Heuristic Optimization Software Systems
Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment

Dissertation zur Erlangung des akademischen Grades
Doktor der technischen Wissenschaften

Angefertigt am Institut für formale Modelle und Verifikation
Eingereicht von:
Dipl.-Ing. Stefan Wagner

Betreuung:
Priv.-Doz. Dipl.-Ing. Dr. Michael Affenzeller

Beurteilung:
Erstbeurteiler: Priv.-Doz. Dipl.-Ing. Dr. Michael Affenzeller
Zweitbeurteiler: Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl

Linz, März 2009
For Heike, my love
Abstract

Many optimization problems cannot be solved by classical mathematical optimization techniques due to their complexity and the size of the solution space. In order to achieve solutions of high quality though, heuristic optimization algorithms are frequently used. These algorithms do not claim to find global optimal solutions but offer a reasonable tradeoff between runtime and solution quality. Therefore, they are especially suitable for practical applications. In the last decades the success of heuristic optimization techniques in various problem domains has led to the development of a broad spectrum of optimization paradigms which often use natural processes as a source of inspiration (as for example evolutionary algorithms, simulated annealing, or ant colony optimization).

Mature and flexible software systems are required for developing and applying heuristic optimization techniques in science and industry. On the one hand, these systems have to support scientists in the development of new algorithms; on the other hand, they should enable users to apply different optimization methods on specific problems easily and should allow a homogeneous integration into existing software environments. The heterogeneous requirements of these two user groups as well as the diversity of different algorithms lead to many challenges for software engineers. Existing systems often suffer from severe drawbacks concerning extensibility, flexibility, modularity, and usability: Algorithm models adapted to specific optimization paradigms prevent the incorporation of new optimization techniques, insufficient modularization complicates the integration into existing systems, and the lack of graphical user interfaces reduces the usability and increases the learning effort.

The main goal of this thesis is to overcome these difficulties. Based on a comprehensive requirements analysis and on a comparison of existing solutions, a new design for heuristic optimization software systems is developed. Thereby, main aspects are a plugin-based architecture, a domain-independent model to represent arbitrary algorithms, support of graphical user interfaces, as well as the integration of parallel algorithms. These aspects serve as a basis for the development of new heuristic optimization software systems. Finally, the thesis is concluded by showing how several heuristic optimization algorithms can be realized in a flexible and reusable way.
Kurzfassung


Acknowledgments

First and foremost I want to thank Priv.-Doz. Dipl.-Ing. Dr. Michael Affenzeller, the head of our research group, who introduced me to metaheuristic optimization strategies. He has not only been a supporting mentor and an encouraging supervisor, but has also become one of my dearest friends. Without his continuous support and advice, this thesis and my scientific work in general would not have been possible.

Furthermore, I would like to express my gratitude to Univ.-Prof. Dipl.-Ing. Dr. Günther Raidl from the Institute of Computer Graphics and Algorithms of the Vienna University of Technology for being my second supervisor. He provided helpful advice and numerous comments which improved the quality of this thesis a lot.

I am also thankful to Univ.-Prof. Dipl.-Ing. Dr. Witold Jacak, the dean of the School of Informatics, Communications and Media in Hagenberg of the Upper Austria University of Applied Sciences, for giving me the opportunity to work at the Department of Software Engineering and for making it possible for me to concentrate on this thesis.

The work described in this thesis would not have been possible without being able to work in a team of encouraged and brilliant researchers. I am very grateful to all members of the Heuristic and Evolutionary Algorithms Laboratory (HEAL) who have always been available for fruitful discussions and provided valuable ideas regarding the design and development of HeuristicLab. In this context, special thanks go to Dipl.-Ing. Dr. Stephan Winkler also for proofreading this thesis and improving its linguistic quality.

Finally, I want to thank my family and all of my friends and colleagues who have encouraged me during my work on this thesis – and of course also my love Heike for standing by my side in easy as well as in hard times.
## Contents

1 Introduction .................................................. 1
   1.1 Motivation and Goal ................................. 1
   1.2 Publications ...................................... 2
   1.3 Structure and Content ............................. 3

2 Heuristic Optimization ..................................... 5
   2.1 Taxonomy of Heuristic Algorithms ................ 6
      2.1.1 Trajectory-Based Metaheuristics .......... 8
      2.1.2 Population-Based Metaheuristics ........ 13
      2.1.3 Hybrid Metaheuristics ..................... 20

3 Requirements Analysis .................................... 23
   3.1 User Groups ....................................... 23
      3.1.1 Practitioners ............................... 24
      3.1.2 Experts .................................. 25
      3.1.3 Students .................................. 26
   3.2 Main Requirements ................................. 27

4 Existing Software ........................................... 31
   4.1 Heuristic Optimization Frameworks ............... 32
      4.1.1 Templar .................................... 32
      4.1.2 HotFrame ................................... 34
      4.1.3 Evolving Objects (EO) ..................... 35
      4.1.4 Open BEAGLE ............................... 36
      4.1.5 ECJ ...................................... 37
      4.1.6 EasyLocal++ ............................... 38
   4.2 Analysis and Comparison ........................... 39

