Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

Parameter Meta-Optimization of Metaheuristic Optimization Algorithms

Diplomarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Christoph Neumüller, BSc

Begutachter: Prof. (FH) DI Dr. Stefan Wagner

September 2011

software engineering
IKM Fakultät Hagenberg

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 4. September 2011

Christoph Neumüller, BSc.

Contents

Abstract

Metaheuristic algorithms are able to find good solutions for complex optimization problems that cannot be solved efficiently by classical mathematical optimization techniques. Many different metaheuristic algorithms have been developed in the recent decades. However, each algorithm has its strengths and weaknesses and until today, no algorithm has been found that can solve all problems better than all other algorithms. This fact is stated in the *no free lunch* theorem (NFL) for optimization and is one of the reasons why metaheuristic algorithms have parameters which allow a user to adapt the behavior of the algorithm.

The choice of parameter values can have great impact on the effectiveness of an algorithm, but that impact depends on the targeted problem. In the worst case, the parameter values of an algorithm need to be tuned for each new problem instance. However, finding good parameter values is not trivial, as some parameters influence the effects of others. Finding good behavioral parameter values requires human expertise and time which are both, expensive and rare. The search for optimal parameter values can be seen as an optimization problem itself which can be solved by a metaheuristic optimization algorithm. This approach is called *meta-optimization*. The drawback of meta-optimization is that it is extremely runtime intensive. Fortunately, the increasing availability of powerful hardware and distributed computing infrastructures make this approach possible.

This thesis shows the concept of meta-optimization in the form of an implementation for the heuristic optimization environment HeuristicLab. The main aspects of the implementation are to be generic, extendable, parallelizable, easy to use, and it should allow the application of different types of meta-optimization algorithms. To demonstrate the effectiveness of the implementation, a number of parameter optimization experiments are performed and analyzed for evolutionary algorithms. These experiments were executed in a distributed computation environment, using a total CPU time of more than 7.5 years.

Kurzfassung

Metaheuristische Algorithmen eignen sich sehr gut zum Lösen von Optimierungsproblemen, die mit klassischen mathematischen Optimierungsverfahren nicht effizient gelöst werden können. In den vergangenen Jahrzehnten wurden viele unterschiedliche Algorithmen entwickelt, die ihre individuellen Stärken und Schwächen haben. Jedoch wurde bis heute kein Algorithmus gefunden, der alle Optimierungsprobleme besser lösen kann als alle anderen Algorithmen. Das *No Free Lunch* Theorem besagt, dass ein solcher Algorithmus nicht existiert. Aus diesem Grund besitzen metaheuristische Algorithmen Steuerparameter, welche es ermöglichen den Algorithmus an das vorliegende Problem anzupassen.

Die Auswahl der Parameterwerte kann großen Einfluss auf die Funktionsweise eines Algorithmus haben. Diese Auswirkungen können sich jedoch von Problem zu Problem unterscheiden. Im schlechtesten Fall müssen die Parameterwerte daher für jede Probleminstanz einzeln optimiert werden. Das Finden optimaler Einstellungen ist jedoch nicht trivial, denn einige Parameter haben Auswirkungen auf die Funktionsweise anderer Parameter. Deshalb ist die Suche nach guten Parametrierungen sehr zeitaufwendig, wenn diese manuell durchgeführt wird. Die Suche nach der optimalen Parameterbelegung kann jedoch selbst als Optimierungsproblem angesehen werden, welches von einem metaheuristischen Algorithmus gelöst werden kann. Dieser Ansatz nennt sich *Meta-Optimierung*. Der Nachteil der Meta-Optimierung ist der große Laufzeitbedarf. Glücklicherweise wird dieser Nachteil durch die immer weiter steigende Verfügbarkeit von Rechenleistung immer kleiner.

Diese Arbeit zeigt das Konzept der Meta-Optimierung in Form einer Implementierung für die heuristische Optimierungsumgebung HeuristicLab. Die wichtigsten Aspekte der Implementierung sind Generizität, Erweiterbarkeit, Parallelisierbarkeit, Benutzbarkeit und Austauschbarkeit der verwendeten Algorithmen. Die Leistungsfähigkeit der Implementierung wird anhand einer Reihe von Optimierungsszenarien demonstriert, deren Ergebnisse präsentiert und analysiert werden. Die Experimente für diese Arbeit wurden verteilt ausgeführt und verbrauchten insgesamt eine Laufzeit von mehr als 7,5 Jahren.

Chapter 1

Introduction

1.1 Motivation and Goal

Optimization is the search for an optimal solution out of a number of feasible solutions. There are many optimization problems which affect our daily life such as finding the fastest tour through a number of destinations or planning the optimal placement of machines in a factory. The number of possible solutions for the many optimization problems grows exponentially when the dimension of the problem (number of destinations, number of machines) is increased. This fact is called the *curse of dimensionality* [8] and quickly leads to an extremely large search space.

Metaheuristic algorithms are able find good solutions even in large search spaces in reasonable time. However, they do not guarantee to find an optimal solution. These algorithms iteratively try to improve one or many solution candidates by applying heuristics which are problem independent. Therefore, they do not require much information about the problem. Metaheuristic algorithms only require information on how to evaluate the quality of a solution candidate and how it can be manipulated. There exists a large variety of metaheuristic algorithms, many of which are inspired by natural processes. Evolution strategies, genetic algorithms, tabu search or simulated annealing are some examples.

Most of these algorithms have a number of behavioral parameters which have effects on their performance. For most algorithms, there exist established default values for these parameters, which are commonly used. However, it turns out that parameter values for an algorithm might work well on one problem instance but not so well on another. Researchers are often in the situation that they need to adapt the parameter values for every new problem instance. Unfortunately, finding the best parameter values is not a trivial task and it is difficult to understand the effect of every parameter. Many parameters have effects on other parameters which makes the problem even more complex. Metaheuristic algorithms exist for decades, but a parameterless algorithm which performs well on all problems has not been found yet. According to the *no free lunch* theorem by

Wolpert and Macready [65] it is impossible to find an algorithm which performs better than all other algorithms on all problems. Therefore, an algorithm needs to have parameters to be adaptable to the problem.

The problem of finding the optimal parameters for an algorithm can be seen as an optimization problem itself. The idea is to use an optimization algorithm as a meta-level optimizer to find the optimal parameter values of another algorithm. This concept is called *parameter meta-optimization* (PMO). There has been research in the area of PMO in the past, but in most of the approaches, specialized implementations were used for the meta-level algorithms which were not exchangeable. In this thesis, a flexible and extendable implementation of the PMO concept for the optimization environment HeuristicLab is presented. The implementation should be highly configurable, user friendly and should allow using any HeuristicLab algorithm as meta-level optimizer. To show the validity of the implementation, a number of parameter optimization scenarios are defined and executed. Since the execution of meta-optimization algorithms is highly runtime intensive, a distributed parallelization environment is used for all experiments in this thesis.

1.2 Structure and Content

This thesis is structured as follows: In Chapter 2 an introduction to metaheuristic optimization and parameter optimization in general is given. A brief overview of related work in the area of PMO is also presented. Chapter 3 provides the technical foundations for this thesis. The optimization environment HeuristicLab as well as the distributed computation infrastructure HeuristicLab Hive are described. In Chapter 4 the requirements for the implementation and the optimization scenarios for this thesis are outlined. Based on these requirements, the implementation details of PMO for HeuristicLab are shown in Chapter 5. In Chapter 6 the results of the parameter optimization scenarios are analyzed. Finally, Chapter 7 summarizes the content of this thesis and gives an outlook on possible improvements for the future.

Chapter 2

Theoretical Foundations

This chapter provides an introduction to the field of optimization with metaheuristic and evolutionary techniques. Furthermore, a selection of optimization problems that are relevant for this thesis is presented. This chapter also gives short explanations for the operators used in the experimental results in Chapter 6. The second part of this chapter describes the concept of parameter meta-optimization and gives an overview of related work.

2.1 Metaheuristic Optimization

For every optimization problem there exists a set of possible solutions which is called the *solution space*. A solution can be seen as an input for a known model. The model – also called *objective function* – calculates an output for a given input. The output is considered the *quality* of a solution. A globally optimal solution of an optimization problem is found, if there exists no other solution which evaluates to a better quality.

The best quality can be either the highest or the lowest, depending if it is a *maximization* or a *minimization* problem. The difficulty of a problem depends, among other factors, on its complexity. Linear optimization problems are problems where the objective function can be described as a linear function. These problems are solvable in polynomial time. There are algorithms such as the Simplex method developed by George Danzig [12] which can solve linear problems efficiently. Non-linear or discrete optimization problems are much harder to solve efficiently, except for some special cases [61]. In the general case, no algorithm is known until today which can solve such problems exactly in polynomial time with a deterministic Turing machine [47].

To search the solution space of combinatorial problems for an optimal solution, a backtracking algorithm might be used which just enumerates all possible solutions [11, 50]. However, the solution space of most problems is so vast that plain enumeration is not feasible in terms of runtime. An approach to reduce the number of solutions to be searched is to cut away search paths which are known

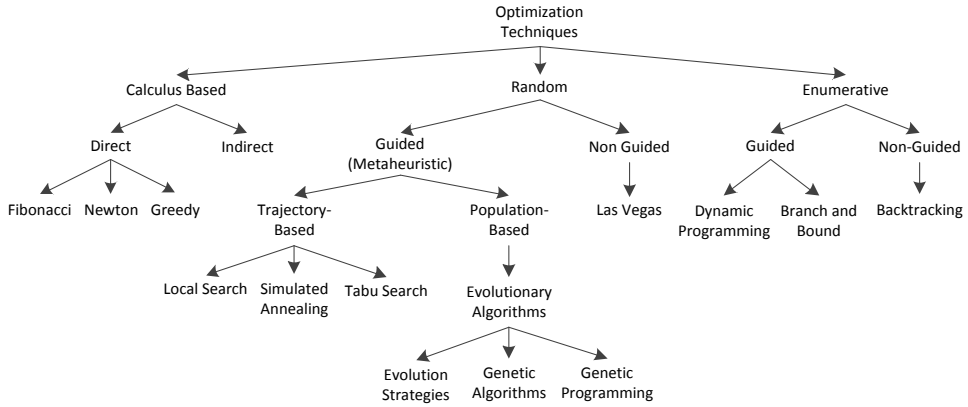

Figure 2.1: Taxonomy of optimization techniques [2]

to be suboptimal as early as possible. A representative of such an approach is *branch and bound* which estimates the lower and upper quality bound [35]. However estimating the quality requires knowledge about the underlying problem. Another way to tackle large solution spaces of non-linear optimization problems is to start with a constructed or randomly created solution and iteratively improve it. To improve the solution, problem-independent heuristics – also called *metaheuristics* – can be applied. Such metaheuristics are usually based on the quality of solutions, which allows the abstraction of the algorithm from the underlying problem. Of course, the evaluation of a solution as well as manipulating a solution are still problem-specific operations, but the algorithmic steps can be abstracted from the problem. Metaheuristic optimization techniques typically consist of some stochastic steps and consequently their results underlie a stochastic distribution. It is not guaranteed that a globally optimal solution is found. Figure 2.1 shows the taxonomy of optimization techniques in a broader context, yet the following sections will describe only metaheuristic methods in more detail, since they are most relevant for this thesis. Metaheuristic algorithms can be categorized into trajectory-based and population-based metaheuristics [10].

2.1.1 Trajectory-Based Metaheuristics

Trajectory-based metaheuristics consider only a single solution at the time. In each iteration step, the algorithm jumps to another solution in the solution space. The goal is to reach promising regions in the solution space without getting stuck in a local optimum. The following list gives a short overview of some well-known trajectory-based metaheuristics:

Local Search

Local search (LS) starts with a single randomly generated solution. LS iteratively searches the neighborhood of the current solution for better solutions. There are two types of LS, which differ in the way of selecting a better neighbor: *First improvement* LS picks the first solution found in the neighborhood that is better than the current solution, while on the other hand *best improvement* LS evaluates all solutions of the neighborhood and selects the best one. The improvement step is repeated until a termination criterion such as CPU time or the number of evaluated solutions is reached. Despite the efficiency and simplicity of this algorithm, it always reaches the next local optimum which it cannot escape due to the lack of a diversification strategy.

Simulated Annealing

Simulated annealing (SA) is inspired by the annealing of metal or glass where the structural properties of these materials become more and more stable as they cool off [33]. SA is an extended version of LS, with the difference that solutions from the neighborhood which are worse than the current solution, are selected with a certain probability. That probability depends on a temperature parameter which decreases over the runtime of the algorithm as well as on the quality difference. In the early stage of the algorithm, it is more likely to escape from a local optimum while in the later stages the algorithm intensifies the search in the current basin of attraction to find an optimum.

Tabu Search

Tabu search (TS) is an extended best improvement neighborhood search [25]. TS stores a history of the last n moves in a FIFO list (the tabu-list). A move represents the manipulation that was applied to the solution. The moves of the tabu-list are not allowed to be applied to the current solution. This modification helps the algorithm to avoid cycles and allows it to escape a local optimum more easily. The length of the tabu-list (tabu tenure) can be configured and steers the intensification and diversification of the search.

2.1.2 Population-Based Metaheuristics

In contrast to trajectory-based techniques, a population-based metaheuristic uses multiple solutions at the time. This set of solutions is also called population. Exploring the solution space at multiple locations at the time increases the diversity and makes these algorithms more robust compared to trajectory-based metaheuristics. Important representatives of population-based metaheuristics are *evolutionary algorithms (EA)*. There are multiple variations of EAs such as *evolution strategies (ES)*, *genetic algorithms (GA)*, and *genetic programming (GP)*.

As the name suggests, EAs are inspired by the natural evolution process which has first been described by Charles Darwin in 1859 [13]. In his concept of the *survival of the fittest*, he identified natural selection as one major aspect of evolution. In an environment which can only host a limited number of individuals, there is a higher chance for individuals to survive and reproduce the better they can compete for the given resources. Another important aspect of evolution is occasionally occurring mutation. By small modifications of the genetic information, individuals can develop new tenures that might or might not help them to adapt to environmental conditions. Individuals strong enough to survive and reproduce will inherit their genetic information to their offspring. Though EAs are inspired by the biological evolution, they are strongly abstracted from it.

In EAs an individual represents a solution. The quality of an individual is called *fitness*. Starting with a population of randomly created individuals, some individuals are selected to be the parents of the next generation. Individuals with high fitness are more likely to be selected as parents than individuals with low fitness. The selected parents are then crossed in the *recombination* step. Each pair of parents combines its genetic information in a way that the offspring contains traits from both parents. After a set of offspring is created, some individuals are manipulated in the *mutation* step. The newly created offspring represent a new generation and the loop continues with the selection of new parents. These steps are repeated until some termination criterion such as CPU time or the number of evaluated solutions is fulfilled. Multiple variations of EAs have been developed over time. The most influential EAs are described in the following.

Evolution Strategies

Evolution strategies (ES) were invented in the early 1960's by Ingo Rechenberg [51] and Hans-Paul Schwefel [52]. ES were designed to solve complex continuous optimization problems with real-vector encoding. The first ES variation used one parent individual and one offspring individual and is known as (1+1)-ES. The creation of a new offspring consists of the duplication and mutation of the parent. In the mutation step each element of the vector of real numbers is manipulated by adding a value independently sampled from a normal distribution $N(0, \sigma)$. Using a normal distribution causes most manipulations to be small while some are large, which aligns with observations in nature. Initially the σ parameter was supposed to be fixed, but Rechenberg soon developed a strategy which adaptively manipulates σ during the optimization process. The adaption is based on the famous 1/5 rule which says:

> "The ratio of successful mutations to all mutations should be 1/5. If it is greater than 1/5, increase the variance; if it is less, decrease the variance."

The rule is executed in periodic intervals of k generations. As computational power increased, a new ES-variant with a population of more than one individual called $(\mu + \lambda)$-ES was developed, where μ denotes the size of the population and λ denotes the number of children. In the selection step, the current population as well as the children are considered. Another ES-variant called (μ, λ)-ES only considers the children in the selection step. Instead of adapting the strength of the mutation by a deterministic rule (the 1/5 rule), the mutation variance can also be encoded into the individuals. Such an approach is called *self-adaptive* which means that the control parameter σ is subject to the same algorithmic operators (selection, mutation, recombination) as the solution itself.

ES are powerful and efficient problem solvers which employ the mutation operator as the main search operator. Recombination is used in some ES variants, but generally plays a less important role.

Genetic Algorithms

The traditional genetic algorithm (GA) was invented by John Holland in 1975 [29]. GAs start with an initial population of randomly created solutions, just like ES. Then a set of individuals is selected to act as parents for the next generation. The probability of being selected is proportional to the fitness of each individual. These parent-individuals are then crossed pairwise to create a set of children. With a certain probability, some of the individuals are mutated. Then the current population is replaced with the new offspring. If *elitism* is applied, one (or more) of the best individuals are not replaced by offspring. Elitism has proven to be helpful to avoid losing successful genetic information. Algorithm 2.1 shows the steps of a GA.

Algorithm 2.1: Standard Genetic Algorithm

1: $P \leftarrow$ generate initial population
2: evaluate P
3: **while** termination criterion not met **do**
4: $\quad P_{selected} \leftarrow$ select solutions from P
5: $\quad P_{offspring} \leftarrow$ recombine individuals from $P_{selected}$
6: $\quad$ mutate some $P_{offspring}$
7: $\quad$ evaluate $P_{offspring}$
8: $\quad P \leftarrow P_{offspring}$
9: **end while**
10: **return** P_0 (best solution)

Other than ES, GAs were originally designed to use binary-strings as solution representation for individuals. If the natural representation of the problem is not binary however (vector of real-values, permutation, etc.), a solution has to be encoded as well as decoded into a binary representation in order to apply the fitness function on it. Encoding and decoding will occur frequently during an

optimization process and therefore uses a lot of resources. Another problem with binary encoding is that crossover and mutation operations are not always semantically meaningful for a solution when applied on the binary representation. For example, adding, subtracting or averaging two real numbers is more meaningful as a crossover operation than combining sub-bit-strings of their binary representation. Therefore, modern GAs often operate on more natural encodings of the problem, thus encoding-specific operators for mutation and crossover are required.

Offspring Selection Genetic Algorithms

The offspring selection genetic algorithm (OSGA) is an enhanced version of traditional GAs developed by Michael Affenzeller and Stefan Wagner [1]. The goal of OSGAs is to avoid losing essential genetic information. Newly created children are evaluated and compared to their parents. Children that have a better fitness value than the parents are considered *successful*. Being better than the parents actually means having better fitness than the better or the worse parent. This comparison is guided by the comparison factor parameter ($CompFact$). A $CompFact$ of 1 means the offspring needs to be better than the better parent, a value of 0.5 means it has to be better than the average fitness of both parents, and a value of 0 means it has to be better than the worse parent. Only if an offspring is successful, it is allowed to be in the next generation, otherwise it is discarded. A new parameter called success ratio ($SuccRatio \in [0, 1]$) indicates the quotient of successful offspring required in a new generation. New children are created by recombination of parents until a sufficient number of successful children is available to form a new generation. The rest of the generation is simply filled up with individuals randomly chosen from the non-successful children. The selection pressure ($ActSelPress$) is defined by the ratio of the population size and how many children were actually created in order to fill the new population. If it becomes more difficult for the algorithm to achieve improvements of the existing individuals, the selection pressure grows. The maximum selection pressure acts as an additional termination criterion in OSGAs.

Genetic Programming

Genetic programming (GP) can be seen as a special variant of EAs that does not strive to find the optimal input values for a model, but instead to evolve a computer program which can solve a certain problem [34]. Such a computer program consists of a tree of statements (terminal and non-terminal) which represent the functions, operators, and variables. GP has been able to yield human competitive results in areas such as data-based modeling, electronic design, game playing, sorting, searching and many more. One form of data-based modeling is called *symbolic regression* which is described in more detail in Section 2.1.3.

