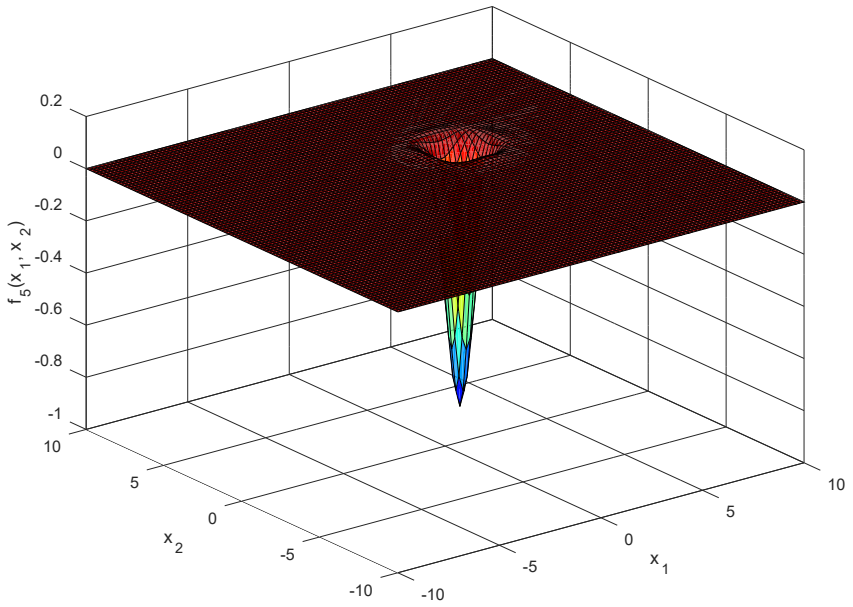


Figure 6.5: Plot of the Easom function in a sub region of the solution space

According to Mühlenbein et al. [MSB91], the Sphere and Rosenbrock functions are commonly used in benchmarks for genetic algorithms. On the other hand, the Rastrigin function is popular for testing "mainstream" optimization algorithms.

To close this section, Table 6.2 gives an overview of all test functions that are mentioned above and that are investigated in the experiments in the sections below.

f	Name	$\boldsymbol{x}_{\min}$	$f(\boldsymbol{x}_{\min})$	Boundaries
f_1	Sphere	$(0, 0, ..., 0)$	0	$-5.12 \leq x_i \leq 5.12$
f_2	Rosenbrock	$(1, 1, ..., 1)$	0	$-2.048 \leq x_i \leq 2.048$
f_3	Rastrigin	$(0, 0, ..., 0)$	0	$-5.12 \leq x_i \leq 5.12$
f_4	Zakharov	$(0, 0, ..., 0)$	0	$-5 \leq x_i \leq 10$
f_5	Easom	(π, π)	-1	$-100 \leq x_i \leq 100$

Table 6.2: Overview of all used test functions

6.3 Experiment setup

This section describes the methodology of how the experiments are carried out, as well as the parameters that are used for comparing the Subset Lattice Optimization with the Genetic Algorithm. Each of the test functions is the objective function, on which the algorithms are trying to find the optimum. For both algorithms, the number of dimensions is fixed $D = 2$ for every test function. All experiments have the same approach for all functions which is once described at this point.

At first, the best configuration of parameters is searched for the Genetic Algorithm. The values for the different parameters are selected based on the experiments of [DM01]:

- $N_{\text{Pop}} \in \{50, 100, 150, 200, 250, 300, 350, 400\}$

- $p_C \in \{0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95\}$
- $p_M \in \{0.001, 0.02, 0.04, 0.06, 0.08, 0.1, 0.18, 0.2\}$
- $p_E \in \{0, 1\}$

This results in $8 * 8 * 8 * 2 = 1024$ different configurations. Each configuration is tested 10 times to counter any inconsistent behavior caused by the random simulations of the algorithm's Monte Carlo part. Thus, the total number of tests is $10,240$. The number of bits used per gene N_{Gene} may be different for each test function. However, it is fixed when comparing the two algorithms on one function. In each experiment, a number of maximum evaluations N_{eval} is defined. This limit may be surpassed, due to the nature of the used Genetic Algorithm framework [2]. A single parameter configuration test is represented by a list of values the Genetic Algorithm achieved after each generation. In each generation, a specific number of evaluations of the objective function was executed, depending on the size of the population. The values of the test are the objective values of the objective function to be tested. They represent the negated fitness of the best individual in a population. Ideally, the objective values should decrease with a growing number of evaluations until the optimum is reached. The downward slope of the values is because all test functions in the experiments are minimization problems.

In order to determine the best parameter configuration for a test function, the arithmetic mean of all retrieved objective values is calculated. The quicker these values reach the optimum, the better. This is why the quality of a parameter configuration is expressed by the area under the curve (AUC) of the corresponding mean objective values. The AUCs of configurations with different population sizes are hard to compare directly. This is because the different population sizes lead to different numbers of evaluation the algorithm can provide. As a result, in each experiment, the objective values of the best configuration for each population are plotted in a figure. Eventually, only the configuration with the overall smallest AUC is taken over to the follow-up experiments of the respective test function.

The second part of an experiment deals with finding the best parameters for the Subset Lattice Optimization. This process is similar to the first part of determining the best Genetic Algorithm parameters. Gelly and Silver have tested high values in the range of powers to ten for the RAVE parameter k in [GS07]. On the other hand, the authors of [SCM12] have tested low values below ten for the UCD parameters d_1, d_2, d_3. Thereupon the following values for the respective parameters are being tested:

- $C_p \in \{0.1, 0.5, 0.75, 1.0, 1.5, 2.0\}$
- $d_1 \in \{0, 1, 2, 3, \infty\}$
- $d_2 \in \{0, 1, 2, 3\}$
- $d_3 \in \{0, 1, 2, 3\}$
- $k \in \{0, 10, 100, 1000\}$

The result is $4 * 4 * 4 * 5 * 6 = 1920$ different parameter configurations. Analogous to the first part, each configuration is executed 10 times which results in a total of 19200 separate tests. An upper limit I is defined, which denotes the number of iterations after which the algorithm terminates. After each 100 iterations, the current result of the algorithm is returned, which leads to a series of data points of objective values, as in the first part of an experiment. The procedure of determining which parameter values are the best for a test function is the same as in the first part for Genetic Algorithms: At first, the AUC of the mean values of each configuration is calculated and sorted. Secondly, the parameter configuration with the lowest AUC is accepted as the best setting for the particular test function. This is illustrated in

[2] As mentioned in Section 6.1, the number of evaluations depends on the population size.

a figure, that shows the best configurations.

The third and last part of each experiment contains the direct comparison between the Genetic Algorithm and the Subset Lattice Optimization on the particular test function. The parameters by which the algorithms are executed were determined in the sub-experiments before. Because in each iteration of the Subset Lattice Optimization exactly one evaluation of the test function is executed, the parameter I can be equalized with N_{eval}. Thus, the effectiveness of both algorithms can be compared to each other as mentioned in Section 6.1. The step size s denotes the number of iterations between the data points in the Subset Lattice Optimization. In each experiment, this value is adapted depending on the population size of the best Genetic Algorithm configuration. Every algorithm is executed 50 times on a test function. With the increased number of reruns, the effects of randomness are averaged. This ultimately leads to better significance. In addition to that, the number of maximum evaluations I and N_{eval} is increased compared to the setup experiments before. Thus, the objective is to present more accurate and significant results. The specific values of that are defined in each experiment.

Both algorithms are compared only at the number of evaluations, which are accessible by both algorithms. The results are shown in a figure of each experiment. An algorithm is more effective than the other if it converges faster to the optimum. After this comparison, the outcomes in every experiment are interpreted and reasons are sought for how the results were achieved.

6.4 Results

This section presents the results of the experiments. Each experiment follows the same pattern, which is explained in the previous section.

6.4.1 Experiment E_1: Sphere function

For the Sphere function, it is assumed that $N_{\text{Gene}} = 15$ and $I = N_{\text{eval}} = 2000$. Figure 6.6 shows the results of the determination of the best parameter of the Genetic Algorithm.

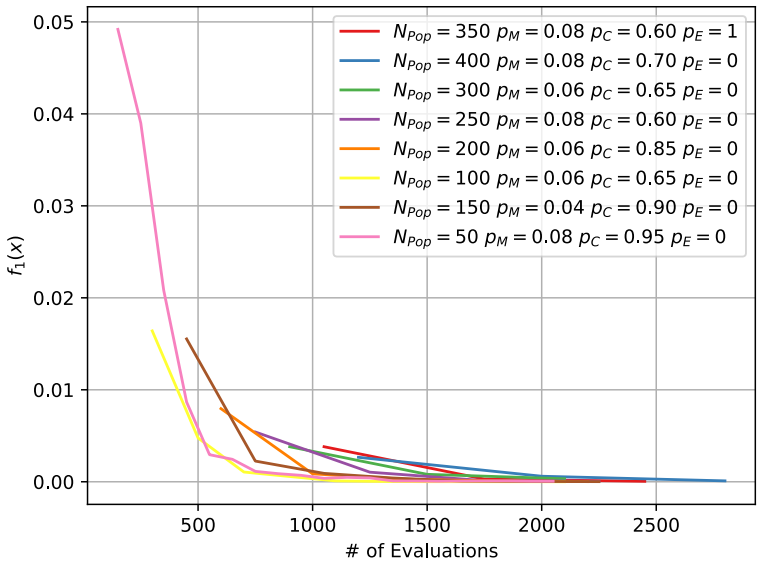


Figure 6.6: Comparison of the best parameters for the Genetic Algorithm for different population sizes on the Sphere function.

The results indicate that across all population sizes the mutation probability $p_M = 0.06$ or $p_M = 0.08$ were among the best configurations. Similarly, the number of elites is $p_E = 0$ achieved the better results than $p_E = 1$ in most cases. In contrast, no statement can be made about which values performed generally better for parameter p_C. Nevertheless, according to the results, the best parameters for the Genetic Algorithm are $N_{\text{Pop}} = 350$, $p_M = 0.08$, $p_C = 0.6$ and $p_E = 1$. These values are used for the comparison below.

The next part of this experiment tries to find the best parameter setup for the Subset Lattice Optimization. Figure 6.7 shows the eight best settings out of all results.

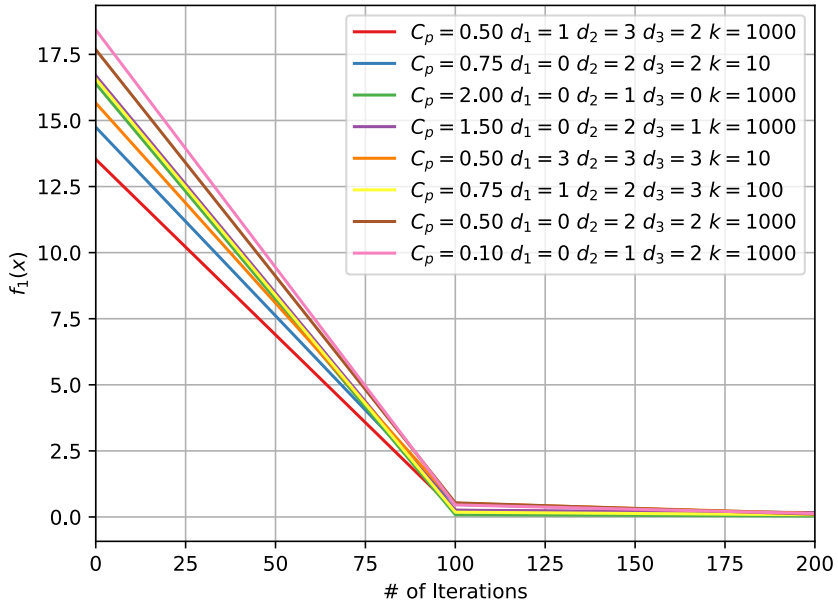


Figure 6.7: The best parameters settings of the Subset Lattice Optimization on the Sphere function.

Because the best settings all came near the optimum after a short time, the diagram is limited on the x-axis to 200 iterations. Although the figure only shows the best results, it illustrates the behavior of the other results: The configurations vary at the start of the algorithm and come closer to the optimum after 100 iterations. At 200 iterations, the mean objective values of the different settings are almost indistinguishable. The only conclusion that can be drawn from this sub-experiment is that the settings of the RAVE parameter $k = 1000$ tend to be among the better results than for other values of k. According to this tests, the best parameter configuration on f_1 is $C_p = 0.5$, $d_1 = 1$, $d_2 = 3$, $d_3 = 2$ and $k = 1000$, which is taken to the final part of this experiment.

Finally, Figure 6.8 shows the direct comparison of the Genetic Algorithm and the Subset Lattice Optimization with the parameters that were determined before. The limits denoting the maximum evaluations allowed are set to $I = N_{\text{eval}} = 10000$. Besides that, the step size of the Subset Lattice Optimization was set to $s = 350$.

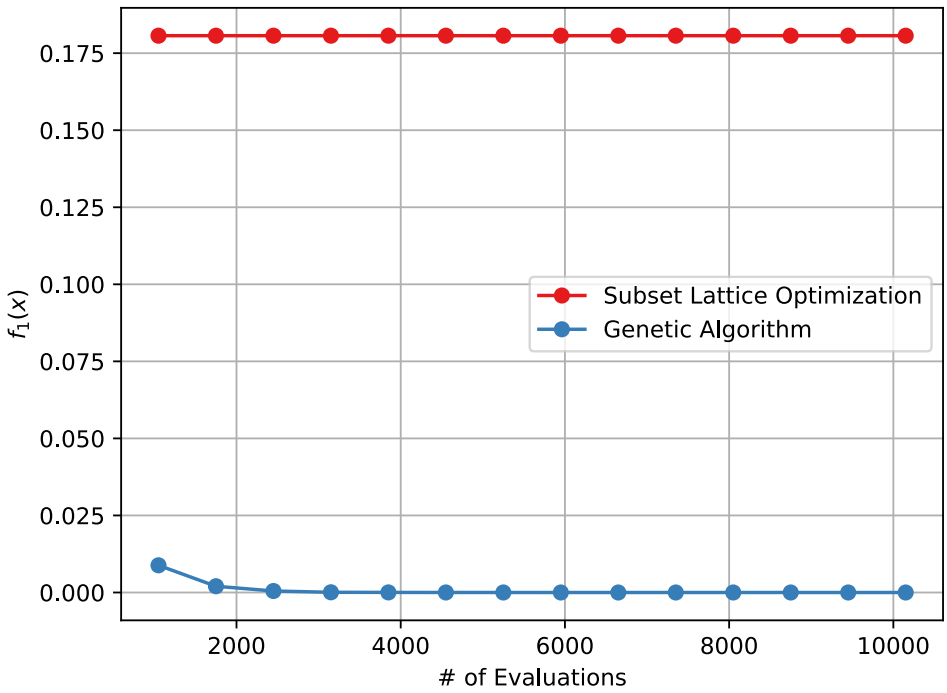


Figure 6.8: Comparison of the Genetic Algorithm and the Subset Lattice Optimization on the Sphere function.

The results clearly show that the Genetic Algorithm is more effective on the Sphere function than the Subset Lattice Optimization. On the one hand, the Genetic Algorithm quickly converges to the optimum. On the other hand, the Subset Lattice Optimization doesn't seem to find the optimum at all. It has to be mentioned, that the points in Figure 6.8 only show the arithmetic mean of all 50 runs of an algorithm. The standard deviation of the Genetic Algorithm is much smaller and converges to zero with increasing number of evaluations. In contrast, the standard deviation of the Subset Lattice Optimization is much higher and doesn't change much over time. This fact is resembled by the figure as well: Although the Subset Lattice Optimization is not as close to the minimum, it doesn't find a better solution. Only some sporadic algorithm executions achieve similarly good results as the Genetic Algorithm, but the average is worse.

6.4.2 Experiment E_2: Rosenbrock function

The second experiment is executed on the Rosenbrock function. As in the experiment before, the number of bits in a gene is denoted by $N_{\text{Gene}} = 15$. But in contrast to E_1, the number of maximum evaluation is set to $I = N_{\text{eval}} = 5000$. Figure 6.9 presents the best configurations for each tested population size of the Genetic Algorithm on the Rosenbrock function.

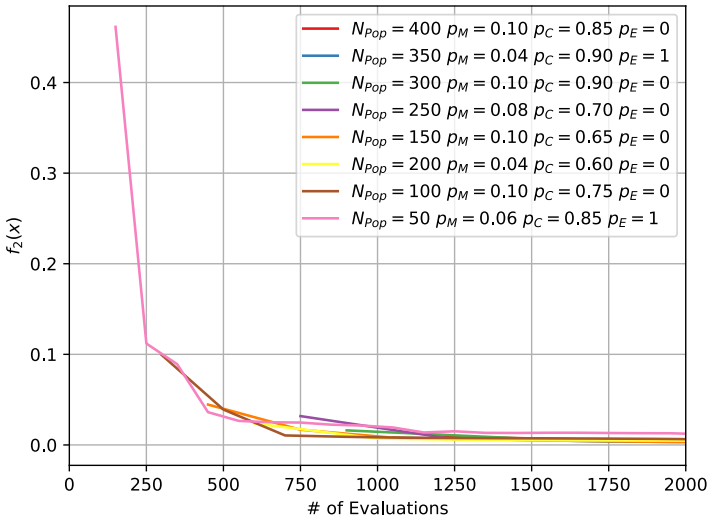


Figure 6.9: Comparison of the best parameters for the Genetic Algorithm for different population sizes on the Rosenbrock function.

According to this, the best parameters that will be considered in the further course of this experiment are $N_{\text{Pop}} = 400$, $p_M = 0.1$, $p_C = 0.85$ and $p_E = 0$. The limit of the x-axis in Figure 6.9 is decreased to 2000 because only little change occurred to the configurations after that. As in Experiment E_1, if the number of elites is $p_E = 0$ the algorithm leads to better results than if $p_E = 1$. In addition to that, the crossover probability tends to find better results if it is in the range $0.7 \leq p_C \leq 0.9$. The figure also confirms that higher population counts are responsible for better results on the Rosenbrock function.

Subsequently, the best results of the setup experiment for the Subset Lattice Optimization are shown in Figure 6.10.

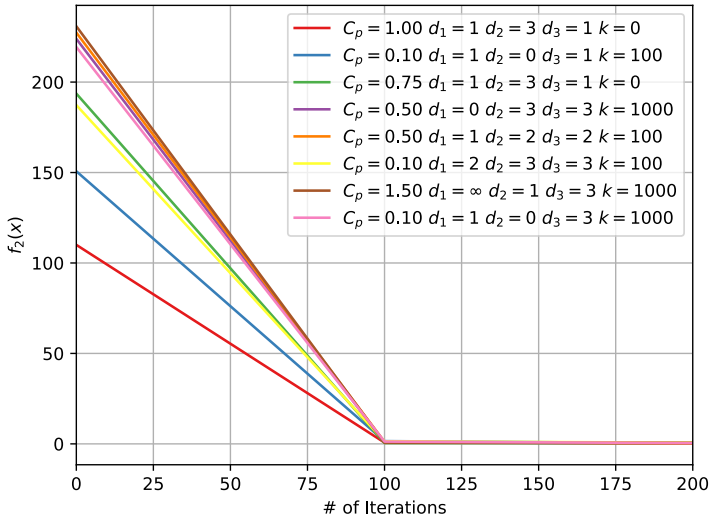


Figure 6.10: The best parameters settings of the Subset Lattice Optimization on the Rosenbrock function.

Like in the determination of the best parameters of the Subset Lattice Optimization in on the Sphere function, this figure is limited on the x-axis at 200 iterations. The different settings vary more at the

start of the algorithm but are closer together than on f_1 after 100 iterations. If this point surpassed, the various parameter setups almost all share the same objective value from there on. Because of this similarity between the configurations, almost no conclusions can be made out of this sub-experiment. Although at the best setup $k = 0$, the larger k values dominate the other best results. Also, among the eight best results it holds $d_3 \geq 1$. The configuration with the lowest AUC, which is compared to the Genetic Algorithm is $C_p = 1.0$, $d_1 = 1$, $d_2 = 3$, $d_3 = 1$ and $k = 0$.