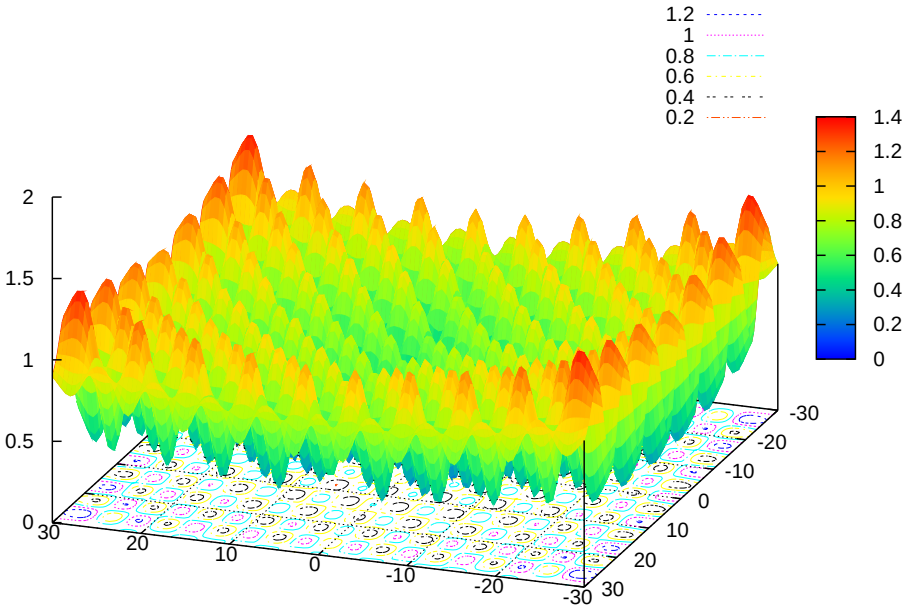

Figure 2.2: Visualization of the two-dimensional Griewank function

2.1.3 Optimization Problems

The following sections describe the optimization problems used in this thesis.

Test-Function Problems

To test and compare optimization algorithms, *test-functions* (or benchmark-functions) are widely used. Test-functions have a number of real-valued input values (dimensions) and are easy to compute. The test-function used in this thesis is the *Griewank* test-function:

$$f(x_1, ..., x_n) = 1 + \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}})$$

for $x_i \in [-600, 600]$. It is a real-valued, single objective minimization problem with an optimal value of 0.0. It is multi-modal which means there are multiple local optima. Figure 2.2 shows the fitness landscape of a two-dimensional Griewank test-function.

Traveling Salesman Problem

The *traveling salesman problem* (TSP) is a combinatorial optimization problem which has the goal to find the shortest round-trip through n cities when all pair-wise distances are known and each city may only be visited once [37, 43]. It has first been formalized in 1930 and is a very well researched optimization problem. Its complexity increases exponentially when the problem dimension is increased. The number of possible solutions is defined by $\frac{(n-1)!}{2}$. There are some methods that solve the TSP exactly, however it can also be solved very efficiently by metaheuristic algorithms. A solution of the TSP is represented as a permutation of indices of the cities. To evaluate a TSP solution the distances of all cities in the path are summed up.

Symbolic Regression and Classification Problem

Symbolic regression is a form of data-based modeling which strives to find a model which expresses the relationship between the input variables and the output variable in the form of a mathematical expression. This expression can consist of terminal and non-terminal symbols. The terminal symbols can be constant values or input variables. The non-terminal symbols can be mathematical functions or conditional expressions.

It is the goal to find a model which provides a maximum fit on the expected output data. When GP is applied to symbolic regression, these models are treated as individuals in a population. The input data for GP usually is a data set with multiple columns, where one column describes the expected output value (θ) and the other columns act as input values. To evaluate a model, it is applied on each row of the dataset. The output of the model ($\hat{\theta}$) is compared to the expected output of each row. θ is also called target variable.

Depending on how the comparison to the target variable is made, GP can be distinguished into *symbolic regression* and *symbolic classification*. In symbolic regression, the target variable is a real number and the error can be measured as the difference between the target variable value and the model output. In symbolic classification, the target value is discrete. There is a set of possible target values, also called classes. Classification works similar to regression, only that one or more threshold values are computed which separate $\hat{\theta}$ into multiple classes.

There are multiple possible ways to compute the quality of a model. A frequently used error measure is the *mean squared error* (MSE) which is defined as

$$MSE(\hat{\theta}) = E[(\hat{\theta} - \theta)^2].$$

Another popular method is called the R-Square correlation coefficient which is defined by:

$$R^2 = 1 - \frac{MSE}{variance}$$

where the variance devotes how irregular the problem is. Another measure for classification is the accuracy of a solution, which is the percentage of instances for which the correct class has been predicted.

The evaluation of a model is only applied to a subset of all data sets which is called the *training partition*. GP tries to find the optimal model to fit the data in the training partition. To test if the model is accurate, it is applied to another subset of datasets called the *test partition*. The test partition is completely independent from the training partition and the results indicate how well the model fits new data.

2.1.4 Operators

This section describes various operators which are used in this thesis. Some of these operators, such as the selection operators, are independent from the problem encoding while others, such as the crossover and the mutation operators, are encoding-specific. The problem encodings that are used in this thesis are *real-vectors* (test-functions), *permutations* (TSP) and *symbolic expression trees*. The names of the parameters mentioned in the following sections are from HeuristicLab.

Selection Operators

The following list provides brief descriptions of the selection operators used in this thesis:

- **LinearRank:** The selection probability of an individual is dependent on its rank in the whole population.
- **Proportional:** The selection probability of an individual is proportional to the fitness of an individual.
- **Random:** Individuals are chosen randomly.
- **Tournament:** A certain number of individuals (*GroupSize*) is chosen randomly. The best one of those is selected.
- **GenderSpecific:** Two parents are selected by using two different selection strategies (*FemaleSelector* and *MaleSelector*) [60].
- **GeneralizedRank:** Individuals are selected based on their rank with a varying focus on better quality which is steered by the *Pressure* parameter [57].
- **NoSameMates:** A conventional selection method is used (parameter *Selector*) to select two individuals. If those individuals have very similar quality (controlled by the parameter *QualityDifferencePercentage*), the selection is repeated a few times (*MaxAttempts* parameter) [28].

- **BestSelector:** Individuals are selected pairwise starting with the best.
- **WorstSelector:** Individuals are selected pairwise starting with the worst.

Mutation Operators for Real-Vector Encoding

- **Breeder:** One position of a real vector is changed by adding or subtracting a value of the interval $[(2^{-15} \cdot range; 2 \cdot range]$, where range is $SearchIntervalFactor \cdot (max - min)$ [41].
- **MichalewiczNonUniformAllPositions:** The manipulation is performed in an additive way and the strength of the manipulation is diminished over time. The *IterationDependency* parameter defines how much the mutation should depend on the progress towards the maximum iteration. A higher value means a stronger dependency on the iterations. This operator manipulates all positions of a real-vector [42].
- **MichalewiczNonUniformOnePosition:** This operator works just like the *MichalewiczNonUniformAllPositions* operator, but it only manipulates one randomly chosen position of the real-vector [42].
- **SelfAdaptiveNormalAllPositions:** This operator samples a new value for each position of the real-vector from a normal distribution where μ is the current value and σ is defined by the *Strategy* parameter [9].
- **PolynomialAllPosition:** Performs the polynomial manipulation on all positions in the real vector. The *Contiguity* parameter specifies the shape of the probability density function that controls the mutation. The *MaximumManipulation* parameter specifies the range of the manipulation [16].
- **PolynomialOnePosition:** Works just like the *PolynomialAllPosition* operator, but it only manipulates one randomly chosen position of the real-vector [16].
- **UniformOnePosition:** Manipulates one position of the real-vector by sampling a new value with a uniformly distributed probability from the range [42].

Crossover Operators for Real-Vector Encoding

- **Average:** Creates a new offspring by calculating the average value of the parents [9].
- **BlendAlphaBeta:** Creates a new offspring by sampling a random value from an interval. The interval is defined by the parent solutions while the interval in the direction of the better parent is extended by the factor *alpha*, in the other direction by the factor *beta* [56].
- **BlendAlpha:** Works just like the *BlendAlphaBeta* operator, but the interval is extended symmetrically in both directions by the factor *alpha* [56].
- **Discrete:** For each position in the real-vector, the value of one of the parents is chosen randomly [9].

- **Heuristic:** Adds the difference between the parent individuals to the better individual. The manipulation is weighted by a random factor in the interval $[0; 1)$ [66].
- **SimulatedBinary:** The positions of the real-vector are manipulated with a probability of 50%. If a position is manipulated it is either crossed, contracted, or expanded with equal probabilities. The *Contiguity* parameter controls the strength of the manipulation [15].
- **SinglePoint:** A random breakpoint is chosen in the interval $[1, N - 1]$ with N = length of the vector. The offspring is constructed from the first part of one parent and the second part of the other parent [42].
- **UniformAllPositionsArithmetic:** An offspring is created by calculating a weighted average of the real-vector positions. The *Alpha* parameter controls if the offspring is more similar to the first or the second parent. The parameter is in the interval $[0; 1]$. The manipulation is applied to all positions [42].
- **UniformSomePositionsArithmetic:** Works just like the *UniformAllPositionsArithmetic* operator, but it is only applied to some randomly chosen positions of the real-vector [18].
- **Local:** Works like the *UniformAllPositionsArithmetic* operator, but the *Alpha* parameter is chosen randomly for each position [18].
- **RandomConvex:** Works like the *Local* operator, but only one value of *Alpha* is chosen for all positions [18].

Mutation Operators for Permutation Encoding

- **Insertion:** Manipulates a permutation by randomly moving one element to another position [22].
- **Inversion:** Randomly selects a part of the permutation and inverts it [21].
- **Scramble:** Randomly selects a part of the permutation and repositions its elements randomly [55].
- **Swap2:** Manipulates a permutation by randomly selecting two elements and swapping those [21].
- **Swap3:** Manipulates a permutation by swapping three randomly chosen elements. It is implemented such that the first three positions are randomly chosen in the interval $[0,N]$ with N = length of the permutation with all positions being distinct from each other. Then position 1 is put in place of position 3, position 2 is put in place of position 1 and position 3 is put in place of position 2.
- **Translocation:** Moves a randomly selected part of the permutation to another position [42].
- **TranslocationInversion:** Moves a randomly selected part of the permutation to another position and inverts it [24].

Crossover Operators for Permutation Encoding

- **Cosa:** Cosa stands for *Cooperative Simulated Annealing* [62]. It manipulates the first parent by inserting random parts in normal and inverted order at different positions which are dependent on the second parent.
- **CyclicCrossover2 (CX2):** This operator is a variant of the classic cyclic crossover [21] and has been described in 2009 by Michael Affenzeller [3]. First, it randomly selects a position of the permutation and copies the first cycle from that position to the offspring. All remaining positions are copied from the second parent.
- **EdgeRecombination (ERX):** Creates an adjacency matrix out of the two permutations containing the outgoing and incoming links for each position. Starting from a random position a new permutation is created by choosing the position with the least number of edges from the adjacency matrix as the next position [64].
- **MaximalPreservation (MPX):** This operator yields to preserve the maximum amount of edges from both parent permutations [40]. It produces an offspring by copying a part of the permutation of the first parent. Further positions are added from the second parent with a hierarchy of rules to avoid infeasibility. Ulder et al. [58] empirically found that the length of the part copied at first should be 1/3 of the length of the permutation.
- **OrderBased (OBX):** Performs a crossover by randomly copying positions (not necessarily adjacent) from the first parent into the offspring permutation keeping the order. The rest of the positions are taken from the second parent also preserving the order [55].
- **Order2 (OX2):** This operator has been described by Affenzeller et al. [3] and is based on the traditional order crossover introduced by Eiben [21]. It creates an offspring by copying randomly chosen interval from the first parent, preserving the positions. Then the missing positions are copied from the start of the second parent in the order they occur.
- **PartiallyMatched (PMX):** First copies all positions from the first parent to the offspring. Then a randomly chosen segment of the second parent is copied to the offspring position by position. Whenever a position is copied, the number at that position currently in the offspring is transferred to the position where the copied number has been [22].
- **PositionBased (PBX):** Randomly chooses each position from either the first or the second parent [55].
- **UniformLike (ULX):** This operator tries to maintain the position in the permutation. Each position is randomly copied from one of the parents. Missing positions are filled randomly [57].

Mutation Operators for Symbolic Expression Tree Encoding

- **OnePointShaker:** Selects one random node of the tree and manipulates it.
- **ReplaceBranchManipulation:** Selects a branch of the tree and replaces it with a newly initialized subtree.
- **ChangeNodeTypeManipulation:** Selects a random tree-node and changes the symbol randomly.
- **FullTreeShaker:** Manipulates each constant and the weight of each variable in the expression tree.

Crossover Operators for Symbolic Expression Tree Encoding

- **SubtreeCrossover:** Selects one random node of each parent tree and swaps them. If the size limitation of one of the parent trees is violated the process is repeated until the size limitation is not violated.

2.2 Parameter Optimization

Different optimization algorithms have different strengths and weaknesses. Some work well on certain problem classes while others may not. Not only do they have different performance on different problem classes, they also behave differently on different problem instances. According to the *no free lunch* theorem (NFL) in heuristic search by Wolpert and Macready [65], there exists no algorithm which performs better than all other algorithms on all problems.

For the task of solving a concrete problem, the first step is to choose a proper algorithm. However since most algorithms have a set of behavioral parameters which affect the performance, it becomes even more difficult to find the *best* algorithm. The choice of parameter values can have a significant impact on the performance of the algorithm as demonstrated for behavioral parameters of GAs in [5] and [46]. Choosing the optimal parameter values for a single algorithm to solve a single problem is already non-trivial. In 1975 DeJong attempted to find the optimal parameters for GAs for all problems [14]. While some of these parameter values had been used as default values for GAs, in most cases the parameters need to be adapted for each problem instance to yield optimal performance.

Eiben [20, 21] distinguishes between two forms of adjusting parameters: parameter *tuning* and parameter *control*. Parameter control is the commonly practiced approach of finding the best parameter values *before* starting the algorithm. The parameters remain fixed throughout the runtime of the algorithm. In parameter control, the algorithm starts with initial parameter values which are changing throughout the run. The taxonomy of parameter setting methods is outlined in Figure 2.3.

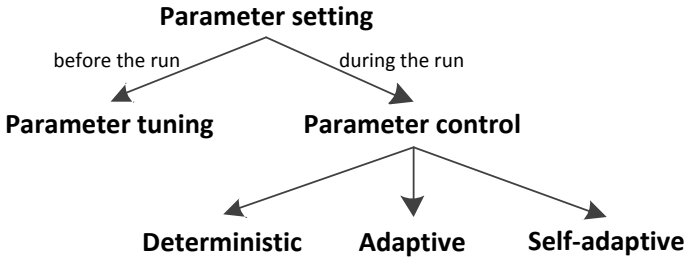

Figure 2.3: Taxonomy of parameter setting [20]

2.2.1 Parameter Control

When the parameter values change during the execution of an algorithm, there has to be some rule which governs this change. There are several different approaches to this [20]:

- ***Deterministic* Parameter Control:** A deterministic rule is used to adapt behavioral parameter values without reacting to any feedback from the search process. These rules are usually based on time or the number of iterations.
- ***Adaptive* Parameter Control:** When an adaptive rule is in place, feedback from the search process is used to control how the parameter values change. For example if the algorithm detects that the population diversity gets too low, it could increase the strength of the mutation operator and vice versa. The 1/5 rule of ES is an example of adaptive parameter control.
- ***Self-Adaptive* Parameter Control:** Self-adaption is inspired by the idea of evolution of evolution. The parameters are encoded into the chromosomes of the individuals and undergo the same mechanisms as the individuals, namely mutation, recombination, and selection. The individuals with better fitness spread their chromosomes within the population, so good parameter values are used more often. They can also change over time due to mutation. However, this approach is only feasible for parameters on the individual-level such as the mutation strength. Population-level parameters such as the population size or the selection operator cannot be adapted in such a way.

2.2.2 Parameter Tuning

Even though parameter control has its advantages, it is much more complex to find rules and methods which can adapt a parameter value in a way that the algorithm performance benefits, than finding a single parameter value upfront. Parameter control methods are also mostly tailored for real-valued parameters.

When it comes to choosing the right operators for an algorithm, parameter tuning is commonly used. However, the choice of parameter values remains a complex task which requires a user to have lots of experience and deep insight into parameter interdependencies and their impacts. Parameter tuning methods can be categorized as follows:

- **Ad-Hoc:** For choosing parameter values, many users rely on conventions and default values [53]. Based on the default values the parameter values are varied until acceptable performance is reached. The problem is that sometimes the optimal parameter values greatly differ from the default values. In addition, the number of repetitions for a parameterization under test is often too low. How well this approach works depends on the experience of the user. However, it is usually a very fast approach.
- **Experimental:** Performing systematic experiments with various parameter settings is a more scientific approach. However, trying all different parameter values and their combinations is impossible in most cases. This leads to the reduction of combinations or the variation of single parameters only. Optimizing parameters one by one does most certainly not lead to the optimal settings, as most parameters tend to have effects on others. Bartz-Beielstein devoted a whole book to the field of testing and experimentation research in evolutionary computation [4].
- **Parameter Meta-Optimization:** Another approach to parameter control is to see the search for optimal parameters as an optimization problem itself. From that perspective, the search for optimal settings has much in common with traditional optimization problems. Values for a set of input variables (parameter values) have to be found which maximize the quality when evaluated (execution of the algorithm). The interdependencies of the input variables and their effects on the quality are unknown and the search space is very large. To solve this optimization problem, an optimization algorithm is applied as a meta-level optimizer. That concept is called *meta-optimization*. It makes sense to use a metaheuristic optimization algorithm to solve that problem, which is the topic of this thesis. The algorithm which solves the parameter optimization problem is called *meta-level* algorithm, whereas the algorithm which is subject to be optimized is called *base-level* algorithm. Figure 2.4 illustrates the structure of this idea. The following section gives an overview of existing approaches in the field of parameter meta-optimization.

2.3 Related Work in Meta-Optimization

Seeing the parameter optimization problem as an optimization problem on its own is not a new idea. Optimizing parameter values with a meta-level algorithm was given many different names in literature such as meta-evolution, super-optimization, automated parameter calibration, meta-optimization, pa-

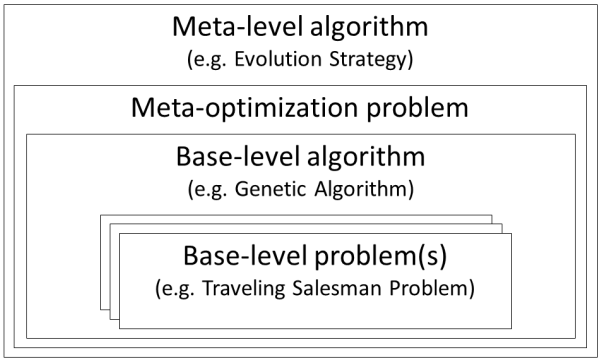

Figure 2.4: Meta-optimization concept

rameter meta-optimization etc. [6, 26, 38, 48]. The first work was done in 1978 by Mercer and Sampson [39]. They tried to find the optimal parameter values for a GA, but their experiments were very limited due to poor computational resources. Another early work was done by Grefenstette in 1986 who also used a GA to optimize the discrete parameters of a GA. He optimized against a set of low dimensional test-function problems to find the generally optimal parameters. Grefenstette used gray-code [63] as solution representation and applied binary mutation and crossover operators. Grefenstette also experienced problems with limited computational resources, which is why he could only perform experiments on very low dimensional problems. In 1994, Bäck [6] optimized behavioral parameters as well as operators of a GA by using a specialized hybrid of GAs and ES as meta-level algorithm. The base-level GA needed to be adapted for his approach. He was the first to use a parallel master-slave approach to overcome computational limitations.

Since meta-optimization is very time-consuming it was not until recently that computational power became available at a scale that allowed to perform realistic experiments. Meissner et al. [38] optimized the parameters of a particle swarm optimization (PSO) algorithm [19, 32] by applying an optimized PSO (OPSO) on meta-level. In 2009, Smit and Eiben [53] performed a comparison of various specialized meta-optimization techniques. They introduced some add-ons to meta-optimization to reduce runtime and the number of meta-level evaluations. A more advanced meta-optimization method called REVAC has been developed by Nannen and Eiben [45], which is not only able to optimize the parameters, but also to estimate the relevance of each parameter. This additional information allows the algorithm to focus on the most relevant parameters and thereby improve runtime speed. More recently Pedersen [48] used the local unimodal sampling (LUS) [49] method as a meta-level optimizer to find the optimal parameters for a differential evolution (DE) [54] algorithm. In 2006, Iunescu [31] showed parameter meta-optimization for the optimization environment HeuristicLab, which is also used in this thesis. Her approach was

capable of exchanging the base-level algorithm and problem, but the meta-level algorithm was a specialized evolutionary algorithm.