In the last part of this experiment, the comparison of the best settings of the Genetic Algorithm and the Subset Lattice Optimization are presented. The evaluation limits are set to $I = N_{\text{eval}} = 20000$ and the step size to $s = 400$. Figure 6.11 shows the results of this experiment.

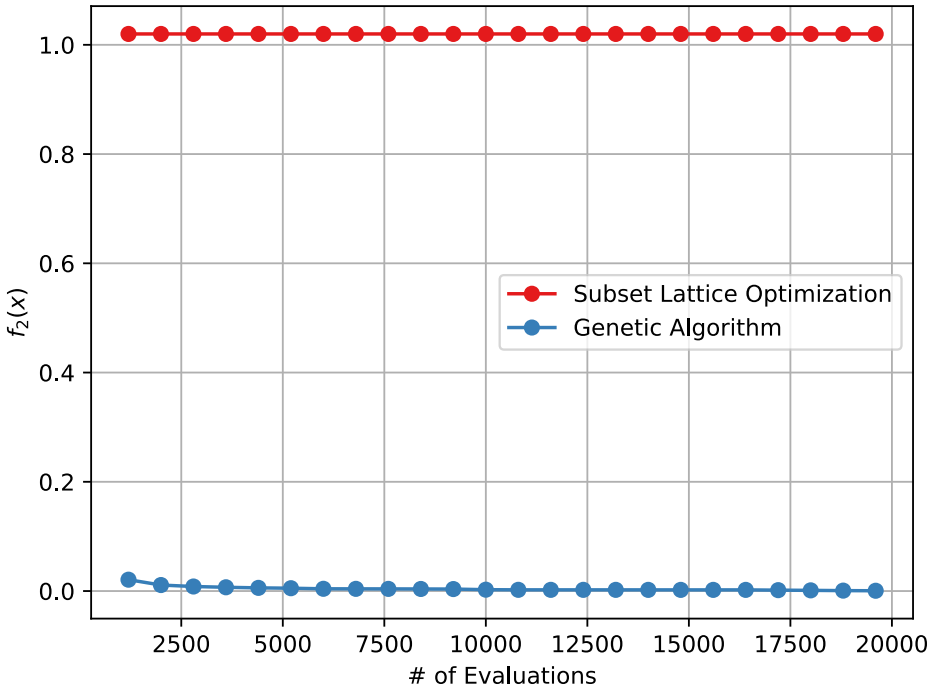


Figure 6.11: Comparison of the Genetic Algorithm and the Subset Lattice Optimization on the Rosenbrock function.

The results are similar to the comparison in the experiment of the Sphere function. The Subset Lattice Optimization doesn't seem to improve over time and the Genetic Algorithm quickly converges to the optimum. This fact has already been discussed in the experiment E_1 in Section 6.4.1.

6.4.3 Experiment E_3: Rastrigin function

The third experiment examines the two algorithms on the Rastrigin function. As parameters for the determination of the best parameters, the limit of maximum evaluations is set to $I = N_{\text{eval}} = 3000$. Furthermore, the number of bits per gene is set to $N_{\text{Gene}} = 15$ in all algorithms throughout this experiment. Figure 6.12 shows the best parameter configuration for each population of the Genetic Algorithm on f_3.

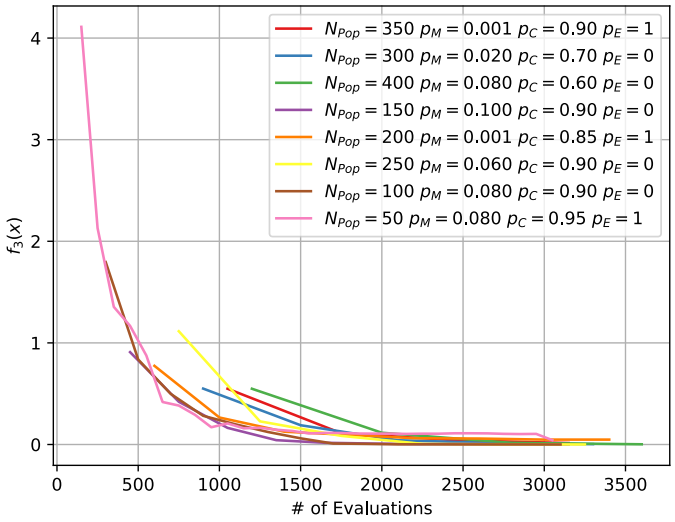


Figure 6.12: Comparison of the best parameters for the Genetic Algorithm for different population sizes on the Rastrigin function.

The results of this function show again, that higher population counts lead to better performance of the algorithm. However, in this case, the largest population size that was tested is only on rank three of the best parameter settings sorted after that. In addition, smaller values for the mutation probability p_M and larger values for the crossover probability p_C tend to result in better performances. The best parameters are $N_{\text{Pop}} = 350$, $p_M = 0.001$, $p_C = 0.9$ and $p_E = 1$.

Next, the search for the best parameters of the Subset Lattice Optimization on f_3 is presented in Figure 6.13. The x-axis was limited at 200 iterations, as in the experiments before.

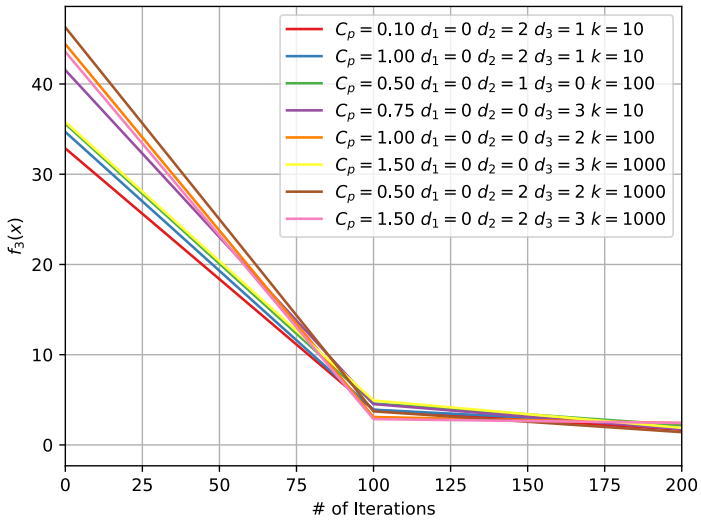


Figure 6.13: The best parameters settings of the Subset Lattice Optimization on the Rastrigin function.

The figure shows again the eight best parameter configurations, which results in a similar figure than in the experiments before. At the first iteration, these best settings can be clustered into two groups. After 100 and more iterations, the objective values of all settings converge to a value near the optimum.

This sub-experiment leads to the conclusion that the parameter setting $d_1 = 0$ achieves the best results. The best configuration on the Rastrigin function is $C_p = 0.1$, $d_1 = 0$, $d_2 = 2$, $d_3 = 1$ and $k = 10$.

Eventually, both algorithms are compared on the Rastrigin function. The evaluations limit is set to $I = N_{\text{eval}} = 10000$ and the step size to $s = 350$. The comparison between the best determined configurations is presented in Figure 6.14.

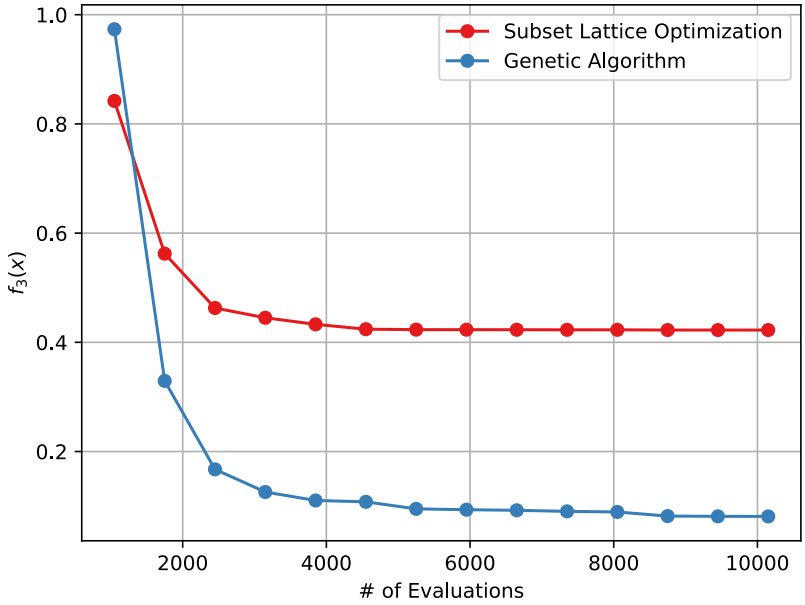


Figure 6.14: Comparison of the Genetic Algorithm and the Subset Lattice Optimization on the Rastrigin function.

At the first point in the diagram, at 1050 evaluations, the Subset Lattice Optimization is closer to the minimum than the Genetic Algorithm. After that, the Genetic Algorithm finds better solutions more quickly. However, both algorithms make significant progress until about 5000 - 6000 evaluations. If this point is exceeded, then the algorithms don't seem to come closer to the solution. Furthermore, both algorithms don't come as close to the solution as in the previous experiments. Nevertheless, the Genetic Algorithm also performs better on the Rastrigin function.

6.4.4 Experiment E_4: Zakharov function

For the experiment that investigates the behavior of the algorithms on the Zakharov function, the bits per gene are set to $N_{\text{Gene}} = 15$ and the maximum evaluations to $I = N_{\text{eval}} = 3000$. In Figure 6.15, the best parameter setting for each particular population size is presented, according to the corresponding sub-experiment.

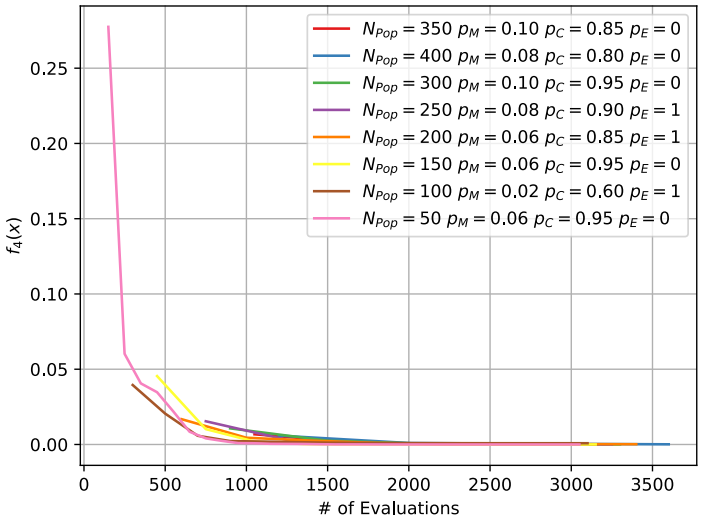


Figure 6.15: The best parameters for the Genetic Algorithm for different population sizes on the Zakharov function.

The figure already suggests that the results are similar to the setup experiments of the Genetic Algorithm on the test functions before. But the results indicate that large values for the crossover probability $p_C \geq 0.8$ are more likely to lead to better performances. In addition, it can also be observed in this experiment that higher population sizes result in better performances. Of all the results, the best configuration, which is considered in the last part of this experiment, is $N_{\text{Pop}} = 350$, $p_M = 0.1$, $p_C = 0.85$ and $p_E = 0$.

In the second part of experiment E_4, the best parameters for the Subset Lattice Optimization on the Zakharov function are searched for. The best eight settings are presented in Figure 6.16, which is again limited to 200 iterations.

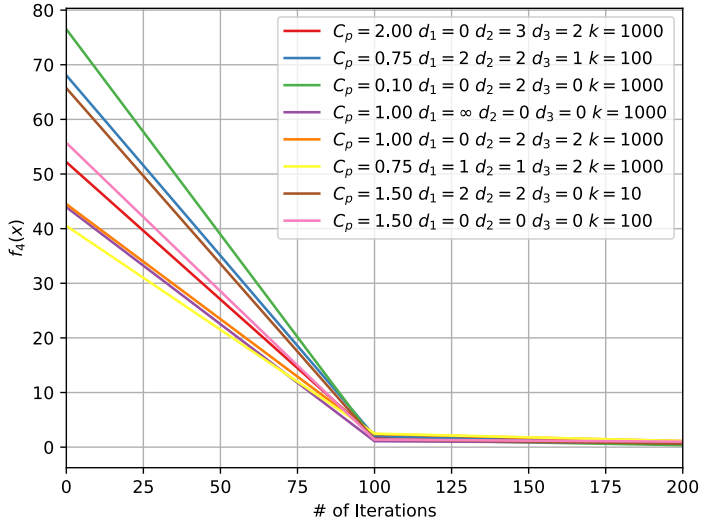


Figure 6.16: The best parameters settings of the Subset Lattice Optimization on the Zakharov function.

Similar to the results of the configuration experiments on the test functions before, only scattered objective values are achieved of the best settings at the start of the algorithm. After 100 iterations the objective values of the parameter settings don't differ much, a fact that applies all the more to an increasing number of iterations. Among the best eight parameter configurations, many different values occur for the individual parameters. For instance, the values for d_1 are 0, 1, 2 and even ∞. The only conclusion that can be drawn from these results is that for $k = 1000$, the configurations tend to lead to better performances. Lastly, the best parameters are $C_p = 2.0$, $d_1 = 0$, $d_2 = 3$, $d_3 = 2$ and $k = 1000$.

The last step of this experiment consists of the direct comparison of the best settings of both algorithms on f_4. For this experiment, the parameters are set as on the experiment E_3, thus $I = N_{\text{eval}} = 10000$ and the step size to $s = 350$. Figure 6.17 illustrates the results of the comparison of the both algorithms.

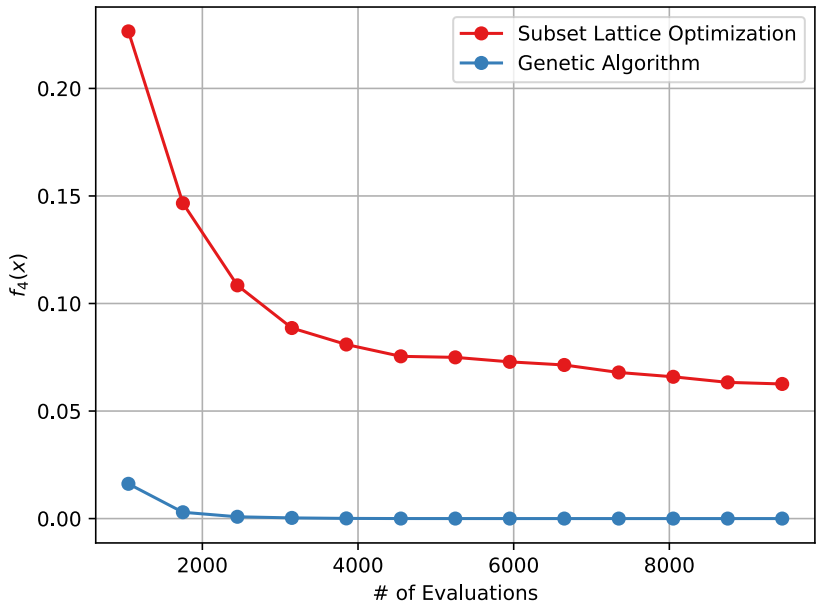


Figure 6.17: Comparison of the Genetic Algorithm and the Subset Lattice Optimization on the Zakharov function.

In the figure, the Genetic Algorithm is closer to the optimum than the Subset Lattice Optimization at all times. First, the Genetic Algorithm quickly converges to the minimum. The Subset Lattice Optimization on the other hand slowly improves its best solution but doesn't come as close as the Genetic Algorithm. After 10000 iterations, the algorithm still has not found the optimum.

6.4.5 Experiment E_5: Easom function

After first tests with hand-picked parameters, the algorithms were not able to find the minimum of the Easom function within limits that were used before. Thus, the maximum allowed evaluations in this experiment are set to $I = N_{\text{eval}} = 10000$. If all parameter combinations were tested as in the experiments before, the experiments would take to much time. Especially the Subset Lattice Optimization has longer runtimes. This is why the different values for the individual parameters are reduced based on the experience of the previous experiments.

In all experiments before, larger population sizes N_{Pop} lead to better performances of the Genetic Algorithm. Also, the two highest values for the crossover probability p_C were never among the best parameter

settings. At these two parameters the selection of values is reduced which leads to the following values being tested on the Easom function:

- $N_{\text{Pop}} \in \{300, 350, 400\}$
- $p_C \in \{0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95\}$
- $p_M \in \{0.001, 0.02, 0.04, 0.06, 0.08, 0.1\}$
- $p_E \in \{0, 1\}$

This results in $3 * 8 * 6 * 2 = 288$ different combinations. With 10 repetitions this leads to 2880 individual algorithm executions. The bit size on all following sub experiments is set to $N_{\text{Gene}} = 15$. Figure 6.18 shows the best parameter settings of the Genetic Algorithm for each population size.

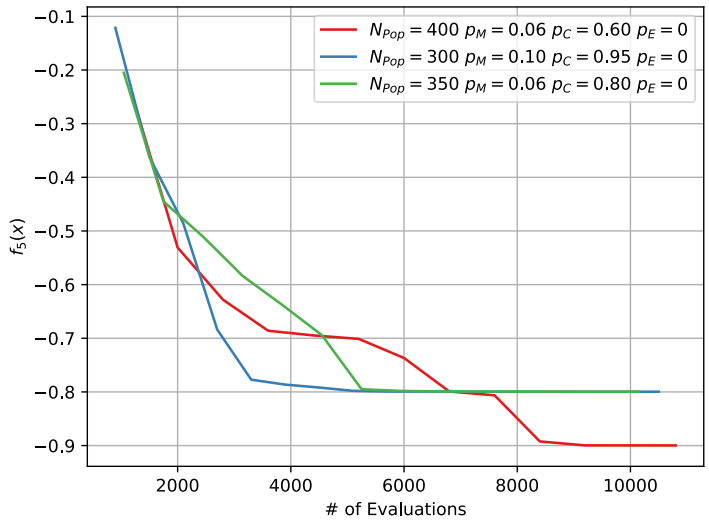


Figure 6.18: The best parameter settings for the Genetic Algorithm for different population sizes on the Easom function.

From this results, it can be seen that no parameter setting reaches the minimum when the objective values are averaged. In some cases, the optimum is found, and in others, the algorithm doesn't even come close. This test function poses a big challenge for the Genetic Algorithm. How often and how quickly the algorithm with a specific setting has found the minimum across all repetitions, defines how good the configuration is. The best setup according to this is $N_{\text{Pop}} = 400$, $p_M = 0.06$, $p_C = 0.6$ and $p_E = 0$. From the three presented settings in the figure, all have in common that no elite is taken into the next generation ($p_E = 0$). In addition to that, in two of the three settings, the mutation probability is $p_M = 0.06$.

The selection of values that are going to be tested with the Subset Lattice Optimization is reduced for every parameter. First, only the best and larger values for C_p are taken because the idea is that the focus on the Easom function should be more on exploration in order to find the minimum. This is also the reason why the two largest values for k are chosen for the determination of the best configuration. The values for the three UCD parameters d_1, d_2 and d_3 are selected out of all the best values from the previous experiments. Thus, the values for the parameters which are tested on the Easom function are the following:

- $C_p \in \{1.0, 1.5, 2.0\}$

- $d_1 \in \{0, 1\}$
- $d_2 \in \{2, 3\}$
- $d_3 \in \{1, 2\}$
- $k \in \{100, 1000\}$

This gives rise to a total of $3 * 2 * 2 * 2 = 48$ different combinations and, with 10 repetitions of each configurations, 480 algorithm executions all together. The best eight results are shown in Figure 6.19.