Many different approaches to meta-optimization have been developed. The following list summarizes some of the peculiarities of these approaches:

- In most previous approaches to meta-optimization the implementation is strongly coupled to the algorithm types that are used. On the meta-level, specialized variants of existing optimization algorithms were implemented. Some approaches were able to optimize multiple base-level at the time. While specialized meta-level algorithms have the advantage that they can be optimized for a low number of evaluations, the drawback is that they are not easily exchangeable by other existing algorithms.
- Another disadvantage of specialized meta-level algorithms is the increased difficulty to reproduce the results by other researchers. When no broadly known and well-documented algorithms are used, it becomes merely impossible to re-implement an algorithm that is only briefly described in a publication.
- Many approaches use binary encoding for parameter values, mainly due to performance advantages [7, 23, 26, 36]. Modern evolutionary algorithms use more natural encodings which represent the problem directly.
- Most meta-optimization approaches are custom implementations with command line interfaces, APIs, or rudimentary graphical user interfaces. This makes using these approaches difficult for people who were not involved in the development.
- Some authors did use parallelization concepts in their approaches, but most of them just mentioned that parallel and distributed computation is highly suitable for meta-optimization.
- Most approaches do only optimize against one optimization goal, which is the average quality achieved by the base-level algorithm. Other goals such as robustness and effort are only considered by few [48].

Chapter 3

Technical Foundations

This chapter introduces the optimization environment HeuristicLab which is used throughout this thesis as well as the distributed computation infrastructure HeuristicLab Hive. Without Hive it would not have been possible to run such large scale experiments for this thesis.

3.1 HeuristicLab

HeuristicLab (HL) is a paradigm independent, extensible, and open environment for heuristic optimization. It was originally created by Stefan Wagner in 2002 [59] and is now available as an open source project which is actively developed by the *Heuristic and Evolutionary Algorithms Laboratory (HEAL)*[1]. It is implemented in the programming language C# and uses version 4.0 of the Microsoft .NET framework. HL has been designed to be easy to use for experts, practitioners, and students. It uses a very open and flexible algorithm model which allows to design algorithms without writing any code. The following sections explain the most important concepts of HL.

3.1.1 Key Concepts

The following list provides a short description of the most important concepts of HL:

- **Plugin-Based Architecture**
  The main architectural concept in HL is its plugin system. In its core, HL only consists of a lightweight mechanism which is able load HL plugins. These plugins are dynamically discovered at runtime and can define dependencies on each other. The advantage of this system is that it is possible to add new functionality by just adding a new plugin. A plugin could add a new algorithm, a new problem, or a new user interface to the environment. In contrast to monolithic systems, it is possible to add new

[1] http://heal.heuristiclab.com

functionality without compiling the whole application. It also enables the deployment of bundles which are tailored to specific scenarios.

- **Graphical User Interface**
  Dealing with metaheuristic optimization algorithms on an API level is common for many frameworks in this field. Although a programming interface provides a high degree of flexibility for an expert user, it might not be the best choice for practitioners and students. That is why usability has always been a core aspect of HL. All components in HL can be viewed and manipulated in a meaningful way. There are easy ways to import problem definitions, configure algorithms, and analyze results so that inexperienced users as well as experts can use the system comfortably.
- **Cloning and Persistence**
  Another core concept in HL is that all items provide a mechanism for cloning and storing as a file. This means that algorithms, problems, and datasets can be stored at any time which meets the requirement of replicability. These mechanisms are designed in such a way, that developing new plugins requires very little implementation overhead.

3.1.2 Algorithm Model

There exists a large number of metaheuristic optimization algorithms with very different ideas and concepts. In addition, new algorithms and variations of existing algorithms are created steadily. HL addresses these demands by using a very generic algorithm model which uses radical abstraction. A HL algorithm is represented by a sequence of operations. Each of these operations manipulates data and the algorithm is executed by an engine. These three components (data, operators, and execution) represent the algorithm model in HL and are described in the following.

Data Model

Data values are used according to the HL data model. It wraps many standard data types such as integers, doubles, strings, or arrays into HL objects. Therefore, each of these values can be persisted, cloned, and viewed. In the traditional imperative programming model, each variable can be accessed via its variable name which points to the actual value in the memory. In the HL data model a variable is represented by a key-value-pair. The key is a string, representing the name of the variable and the value is a HL object.

These variables are arranged in scopes which are collections of variables. Scopes are combined in a hierarchical way, representing different layers of abstraction. An Operator is applied on a single scope. The operator can read and manipulate variables from that scope. It also has access to the parent-scope and to the sub-scopes. If a variable cannot be found in the current scope, the lookup is repeated in the parent scopes. Therefore, a variable is visible to all sub-scopes,

but can be overridden in one of them by a variable with the same name.

Operator Model

As already mentioned an algorithm consists of a sequence of executable instructions (operators). An operator can also be seen as a statement in imperative programming. In the operator model of HL operators are represented as objects. The major tasks of an operator is to manipulate data and to define which operator will be executed next. To manipulate data, each operator can define a list of parameters. Just as functions in imperative programming languages, these operators have formal and actual parameters. Formal parameter names define how the parameter is used inside the operator, whereas the actual parameter name defines by which name the parameter should be looked up in the scope tree. This concept makes the implementation of an operator reusable and independent from the algorithm they are used in.

The second purpose of HL operators is to decide which operator should be executed next. This mechanism essentially defines the flow of an algorithm. Each operator-instance may contain references to other operators. When an operator is executed, it decides which of its sub-operators should be executed next. Therefore, operators can also be used just like if- or switch-statements in imperative programming. Since such an operator-graph may contain cycles, it is possible to create loops. With the HL algorithm model it is possible to represent any algorithm that can be described in imperative programming languages, since major concepts such as sequences, branches, loops and recursion can be used.

Execution Model

A HL algorithm is an operator-graph which can be executed by an engine. A step in the algorithm consists of a tuple containing an operator and the scope it will be applied on. This tuple is called operation. An algorithm starts with an initial operation and an empty scope. The initial operation is put on a stack. In each step of the algorithm, the engine pulls an operation from the stack and executes it. An operation may return one or more successor operations which are pushed onto the stack in reversed order. The engine continues to execute operations from the stack until it is empty.

The separation of the algorithm definition and the execution engine has the advantage that the same algorithm can be executed by different kinds of engines. As many real-world applications of heuristic optimization are very runtime consuming, parallelization is a key requirement for optimization environments. HL provides such a capability through a parallel engine. Some operators produce multiple successor operations which are applied on different sub-scopes. If these operations are independent from each other, they can be parallelized. The evaluation of solution candidates in population-based algorithms is one example for that. Going even further, HL provides another engine – the Hive engine – which

is capable of executing these operations in the distributed computation environment *HeuristicLab Hive* which is described in more detail in the following section.

3.2 HeuristicLab Hive

HeuristicLab Hive is an elastic, scalable, and secure infrastructure for distributed computation. It is an open source system implemented in C# using version 4.0 of the Microsoft .NET framework. Hive consists of a server with a database and a number of computation *slaves*. A user can upload a job to the Hive server via a web service interface. Then the server distributes the jobs among the available slaves and retrieves the results after they are finished. The following list shows the most important aspects and features of HeuristicLab Hive:

- **Generic Usage**
  Though the main purpose of HeuristicLab Hive is to execute HL algorithms, it is designed to be completely independent from HL. Any type of job can be executed on Hive. A job is a .NET object which implements a certain interface. Hive offers a web service where jobs can be added, controlled, and removed.
- **Elasticity**
  Slaves can be added and removed at any time, even while they are calculating jobs. Therefore, jobs have the ability to be paused and persisted. The advantage of an elastic system is to be able to add resources at peak demand times and remove them if they are needed otherwise. There is no automatic snapshotting of job results, but a user can pause, modify, and resume a single job any time.
- **Heterogeneity**
  Slaves with different properties and constraints are supported. CPU performance and available memory can differ in every slave. Job scheduling respects these constraints. Hive is also independent of the network infrastructure. A slave can be directly connected through a fast high-speed link or it can be far away in a secured company network. The only requirement for slaves is to have the .NET 4.0 framework installed. The importance of network speed depends on the kind of jobs. For long running jobs, the network performance is more irrelevant while for short running jobs it can be a major overhead.
- **Schedule**
  It is possible to define schedules for slaves and groups of slaves. A schedule defines when a slave is allowed to calculate and when it is not. This is particularly useful to use office computers at night which would be unused otherwise.

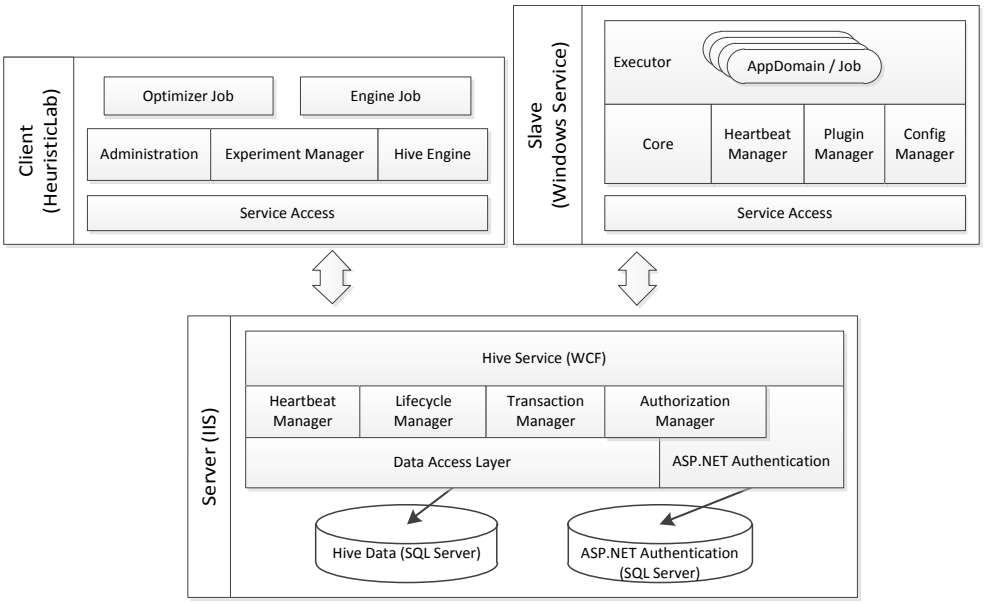

Figure 3.1: Hive components

- **Plugin-Independency**
  Slaves are lightweight applications and do not contain the assemblies needed to execute a job. When a job is uploaded, the files and assemblies that are needed to deserialize and execute the job are automatically discovered and also uploaded. Assemblies and files are grouped in *plugins* which are versioned. Hive takes care of distributing, storing, and caching those files efficiently. On the slave, these plugins are loaded inside a *.NET AppDomain* and the job can be executed. The possibility to submit a job with versioned plugins makes Hive independent of already deployed plugins and eliminates the need to update slaves.
- **Security**
  Hive puts emphasis on security. Users can control on which slaves their jobs should be computed. Hive ensures confidentiality and integrity of all communication by using X.509 certificates and message level encryption. Since a user can upload custom assemblies to the system, it is crucial that on a slave each job is executed in its own sandbox. Sandboxes are realized by *.NET AppDomains* with restricted permissions.

3.2.1 Components

Figure 3.1 illustrates the main components of HeuristicLab Hive. The following section describes these components.

- **Server**
  The server API is exposed as a WCF[2] web service which is hosted in IIS[3] 7.5. It offers methods to upload jobs and plugins, to fetch state information about jobs and to download results. It also exposes methods to control and administer the system including tasks such as grouping slaves or analyzing system performance. *ASP.NET Authentication*[4] is used to authenticate users and determine roles. The *Authorization Manager* authorizes users to access certain pieces of data. The *Transaction Manager* helps to encapsulate business processes in transactions and applies exception handling and rollbacks if necessary. The *Lifecycle Manager* contains methods which need to be executed periodically, usually once every minute. Among these lifecycle methods is the detection of timeouts of slaves, which leads to rescheduling of lost jobs. In addition, statistical data is computed every minute to enable the analysis of the system utilization over time. The *Heartbeat Manager* handles incoming heartbeats from slaves. It decides if a slave gets another job or not depending on the time schedule, free resources and the assignment of resource groups for the jobs. The *Data Access Layer* abstracts the details of the database which is accessed using Linq-To-Sql[5].
- **Slave**
  The slaves run as a windows service. The reason for this choice is that the slave can run no matter which user is logged in, even if no user is logged in. The *Config Manager* maintains state measures about the slave such as CPU usage, used memory, and available CPU cores. The *Heartbeat Manager* sends periodic messages to the server, reporting which jobs are calculated and how much resources (CPU cores, memory) are available. The server responds with a list of messages containing the next actions for the slave. These messages are queued and dispatched by the *Core* component. Communication between server and slave is always initiated by the slave to avoid problems which may be caused by NAT[6] and firewalls. Figure 3.2 shows an exemplary infrastructure with slaves deployed on various different locations behind firewalls. It also shows that it is possible to hook up external computers to be Hive slaves. It could enable companies to use their own computational resources at specific times (at night) whereas scheduling and distribution is done by the Hive server. The core controls a list of *Executors* which are responsible for sandboxing and executing jobs. The *Plugin Manager* takes care of fetching, caching, and copying plugins for a sandbox. The *Executor* needs to watch the calculated jobs, listen to certain events and observe failed jobs.

[2] Windows Communication Foundation
[3] Microsoft Internet Information Server
[4] http://msdn.microsoft.com/en-us/library/eeyk640h.aspx
[5] http://msdn.microsoft.com/en-us/library/bb386976.aspx
[6] Network address translation - http://en.wikipedia.org/wiki/Network_address_translation

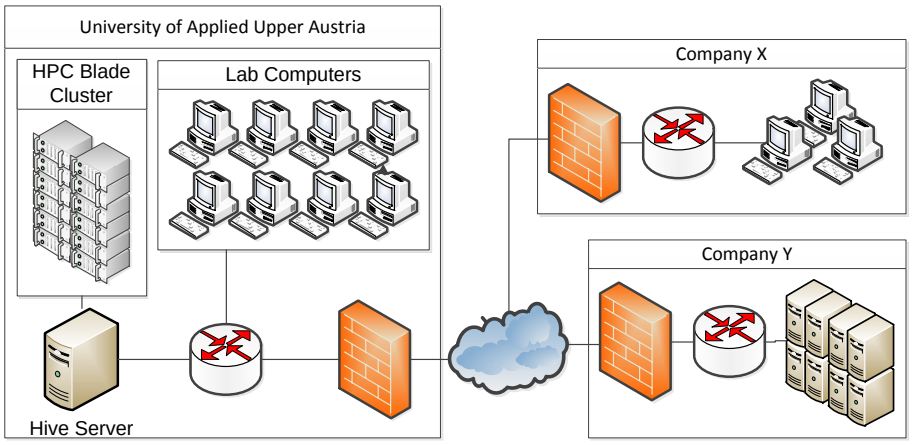

Figure 3.2: Exemplary Hive slave deployment with external companies

- **Client**
  The HL client for Hive features an administration user interface. It allows creating and arranging slave groups and observing the slaves' current state. HL currently supports two ways of executing algorithms in the Hive. Either a complete HL algorithm can be sent to Hive (Optimizer Job), or parts of an algorithm are executed on the Hive (Engine Job):
  - **Optimizer Job**
    An optimizer job encapsulates a HL optimizer which may be a single algorithm or an experiment containing a set of algorithms.
  - **Engine Job**
    An engine job encapsulates a HL Engine which executes an operator graph. Such an engine can be initialized with parts of an algorithm. This is what the Hive Engine does when a UniformSubScopesProcessor occurs in an operator graph. A UniformSubScopesProcessor means that an operator needs to be executed on a number of scopes. For each scope, an Engine Job is created and encapsulates the operator and the corresponding scope. Evolutionary algorithms for example need to evaluate each solution candidate of a population. If such an algorithm is executed by the Hive Engine, an Engine Job is created for each solution candidate and submitted to the Hive. The Hive Engine then waits until all jobs are finished and continues executing the algorithm.

All meta-optimization experiments presented in Chapter 6 are executed using the Hive Engine, which helped a lot to do so much computation. Also most of the cross-test experiments were executed using Hive. The computational resources were provided by the University of Applied Sciences Upper Austria.

Chapter 4

Requirements

This chapter is dedicated to the goals of this thesis and the requirements for the implementation. Based on the analysis of previous approaches in the field of parameter meta-optimization (PMO) in Chapter 2, the following requirements have been identified:

- **Exchangeability:** The implementation should be as generic as possible. It should not be necessary to do any adaptations to the meta-level and the base-level algorithm. Any meta-level algorithm should be able to optimize the parameters of any base-level algorithm. This would allow using the power of existing algorithms and new algorithms on the meta-level in the future.
- **Multiple Problems:** It should be possible to optimize algorithm parameters for multiple base-level problem instances at the same time, since it is often desirable to find the optimal settings for a class of problems and not for a single instance.
- **Multi-Objective:** The average solution quality that is achieved by an algorithm with a certain parameterization should not be the only optimization goal. There should also be the possibility to optimize for maximum robustness or minimum effort.
- **Appropriate Encoding:** Since many researchers in the past used binary encoding for parameter values, the implementation should use an encoding which represents the parameters in a more natural way (integers, doubles, booleans, operators). Also the optimization of parameters of operators should be possible.
- **Modularity:** The implementation should use exchangeable operators for manipulating parameter values in solution candidates. For each data type there should be specialized operators which can be selected by the user. The implementation should also use the HL plugin infrastructure and its automatic type discovery system, so that new operators can easily be added by new plugins, without changing any existing code. It should also be easy to add support for new parameter data types.

- **Symbolic Expression Grammar:** Despite supporting standard data types, the implementation should also be capable of optimizing symbolic expression grammars for symbolic regression and classification.
- **Configurability:** A user should be able to decide which parameters should be optimized. Further, it should be possible to define in which range each parameter value should be optimized. In that way a user can use his expertise and narrow search ranges to reduce the size of the search space. It should also be possible to define which operators should be used in the base-level algorithm.
- **Exhaustive Search:** It should be possible to use the configuration of search ranges to perform an exhaustive search within these ranges. This should be possible for multiple parameters and all of their combinations. Of course, there has to be some sort of step size to keep the number of possible parameter combinations tolerable.
- **Parallelization:** The implementation has to be designed in such a way that parallel and distributed execution of the meta-level algorithm is possible.
- **Usability:** Following one of HL's most important concepts, the implementation should have a rich and easy to use graphical user interface. It should allow even non-experienced users to get started quickly.
- **Analytics:** There should be possibilities to analyze and observe the progress of the meta-level algorithm.

Despite the functional requirements for the implementation, this thesis should prove the viability of the implementation by performing a number of different optimization scenarios:

- **Varying Problem Dimensions:** The optimal parameters for a GA should be found for multiple base-level problems with different problem complexities.
- **Varying Generations:** The optimal parameters for a GA should be found for multiple base-level problems with a different number of base-level algorithm generations.
- **Varying Problem Instances:** The optimal parameters for a GA should be found for different instances of the TSP.
- **Different Meta-Level Algorithms:** Different meta-level algorithms should be used to optimize the same base-level algorithm.
- **Symbolic Expression Grammar for Regression:** The symbolic expression grammar should be optimized for a symbolic regression problem.
- **Symbolic Expression Grammar for Classification:** The symbolic expression grammar should be optimized for a symbolic classification problem.

Chapter 5

Implementation

Based on the requirements defined in Chapter 4 the PMO problem for HL was implemented. This chapter describes the design of this implementation.

5.1 Solution Encoding

Any optimization problem in HL has to specify how solution candidates are represented. This representation is called *solution encoding*. An optimization algorithm may need to apply various operations on solution candidates, which manipulates them (e.g., mutation, or recombination of solution candidates). These encoding specific operations have to be provided by the solution encoding implementation. Several solution encodings are available in HL such as binary encoding, permutation encoding, vectors of real values, or vectors of integer values.

However, none of these existing solution encodings can be used to represent the parameters of an algorithm. Therefore, a new solution encoding had to be implemented. In the following sections, the parameter system of HL and the solution encoding for PMO are described.