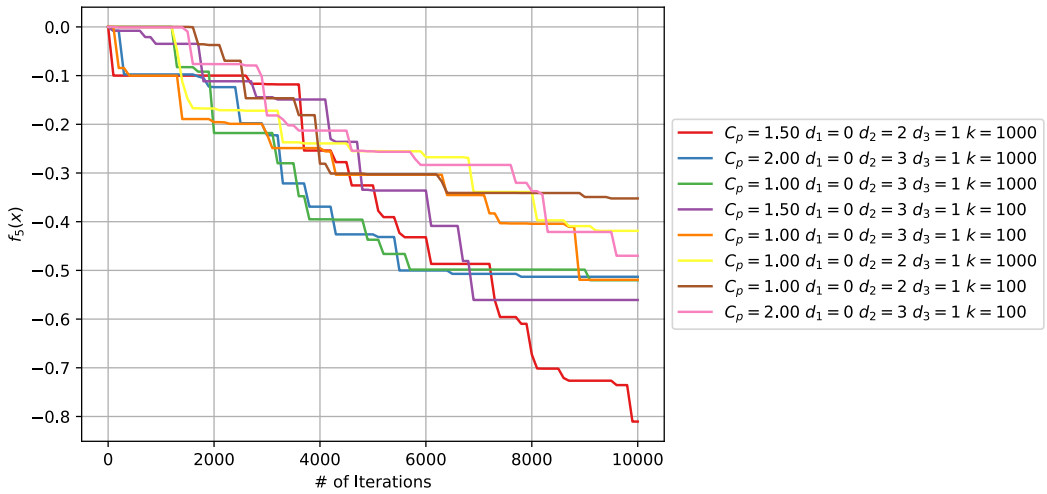


Figure 6.19: The best parameters settings of the Subset Lattice Optimization on the Easom function.

The best parameter settings, illustrated in the figure, show that the Subset Lattice Optimization iteratively improves its best solution over the whole recorded time. According to the results, the parameters $d_1 = 0$ and $d_3 = 1$ lead to better performance of the algorithm. In addition to that, if the RAVE parameter is set to $k = 1000$, the performance of the algorithm tends to be better than for $k = 100$, which corresponds to the conclusions of the other experiments. The best parameter configuration is $C_p = 1.5$, $d_1 = 0$, $d_2 = 3$, $d_3 = 2$ and $k = 1000$. This configuration has a significantly lower AUC and with that a significantly better performance, than the other parameter settings. In the figure this fact is illustrated and, furthermore, it becomes clear that the algorithm could have improved its solution even more if more iterations would be allowed.

In the last comparison of the experiments on test functions, the execution parameters are set to $I = N_{\text{eval}} = 20000$ and $s = 400$. Figure 6.20 shows the result of the best parameter settings of each algorithm on the Easom function.

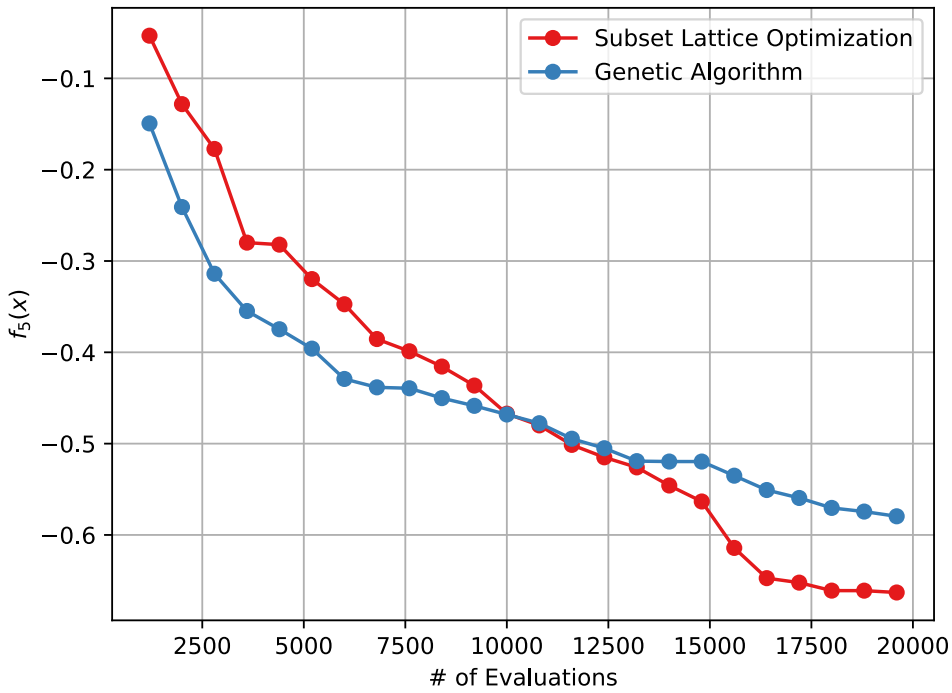


Figure 6.20: Comparison of the Genetic Algorithm and the Subset Lattice Optimization on the Easom function.

The results clearly show that the Subset Lattice Optimization performs better than the Genetic Algorithm on the Easom function. After the first evaluations, the Genetic Algorithm is closer to the minimum across all repetitions in average. Both algorithms improve at about the same rate until 10000 evaluations. At this point, the Subset Lattice Optimization passes the Genetic Algorithm and comes closer to the minimum. The latter algorithm doesn't improve as much as the Subset Lattice Optimization. Some of the individual algorithm executions don't come near the optimum, instead, they stay at the large flat area that characterizes the Easom function. On the other hand, some algorithm executions find the optimum, whereas a few stay in the small area between the flat area and the minimum. From the results, it can be concluded that the Subset Lattice Optimization is more likely to find the minimum on the Easom function because the figure illustrates the better arithmetic mean in each point across all repetitions.

6.5 Conclusion of the experiments

There are two different categories of conclusions which will be described in this section. Firstly, the parameter settings of both algorithms are summarized and examined. Secondly, the performances of the algorithms on the different test functions are summarized. In addition, suggestions are made concerning the results of the algorithms in the experiments and across all test functions. The focus in this section is on the Subset Lattice Optimization because it is the proposed algorithm of this thesis.

First, in order to conclude on the parameters of the Genetic Algorithm, the best parameter setting for each test function is summarized in Table 6.3.

f	N_{Pop}	p_M	p_C	p_E
f_1	350	0.08	0.6	1
f_2	400	0.1	0.85	0
f_3	350	0.001	0.9	1
f_4	350	0.1	0.85	0
f_5	400	0.06	0.6	0

Table 6.3: Best parameters on each function for the Genetic Algorithm.

This table resembles the results of all experiments. The only definite statement that can be made, is that larger population sizes lead to better algorithm performances, with $N_{\text{Pop}} = 350$ and $N_{\text{Pop}} = 400$ being the only values present in the table. The mutation probability would lead to good results if it was smaller, that is $p_M \leq 0.1$. Such conclusions cannot be made for the crossover probability p_C, as all different values are among the best results across all experiments. The best results of each experiment also contain the two different values for the elite parameter p_E. But when looking at the best results of the individual experiments, the algorithm tends to perform better if $p_E = 0$.

Analogous to the Genetic Algorithm, the best parameter configuration of the Subset Lattice Optimization on each test function is listed in Table 6.4.

f	C_p	d_1	d_2	d_3	k
f_1	0.5	1	3	2	1000
f_2	1.0	1	3	1	0
f_3	0.1	0	2	1	10
f_4	2.0	0	3	2	1000
f_5	1.5	0	2	1	1000

Table 6.4: Best parameters on each function for the Subset Lattice Optimization

The parameter in the first column is the exploration parameter C_p, which originates from the UCT formula (see Section 3.4). Across all experiments, no conclusions can be drawn regarding the general value for this parameter that should be chosen for the Subset Lattice Optimization. In the table above, as well as in the specific experiments, different values were among the best parameter configurations. A reason for that result could be that this value strongly depends on the solution landscape and in the degree, this space should be explored. According to the best configurations, for the UCD parameters d_1, d_2 and d_3, only two different values are considered among the best results. On the one hand, these parameters have a more diverse range in individual experiments. On the other hand, in other experiments (for instance E_3 and E_5), only a few values are among the best settings. Thus, it can be assumed, that the values in the table tend to better performances than the others: $d_1 \in \{0,1\}$, $d_2 \in \{2,3\}$ and $d_3 \in \{1,2\}$. These parameters seem to be less dependent of the test function. They reflect how the algorithm creates and traverses the lattice. The last column lists the RAVE parameter. In three out of five test functions, the value 1000 is the best value. Additionally, at all experiments including E_2 and E_3, if the parameter is set to $k = 1000$, the respective configurations will always be among the best settings. This leads to the conclusion, that $k = 1000$ is the best choice for k and improves the performance of the Subset Lattice Optimization.

When comparing the two algorithms, in four out of five test functions, the Genetic Algorithm is more effective than the Subset Lattice Optimization. On the Sphere and the Rosenbrock function, the comparison shows similar results. In both cases, the Genetic Algorithm quickly converges to the optimum, while the Subset Lattice Optimization does not seem to improve. What is not visible in the Figures 6.8

and 6.11, is the fact that the SLO does improve its best solution, but only in evaluation counts that are not shared with the Genetic Algorithm. At the first shared point, the Subset Lattice Optimization has reached its best state. This is the reason why the figures don't show any improvement. On the Rastrigin function, the Subset Lattice Optimization is closer to the minimum in the first 1000 evaluations but is overtaken by the Genetic Algorithm after that. The result is comparable on the Zakharov function, where the Genetic Algorithm is always closer to the optimum. In the latter two test functions, the improvement of the Subset Lattice Optimization is visible in the Figures 6.14 and 6.17. This is in contrast to the comparisons of the first two experiments. The set of experiments were completed on the Easom function. At the beginning, the Genetic Algorithm was closer to the optimum. However, in the following evaluations, both algorithms improved its best solution and, eventually, the Subset Lattice Optimization came closer to the minimum after 10000 evaluations.

An observation of the behavior across all test functions, except the Easom function, is that the Subset Lattice Optimization improves its best solution quickly at the first few iterations. However, the improvements come to a sudden halt, when a certain point is reached. This point is reached earlier on the Sphere and Rosenbrock function and later on the Rastrigin and Zakharov function. There are at least two possible explanations for that:

- The algorithm finds a good solution in a region of the lattice and builds the lattice in that region while improving the best solution. However, the disadvantage would be that the global optimum could be in another region that is not built at the moment. It could only be found through randomly selecting a path that leads to this solution.
- A good solution is found after the first few iterations. However, the average reward of a state could decrease when the neighborhood of this good solution only contains bad individuals. Thus, the algorithm would decide to more likely exploit an area where the average reward is better. This new region could be the focus of the extension of the lattice, although no better solution was found there.

The Subset Lattice Optimization maintains only one single solution candidate, while iteratively building the lattice. In contrast, the Genetic Algorithm maintains a population of multiple solution candidates, which may be the reason it converges more. Based on the points mentioned above, it can be concluded, that the Subset Lattice Optimization isn't good at exploiting promising regions in the lattice. Thus, the solutions are not improved after a certain point is reached.

The results on the Easom function show that the Subset Lattice Optimization is able to find an optimum within a larger solution space. After this fact, it can be concluded that the algorithm is successful at the exploration, but on the other hand is not fit at exploiting the found solutions. This circumstance needs further enhancements of the algorithm, which are discussed in an outlook in Section 8.2.

7 Lennard-Jones Atomic Clusters Problem

In the previous chapter, the proposed Subset Lattice Optimization was tested on artificial benchmark functions. This is a common procedure typically used in the literature when it is investigated how well a new algorithm performs in certain scenarios. The scenarios, which Global Optimization algorithms are typically applied to in order to find optima are called *real-world problems*. In this chapter, the Subset Lattice Optimization is tested against a Genetic Algorithm on a real-world problem, namely the Lennard-Jones Atomic Clusters problem (or for short: *Lennard-Jones* problem).

The task of the Lennard-Jones problem is to find the positions of a set of neutral atoms, such that the potential energy between the atoms is minimized. This energy is also referred to as *Lennard-Jones potential energy*. The cluster is in a more stable state if the potential is at a minimum. For a given number of atoms, there exists one single global minimum [Fan02]. In addition, there are many local minima whose number grows exponentially as the number of atoms increases. For instance, in a cluster of 13 atoms there are 988 minima and, if the number of atoms comes close to 100, there exist already 10^{140} local minima [Dav+96]. This fact makes the Lennard-Jones problem extremely hard to solve. In the literature, the search for the optima of particularly large clusters is often performed. For instance, the authors of [WD97] present the global minimum of each cluster up to 110 atoms [1]. Table 7.1 shows a selection of this source by presenting the minima of clusters from a size of three up to ten atoms.

Number of Atoms	Global potential energy minimum in [J]
3	−3
4	−6
5	−9.103852
6	−12.712062
7	−16.505384
8	−19.821489
9	−24.113360
10	−28.422532

Table 7.1: Overview of a few global Lennard-Jones energy potential minima (Own representation based on [WD97, p. 3]).

Although the Subset Lattice Optimization is applied on only one real-world problem in this thesis, it has the potential of being executed on any optimization problem. The condition that has to be met for this is that the states of the problem need to be encoded as a binary string, like in the classic Genetic Algorithm. This is not the case, for instance, in solving the Bin-packing problem. To this point, there exist only viable approaches to solve the Bin-packing problem with Genetic Algorithms, that have applied a permutation encoding [2], for example in [Ree96]. Many applications of Genetic Algorithms rely on the permutation encoding, like scheduling problems or the mentioned Bin-packing problem. These problems cannot be used because the permutation encoding is not defined to be used in the Subset Lattice Optimization.

[1] The authors have a list of global optima in clusters up to 1000 on their website: `http://doye.chem.ox.ac.uk/jon/structures/LJ.html`, last accessed on September 25th, 2017.

[2] An overview of different encoding strategies was given in Section 2.2.1.

In the further course of this chapter, the Lennard-Jones problem itself is described. After that, an experiment is carried out that compares the performance of the Genetic Algorithm and the Subset Lattice Optimization on this problem.

7.1 Problem description

This section describes the Lennard-Jones Atomic Clusters problem and defines it mathematically. The definitions are based on Fans work in [Fan02]. Firstly, the task of an optimization is to find the global minimum of the Lennard-Jones energy potential of the whole atomic cluster. The Lennard-Jones potential depends on the positions of the individual atoms. That's why the correct positions of all atoms are searched that result in the global Lennard-Jones energy potential minimum.

The problems are scaled with the number of atoms per cluster. A cluster consists of A atoms, which are defined in the three-dimensional Euclidean space: Atom i has the coordinates $\boldsymbol{y}^i = (y_1^i, y_2^i, y_3^i)^T$ with $\boldsymbol{y}^i \in \mathbb{R}^3$. The coordinates of all atoms of a cluster is then given by $Y = (\boldsymbol{y}^1, \boldsymbol{y}^2, ..., \boldsymbol{y}^A)^T$. Next, the distance between two atoms i and j can be defined, which is denoted by the Euclidean distance $r_{ij} = ||y^i - y^j||$. With the help of that, the Lennard-Jones energy potential between two atoms i and j is denoted by:

$$v(r_{ij}) = \frac{1}{r_{ij}^{12}} - \frac{2}{r_{ij}^{6}}$$

The energy potential, in dependency of the distance between two atoms, is illustrated in Figure 7.1. In this figure, the x-axis stands for the distance between the two atoms and the y-axis shows the Lennard Jones energy potential between the two. The further away these two atoms are from each other, the more this energy potential approaches zero. This is because the atoms attract each other. At distance 1 the global minimum with energy -1 Joule is reached, which denotes the stable state. If the two atoms get closer, the energy will go to infinity with the two atoms being in a repulsion.

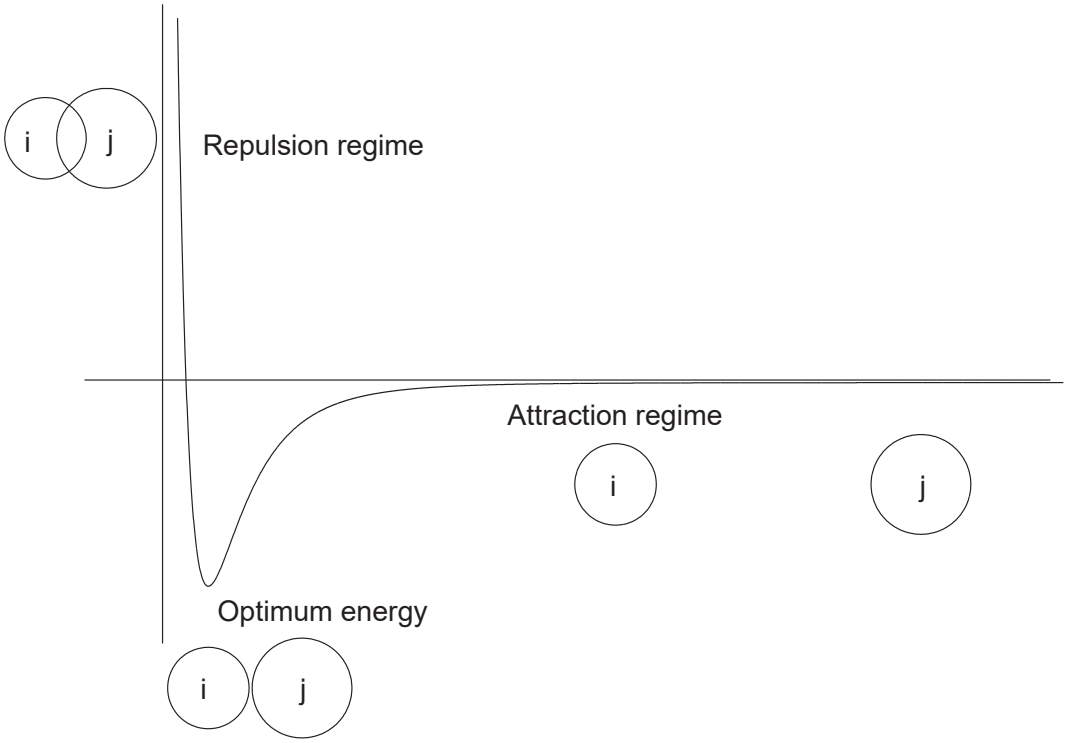


Figure 7.1: Lennard-Jones potential of two atoms dependent of their distance. (Source: [Fan02, p. 10])

Eventually, the Lennard-Jones energy potential of the whole cluster can be calculated by summing up all energy potentials between each pair of atoms in the cluster. Thus, the potential is denoted by:

$$V(Y) = \sum_{i=1}^{A-1} \sum_{j=i+1}^{A} v(r_{ij}) = \sum_{i=1}^{A-1} \sum_{j=i+1}^{A} \frac{1}{r_{ij}^{12}} - \frac{2}{r_{ij}^{6}}$$

In the further course of this chapter, this function is used as the objective function in the optimization algorithms.

7.2 Experiment

In this section, the experiment is carried out that has the goal to test the Subset Lattice Optimization on the Lennard-Jones Atomic Clusters problem. The algorithm's performance is compared with the performance of the Genetic Algorithm, that was used in the experiments before (see Chapter 6). First, the configurations of both algorithms and of the experiment itself are explained. Subsequently, the results are presented, followed by concluding remarks.

7.2.1 Configuration

The process of determining the potential energy of a cluster is the same in both algorithms in order to compare their effectiveness directly. For the minimization, the objective function that is used is the function that calculates the energy based on the positions of all atoms $V(Y)$. Thus, the solution space of the two algorithms is a set of coordinates. This approach is rather simple and does not use any other information to create candidate solutions. For instance in [Bar+99], the authors use another algorithm to create more sophisticated coordinates instead. Because the Subset Lattice Optimization builds a state lattice iteratively instead of randomly creating states, this approach couldn't be implemented in the algorithm. For the Genetic Algorithm, on the other hand, this type of enhancements won't be implemented, in order to satisfy that the comparison of effectivenesses between the two algorithms is ensured.

The coordinates of the atoms are stored as a binary string in a chromosome in both algorithms. Each coordinate of an atom is represented by a gene. With a given number of bits per gene, the chromosome length (or lattice base size) can be calculated with $b = N_{\text{Gene}} * A * 3$. In this experiment, the number of atoms in a cluster that will be tested is limited to $A = 3$. Additionally, the number of bits in a gene is set to $N_{\text{Gene}} = 8$. Thus, the chromosome length in this scenario is $b = 8 * 3 * 3 = 72$. This number is larger than in the experiments on the test functions where the chromosome length was 30 bits. That fact increases the complexity of the problem. Furthermore, for this small number of atoms, it is assumed that the range in which each coordinate is in, is bounded by $\forall y_j^i \in Y : -1 \leq y_j^i \leq 1$.