5.1.1 Parameter Trees in HeuristicLab

HL features a generic concept for parameterization. Objects that are parameterizable implement the interface *IParameterizedItem*. It exposes a collection of *IParameter* objects. A class implementing this interface is responsible for adding *IParameter* objects to the collection when it is constructed. In HL many objects are parameterizable. Naturally, algorithms are among them. But also problems do have parameters which are relevant for problem specific operators. Even some operators have parameters. An example is the *TournamentSelector* operator which has a *GroupSize* parameter. Since some algorithms have operators as parameters, a composite tree structure of parameters emerges. The following list explains the different types of parameters in HL:

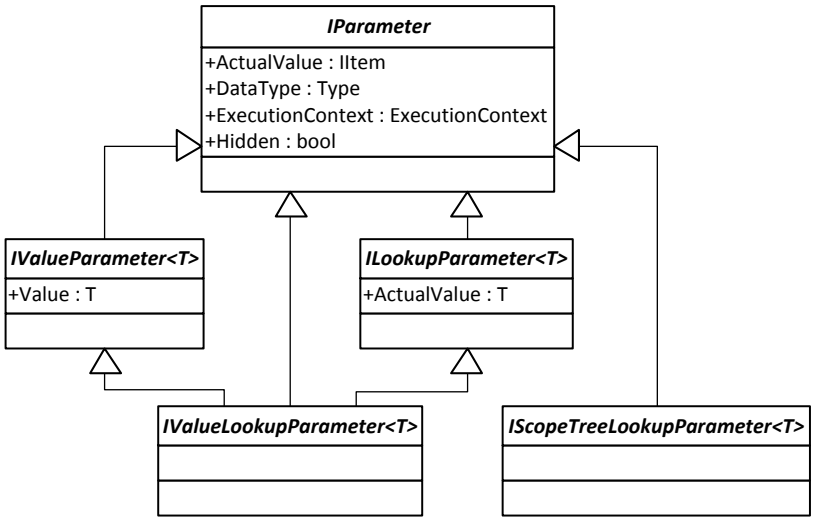

Figure 5.1: Interfaces for parameters in HL

- **IParameter:** Represents the base interface for all other types of parameters. Offers the property *ActualValue*, but does not necessarily store any value. It defines the data type of the value. The data type restricts the value which is assignable to the parameter. Figure 5.1 shows the inheritance structure of parameter types.
- **IValueParameter:** Stores the actual value and exposes it as the property *Value*. The property *ActualValue* just returns that value.
- **ILookupParameter:** Does not store the value, but rather searches the scope tree upwards for a variable with the name defined by the *ActualName* property. Reading and writing the *ActualValue* actually means reading and writing that found variable value. A lookup parameter is like a reference to another variable.
- **IValueLookupParameter:** Derives from *IValueParameter* and *ILookupParameter*. It is a combination of these two parameter types. If a value is set it behaves like a value parameter, but if the value is null it behaves like a lookup parameter.
- **IScopeTreeLookupParameter:** Searches the scope tree downwards and returns a collection of all occurrences of variables with a name defined by *ActualName*. A search depth specifies how deep the scope tree should be searched.

It is important to distinguish between these parameter types since it does not make sense to optimize all of them. It actually only makes sense to optimize *IValueParameters* and *IValueLookupParameters*. They are the only types that contain an actual value. An *IScopeTreeLookupParameter* is more of a collector

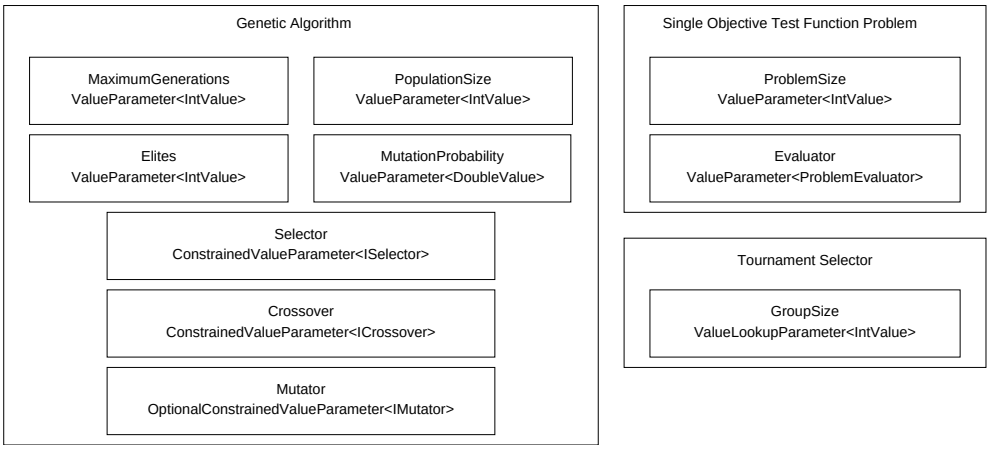

Figure 5.2: This figure shows the parameters and the parameter types of a GA, a test function problem and a tournament selection operator.

of values from subscopes and shall not be optimized. An *ILookupParameter* is just a reference to another variable. Therefore, it does not need to be optimized either.

A typical algorithm in HL has a problem and a set of parameters. As an example, Figure 5.2 shows the parameters of a GA (see Section 2.1.2), a test-function problem and a tournament selection operator. Some parameter values of the GA are parameterizable, but this is dependent on the actual value. For example there are some selection operators which do not have parameters, but others do have parameters such as the tournament selector. The GA and the test-function problem just have value parameters whereas the tournament selector has one value lookup parameter. Some of these value parameters are constrained which means there is a limited set of types which are assignable to them. The mutation and crossover operators for example are limited to the operators which are compatible with the solution encoding of the problem. Since a GA can also run without mutation, the mutation operator is optional which means this parameter can be null.

5.1.2 Parameter Configuration Trees

As described in the beginning of this chapter, optimization problems in HL need a solution encoding which provides the representation of a solution candidate as well as operations to manipulate it. For PMO, the representation of a solution candidate would be a tree of concrete parameter values for a parameterizable object. An additional requirement for PMO is that the way how each parameter should be optimized needs to be configurable. It should be possible to define which parameters of a parameterizable object should be optimized and for each of these parameters different configuration options should be available, depend-

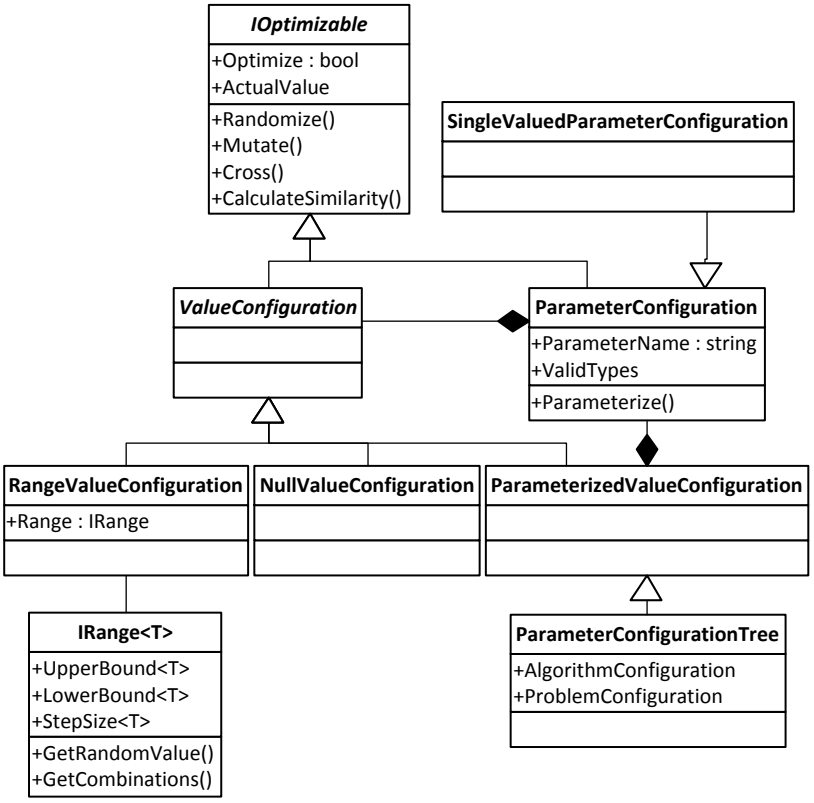

Figure 5.3: Classes of the parameter configuration tree solution encoding for PMO

ing on the type of parameter.

The solution representation for PMO for HL is called *parameter configuration tree*. Just as parameter trees, parameter configuration trees use a composite tree data structure. As shown in Figure 5.3, all elements of the tree do implement the interface *IOptimizable*. If an *IOptimizable* will be optimized or not optimized, depends on the property *Optimize*. The interface also offers the property *ActualValue* which returns the current value of a parameter. The method *Randomize()* samples a new random value. *Cross()* and *Mutate()* are needed for recombination and mutation in evolutionary optimization algorithms. The method *CalculateSimilarity()* calculates a value between zero and one which expresses the similarity between two parameter trees. This is needed for population diversity analysis in population-based algorithms, which is further discussed in Section 5.4.3. The following list explains the elements of the parameter configuration tree:

- **ParameterConfiguration:** One parameter is represented by a *ParameterConfiguration* object. It contains information about the parameter such as the name, the allowed data types, and a list of possible values. Each of

these possible values is represented as a *ValueConfiguration*. It also stores the index of the currently selected *ValueConfiguration*.

- **ValueConfiguration:** A *ValueConfiguration* represents a possible value for a parameter. It acts as an abstract base class for different types of configurations and encapsulates a concrete HL data type.
- **RangeValueConfiguration:** This type of value configuration is used for numeric data types of parameter values. It allows to define a numeric range (*IRange)* in which the optimal parameter value may be searched. The range is used for sampling random values as well as for checking violations of the range constraint after a recombination or mutation step.
- **IRange<T>:** A range has a lower bound, an upper bound and a step size (further explained in Section 5.1.3). It supports sampling new random values from that range (*GetRandomValue()*) as well as enumerating all possible values of that range with respect to the step size. This enumerative approach can be used to perform an exhaustive search (see Section 5.4.5).
- **ParameterizedValueConfiguration:** This type is used for types of parameter values which happen to be parameterizable themself. It contains a collection of parameter configurations, which corresponds to the parameters the value has.
- **NullValueConfiguration:** This type of value configuration represents the value null for a parameter. Since some parameters allow null as a value, there needs to be a special value configuration in such a case.
- **SingleValuedParameterConfiguration:** This type allows only one value configuration. In some cases it does not make sense to configure different values for a parameter value. The *SingleValuedParameterConfiguration* also allows to simplify the user interface by omitting the choice of different value configurations.
- **ParameterConfigurationTree:** This type is the root element for a parameter configuration tree. It derives from *ParameterizedValueConfiguration* and contains two single-valued parameter configurations. One stands for the algorithm and one for the problem. They are single-valued, because it would be too confusing for a user to be able to configure multiple instances of algorithms and problems for an optimization process.

To construct a parameter configuration tree, the parameter tree of the algorithm and the problem are traversed. For each *IParameterizedItem* a new *ParameterizedValueConfiguration* is created. Out of each parameter of the parameterized item, a new *ParameterConfiguration* is instantiated. When a *ParameterConfiguration* is constructed, it analyzes the assignable data types for the parameter and creates a list of valid values (one for each possible data type). Out of the valid types for the parameter, a list of *ValueConfigurations* is created. Depending on the type of each value, specialized value configurations are instantiated. For numeric types, *RangeValueConfigurations* are created and for

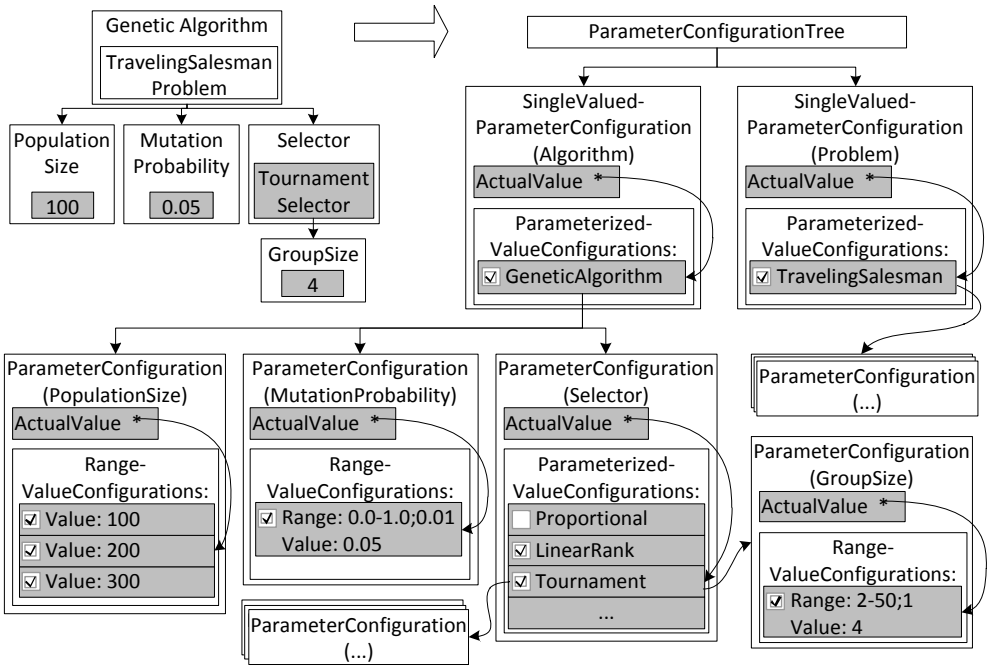

Figure 5.4: Object graph of a simplified example of a PMO solution encoding for the parameters of a GA

IParameterizedItems, the construction continues with new *ParameterizedValueConfigurations*. Figure 5.4 shows a simplified parameter tree of a GA along with the corresponding parameter configuration tree and Figure 5.6 shows the user interface for the configuration options of a GA.

5.1.3 Search Ranges

For numeric values, a search range can be specified. Search ranges enable a user to utilize knowledge about some parameters to reduce the size of the solution space and to reduce runtime. If for example a user wants to optimize the population size of a GA, the meta-level algorithm needs to know in which range the value should be, because it cannot create infinite values. Using maximum integer or long values as upper limits would result in extremely long runtimes of the base-level algorithm, so this is not an option either. Instead, each *RangeValueConfiguration* contains an object of the type *IRange<T>*. A range has a lower bound, an upper bound, and a step size. The step size defines what the resolution of the range should be. For example if the step size is 5, the lower bound is 100 and the upper bound is 120 then the optimization process can only use the values {100, 105, 110, 115, 120}. Ranges can be used both for integer and for

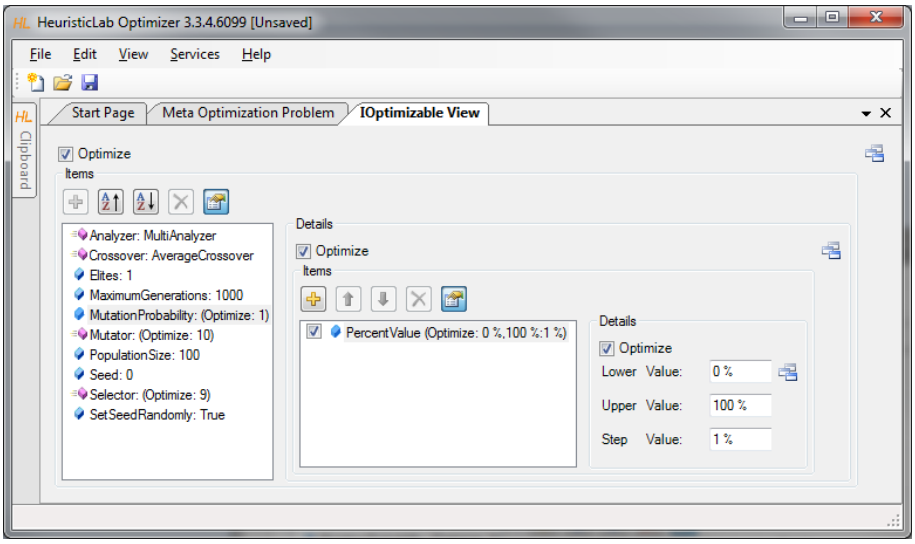

Figure 5.5: Shows the configuration options for the *MutationProbability* parameter. The left list-box shows the parameter configurations of a GA. The middle list-box shows a list of value configurations and the right panel shows the configuration options for a range.

double values. Figure 5.5 shows the user interface for the configuration options of the *MutationProbability* parameter of a GA.

Generally, it is a good idea to use very small step sizes. However, if a user wants to use the parameter configuration tree to do an exhaustive search of parameter values (see Section 5.4.5), the step size needs to be sufficiently large, or otherwise the number of combinations grows too large. Another reason to use large step sizes is to increase the probability to hit a cached solution (see Section 5.4.4).

5.1.4 Symbolic Expression Grammars

As stated in Chapter 4, PMO should support the optimization of symbolic expression grammars. As the grammar is a complex data type, some special extensions had to be made to make such grammars configurable and optimizable. Due to the flexible design of parameter configuration trees, such an extension was easily possible.

Symbolic Expressions in HeuristicLab

Symbolic expression grammars in HL are complex value types, which are used as parameters in symbolic regression and classification problems. A grammar contains a list of symbols which are used to construct symbolic expressions (solutions). The symbols can be mathematical and logical expressions as well as

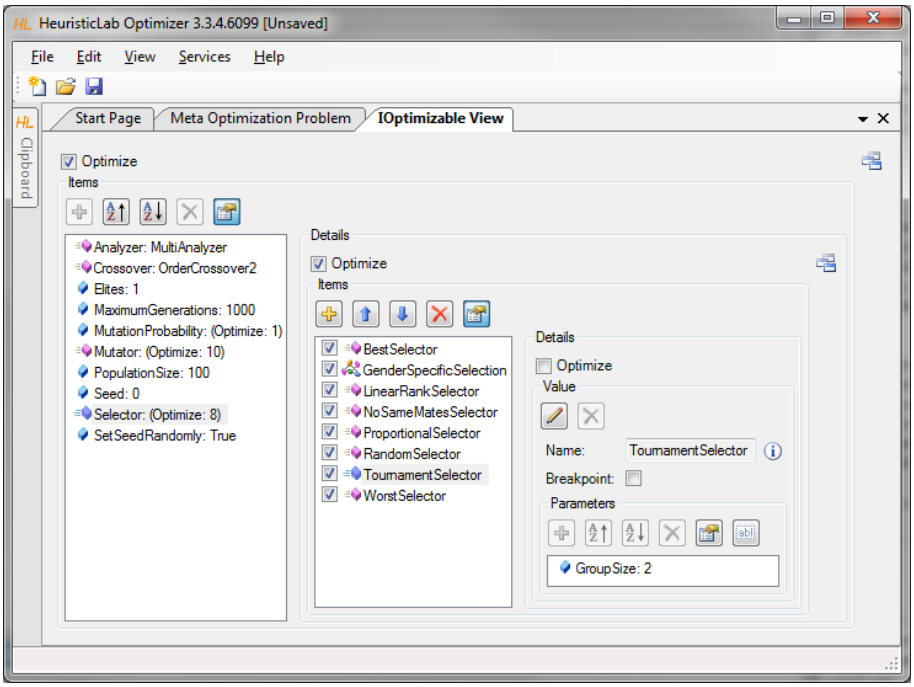

Figure 5.6: Shows the configuration options for the *Selectior* parameter. The middle list-box shows a list of possible values (value configurations). Each of them can have child-parameters which can also be optimized.

constants and variables. The grammar defines the arity of the symbols and how they can interact with each other. An *IfThenElse* symbol for example has to have three child symbols – one is restricted to boolean return values, the others (*then* and *else*) can be arbitrary expressions. Each symbol defines an *initial frequency* which affects the probability of a symbol being used in randomly created expressions. The *Constant* symbol has properties that affect the strength of the manipulation when such a node is mutated:

- **MinValue:** The minimum value the constant can have.
- **MaxValue:** The maximum value the constant can have.
- **ManipulatorMu/Sigma:** Used in an additive manipulation operation where a value is sampled from a normal distribution $\mathcal{N}(\mu, \sigma)$. The sampled value is added to the constant.
- **MultiplicativeManipulatorSigma:** Used in a multiplicative manipulation. In this case the constant is multiplied with a value sampled from a normal distribution $\mathcal{N}(1.0, \sigma)$.

Additive and multiplicative manipulations are used randomly with a chance of 50% for each constant in a symbolic expression. The *Variable* symbol also has properties that affect the strength of manipulation:

- **WeightMu/Sigma:** Each variable has a weight which is also modified in mutation operations. *WeightMu* and *WeightSigma* are used when the weight is first initialized from a normal distribution $\mathcal{N}(\mu, \sigma)$.
- **WeightManipulatorMu/Sigma:** Used in an additive manipulation operation of the weight where a value is sampled from a normal distribution $\mathcal{N}(\mu, \sigma)$. The sampled value is added to the constant.
- **MultiplicativeWeightManipulatorSigma:** Used in a multiplicative manipulation of the weight. In this case the constant is multiplied with a value sampled from a normal distribution $\mathcal{N}(1.0, \sigma)$.

Encoding of Symbolic Expression Grammars

The fact that symbolic expression grammars are custom types that do not use the traditional parameter/value paradigm, some special value configurations had to be implemented:

- **SymbolicExpressionGrammarValueConfiguration:** This class is derived from *ParameterizedValueConfiguration* and represents the configuration for a symbolic expression grammar. When it is instantiated, it iterates over all symbols of the grammar and creates a *SingleValuedParameterConfiguration* for each one. Each symbol is therefore treated as a parameter. It is single-valued because it does not make sense to add different values (symbols) and it is more convenient in the user interface.
- **SymbolValueConfiguration:** This class is also derived from *ParameterizedValueConfiguration*. It represents the configuration for a single symbol. When it is instantiated, it creates a value configuration for each property the symbol has. In any case, this involves the initial frequency of a symbol. In the case of the *Constant* and the *Variable* symbol this also involves the properties mentioned above.

These simple extensions of the parameter configuration tree make further adaptations obsolete. Crossover and mutation operations work just the same, as all symbols and properties are treated as parameters with basic data types. Therefore, the existing crossover and mutation operators (described in Section 5.3) can be used. The user interface is able to display the symbols just as if they were normal parameters. Figure 5.7 shows the configuration of a property of the *Variable* symbol in a symbolic expression grammar.

5.2 Fitness Function

In heuristic optimization, one step of the optimization process is to evaluate the quality of a solution candidate. This quality is also called fitness of a solution candidate. Based on the fitness of the solution candidates the selection step is performed in evolutionary algorithms. To evaluate the quality of a set of parameter values the base-level algorithm needs to be executed. Since the result

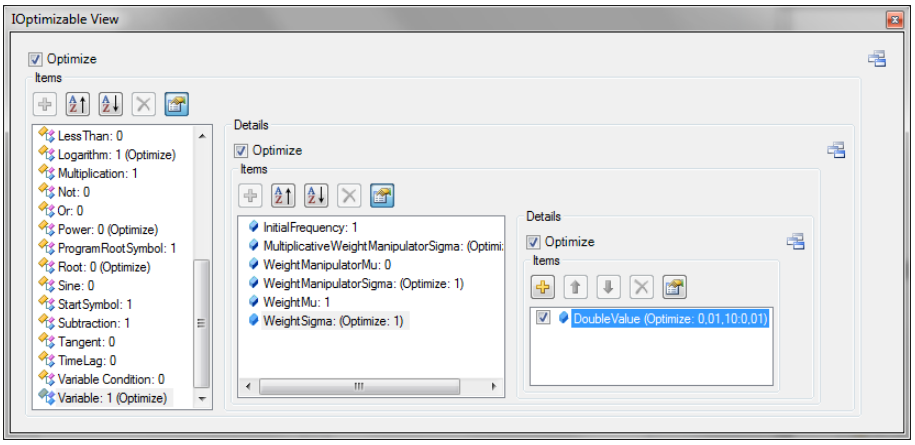

Figure 5.7: Configuration options of the *Variable* symbol

of an optimization algorithm underlies a stochastic distribution, it is necessary to repeat the execution n times. Based on the requirements defined in Chapter 4, the quality of a PMO solution candidate consists of the following components:

- **Solution Quality** (q): The average achieved quality of n base-level algorithm runs.
- **Robustness** (r): The standard deviation of the qualities of n base-level algorithm runs.
- **Effort** (e): The average number of evaluated solutions of n base-level algorithm runs. The number of evaluated solutions was chosen because the execution time is not a reliable measure in a heterogeneous distributed computation environment (see Section 3.2).

There are two options to tackle multi-objective problems. The simplest one is to compute a weighted sum or average out of the objective values. Another option would be to compute a pareto front [30] with the NSGA-II [17]. For the sake of simplicity, a weighted average was chosen for this implementation. Another requirement defined in Chapter 4 is that optimal parameter values for multiple problem instances should be found. Therefore, PMO allows a user to add multiple base-level problem instances. In order to evaluate the solution quality the base-level algorithm needs to be executed n times for each base-level problem instance.

However, different problem instances might yield quality values in different dimensions. An example would be to optimize a GA for three Griewank [27] test-functions in the dimensions 5, 50 and 500. A GA with a given parameterization might result in the quality values 0.17, 4.64 and 170.84. Using a simple arithmetic mean for the fitness function would overweight the third problem a lot. It is the goal to find settings which are suited for each problem equally well. To tackle this issue, normalization has to be applied on all results of each run (q, r, e). The

reference values for normalization are the best values from the first generation (r_q, r_r, r_e). Furthermore each objective needs to be weighted (w_q, w_r, w_e). The quality (Q) of a solution candidate (c) for m base-level problems is thereby defined as:

$$Q(c) = \frac{1}{m} \sum_{i=1}^{m} \frac{\frac{q_i}{r_q} w_q + \frac{r_i}{r_r} w_r + \frac{e_i}{r_e} w_e}{w_q + w_r + w_e}$$

5.2.1 Handling of Infeasible Solutions

There are cases when parameter values are set in such a way that the algorithm cannot run. Some parameters in HL are constrained so that setting the wrong values is impossible. However not all of these cases can be verified before actually executing the algorithm. For example, there is currently no method in HL that verifies that the population size has to be larger than the number of elites. In evolutionary computing there are different approaches on how to handle infeasible solutions as discussed in [44]:

- **Preserving Feasibility:** The transformation operators of the algorithm guarantee to create only feasible solutions. However due to the limitations of parameter validation in HL this method is not applicable.
- **Repairing Infeasible Solutions:** This approach allows the creation of infeasible solutions, but it guarantees that each individual is repaired by a repair function before being evaluated. Implementing such a repair function for HL would be tricky. The best way to do it would be to manipulate parameters and try to evaluate until no exception is thrown by the algorithm.
- **Penalty Function:** When a penalty function is used operators are allowed to produce infeasible solutions. These solutions are penalized when they are evaluated. The penalty value may be dependent on how infeasible a solution is or it may be fixed.
- **Repeating:** If the operators are stochastic, the manipulation of a solution could be repeated until it is feasible.

In PMO for HL the penalty function approach has been chosen. The penalty value is defined by the worst quality value from the first generation.

5.3 Operators

Algorithms in HL are very abstracted from the optimization problem and the problem encoding. However, some algorithm operation need to manipulate solution candidates and therefore they need to be implemented specifically for each encoding. Among these encoding-specific operators are operators that create solutions, crossover-, and mutation-operators for population-based algorithms as well as move-operators for trajectory base algorithms. For this thesis, only

operators for population-based evolutionary algorithms have been implemented. In the following sections these operators are described.

5.3.1 Solution Creator

The solution creator is responsible for creating a random solution candidate. This operator is mainly used to initialize an initial generation of randomized individuals. The solution creator for PMO needs one initial parameter configuration tree which can be generated from a given algorithm and a problem (as described in Section 5.1.2). A user can configure the tree by enabling optimization for certain parameters and by adapting search ranges and possible values. To create one random solution candidate the initial parameter configuration tree is cloned. Then the cloned parameter configuration tree is traversed. For each optimizable element the method *Randomize()* is called. Depending on the type of element, different actions are performed:

- **ParameterConfiguration**: The *Randomize()* method is called on all child value configuration elements. A new actual value is selected randomly from the list of available value configurations.
- **ParameterizedValueConfiguration**: The *Randomize()* method is called on all child parameter configuration elements.
- **RangeValueConfiguration**: A new uniformly distributed random value is sampled out of the specified range.

5.3.2 Evaluator

The evaluation operator applies the fitness function on a solution candidate in order to compute its quality. In the case of PMO, the base-level algorithm needs to be parameterized and executed n times for each base-level problem. The *PMOEvaluator* therefore creates an instance of the base-level algorithm for each repetition and for each base-level problem. Then all these instances are parameterized with the current parameter values of the parameter configuration tree. The base-level algorithm instances are configured with the corresponding base-level problems. Then each of the base-level algorithm instances is executed (sequential, parallel, or distributed depending on the selected execution engine). After all runs have finished, the results of each run are collected. For each base-level problem the average quality, the standard deviation of the qualities, and the average number of evaluated solutions is computed. The last step in the evaluation is the normalization of these values. As described in Section 5.2 the qualities of the different base-level problems need to be divided by the reference value for each base-level problem. The average of the normalized fitness measures represents the final quality value for one solution candidate.

The evaluation of solution candidates is the computationally most intense part of meta-optimization. Fortunately, the solution evaluations are independent from each other, which makes parallelization a viable option. In HL, parallel

evaluation of solution candidates is possible either locally with multiple threads or distributed using HL Hive (see Section 3.2). For this thesis, HeuristicLab Hive was used excessively to overcome the massive runtime requirements. The used resources are split up into high performance computing infrastructure and regular desktop computers. Up to 120 CPU cores were used to compute the experiments shown in Chapter 6.

5.3.3 Mutation

The mutation operation in evolutionary algorithms is supposed to manipulate one solution candidate. It is applied with a certain probability (*MutationProbability*). To manipulate a parameter configuration tree, the elements of the tree need to be selected. The selected elements need to be manipulated in different ways, depending on their data types. The following operators were implemented for selecting the nodes of the tree that should be manipulated:

- **ParameterConfigurationOnePositionManipulator**: Randomly chooses one parameter out of the parameter configuration tree which is selected to be optimized. On this parameter a type-specific manipulation operator is applied.
- **ParameterConfigurationAllPositionsManipulator**: Applies the type-specific manipulation operators to all parameters of the parameter configuration tree which are selected to be optimized.

After the elements nodes that should be mutated are selected, type-specific manipulation operators are applied on each of them:

Mutation of Parameter Configurations

When a parameter configuration is mutated, a new current value is selected randomly from the list of available value configurations. The probability for each value to be selected is the same.

Mutation of Boolean Values

Since boolean values can only have two different values (*true*, *false*), one of these values is randomly chosen by the mutation operator.

Mutation of Integer Values

- **UniformIntValueManipulator**: Sets the parameter value to a new random value that is sampled from the specified range.
- **NormalIntValueManipulator**: Sets the parameter value to a new random value that is sampled from a normal distribution $\mathcal{N}(\mu, \sigma)$ where the mean μ is the current value and the variance σ is 10% of the size of the

specified range. If the newly sampled value happens to lie outside of the range, it is re-sampled until it lies within the range.

Mutation of Double Values

- **UniformDoubleValueManipulator**: Apart from generating a double value, this mutation operator works exactly like the *UniformIntValueManipulator* described earlier.
- **NormalDoubleValueManipulator**: Apart from generating a double value, this mutation operator works exactly like the *NormalIntValueManipulator* described earlier.

5.3.4 Crossover

The crossover (or recombination) operation is supposed to combine the genes of two solution candidates to create a new one. The two parent solution candidates are selected by the selection operator of the meta-level algorithm. For PMO this means to combine two parameter configuration trees. To do so, one of the parameter configuration trees is cloned. Then the newly created offspring is traversed along with the second parent. Each *IOptimizable* node is then crossed with the corresponding node of the second tree. The concrete crossover operation is dependent on the data type of the node:

Crossover of Parameter Configurations

When two parameter configurations are crossed, the index of the currently selected value configuration is chosen from one of the parents randomly.

Crossover of Boolean Values

The boolean value of either of the two parent solution candidates is chosen randomly.

Crossover of Integer Values

- **DiscreteIntValueCrossover**: The value of one of the parents is chosen randomly.
- **AverageIntValueCrossover**: The average of the values of the parents is computed.
- **NormalIntValueCrossover**: A new value is sampled from a normal distribution $\mathcal{N}(\mu, \sigma)$ where the mean μ is the value of the better parent and the variance σ is the absolute difference of the values of the parents. If the newly sampled value lies outside of the range, it is re-sampled until it lies within the range. This operator strongly emphasizes the better parent

solution candidate and correlates the strength of manipulation with the difference of the parents.

Crossover of Double Values

- **DiscreteDoubleValueCrossover**: Apart from generating a double value, this crossover operator works exactly like the *DiscreteIntValueCrossover* described earlier.
- **AverageDoubleValueCrossover**: Apart from generating a double value, this crossover operator works exactly like the *AverageIntValueCrossover* described earlier.
- **NormalDoubleValueCrossover**: Apart from generating a double value, this crossover operator works exactly like the *NormalIntValueCrossover* described earlier.

5.4 Analysis

In HL, each problem implementation can provide analyzers for an algorithm. Analyzers are executed after each iteration of an algorithm. They can create result values which are displayed in the graphical user interface. The analyzers which were implemented for PMO are described in the following sections.

5.4.1 Population Analyzer

This analyzer adds the individuals of the current population to the results collection. It enables a user to inspect the properties of all solution candidates of the population during runtime. Figure 5.8 shows a list of individuals of a population.

5.4.2 Best Solution History Analyzer

Every time a new best solution candidate is found, it is inserted into a list of best solution candidates. This list is inserted into the results collection along with the generations in which they were produced. This list makes it possible to analyze how the best solution evolved over the course of multiple generations.

5.4.3 Population Diversity Analyzer

The population diversity analyzer computes the similarity between each pair of solution candidates of the current population. The similarity between two solution candidates is defined as follows:

- **ParameterConfiguration**: The average similarity value of all selected value configurations.

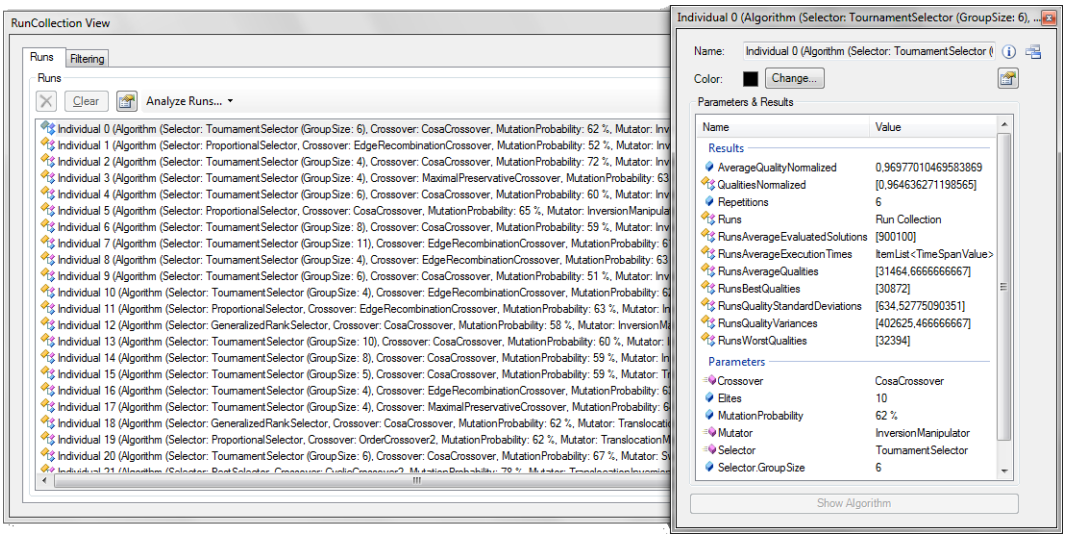

Figure 5.8: The population analyzer shows a list of individuals of the current population of a meta-optimization run. Detailed information about the fitness and the parameter values of each individual are available.

- **ParameterizedValueConfiguration**: The average similarity of all parameter configurations.
- **RangeValueConfiguration**: The similarity (S) of two numeric value configurations (c_1, c_2) which have a range with a lower bound (α) and an upper bound (β) is defined as follows:

$$S(c_1, c_2) = \max(0, \frac{d(c_1 - c_2)}{\beta - \alpha})$$

  d is a parameter which affects how similar two values are in relation to the range. In the experiments for this thesis d was set to two, which means that two values have a similarity of zero if their difference is the same or larger than half of the range size.

The result is a similarity matrix with values ranging from zero to one. Figure 5.9 shows such a similarity matrix as a heat map. The similarity matrix can be stored for each generation, which allows analyzing the diversity of the population over time.

5.4.4 Solution Cache Analyzer

This analyzer keeps a history of previously evaluated solutions. Its purpose is to avoid the evaluation of solutions which have already been evaluated. Because the solution evaluation might be executed in a distributed environment, the solution cache has to be filled after the whole population was evaluated. This is done by

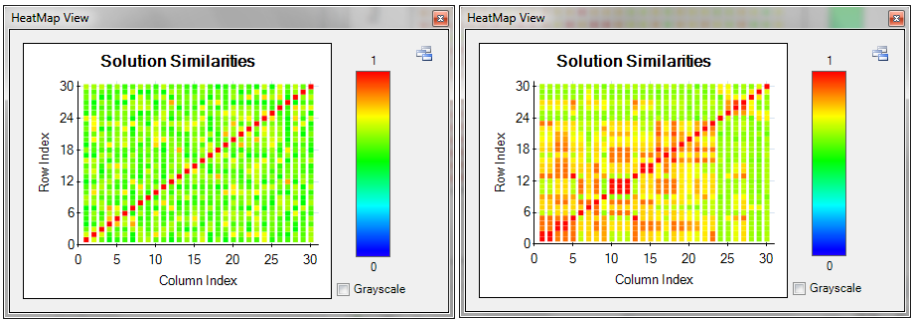

Figure 5.9: The image on the left shows the diversity of a population of 30 solution candidates in an early stage of the optimization process. The red diagonal indicates that every individual has a similarity of one to itself while the similarity to the other solutions is quite low. The image on the right shows the population diversity in a later stage of the optimization process. Some groups of individuals are very similar to each other, indicated by red color.

the solution cache analyzer. Furthermore, the solution cache analyzer allows running statistical analysis of all solution evaluations that occurred during the optimization process. However, due to the high memory demand of storing all solution results, the analyzer can be configured to store only the solutions of the previous generation.

5.4.5 Exhaustive Search

This thesis focuses on the automatic optimization of parameters for heuristic optimization algorithms by using a meta-optimization approach. However, in some scenarios it might as well be interesting to explore the whole search space. Of course, this is only possible if a small amount of parameters and narrow search ranges are explored. For this requirement an enumerator (*ParameterCombinationsEnumerator*) which implements the interface *IEnumerator<IOptmizable>* was created. This special enumerator is able to enumerate all possible combinations of parameter values of a parameter configuration tree. It is initialized with an *IOptimizable* object. Depending on the type of the object, the enumerator initializes a list of child-enumerators:

- **ParameterConfiguration**: A *ParameterCombinationsEnumerator* is created for each value configuration which is selected to be optimized.
- **RangeValueConfiguration**: The range offers the method *GetCombinations()* which returns the list of values from that range. It starts with the lower bound value and adds the step size until it reaches the upper bound value. An enumerator of this list is added to the child-enumerators.
- **ParameterizedValueConfiguration**: A *ParameterCombinationsEnumerator* is created for each parameter configuration.

```
var alg = new GeneticAlgorithm();
var problem = new TravelingSalesmanProblem();
var config = new ParameterConfigurationTree(alg, problem);

// configure search ranges
Configure(config);

// iterate over all possible parameter combinations
var experiment = new Experiment();
foreach (ParameterConfigurationTree curConfig in config) {
  var curAlg = new GeneticAlgorithm();
  curConfig.Parameterize(curAlg);
  experiment.Optimizers.Add(curAlg);
}

// run the created algorithms
experiment.Start();
```

Program 5.1: This code shows how easy it is to iterate over all possible parameter combinations of a parameter configuration tree.

Having added the child-enumerators, each enumerator is moved to the first element. In this state, all parameter values with ranges have their values set to the lower bound value and all parameter configurations have the first value configuration as actual value. In the *MoveNext()* method the first child-enumerator is moved to the next element and the actual value of the *IOptimizable* is updated. Subsequent calls of *MoveNext()* do move the first enumerator forward until it reaches the last element. When this happens, the second child-enumerator is moved forward and the first enumerator is reset to the first element. When the second child-enumerator reached the last element, the next enumerator is moved forward and the first and second are reset. This process continues until all child-enumerators have reached the last element.

Doing an exhaustive search is as easy as iterating over all combinations by using the *ParameterCombinationsEnumerator* and parameterizing a new algorithm instance in every iteration which is shown in Program 5.1. Since the iterated parameter configuration tree is modified and not cloned in every iteration, it is very fast and can iterate hundreds of thousands of combination in few seconds. Figure 6.8 shows the results of such an exhaustive search.

Chapter 6

Experimental Results

One of the goals of this thesis is to find the optimal parameterization for some optimization algorithms and problems in order to test and validate the implementation presented in Chapter 5. Several different optimization scenarios have been selected. They are described in the following sections.

6.1 Scenario 1: Varying Problem Dimensions, Short Runtime

In this scenario (S_1) the parameters of a GA (see Section 2.1.2) are optimized. Multiple different Griewank test-functions (see Section 2.1.3) are used as base-level problems. As a meta-level optimizer, a GA is applied. The goal of this scenario is to find out how the optimal parameter values differ when different problem dimensions of test-functions are used. The base-level algorithm is configured in such a way that the runtime is kept quite short (in contrast to S_2 where runtime is longer). The following sections describe the parameters that will be used for each level as well as the different base-level problem sets which are used.