The settings that are used for the algorithms are derived from the results of the experiments in the previous chapter. Tables 6.3 and 6.4 contain the best parameters across all test functions, from which the values for the parameters in this experiment are based on. Each algorithm is configured as follows:

- Genetic Algorithm: From all four parameters of the Genetic Algorithm, two are fixed and two are varied. First, the best results were achieved for the biggest two population sizes. With such a certainty, the mutation probability cannot be estimated. However, based on Table 6.3, the value 0.1 is listed the most over all functions. This is also true for the number of elites, where the best choice is to not use any elites. Regarding the crossover probability, two values were among the best. Thus,

the configurations of the parameters on the Lennard-Jones problem are given by $N_{\text{Pop}} \in \{350, 400\}$, $p_C \in \{0.6, 0.85\}$, $p_M = 0.1$ and $p_E = 0$.

- Subset Lattice Optimization: In Table 6.4, the values for the parameter C_p are very diverse. Instead of choosing all six values, as in the experiments E_1 - E_4, they are reduced to the two largest values. This limitation is due to the fact that in the Lennard-Jones problem, the focus should be more on the exploration of the solution space. The values for d_1 and d_3 are fixed at the values which are most represented among the best results. This decision is also supported from the results of the Easom function, where these values are also widely represented in the best configurations. In addition to that, the k parameter is fixed as well because the best choices for this are large values, as was already discussed before, in Section 6.5. These considerations lead to the following parameter settings: $C_p \in \{1.5, 2.0\}$, $d_1 = 0$, $d_2 \in \{2, 3\}$, $d_3 = 1$ and $k = 1000$.

Accordingly, a total of four configurations are tested per algorithm. Each configuration is repeated 50 times to increase the quality of the results. This means that, altogether, every algorithm has 200 executions. Furthermore, the limit of maximum evaluations is set to $I = N_{\text{eval}} = 10000$.

7.2.2 Results

The results of this experiment are shown in Figure 7.2. Only the best of all four parameter settings that an algorithm can have in this experiment, is displayed. Furthermore, the average objective values, which are calculated over all 50 repetitions, form the mean results that are shown. Thus, the process of creating the figure out of the best results is the same as for the test functions in Section 6.4.

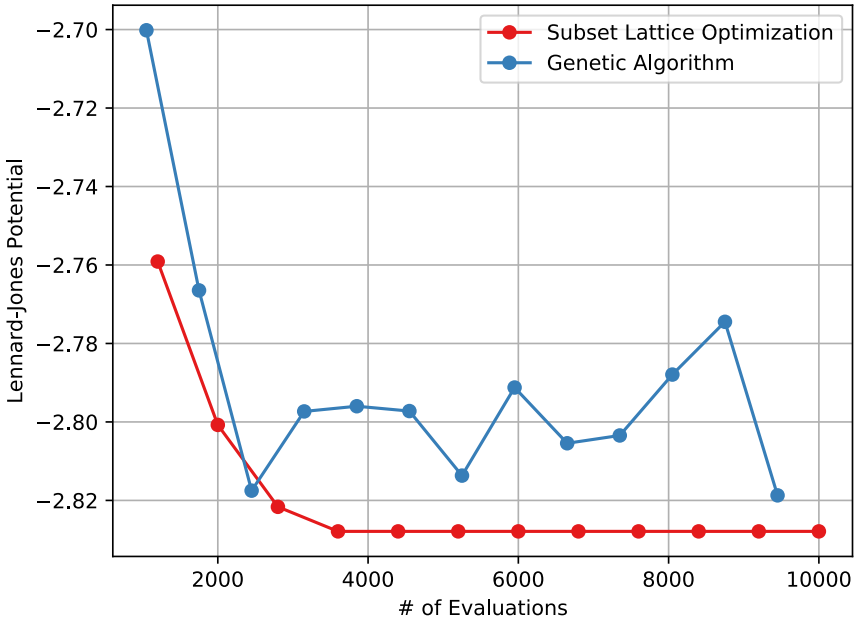


Figure 7.2: Comparison of the Genetic Algorithm and the Subset Lattice Optimization on the Lennard-Jones Atomic Clusters problem.

The data points in the graph have different coordinates on the x-axis, which is the result of different sampling rates of the algorithms. In this case, the objective value from the Subset Lattice Optimization was taken every 800 and from the Genetic Algorithm every 700 evaluations. The best configuration for the Genetic Algorithm is with the parameter values $N_{\text{Pop}} = 350$, $p_M = 0.1$, $p_C = 0.6$ and $p_E = 0$. For the

Subset Lattice Optimization, the parameter setting $C_p = 2.0$, $d_1 = 0$, $d_2 = 3$, $d_3 = 1$ and $k = 1000$ leads to the best results.

The Subset Lattice Optimization has a better performance than the Genetic Algorithm on the Lennard-Jones Atomic Clusters problem. Although the minimum of -3 was not reached in average by both algorithms, the Subset Lattice Optimization converges more securely in direction of the optimum. In all points, except one at about 2600 evaluations, the proposed algorithm is closer to the optimum than the Genetic Algorithm. From the first sample point at 800 evaluations, it seems that the algorithm quickly finds a local minimum with the objective value -2.83 and converges to it after 4000 evaluations. After that, the algorithm doesn't make any improvements on its best solution. The Genetic Algorithm, on the other hand, doesn't seem to find a stable solution. It also converges in direction to the optimum, but after 2300 evaluations it doesn't make any progress in finding good solutions. Rather than that, the objective value of the best solution jumps between -2.82 and -2.77

In addition, it should be mentioned that the determination of the best configuration was achieved by selecting the parameter values that lead to the lowest AUC, similar to the experiments on the test functions. Tables 7.2 and 7.3 show the values of the parameters that are varied in the experiment, sorted by the AUC that the individual configuration obtained. According to this results, the best values are on the one hand $N_{\text{Pop}} = 350$ and $p_C = 0.6$ for the Genetic Algorithm, on the other hand $C_p = 2$ and $d_2 = 3$ for the Subset Lattice Optimization.

AUC	N_{Pop}	p_C
1742	350	0.60
1767	400	0.60
1948	400	0.85
1966	350	0.85

Table 7.2: Best parameter settings of the Genetic Algorithm

AUC	C_p	d_2
1568	2.0	3
1570	1.5	2
1580	2.0	2
1596	1.5	3

Table 7.3: Best parameter settings of the Subset Lattice Optimization

7.3 Conclusion from the Real World Application

This chapter demonstrated that the proposed Subset Lattice Optimization can also be applied in real-world scenarios. The Lennard-Jones Atomic Clusters problem provided a good example of how real-world problems can be more difficult to solve. Even with a low number of atoms in a cluster, the chromosome length was more than twice as long as in the test functions. Another challenge was the highly multimodal objective function.

The results showed, that the Genetic Algorithm was outperformed by the Subset Lattice Optimization. One reason for this could be that the Genetic Algorithm is not able to cope with the larger solution space so well. In addition to that, the results also confirmed one conclusion that was made in the experiments on the test functions: The Subset Lattice Optimization tends to improve its solution only up to a certain point. When this point is reached, the algorithm doesn't seem to get out of the local optimum. In Section 6.5 some ideas of how this behavior can be explained were presented.

According to the experiment, the Subset Lattice Optimization works well on the Lennard-Jones problem. This leads to the conclusion, that the algorithm can handle more complex scenarios well, that are characterized by larger lattice base sizes. It can also be concluded that the algorithm has a good ability to find an optimum in a large solution space, which is the case in the tested problem.

8 Conclusion and Outlook

In this final chapter, a conclusion of the thesis is given. Afterwards, an outlook will be presented which contains suggestions for future work on the topic of the thesis.

8.1 Conclusion

The key contribution of this thesis is the creation of a new Global Optimization algorithm, namely the Subset Lattice Optimization. It was presented in Chapter 4 and is based on the fundamentals in the previous chapters. In addition to the basic version, two enhancements of the algorithm were introduced as well. After that, the performance of the algorithm was compared with a Genetic Algorithm in experiments on test functions in Chapter 6. Furthermore, the performances of the algorithms were investigated on the more complex Lennard-Jones Atomic Clusters problem.

All experiments reveal that the algorithm is able to find global optima or that it comes close to it and converges in local optima. According to the conclusions that were derived from the experiments, the algorithm is able to find an optimum at even large solution spaces. It outperforms the Genetic Algorithm at the latter case, more specifically on the Easom function and the Lennard-Jones problem. One downside of the algorithm is, that it does not seem to make use of the found optima. This can be seen at simpler problems like the Sphere function: A relative good solution is found quick. However, it is not improved after a certain amount of time. The convergence stops although the global optimum is not found.

Another key contribution of this thesis is the use of Rapid Value Estimation inside the UCD algorithm. The UCD algorithm, based on the Monte Carlo Tree Search with the UCT policy, is the core of the Subset Lattice Optimization and utilizes transpositions. In [SCM12], the authors of the UCD algorithm suggest using RAVE to improve the algorithm. After careful research, this idea was investigated in this thesis for the first time. The results of the experiments indicate that the addition of RAVE improved the performance of the algorithm significantly, at least in graphs that have many transpositions, like the used binary subset lattice. In almost all test functions, high values for the RAVE parameter k lead to better results. Thus, $k = 1000$ was used as the best choice for the algorithm in the real-world application (see Chapter 7).

8.2 Outlook

This work presents the first concept of a new optimization algorithm. Thus, there exist many possibilities of how to improve it. Although the algorithm performed well in the individual experiments, there are a few ideas that could be investigated in future works in order to improve the Subset Lattice Optimization:

- The first possible extension of the algorithm would be to support permutation encoding. With this, each state would consist of n different natural numbers from 0 to $n-1$. The states differ depending on the permutation of the numbers. In order to develop this encoding for the algorithm, it must be defined how the states relate to each other so that they form a lattice.
- Another extension would be to support constraints of objective functions. Constraints that restrict variable values are commonly used, specifically in more complex functions. This would result in many nodes of the lattice being forbidden by such hard constraints. It could be the case that the integration of constraints in the framework would not affect the algorithm itself. Instead, only the objective function that decodes the states would have to be changed.