6.1.1 Meta-Level Algorithm

The parameters of the meta-level GA are shown in Table 6.1. These settings were tuned manually. The rather small population size and maximum generations are related to the immense runtime requirements. A repetition-count of six has been chosen as a trade-off between high runtime demand and high evaluation accuracy. The average quality is used as the only optimization objective in this scenario, because weighted fitness evaluation has not yet been implemented at the time of these experiments.

Parameter Name	Value
Algorithm	GA
Maximum generations	100
Population size	30
Mutation probability	10%
Elites	1
Selection operator	Proportional
Mutation operator	OnePositionsManipulator
Mutation operator (IntValue)	NormalIntValueManipulator
Mutation operator (DoubleValue)	NormalDoubleValueManipulator
Crossover operator (IntValue)	NormalIntValueCrossover
Crossover operator (DoubleValue)	NormalIntValueCrossover
Evaluation repetitions	6
Quality weight	1.0
Robustness weight	0.0
Effort weight	0.0

Table 6.1: Parameters of the meta-level algorithm (m_1) for S_1

6.1.2 Base-Level Algorithm

The parameter configuration (c_1) of the base-level GA is shown in Table 6.2. The population size and the number of maximum generations are fixed for this scenario in order to keep the runtime approximately equal for all solution candidates (100'000 solution evaluations, 1–2 minutes runtime). When ranges are specified, the number after the colon represents the step size. For some parameters, such as the *iteration dependency* or the *maximum manipulation*, a set of concrete values has been chosen instead of ranges to reduce the search space and simplify the problem. An exhaustive search in the space of all possible parameter values would be rather impossible. Due to the curse of dimensionality [8] the number of possible combinations increases dramatically as parameters are added to be optimized. The number of possible parameter value combinations for c_1 is more than 29 billion. Assuming an average base-level algorithm runtime of 90 seconds, it would take 30 million years to do an exhaustive exploration of the search space.

6.1.3 Base-Level Problems

The following base-level problems were used in this scenario:

- f_1: $griewank(500)$
- f_2: $griewank(1'000)$
- f_3: $griewank(1'500)$
- f_4: $griewank(2'000)$

The Griewank test-function is a minimization problem which has an optimal value of 0.0 (see Section 2.1.3 for more details).

Parameter Name	Values
Algorithm	GA
Maximum generations	1'000
Population size	100
Mutation probability	0%–100%:1%
Elites	0–100:1
Selection operator	LinearRank, Proportional, Random, Tournament (Group size: 2–100:1), GenderSpecific (Female selector: Proportional, Male selector: Random), BestSelector, WorstSelector
Mutation operator	Breeder, MichalewiczNonUniformAllPositions (Iteration dependency: 2, 5, 10), MichalewiczNonUniformOnePosition (Iteration dependency: 2, 5, 10), SelfAdaptiveNormalAllPositions (Strategy parameter: 1, 12), PolynomialAllPosition (Contiguity: 2, Maximum manipulation: 1, 12, 120), PolynomialOnePosition (Contiguity: 2, Maximum manipulation: 1, 12, 120), UniformOnePosition, null (no mutation)
Crossover operator	Average, BlendAlphaBeta (Alpha: 0.75, Beta: 0.25), BlendAlpha (Alpha: 0.5), Discrete, Heuristic, Local, RandomConvex, SimulatedBinary (Contiguity: 2), SinglePoint, UniformAllPositionsArithmetic (Alpha: 0.33), UniformSomePositionsArithmetic (Alpha: 0.5, Probability: 0.5)

Table 6.2: Parameter configuration (c_1) of the base-level algorithm for S_1

6.1.4 Results and Discussion

One meta-optimization run was performed for each base-level problem f_1–f_4 as well as for the problem set $\{f_1 f_2 f_3 f_4\}$. The parameter configuration c_1 defined in Table 6.2 was used. Table 6.3 shows the best parameter values (p) found in each run. All runs were executed at the same time using HeuristicLab Hive and finished within 3 days.

Interestingly the solutions $p(c_1, f_2)$, $p(c_1, f_3)$, $p(c_1, f_4)$, and $p(c_1, f_5)$ seem to be very similar. Since *Best-* and *WorstSelector* behave almost the same, the only significant outlier is the *mutation probability* value of 46% in $p(c_1, f_3)$. In this case, the optimization process could have done further improvement as the quality value is worse than the quality for $p(c_1, f_4)$.

To validate if each parameterization is really optimal for the problem set it has been optimized for, cross-testing was performed for all results. In these cross-tests, each parameterization was applied to a different base-level problem. Figure 6.1 shows that $p(c_1, f_1)$ performs significantly better on f_1 (as expected) than on the other problems, while $p(c_1, f_2)$ to $p(c_1, f_5)$ perform almost equally well on f_2 to f_5, but not so well on f_1.

To validate the outcome of the meta-optimization runs, one-dimensional explorations of the search space were performed for some parameter values of $p(c_1, f_1)$. To do this kind of exploration, one parameter value is varied while all

	$p(c_1, f_1)$	$p(c_1, f_2)$	$p(c_1, f_3)$	$p(c_1, f_4)$	$p(c_1, \{f_1 f_2 f_3 f_4\})$
Elites	1	1	1	1	1
Crossover operator	BlendAlpha	Average	Average	Average	Average
Mutation operator	SelfAdaptive-NormalAllPos	Michalewicz-NonUniform-AllPos	Michalewicz-NonUniform-AllPos	Michalewicz-NonUniform-AllPos	Michalewicz-NonUniform-AllPos
Iteration dependency		10	2	2	10
Mutation probability	27%	30%	46%	25%	25%
Selection operator	Tournament	WorstSelector	BestSelector	BestSelector	WorstSelector
Tournament group size	5				
Generations	100	100	100	100	100
CPU time (days)	23	9.9	19.6	21.6	86
Average qualities	f_1: 1.3952	f_2: 12.5546	f_3: 25.3463	f_4: 20.18	f_1: 5.86 f_2: 11.79 f_3: 17.51 f_4: 23.58

Table 6.3: Solutions of the meta-optimization runs for S_1

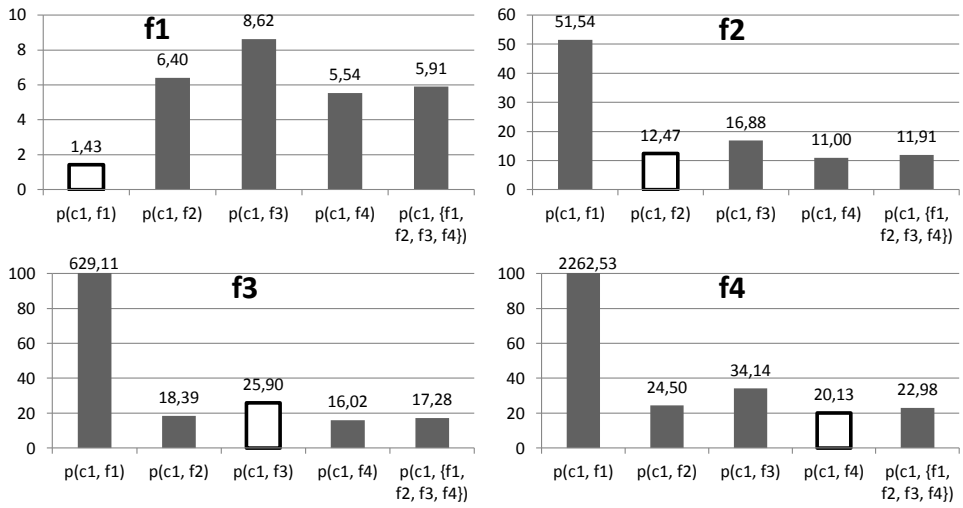

Figure 6.1: Average qualities achieved by different parameterizations for the problems f_1 to f_4. Each parameterization (p) was repeated 10 times on each base-level problem.

other parameter values are fixed. These experiments do not prove that $p(c_1, f_1)$ represents the optimal parameterization (global optimum), but they provide some insight into the meta-fitness-landscape in a single dimension and if the best value in that dimension was found (local optimum).

Figure 6.2 shows the exploration of the *Elites* parameter. Each dot in the chart shows the best quality of the execution of the base-level algorithm (genetic

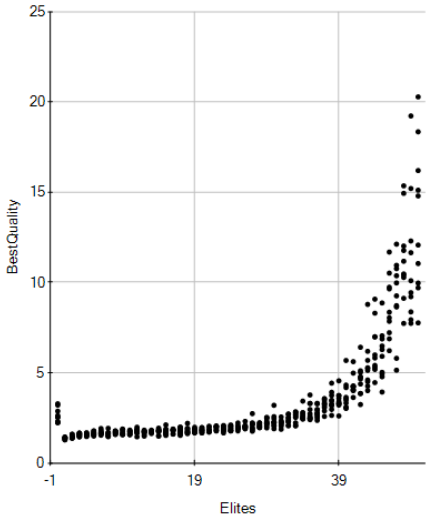

Figure 6.2: One-dimensional exploration of the meta-fitness-landscape for the *Elites* parameter of $p(c_1, f_1)$. Each dot represents a run. 10 repetitions were performed for each value.

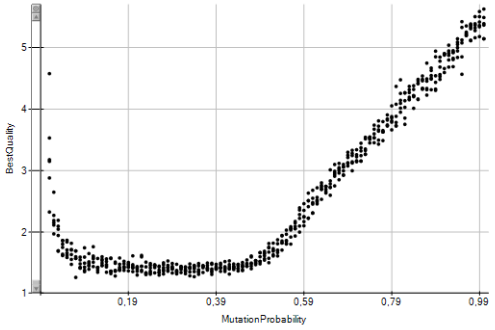

Figure 6.3: One-dimensional exploration of the meta-fitness-landscape for the *MutationProbabilty* parameter of $p(c_1, f_1)$. Each dot represents a run. 10 repetitions were performed for each value.

algorithm) parameterized with $p(c_1, f_1)$, only the *Elites* parameter was varied with values from 0–50. Obviously the best value seems to be 1 as suggested by $p(c_1, f_1)$. In a similar way the *GroupSize* parameter was tested in the range of 2–50. The best value according to $p(c_1, f_1)$ is 5. Figure 6.4 shows that 5 is indeed the optimal value, assuming all other parameter values fixed. Both the *Elites* and the *GroupSize* parameters have a clear optimum in contrary to the *MutationProbability* parameter. Figure 6.3 shows that any value for the *MutationProbabilty* between 20% and 50% yields approximately the same quality. The value of 27% which was identified by $p(c_1, f_1)$ lies in the middle of this range.

The results presented in this scenario clearly show that the choice of parameter values is significantly influenced by the dimension of the problem. A good parameterization for a problem is not necessarily good on another problem, even if only the dimension is varied.

6.2 Scenario 2: Varying Problem Dimensions, Long Runtime

In this scenario (S_2) the optimization scenario S_1 is repeated with the same algorithms and parameters, only the maximum number of generations of the base-level algorithm is increased to 10'000 (resulting in 1'000'000 solution eval-

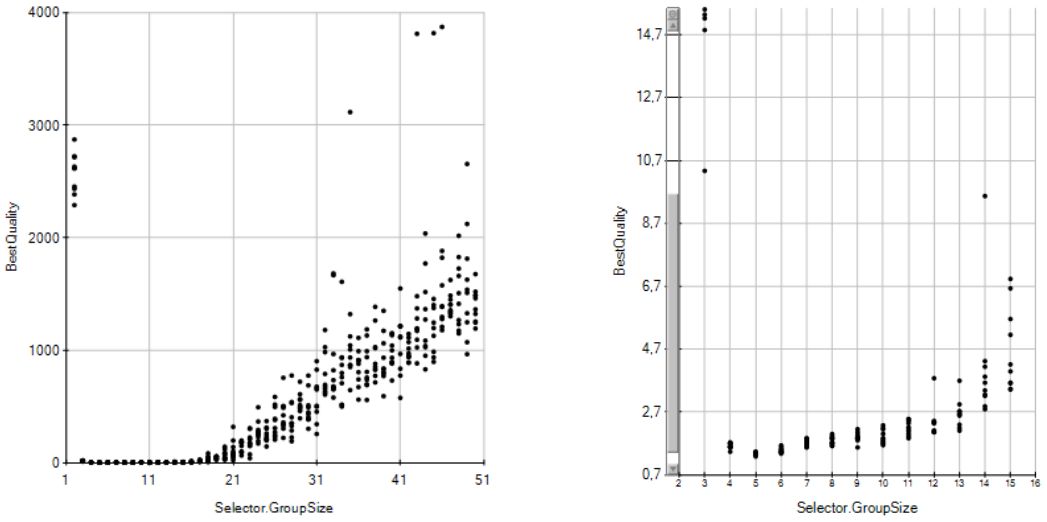

Figure 6.4: One-dimensional exploration of the meta-fitness-landscape for the *GroupSize* parameter of $p(c_1, f_1)$. Each dot represents a run. 10 repetitions were performed for each value. Both charts show the same runs, only the right one is zoomed to make the best value of 5 visible.

uations). This parameter configuration will be referred to as c_2. That change resembles a more realistic scenario, with an average base-level algorithm runtime of around 10 minutes.

6.2.1 Results and Discussion

Just as in S_1 one meta-optimization run was performed for each problem f_1–f_4 (as defined in Section 6.1.3) and one run was performed for multiple problems $\{f_1 f_2 f_3 f_4\}$. Table 6.4 shows the results of each run. All runs were performed at the same time using HeuristicLab Hive. $p(c_2, f_1)$ finished after 9 days, $p(c_2, f_2)$ after 26 days and $p(c_2, f_3)$–$p(c_2, f_5)$ were stopped after 33 days of execution due to limited time and computational resources.

The *BlendAlphaCrossover* was identified as the best crossover operator and the *BreederGeneticAlgorithmManipulator* as the best mutation operator for all problem sets. Remarkably, the *Elites* parameter seems to be negatively correlated to the problem dimension. A significant difference between $p(c_2, f_1)$ and the other results is the *LinearRankSelector* and the lower *MutationProbability*. To analyze the results each parameterization was tested against each problem set in the form of cross-tests (Figure 6.5). The cross-tests show that $p(c_2, f_1)$ and $p(c_2, f_4)$ perform best on the problem set they have been optimized for. $p(c_2, f_2)$ and $p(c_2, f_3)$ perform well but not best on the problem sets they have been optimized for. $p(c_2, f_5)$, which is supposed to work well on all problems, performs

	$p(c_2, f_1)$	$p(c_2, f_2)$	$p(c_2, f_3)$	$p(c_2, f_4)$	$p(c_2, \{f_1 f_2 f_3 f_4\})$
Elites	10	7	1	0	8
Crossover operator	BlendAlpha	BlendAlpha	BlendAlpha	BlendAlpha	BlendAlpha
Mutation operator	Breeder	Breeder	Breeder	Breeder	Breeder
Mutation probability	24%	68%	57%	80%	75%
Selection operator	LinearRank	Tournament	Tournament	Tournament	Tournament
Tournament group size		5	5	10	3
Generations	100	100	79	66	25
CPU time (days)	66.7	147.7	197.2	228.2	329.9
Average qualities	f_1: 6.6×10^{-10}	f_2: 1.0×10^{-6}	f_3: 5.0×10^{-5}	f_4: 0.02	f_1: 4.6×10^{-6}, f_2: 0.0038, f_3: 0.0054, f_4: 0.09548

Table 6.4: Solutions of the meta-optimization runs for S_2. Note that PMO for f_3, f_4, and f_5 has been stopped before reaching 100 generations due to time constraints.

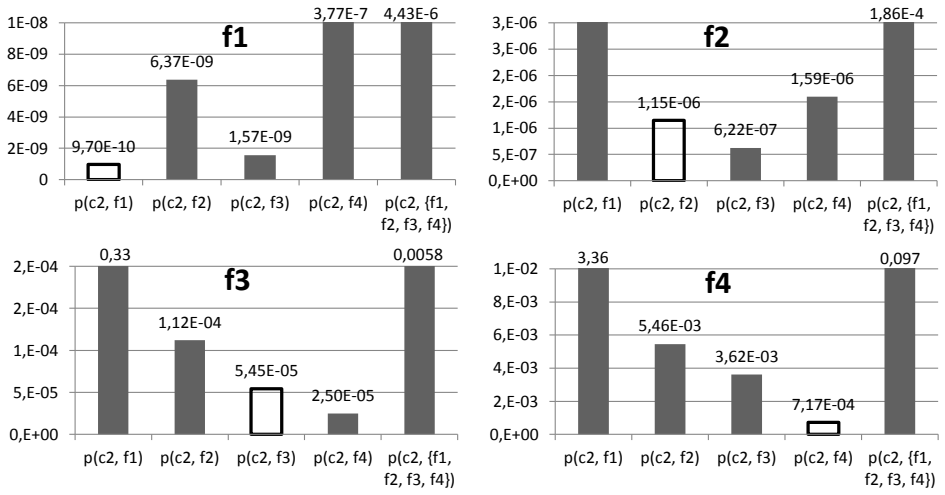

Figure 6.5: Average qualities achieved by different parameterizations for the problems f_1 to f_4. Each parameterization p was repeated 10 times on each base-level problem.

not very good which might be caused by stopping it after only 25 generations.

The results in this scenario clearly show that problems in different dimensions require different parameterizations. The resulting parameters may also be surprising for some researchers, because they differ significantly from the *default* suggestions. The *Elites* parameter for instance is usually set to one, but the results show (except for $p(c_2, f_3)$) that this setting is not always the best – at least in combination with the other values found here.

Problem	Generations	Average quality	
		optimized for 1'000 generations	optimized for 10'000 generations
f_1	1'000	1.4275	32.5088
f_2	1'000	12.4656	233.6063
f_3	1'000	25.8992	1840.8791
f_4	1'000	20.1268	5720.0282
f_1	10'000	0.6300	9.7025×10^{-10}
f_2	10'000	2.2378	1.1464×10^{-6}
f_3	10'000	4.4841	5.4466×10^{-5}
f_4	10'000	3.1242	7.1727×10^{-4}

Table 6.5: Shows the qualities (average of 10 runs) of a multiple GA runs with different settings for a different number of generations.

6.3 Scenario 3: Varying Generations

In this scenario (S_3) the effects of the number of generations is analyzed for GAs. In order to do so, the results from S_1 and S_2 are compared. In both scenarios, the parameters of a GA are optimized to solve the Griewank test-function problem in different dimensions. In S_1 the base-level GA runs for 1'000 generations, while in S_2 it is configured to run for 10'000 generations. The results of these optimizations have been shown in Table 6.3 and 6.4 in the previous sections.

To validate these results, the settings were cross-tested. The results of these cross-tests are shown in Table 6.5. It is evident that the parameter values from S_1 clearly outperform the parameter values of S_2 for 1'000 generations at every problem. The opposite is the case for 10'000 generations. For further analysis, the quality charts for f_1 and 10'000 generations is shown with the settings from S_1 (Figure 6.6) and S_2 (Figure 6.7). In Figure 6.6 the quality improves significantly until generation 1'000, but then it stagnates and does not improve anymore. In contrast, the quality chart in Figure 6.7 shows slower convergence, but it does not suffer from stagnation at all. The quality improves in almost every iteration until the last generation.

6.4 Scenario 4: Varying Problem Instances

In scenario 4 (S_4) the parameters of a GA for different instances of the TSP (see Section 2.1.3) are optimized. Two different algorithms are used on the meta-level. The goal is to see how the optimal parameter values differ for different TSP instances and also how the two different meta-level algorithms perform in comparison.

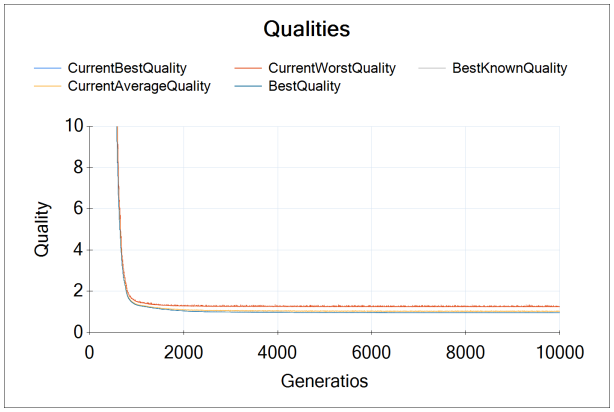

Figure 6.6: Shows the quality history of a GA run for f_1 over 10'000 generations. The settings that were used in this run were optimized for 1'000 generations.