- One idea, which does not alter the algorithm, is to test it on more complex functions. Specifically, these tests could be carried out on benchmark functions that are scalable to an arbitrary number of dimensions. The real-world problem has already shown that the Subset Lattice Optimization handles the larger lattice base size quite well. It would be of great interest to see how the algorithm performs on states that have a size of hundreds of bits.
- The integrated Rapid Value Estimation improved the algorithm. An area of investigation could be to swap this part of the algorithm with other All-Moves-As-First technologies. Many of these methods are discussed in the literature and could be tested. Even other RAVE methods could be promising, for example, KillerRAVE or PoolRAVE.
- The last idea would be to dynamically change the algorithm's parameters during the runtime. This has the goal to counter the halt in the convergence of the algorithm. At first, this halt needs to be detected. After that, the parameter could be changed in order to find better solutions. For instance, the RAVE parameter k or the exploration parameter C_p could be increased with the goal to find better regions in the solution space.

Bibliography

[ABB14] A. Ahmadi, F. E. Bouanani, and H. Ben-Azza. “Four Parallel Decoding Schemas of Product Block Codes”. In: *Transactions on Networks and Communications* 2.3 (2014), pp. 49–69.

[ACF02] P. Auer, N. Cesa-Bianchi, and P. Fischer. “Finite-time Analysis of the Multiarmed Bandit Problem”. In: *Machine Learning* 47.2-3 (May 2002), pp. 235–256.

[Bar+99] C. Barrón, S. Gómez, D. Romero, and A. Saavedra. “A Genetic Algorithm for Lennard-Jones Atomic Clusters”. In: *Applied Mathematics Letters* 12.7 (1999), pp. 85–90.

[Bar16] T. Bartz-Beielstein. “A Survey of Model-based Methods for Global Optimization”. In: *Bioinspired Optimization Methods and their Applications*. 2016, pp. 1–18.

[BC02] T. Britz and P. Cameron. *Partially ordered sets*. Last accessed on September 29th 2017. 2002. URL: `http://www.maths.qmul.ac.uk/~pjc/csgnotes/posets.pdf`.

[BC95] S. Baluja and R. Caruana. “Removing the genetics from the standard genetic algorithm”. In: *Machine Learning: Proceedings of the Twelfth International Conference on Machine Learning*. 1995, pp. 38–46.

[BP11] R. Brüggemann and G. P. Patil. “Partial Order and Hasse Diagrams”. In: *Ranking and Prioritization for Multi-indicator Systems: Introduction to Partial Order Applications*. Springer, 2011. Chap. 2, pp. 13–23. ISBN: 978-1-4419-8477-7.

[Bro+12] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43.

[BS13] A. Benbassat and M. Sipper. “EvoMCTS: Enhancing MCTS-Based Players through Genetic Programming”. In: *2013 IEEE Conference on Computational Intelligence in Games (CIG)*. IEEE, 2013, pp. 1–8.

[CBK08] B. E. Childs, J. H. Brodeur, and L. Kocsis. “Transpositions and Move Groups in Monte Carlo Tree Search”. In: *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Perth, Australia, 15-18 December, 2008*. IEEE, 2008, pp. 389–395.

[CBM11] D. B. Carvalho, J. C. N. Bittencourt, and T. D. Maia. “The Simple Genetic Algorithm Performance: A Comparative Study on the Operators Combination”. In: *The First International Conference on Advanced Communications and Computation*. 2011.

[CBP09] T. Cazenave, F. Balbo, and S. Pinson. “Using a Monte-Carlo approach for Bus regulation”. In: *12th International IEEE Conference on Intelligent Transportation Systems*. Oct. 2009, pp. 1–6.

[Cha+10] N. Chase, M. Redemacher, E. Goodman, R. Averill, and R. Sidhu. *A Benchmark Study of Optimization Search Algorithms*. Tech. rep. Red Cedar Technology, 2010.

[CZN09] C. M. Chan, L. M. Zhang, and J. T. Ng. “Optimization of Pile Groups Using Hybrid Genetic Algorithms”. In: *Journal of Geotechnical and Geoenvironmental Engineering* 135.4 (2009), pp. 497–505.

[Dar59] C. Darwin. *On the Origin of Species*. 1859.

[Dav+96] D. Daven, N. Tit, J. Morris, and K. Ho. “Structural Optimization of Lennard-Jones Clusters by a Genetic Algorithm”. In: *Chemical Physics Letters* 256.1 (1996), pp. 195–200.

[DH12] J. M. Dieterich and B. Hartke. “Empirical Review of Standard Benchmark Functions Using Evolutionary Global Optimization”. In: *Applied Mathematics* 3.10A (2012), pp. 1552–1564.

[DM01] J. G. Digalakis and K. G. Margaritis. “On Benchmarking Functions for Genetic Algorithms”. In: *International Journal of Computer Mathematics* 77.4 (2001), pp. 481–506.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002. ISBN: 978-0-521-78451-1.

[DP98] R. Delgoda and J. D. Pulfer. “A Guided Monte Carlo Search Algorithm for Global Optimization of Multidimensional Functions”. In: *Journal of Chemical Information and Computer Sciences* 38.6 (1998), pp. 1087–1095.

[Fan02] E. Fan. “Global Optimization of Lennard-Jones Atomic Clusters”. MA thesis. Hamilton, Ontario: McMaster University, 2002.

[GS07] S. Gelly and D. Silver. “Combining Online and Offline Knowledge in UCT”. In: *Proceedings of the 24th International Conference on Machine Learning*. Vol. 227. ACM International Conference Proceeding Series. ACM, 2007, pp. 273–280.

[GS10] R. Gaudel and M. Sebag. “Feature Selection as a One-Player Game”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. 2010, pp. 359–366.

[Hav15] Š. Havránek. “Genetic Algorithms driven by MCTS”. MA thesis. Charles University in Prague, Faculty of Mathematics and Physics, 2015.

[HH98] R. L. Haupt and S. E. Haupt. *Practical Genetic Algorithms*. John Wiley & Sons, 1998. ISBN: 047-1188735.

[Hol92] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, 1992. ISBN: 0262082136.

[HP09] D. P. Helmbold and A. Parker-Wood. “All-Moves-As-First Heuristics in Monte-Carlo Go”. In: *Proceedings of the 2009 International Conference on Artificial Intelligence*. 2009, pp. 605–610.

[JY13] M. Jamil and X.-S. Yang. “A Literature Survey of Benchmark Functions For Global Optimisation Problems”. In: *International Journal of Mathematical Modelling and Numerical Optimisation* 4.2 (2013), pp. 150–194.

[KS06] L. Kocsis and C. Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Proceedings of the 17th European Conference on Machine Learning*. Berlin, Germany: Springer, 2006, pp. 282–293.

[LK75] J. K. Lenstra and A. H. G. R. Kan. “Some Simple Applications of the Travelling Salesman Problem”. In: *Journal of the Operational Research Society* 26.4 (1975), pp. 717–733.

[Mar03] A. C. Marta. “Parametric Study of a Genetic Algorithm using a Aircraft Design Optimization Problem”. In: *Genetic Algorithms and Genetic Programming at Stanford*. Stanford Bookstore, 2003, pp. 133–142.

[MSB91] H. Mühlenbein, M. Schomisch, and J. Born. “The Parallel Genetic Algorithm as Function Optimizer”. In: *Parallel Computing* 17.6-7 (1991), pp. 619–632.

[Mun14] R. Munos. “From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning”. In: *Foundations and Trends in Machine Learning* 7.1 (2014), pp. 1–129.

[New] J. Newcombe. *The Genetic Algorithm Framework for .NET*. Last accessed: August 9th, 2017. URL: `https://www.gaframework.org`.

[NM06] N. Nedjah and L. de Macedo Mourelle. *Swarm Intelligent Systems*. Vol. 26. Springer, 2006. ISBN: 978-3-540-33868-0.

[Ree96] C. Reeves. “Hybrid Genetic Algorithms for Bin-packing and Related Problems”. In: *Annals of Operations Research* 63.3 (1996), pp. 371–396.

[RN09] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson Education, 2009. ISBN: 978-0136042594.

[Rob14] C. P. Robert. “Monte Carlo Methods”. In: *Wiley StatsRef: Statistics Reference Online*. Wiley Online Library, 2014, pp. 1–13. ISBN: 9781118445112.

[RTC11] A. Rimmel, F. Teytaud, and T. Cazenave. “Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows”. In: *Applications of Evolutionary Computation - EvoApplications 2011: EvoCOMNET, EvoFIN, EvoHOT, EvoMUSART, EvoSTIM, and EvoTRANSLOG, Torino, Italy, April 27-29, 2011, Proceedings, Part II*. 2011, pp. 501–510.

[SCM12] A. Saffidine, T. Cazenave, and J. Méhat. “UCD: Upper Confidence Bound for Rooted Directed Acyclic Graphs”. In: *Knowledge-Based Systems* 34 (2012), pp. 26–33.

[SD07] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 2007. ISBN: 978-3-540-73190-0.

[Sil+16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (2016), pp. 484–489.

[SSM15] M. Sidorov, E. Semenkin, and W. Minker. “Unconstrained Global Optimization: A Benchmark Comparison of Population-based Algorithms”. In: *Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics*. Vol. 1. 2015, pp. 230–237.

[TC16] N. C. Tang and A. Chilkoti. “Combinatorial codon scrambling enables scalable gene synthesis and amplification of repetitive proteins”. In: *Nature materials* 15.4 (2016), p. 419.

[WD97] D. J. Wales and J. P. K. Doye. “Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms”. In: *The Journal of Physical Chemistry A* 101.28 (1997), pp. 5111–5116.

[Wei09] T. Weise. *Global Optimization Algorithms - Theory and Application*. Last Accessed on October 1st, 2017. 2009. URL: `http://www.it-weise.de/projects/book.pdf`.

[Zil96] S. Zilberstein. “Using Anytime Algorithms in Intelligent Systems”. In: *AI magazine* 17.3 (1996), pp. 73–83.