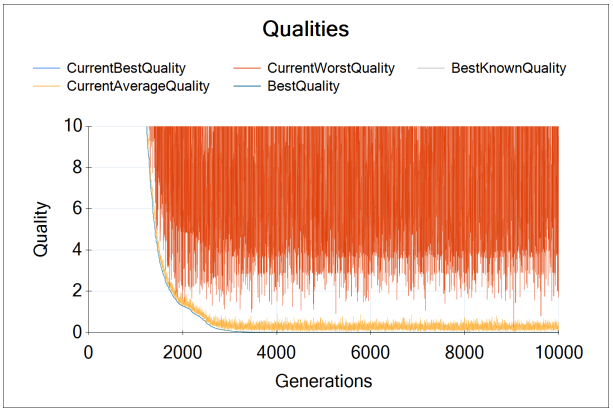

Figure 6.7: Shows the quality history of a GA run for f_1 over 10'000 generations. The settings that were used in this run were optimized for 10'000 generations.

6.4.1 Meta-Level Algorithm

On the meta-level, a normal GA and an offspring selection genetic algorithm (OSGA) are used. The configuration of both meta-level algorithms is shown in Table 6.6. As described in Section 2.1.2 OSGAs compare the quality of a new offspring to the quality of the parent individuals. It is then decided if the new offspring may pass into the new generation. In m_3 a lower- and upper bound of 1.0 is used for the comparison factor, which results in a *strict offspring selection*. That means that only offspring which are better than both parents may survive. A *success ratio* of 1.0 means that 100% of the new generation have to fulfill that criterion. Since the evaluation of a solution candidate for PMO takes a long time, it would be inefficient to evaluate new offspring one after the other. Instead, a number of offspring is created and evaluated at once. This evaluation can then be parallelized. In m_3 the number of offspring created at once is 60.

Other than in S_1 and S_2, the robustness and the effort are also included in the fitness function. However, their weights are chosen very low, since the main objective should still be the best quality. The meta-optimization algorithm should not primarily optimize for a low number of evaluated solutions (effort) or a low standard deviation (robustness). Instead, the goal should be to find a solution candidate with good quality, but if there are several candidates with similar quality, the one with a better robustness and lower effort should be ranked better.

6.4.2 Base-Level Algorithm

Table 6.7 describes the parameter configuration of the GA which should be optimized. The configuration is similar to the one used in S_1, only that the sets

Parameter Name	m_2	m_3
Algorithm	GA	OSGA
Maximum generations	100	100
Population size	30	30
Mutation probability	15%	15%
Elites	1	1
Selection operator	Proportional	Proportional
Mutation operator	OnePosition	OnePosition
Mutation operator (IntValue)	Normal	Normal
Mutation operator (DoubleValue)	Normal	Normal
Crossover operator (IntValue)	Normal	Normal
Crossover operator (DoubleValue)	Normal	Normal
Comparison factor LB		1.0
Comparison factor UB		1.0
Success ratio		1.0
Evaluation repetitions	6	6
Quality weight	1.0	1.0
Robustness weight	0.01	0.01
Effort weight	0.0005	0.0005

Table 6.6: Parameters of the meta-level algorithms for S_4

of operators for crossover and mutation are different. The reason for this is that the TSP has a different solution encoding than test-functions and mutation as well as crossover operators are encoding-specific. The population size and the maximum generations are fixed to avoid individuals with very different runtime. The maximum number of generations is 10'000, which results in an approximate runtime of 1–3 minutes for f_5 and 1–4 minutes for f_6.

6.4.3 Base-Level Problems

The TSPs which were used as base-level problems in this scenario are benchmark problems taken from the TSPLIB[1]:

- f_5: $ch130$ (cities: 130, best quality: 6110)
- f_6: $kroa200$ (cities: 200, best quality: 29368)

6.4.4 Results and Discussion

The parameters of the base-level algorithm are optimized for each base-level problem with a GA (m_2) and an OSGA (m_3) resulting in four meta-optimization runs. The results of these runs are presented in Table 6.8. While the GA runs calculated 100 generations, the OSGAs both stopped after 5 generations because the selection pressure reached the limit of 30. The OSGA runs required approximately the same number of evaluated solutions as the GA runs.

[1] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95

Parameter Name	Values
Algorithm	GA
Maximum generations	10'000
Population size	100
Mutation probability	0%–100%:1%
Elites	0–100:1
Selection operator	LinearRank, Proportional, Random, Tournament (Group size: 2–100:1), GenderSpecific (Female selector: Proportional, Male selector: Random), GeneralizedRank (Pressure: 2), NoSameMates (Selector: Tournament (Group size: 2), Difference: 5%), BestSelector, WorstSelector
Mutation operator	Insertion, Inversion, Scramble, Swap2, Swap3, TranslocationInversion, Translocation, null (no mutation)
Crossover operator	Cosa, CyclicCrossover2 (CX2), EdgeRecombination (ERX), MaximalPreservation (MPX), OrderBased (OX), Order2 (OX2), PartiallyMatched (PMX), PositionBased (PBX), UniformLike (ULX)

Table 6.7: Parameter configuration (c_3) of the base-level algorithm for S_4

	$p(c_4, f_5, m_2)$	$p(c_4, f_5, m_3)$	$p(c_4, f_6, m_2)$	$p(c_4, f_6, m_3)$
Elites	8	0	15	18
Crossover operator	ERX	ERX	OX2	OX2
Mutation operator	Translocation-Inversion	Inversion	Inversion	Inversion
Mutation probability	88%	92%	52%	33%
Selection operator	Tournament	Tournament	NoSameMates	GeneralizedRank
Tournament group size	11	14		
Generations	100	5	100	5
Termination	Generations	Sel. pressure	Generations	Sel. pressure
Evaluated solutions	3000	3330	3000	2460
CPU time (days)	17.3	32.1	25.7	23.7
Average quality	f_5: 6360.16	f_5: 6303.33	f_6: 31613.5	f_6: 31648.5
Diff. to best known	f_5: 250,16	f_5: 193,33	f_6: 2245,5	f_6: 2280,5
Standard deviation	f_5: 72.62	f_5: 42.76	f_6: 389.08	f_6: 627.49
Evaluated solutions	f_5: 920100	f_5: 1000100	f_6: 850100	f_6: 820100

Table 6.8: Solutions of the meta-optimization runs for S_4

The results show that the *ERX* seems to be a better crossover operator for f_5 whereas the *OX2* works better for f_6. The *Inversion* operator which inverts a random part of a permutation seems to be a good fit for all problem instances. The *TranslocationInversion* operator which was selected in $p(c_3, f_5, m_2)$ is similar to the *Inversion* operator only that it also shifts the inverted part of the permutation to another position. Due to the extremely high mutation probabilities chosen in $p(c_3, f_5, m_2)$ and $p(c_3, f_5, m_3)$ the mutation operator has a large impact on the algorithm. Combined with the large number of elites and a large

group size of the tournament selection, the probability of being selected is extremely high for the best individuals and almost zero for the worst. In fact, the algorithm almost turns into a local search algorithm (see Section 2.1.1).

Obviously the parameters for f_6 seem to require a lower mutation probability. The crossover operator is more relevant in the search process, so it is interesting to see that the *OX2* is favored. Instead of the tournament selection operator of $p(c_3, f_5, m_2)$ and $p(c_3, f_5, m_3)$, the *NoSameMates* and the *GeneralizedRank* selection operators are chosen. Interestingly, the selection operators are quite different in these two solutions, yet the resulting average quality seems to be very similar. It is remarkable that the standard deviation of $p(c_3, f_6, m_3)$ is much higher than the one of $p(c_3, f_6, m_2)$ which may be related to the different selection operator. Compared to the results for f_5 the number of elites is much higher in the results for f_6. This may also be related to the fact that $p(c_3, f_5, m_2)$ and $p(c_3, f_5, m_3)$ already employ a tournament selector with a large group size which also increases the probability of the top individuals to be selected.

$p(c_3, f_5, m_3)$ is kind of an outlier regarding the *Elites* parameter value of zero. To analyze if this result is feasible, a two-dimensional exploration of the parameters *Elites* and *GroupSize* was performed. The base-level algorithm was parameterized with the results from $p(c_3, f_5, m_3)$. The parameters *Elites* and *GroupSize* were varied in the ranges 0–30 and 2–30 resulting in 899 different combinations. Each combination was executed 10 times using HeuristicLab Hive. The results of this grid search are shown in Figure 6.8. The total runtime of that experiment was 13.57 days. The plot shows extremely bad quality values for very low values of *Elites* and *GroupSize*. The best quality values can indeed be achieved with zero elites and a tournament group size of 14.

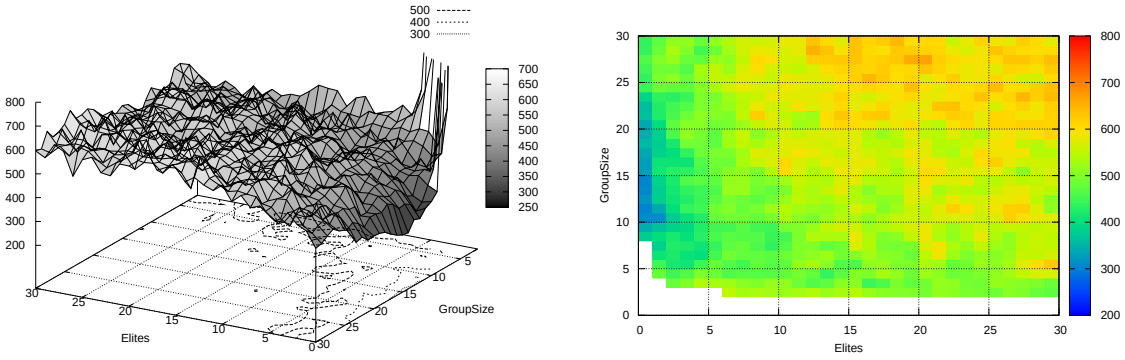

Figure 6.8: Exploration of the parameters *Elites* and *GroupSize* of $p(c_3, f_5, m_3)$. The *Elites* parameter is varied in the range of 0–30, the *GroupSize* parameter in the range of 2–30. The z-axis shows the average difference to the best known quality of 10 repetitions for each combination.

The results in this scenario show that high mutation probabilities along with a high number of elites and tournament group size can achieve very good results. However, it was not possible to tune the parameters in such a way that they find the global optimum of both TSPs and also the results should be taken with a grain of salt, since the standard deviations shown in Table 6.8 are very high. This means that the meta-level fitness landscape is very noisy and there may be many parameter values which yield similar qualities.

6.5 Scenario 5: Symbolic Expression Grammar for Regression

One way to affect the performance of symbolic regression is to change which symbols are allowed for building expression trees. As described in Section 5.1.4, these symbols are defined in the symbolic expression grammar. In this scenario (S_5) some properties of the grammar are optimized for a symbolic regression problem.

6.5.1 Meta-Level Algorithm

As meta-level optimizers, GAs with different parameterizations are used. Table 6.9 shows the different configurations. In m_4 a *Normal* crossover is used, whereas the *DiscreteCrossover* is used for the others. Additionally the amount of mutation is increased for m_5 by using the *AllPositions* mutator for the configurations that use the *DiscreteCrossover*. In m_6 and m_7 a *Tournament* selector with a group size of three and five is used.

6.5.2 Base-Level Algorithm

A GA is used as the base-level algorithm. Table 6.10 shows the parameters of the algorithm as well as the settings for the symbolic expression grammar. The symbols that are enabled have an initial frequency of one by default. Beside the *Variable* and *Constant* symbol there are arithmetic symbols enabled. The initial frequencies of the *Logarithm*, *Root* and *Power* symbols are being optimized in the range of zero to five. The goal is to find out if these operators, which can describe non-linear correlations, should be weighted more or less than the other symbols to find good solutions. As for the *Constant* and *Variable* symbol the goal is to find out how strong the manipulation operations should be on them when they are mutated. Therefore, their sigma properties (see Section 5.1.4) are optimized. The approximate runtime of the base-level algorithm is 2–3 minutes.

6.5.3 Base-Level Problem

The base-level problem (f_7) is a real world benchmark data set which comes from an industrial problem on modeling gas chromatography measurements of

Parameter Name	m_4	m_5	m_6	m_7
Algorithm	GA	GA	GA	GA
Maximum generations	100	100	100	100
Population size	30	30	30	30
Mutation probability	15%	15%	15%	15%
Elites	1	1	1	0
Selection operator	Proportional	Proportional	Tournament	Tournament
Tournament group size			3	5
Mutation operator	OnePosition	AllPosition	AllPosition	AllPosition
Mutation operator (IntValue)	Normal	Normal	Normal	Normal
Mutation operator (DoubleValue)	Normal	Normal	Normal	Normal
Crossover operator (IntValue)	Normal	Discrete	Discrete	Discrete
Crossover operator (DoubleValue)	Normal	Discrete	Discrete	Discrete
Evaluation repetitions	6	6	6	10
Quality weight	1.0	1.0	1.0	1.0
Robustness weight	0.01	0.01	0.01	0.01
Effort weight	0.0005	0.0005	0.0005	0.0005

Table 6.9: Parameters of the meta-level algorithms for S_5

Parameter Name	Values
Algorithm	GA
Maximum generations	100
Population size	500
Mutation probability	15%
Elites	1
Selection operator	Tournament
Mutation operator	ReplaceBranchManipulation, ChangeNodeTypeManipulation, OnePointShaker
Crossover operator	SubtreeCrossover
Evaluation operator	R-Square
Training partition	0–3999
Test partition	4000–4999
Enabled Symbols	Constant, Variable, Addition, Subtraction, Multiplication, Division, Average, Logarithm, Root, Power
Init. freq. (Logarithm)	0–5:0.01
Init. freq. (Root)	0–5:0.01
Init. freq. (Power)	0–5:0.01
Additive sigma (Constant)	0.01–10:0.0
Multiplicative sigma (Constant)	0.01–10:0.0
Weight sigma (Variable)	0.01–10:0.0
Additive weight sigma (Variable)	0.01–10:0.0
Multiplicative weight sigma (Variable)	0.01–10:0.0

Table 6.10: Parameter configuration (c_4) of the base-level algorithm for S_5

the composition of a distillation tower. It consists of 13 attributes and 5000 instances. The goal is to use regression to find an expression which describes a target variable as accurately as possible. All variables are normalized to a scale of 0.0 – 1.0, so that the resulting sigma-values can be related to the actual variable ranges. As the R^2 evaluator is used the theoretical best quality value is 1.0.

6.5.4 Results and Discussion

Table 6.11 shows the optimal parameters of the symbolic expression grammar, identified by four independent meta-optimization runs performed for this scenario. The table also shows the average quality (R^2 on the test-partition) of the best solution found. To validate the best solution found by the meta-optimization the settings were repeated in an independent experiment with 30 repetitions each. The results are also shown in the table. To be able to compare the success of this scenario a comparison with the default settings for symbolic expression grammars is also performed ($p(default)$). The qualities of the best solutions do yield above 0.9 which is an excellent value.

The initial frequencies of the *Logarithm*, *Root* and *Power* symbol range between 0.1 and 4.6 which seems to be very random considering that the allowed optimization range is between 0.0 and 5.0. Although most values lie above 1.0, which is the default, it seems that the base-level algorithm is not very sensitive to the initial frequencies of non-linear symbols. The additive and multiplicative sigma values for the *Constant* symbol are also very randomly distributed and significantly higher than the default values (1.0 and 0.03). Especially the multiplicative sigma is many factors higher than the default value. In addition, the sigmas for the manipulation of variable weights are significantly higher than what is suggested as default value. All optimized grammars yield higher qualities than the default grammar settings. One particular model has been chosen from one of the test runs of $p(c_4, f_7, m_6)$ to show how the model can predict the expected output. Figure 6.9 and 6.10 show a scatter plot and a line chart with the expected and the predicted value for training and test data.

Generally what the results reveal is that the GP algorithm in place is quite robust regarding the sigmas and initial frequencies in the grammar. However, some improvement is possible if the sigma values are increased significantly compared to the default values in HL.

6.6 Scenario 6: Symbolic Expression Grammar for Classification

This scenario (S_6) is aimed at analyzing the impact of changes to the symbolic expression grammar for classification problems. In particular, the initial frequencies of conditional symbols and constants are optimized for four different

	$p(default)$	$p(c_4, f_7, m_4)$	$p(c_4, f_7, m_5)$	$p(c_4, f_7, m_6)$	$p(c_4, f_7, m_7)$
Init. freq. (Logarithm)	1.00	4.53	3.31	4.07	4.53
Init. freq. (Root)	1.00	4.61	3.03	3.27	0.11
Init. freq. (Power)	1.00	2.57	2.29	1.59	1.14
Additive sigma (Constant)	1.00	3.79	1.22	8.43	5.34
Multiplicative sigma (Constant)	0.03	4.74	7.54	0.46	4.24
Weight sigma (Variable)	1.00	7.28	7.38	4.36	3.97
Additive weight sigma (Variable)	0.05	6.64	1.68	3.06	5.03
Multiplicative weight sigma (Variable)	0.03	2.14	3.52	1.65	5.86
Generations		100	100	100	100
Evaluated solutions		2930	2930	2930	2930
CPU time (days)		29.5	41.1	41.3	70.6
Average quality		0.8992	0.9041	0.9033	0.8915
Standard deviation		0.0074	0.0087	0.0047	0.0109
Average quality (30 repetitions)	0.8486	0.8729	0.8736	0.8642	0.8741
Standard deviation (30 repetitions)	0.0330	0.0250	0.0282	0.0391	0.0353

Table 6.11: Best solutions of the meta-optimization runs for S_5

Figure 6.9: Shows how the estimated values correlate with the target values.

Figure 6.10: Shows a small section of the line chart with the target values and the estimated values for the training and test partition.

classification problem instances. A GA is used as meta-level algorithm and an OSGA is used as base-level algorithm. The goal of this scenario is to find the optimal proportions of constant and conditional symbols in symbolic expression trees. If the initial frequency of symbols that are needed very often is too low, the OSGA needs more iterations to find good solutions. If it is too high, it might produce large and inefficient expression trees too often. Furthermore, the optimal initial frequencies for boolean symbols, which are used in conditional

Parameter Name	m_8
Algorithm	GA
Maximum generations	100
Population size	30
Mutation probability	30%
Elites	1
Selection operator	Proportional
Mutation operator	OnePosition
Mutation operator (IntValue)	Normal
Mutation operator (DoubleValue)	Normal
Crossover operator (IntValue)	Normal
Crossover operator (DoubleValue)	Normal
Evaluation repetitions	6
Quality weight	1.0
Robustness weight	0.01
Effort weight	0.0001

Table 6.12: Parameters of the meta-level algorithm for S_6

symbols, should be optimized.

6.6.1 Meta-Level Algorithm

The parameters of the GA that is used on the meta-level are shown in Table 6.12.

6.6.2 Base-Level Algorithm

The parameters of the OSGA that is used as base-level algorithm are shown in Table 6.13. The table also shows the optimization configuration for the symbolic expression grammar. The initial frequency of the *Constant* symbol is optimized in the range of 1–20, and the conditional symbols are optimized in the range of 0–20. The initial frequencies of all other symbols are fixed at 1.0. The training and test partitions are different for each base-level problem as each problem has a different number of rows. The approximate runtime of the base-level algorithm varies is between 15 and 60 minutes, depending on the problem.

6.6.3 Base-Level Problems

In this scenario, four classification problem instances are used as base-level problems. All data sets have been scaled to values from -1.0 to 1.0. The following list gives some background information about each problem instance:

- f_8: **Melanoma:** This data set is provided by the Department of Dermatology of the Medical University Vienna and contains medical measurements from potential skin cancer patients. It contains 1311 instances with 30 attributes, from which 90.5% are diagnosed with skin cancer and 9.5% are healthy.

Parameter Name	Values
Algorithm	OSGA
Maximum generations	100
Population size	500
Mutation probability	15%
Elites	1
Selection operator	GenderSpecific (Random, Proportional)
Mutation operator	ReplaceBranchManipulation, ChangeNodeTypeManipulation, OnePointShaker, FullTreeShaker
Crossover operator	SubtreeCrossover
Evaluation operator	Mean Squared Error
Training partition	f_8: 0–999, f_9: 0–999, f_{10}: 0–699, f_{11}: 0–399
Test partition	f_8: 1000–1311, f_9: 1000–1517, f_{10}: 700–999, f_{11}: 400–569
Enabled Symbols	Constant, Variable, Addition, Subtraction, Multiplication, Division, IfThenElse, GreaterThan, LessThan, And, Or, Not
Init. freq. (Constant)	1–20:0.01
Init. freq. (IfThenElse)	0–20:0.01
Init. freq. (GreaterThan)	0–20:0.01
Init. freq. (LessThan)	0–20:0.01
Init. freq. (And)	0–20:0.01
Init. freq. (Or)	0–20:0.01
Init. freq. (Not)	0–20:0.01

Table 6.13: Parameter configuration (c_5) of the base-level algorithm for S_6

- f_9: **Prostate:** This data set is provided by the Central Blood Laboratory of the General Hospital Linz, Austria and was measured in the years 2005–2008. It contains 1517 instances with 27 routinely measured blood values of patients diagnosed with prostate cancer (75%) as well as from healthy persons (25%).
- f_{10}: **Respiratory:** This data set is also provided by the Central Blood Laboratory of the General Hospital Linz, Austria and was measured in the years 2001–2008. It contains 999 instances with 27 routinely measured blood values of patients diagnosed with cancer in the respiratory system (75%) as well as from healthy persons (25%).
- f_{11}: **Wisconsin:** The Wisconsin breast cancer dataset is provided by the UCI machine learning repository[2] and contains and 569 instances with 32 attributes of breast cancer patients (37%) as well as from healthy persons (63%).

[2] http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

	$p(c_5, f_8)$	$p(c_5, f_9)$	$p(c_5, f_{10})$	$p(c_5, f_{11})$
Init. freq. (Constant)	1,75	16,66	1,00	2,59
Init. freq. (IfThenElse)	7,67	0,22	13,96	1,86
Init. freq. (GreaterThan)	3,09	18,59	12,90	13,92
Init. freq. (LessThan)	13,26	11,87	5,78	13,11
Init. freq. (And)	8,87	15,84	4,61	3,39
Init. freq. (Or)	17,05	17,04	13,90	16,13
Init. freq. (Not)	3,97	14,68	3,38	6,35
Generations	75	85	58	94
Evaluated solutions	2205	2495	1712	2756
CPU time (days)	293.4	456.4	261.4	278.6
Average quality (MSE on test)	0.1052	0.6141	0.3121	0.1244
Quality standard deviation	0.0166	0.0127	0.0227	0.0378
Average evaluated solutions	981'916	1'113'966	915'816	1'020'616
Average MSE on test (30 repetitions)	0.1378	0.6406	0.3312	0.2057
Standard deviation of MSE on test (30 repetitions)	0.0212	0.1823	0.0401	0.0558
Average accuracy on test (30 repetitions)	96.3%	78.9%	90.2%	93.6%

Table 6.14: Shows the best solutions of the meta-optimization runs for S_6 as well as the results of independent experiments with 30 repetitions.

6.6.4 Results and Discussion

For each of the base-level problems one meta-optimization run was executed with the configuration c_5. Table 6.14 shows the best solutions found in each run. Due to extremely high runtime demands, the runs had to be aborted prematurely so none of them reached the 100^{th} generation. The results were validated by running an independent experiment with 30 repetitions for each parameterization. Just as in S_5, it can be observed that the results from these experiments are slightly worse than the average qualities from the meta-optimization runs. The problem could be that because elitism is used and the evaluation of a solution candidate is stochastic, one evaluation might yield a higher quality and the solution candidate becomes the elite individual. It therefore does not necessarily represent the quality to be expected with these settings, but the best quality that was achieved with this parameterization during the whole meta-optimization run.

The results show quite different and seemingly random initial frequencies for each problem instance. To validate these results, they were cross-tested with all problem instances. Figure 6.11 shows the results of the cross-tests. The figure clearly shows that the optimized settings are always equal or better than all other settings (including the default settings) on each problem. A little bit of an outlier is $p(c_5, f_9)$ which has an extremely high initial frequency for the *Constant* symbol and an extremely low initial frequency for the *IfThenElse* symbol. Interestingly the cross-tests show that these parameter values perform indeed very well on f_9, while they perform bad on all other problems.

OSGAs are very robust algorithms and the optimization of initial frequen-

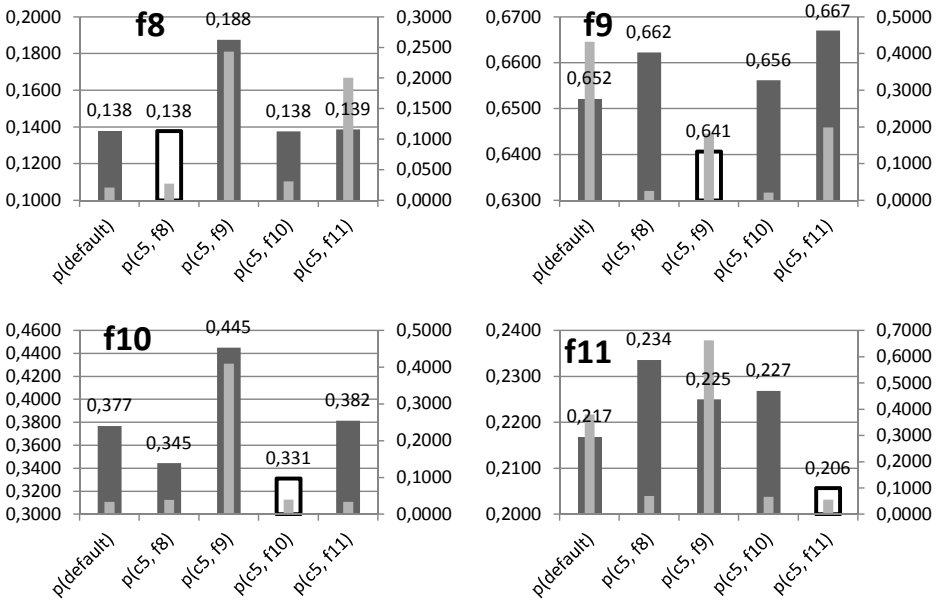

Figure 6.11: The thick bars show the average MSE on the test partition on the problem instances f_8, f_9, f_{10} and f_{11}. The thin bar shows the standard deviations on the secondary scale. Each problem instance was optimized with the default settings as well as with the settings that resulted from the meta-optimization runs of S_6. Each combination was repeated 30 times.

cies for constants and conditional symbols yielded very humble improvements. Nevertheless the meta-optimization approach was able to find parameter values which are slightly better than the default values and it has been shown the different results for each problem instance are legitimate.

Chapter 7

Conclusion and Outlook

The main goal of this thesis was to automate the search for optimal parameters for metaheuristic optimization algorithms by using a meta-algorithm. In order to achieve this goal a parameter optimization problem was implemented for the optimization environment HeuristicLab.

As described in Chapter 2 there exist several approaches to parameter meta-optimization (PMO). The solution presented in this thesis combined the positive aspects of these approaches. A novel concept introduced in this thesis is the generic design that makes meta- and base-level algorithms arbitrarily exchangeable as long as the necessary operators are implemented. The flexible and open architecture of HL made this generic approach possible. Besides the obvious advantage of being able to optimize any base-level algorithm, it is also beneficial to be able to choose the meta-level optimizer from existing algorithms that are well known. Inspired by HL's flexible plugin infrastructure, the PMO implementation was designed to be easily extendable by new operators and parameter data types, without changing any existing code. It has been shown in the example of symbolic expression trees that such extensions are possible without much effort. Driven by HL's focus on usability, a simple yet flexible user interface has been created for PMO. A user can select the algorithm and the problem that should be optimized. Further, it is possible to define which parameters should remain fixed and which should be optimized. Search ranges for individual parameters can be defined to shrink the search space. A user can select multiple base-level problems for which the optimal behavioral algorithm parameters should be found. To avoid over- or underweighting problems, normalization is applied to the results of each base-level problem. The presented PMO implementation supports multi-objective parameter optimization by using a normalized weighted sum of the three objectives *average quality*, *robustness* and *effort*.

A novel feature for meta-optimization that has been introduced in this thesis is the optimization of the properties of symbolic expression grammars for symbolic regression and classification. Symbolic expression grammars are repre-

sented as custom data types in HL, yet the effort for extending PMO to support them was minimal which has clearly shown the flexible design of the PMO approach.

To show the validity of the PMO implementation presented in this thesis a number of optimization scenarios, which resemble the complexities of real-world problems, was performed. Since this is a very runtime intensive task, these experiments were executed in the distributed computation environment HeuristicLab Hive which is deployed at the University of Applied Sciences Upper Austria. The experiments in Scenario 1 have shown that the optimal parameters can differ significantly when the same test-function problem with different problem dimensions is used. In Scenario 2, very interesting parameters with extremely high mutation probabilities and a high number of elites were identified as best parameter settings. It indicates that the best parameters can be far off the default and commonly used settings. In Scenario 3, the effect of using different numbers of iterations on the parameters was analyzed. It was shown that in the case of real-value test-functions with 10'000 iterations, it is beneficial to use operators which perform small manipulations on the solution candidates. In Scenario 4 and 5, the symbolic expression grammars for regression and classification problems were optimized. It has been shown that the default values for the manipulation sigmas of variable weights and constants are set too low in HeuristicLab. Of course, this fact can differ depending on the actual problem instance. In the case of classification, the optimization of initial frequencies of constants and conditional symbols yielded very humble results, especially in respect to the huge amounts of runtime spent on these runs. In that case, the default settings of HeuristicLab are appropriate.

Concluding, the approach of using a meta-level algorithm to find the optimal parameters has proven to work very well on some problems. It is possible to find parameter value combinations that are very different from commonly used settings. This functionality comes at the cost of huge runtime demands. The total sum of CPU time used for the experiments in this thesis amounts to over 7.5 years.

The advancements in computing power in the recent years have made this thesis possible at that scale. Parallelization and distributed computing has been used to perform experiments. However, there is room for improvement in terms of runtime performance. Two ways to optimize runtime would be *racing* and *sharpening* [53]. When *racing* is used, promising solution candidates are evaluated more often than bad solution candidates. With *sharpening*, the number of repetitions is increased depending on the current generation. In this way, the solution evaluation is faster in the beginning of the optimization process and gets more and more precise towards the end.

An idea to make future development of PMO easier would be to provide benchmark problems for parameter settings. Such a benchmark problem could have a generated meta-fitness landscape. Evaluating a solution candidate would only require to lookup a value, so that the evaluation of solution candidates

would become extremely fast. Of course, the fitness evaluation should underlie a stochastic distribution, just as real evaluations of parameter settings. This would make it much easier to tune the parameters of a meta-level optimizer.

Bibliography

[1] Affenzeller, M. and S. Wagner: *Offspring selection: A new self-adaptive selection scheme for genetic algorithms.* In Ribeiro, B., R.F. Albrecht, A. Dobnikar, D.W. Pearson, and N.C. Steele (eds.): *Adaptive and Natural Computing Algorithms*, Springer Computer Series, pp. 218–221. Springer, 2005.

[2] Affenzeller, M., S. Winkler, and S. Wagner: *Evolutionary systems identification: New algorithmic concepts and applications.* In Kosinski, W. (ed.): *Advances in Evolutionary Algorithms*, ch. 2, pp. 29–48. IN-TECH, 2008.

[3] Affenzeller, M., S. Winkler, S. Wagner, and A. Beham: *Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications.* Numerical Insights. CRC Press, 2009.

[4] Bartz-Beielstein, T., C. Lasarczyk, and M. Preuss: *Sequential Parameter Optimization.* IEEE, 2005.

[5] Bäck, T.: *Optimal mutation rates in genetic search.* In *Proceedings of the fifth International Conference on Genetic Algorithms*, pp. 2–8. Morgan Kaufmann, 1993.

[6] Bäck, T.: *Parallel optimization of evolutionary algorithms.* Lecture Notes In Computer Science, 866:418–427, 1994.

[7] Bäck, T. and H.P. Schwefel: *An overview of evolutionary algorithms for parameter optimization.* Evolutionary Computation, 1(1):1–23, 1993.

[8] Bellman, R. and R. Corporation: *Dynamic programming.* Rand Corporation research study. Princeton University Press, 1957.

[9] Beyer, H.G. and H.P. Schwefel: *Evolution strategies - A comprehensive introduction.* Natural Computing, 1(1):3–52, 2002.

[10] Blum, C. and A. Roli: *Metaheuristics in combinatorial optimization: Overview and conceptual comparison.* ACM Compututing Surveys, 35:268–308, 2003.

[11] Cormen, T.H., C.E. Leiserson, R.L. Rivest, and C. Stein: *Introduction to Algorithms*. MIT Press, 2nd ed., 2001.

[12] Dantzig, G.B.: *Linear Programming and Extensions*. Princeton University Press, 1963.

[13] Darwin, C.: *The Origin of Species*. Wordsworth Classics of World Literature. Wordsworth Editions, 1998.

[14] De Jong, K.A.: *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, 1975.

[15] Deb, K. and R.B. Agrawal: *Simulated binary crossover for continuous search space*. Complex Systems, 9:115–148, 1995.

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

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

[18] Dumitrescu, D., B. Lazzerini, L.C. Jain, and A. Dumitrescu: *Evolutionary Computation*. CRC Press, 2000.

[19] Eberhart, R. and J. Kennedy: *A new optimizer using particle swarm theory*. In *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pp. 39–43, 1995.

[20] Eiben, A.E., Z. Michalewicz, M. Schoenauer, and J.E. Smith: *Parameter control in evolutionary algorithms*. IEEE Transactions on Evolutionary Computation, 3:124–141, 1999.

[21] Eiben, A.E. and J.E. Smith: *Introduction to Evolutionary Computation*. Natural Computing Series. Springer, 2003.

[22] Fogel, D.: *An evolutionary approach to the traveling salesman problem*. Biological Cybernetics, 60:139–144, 1988.

[23] Fogel, D., L. Fogel, and J. Atmar: *Meta-evolutionary programming*. In *Signals, Systems and Computers*, pp. 540–545. IEEE Computer Society Press, 1991.

[24] Fogel, D.B.: *Applying evolutionary programming to selected traveling salesman problems*. Cybernetics and Systems, 24:27–36, 1993.

[25] Glover, F. and G.A. Kochenberger: *Handbook of Metaheuristics*, vol. 57 of *International Series in Operations Research & Management Science*. Kluwer, 2003.

[26] Grefenstette, J.: *Optimization of control parameters for genetic algorithms.* IEEE Transactions on Systems, Man, and Cybernetics, 16(1):122–128, 1986.

[27] Griewank, A.O.: *Generalized descent for global optimization.* Journal of Optimization Theory and Applications, 34:11–39, 1981.

[28] Gustafson, S., E. Burke, and N. Krasnogor: *On improving genetic programming for symbolic regression.* In *IEEE Congress on Evolutionary Computation*, pp. 912–919, 2005.

[29] Holland, J.H.: *Adaption in Natural and Artificial Systems.* University of Michigan Press, 1975.

[30] Horn, J., N. Nafpliotis, and D. Goldberg: *A niched pareto genetic algorithm for multiobjective optimization.* In *IEEE World Congress on Computational Intelligence*, pp. 82–87, 1994.

[31] Iunescu, M.: *Parameter Optimization of Genetic Algorithms by Means of Evolution Strategies in a Grid Environment.* PhD thesis, Johannes Kepler Universität Linz, 2006.

[32] Kennedy, J. and R. Eberhart: *Particle swarm optimization.* In *IEEE International Conference on Neural Networks*, pp. 1942–1948, 1995.

[33] Kirkpatrick, S., C.D. Gelatt, and M.P. Vecchi: *Optimization by simulated annealing.* Science, 220:671–680, 1983.

[34] Koza, J.R.: *On the programming of computers by means of natural selection.* MIT Press, 1992.

[35] Land, A.H. and A.G. Doig: *An automatic method of solving discrete programming problems.* Econometrica, 28:497–520, 1960.

[36] Landgraaf, W. de, A. Eiben, and V. Nannen: *Parameter calibration using meta-algorithms.* IEEE, 2007.

[37] Larranaga, P., C.M.H. Kuijpers, R.H. Murga, I. Inza, and D. Dizdarevic: *Genetic algorithms for the travelling salesman problem: A review of representations and operators.* Artificial Intelligence Review, 13:129–170, 1999.

[38] Meissner, M., M. Schmuker, and G. Schneider: *Optimized particle swarm optimization (opso) and its application to artificial neural network training.* BMC Bioinformatics, 7(1):125, 2006.

[39] Mercer, R. and J. Sampson: *Adaptive search using a reproductive metaplan.* Kybernetes, 7(3):215–228, 1978.

[40] Mühlenbein, H.: *Evolution in time and space - the parallel genetic algorithm.* In *Foundations of Genetic Algorithms*, pp. 316–337. Morgan Kaufmann, 1991.

[41] Mühlenbein, H. and D. Schlierkamp-Voosen: *Predictive models for the breeder genetic algorithm i. continuous parameter optimization.* Evolutionary Computation, 1:25–49, 1993.

[42] Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, 3rd ed., 1999.

[43] Michalewicz, Z. and B. Fogel: *How to Solve It: Modern Heuristics.* Springer, 2000.

[44] Michalewicz, Z. and M. Schoenauer: *Evolutionary algorithms for constrained parameter optimization problems.* Evolutionary Computation, 4:1–32, 1996.

[45] Nannen, V. and A. Eiben: *A method for parameter calibration and relevance estimation in evolutionary algorithms.* Genetic And Evolutionary Computation Conference, pp. 183–190, 2006.

[46] Ochoa, G., I. Harvey, and H. Buxton: *Optimal mutation rates and selection pressure in genetic algorithms.* In *Genetic And Evolutionary Computation Conference*, 2000.

[47] Papadimitriou, C.H.: *Computational Complexity.* Addison-Wesley, 1994.

[48] Pedersen, E.M.H.: *Tuning & Simplifying Heuristical Optimization.* PhD thesis, University of Southampton, 2010.

[49] Pedersen, M. and A. Chipperfield: *Local unimodal sampling.* Techn. rep., Hvass Laboratories, 2008.

[50] Pomberger, G. and H. Dobler: *Algorithmen und Datenstrukturen: Eine systematische Einführung in die Programmierung.* Pearson Studium, 2008.

[51] Rechenberg, I.: *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog, 1973.

[52] Schwefel, H.P.P.: *Evolution and Optimum Seeking: The Sixth Generation.* John Wiley & Sons, Inc., 1993.

[53] Smit, S.K. and A.E. Eiben: *Comparing parameter tuning methods for evolutionary algorithms.* In *IEEE Congress on Evolutionary Computation*, pp. 399–406, 2009.

[54] Storn, R. and K. Price: *Differential evolution – A simple and efficient heuristic for global optimization over continuous spaces.* Journal of Global Optimization, 11:341–359, 1997.

[55] Syswerda, G.: *Schedule optimization using genetic algorithms.* Handbook of Genetic Algorithms, pp. 332–349, 1991.

[56] Takahashi, M. and H. Kita: *A crossover operator using independent component analysis for real-coded genetic algorithms.* In *Proceedings of the 2001 Congress on Evolutionary Computation*, pp. 643–649, 2001.

[57] Tate, D.M. and A.E. Smith: *A genetic approach to the quadratic assignment problem.* Computers & Operations Research, 22:73–83, 1995.

[58] Ulder, N.L.J., E.H.L. Aarts, H.J. Bandelt, P.J.M. van Laarhoven, and E. Pesch: *Genetic local search algorithms for the travelling salesman problem.* In *Parallel Problem Solving from Nature*, pp. 109–116. Springer, 1991.

[59] Wagner, S.: *Heuristic optimization software systems-Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment.* PhD thesis, Johannes Kepler University, Linz, Austria, 2009.

[60] Wagner, S. and M. Affenzeller: *SexualGA: Gender-specific selection for genetic algorithms.* In *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, vol. 4, pp. 76–81. International Institute of Informatics and Systemics, 2005.

[61] Walsh, G.R.: *Methods of Optimization.* John Wiley and Sons, 1975.

[62] Wendt, O.: *COSA: Cooperative Simulated Annealing - Integration von Genetischen Algorithmen und Simulated Annealing am Beispiel der Tourenplanung.* PhD thesis, IWI Frankfurt, 1994.

[63] Whitley, D.: *A free lunch proof for gray versus binary encodings.* In *Proceedings of the Genetic and Evolutionary Computation Conference*, vol. 1, pp. 726–733. Morgan Kaufmann, 1999.

[64] Whitley, D., T. Starkweather, and D. Shaner: *The traveling salesman and sequence scheduling: Quality solutions using genetic edge recombination.* In *Handbook of Genetic Algorithms*, pp. 350–372, 1990.

[65] Wolpert, D.H. and W.G. Macready: *No free lunch theorems for optimization.* IEEE Transactions on Evolutionary Computation, 1(1):67–82, 1997.

[66] Wright, A.H.: *Genetic algorithms for real parameter optimization.* In *Foundations of Genetic Algorithms*, pp. 205–218. Morgan Kaufmann, 1991.

List of Figures

List of Tables