5 HeuristicLab Architecture ............................... 43
   5.1 HeuristicLab 1.1 .................................. 43
   5.2 HeuristicLab 2.0 .................................. 46
   5.3 HeuristicLab 3.0 .................................. 49
      5.3.1 Plugin Infrastructure ....................... 50
CONTENTS

5.3.2 Object Model ........................................... 54
5.3.3 Algorithm Model ........................................ 58
5.3.4 Parallelism ............................................ 64
5.3.5 Layers of User Interaction .............................. 66
5.4 Analysis and Comparison ................................... 68
5.5 Applications of HeuristicLab ............................. 70

6 Algorithm Modeling ............................................... 75
6.1 Basic Operators ........................................... 75
6.2 Control Operators ......................................... 78
6.3 Exemplary Problem Representation ..................... 81
   6.3.1 Solution Encoding .................................... 82
   6.3.2 Quality Evaluation .................................. 84
6.4 Selection and Reduction ................................... 85
   6.4.1 Selection Operators .................................. 86
   6.4.2 Reduction Operators ................................ 87
   6.4.3 Sorting Operators ................................... 88
6.5 Modularity .................................................. 88
6.6 Greedy Local Search ....................................... 90
6.7 Simulated Annealing ....................................... 95
6.8 Evolution Strategies ....................................... 98
6.9 Genetic Algorithms ........................................ 100
6.10 Hybrid Algorithms ......................................... 105
6.11 Parallel Algorithms ....................................... 107

7 Conclusion ..................................................... 113

Bibliography ..................................................... 117

List of Tables ................................................... 135

List of Figures .................................................. 136

List of Listings ................................................ 139
Chapter 1

Introduction

1.1 Motivation and Goal

In the last decades a steady increase of computational resources and concurrently an impressive drop of hardware prices could be observed. Nowadays, very powerful computer systems are found in almost every company or research institution, providing huge processing power on a broad basis which was unthinkable some years ago. This trend opens the door for attacking complex optimization problems of various domains that were not solvable in the past. Concerning problem solving methodologies, especially heuristic algorithms are very successful in that sense, as they provide a reasonable tradeoff between solution quality and required runtime.

In the research area of heuristic algorithms a broad spectrum of optimization techniques has been developed. In addition to problem-specific heuristics, particularly the development of metaheuristics is a very active field of research, as these algorithms represent generic methods that can be used for solving many different optimization problems. A variety of often nature inspired archetypes has been used as a basis for new optimization paradigms such as evolutionary algorithms, ant systems, particle swarm optimization, tabu search, or simulated annealing. Several publications show successful applications of such metaheuristics on benchmark and real-world optimization problems.

However, Jürg Nievergelt stated in his article in 1994 “No systems, no impact!” [130] and pointed out that well-engineered software systems are a fundamental basis to transfer research results from science to industry. Of course, this statement is also true in the area of heuristic optimization. In order to apply effective and enhanced metaheuristics on real-world optimization problems, mature software systems are required which meet demands of software quality criteria such as reliability, efficiency, usability, maintainability, modularity, portability, or security. Although these requirements are well-known and considered in enterprise software systems, they are not yet satisfactorily respected in the heuristic
optimization community. Most heuristic optimization frameworks are research prototypes and are funded by national or international research programs. In such scenarios it is very hard to establish a continuous development process, to take into account many different users, to provide comprehensive support for users, or to reach the maturity of a software product. Therefore, these systems cannot be easily integrated into an enterprise environment.

Another major difficulty regarding the design of general purpose heuristic optimization software systems is that there is no common model for metaheuristics. Due to the heterogeneous nature of heuristic optimization paradigms, it is hard to identify and generalize common concepts which imposes a challenging problem on software developers. Many existing software frameworks focus on one or a few particular optimization paradigms and miss the goal of providing an infrastructure which is generic enough to represent all different kinds of algorithms. The variety of existing frameworks makes it very difficult for researchers to develop and compare their algorithms to show advantageous properties of a new approach. A unified software platform for heuristic optimization would improve this situation, as it would enable algorithm developers to assemble algorithms from a set of ready-to-use components and to analyze and compare results in a common framework.

Therefore, the development of high quality and mature heuristic optimization software systems would lead to a win-win situation for industry and for science. In this thesis, the author aims at this goal by proposing a flexible architecture and a generic design for heuristic optimization applications. Instead of trying to incorporate different heuristic optimization algorithms into a common model, a generic algorithm (meta-)model is presented that is capable of representing not only heuristic optimization but arbitrary algorithms. By this means the model can be used for developing custom algorithm models for various optimization paradigms. Furthermore, state-of-the-art software engineering methodologies are used to satisfy additional requirements such as parallelism, user interaction on different layers of abstraction, flexible deployment, or integration into existing applications.

1.2 Publications

In the context of this thesis five peer-reviewed conference papers have been published that outline the author’s work on heuristic optimization software systems [172, 169, 176, 174, 175]. Furthermore, [169] has also been published in the Journal of Systems Science [170]. These publications address various aspects regarding the development of the HeuristicLab optimization environment and represent the basis of this thesis. Besides, the author has authored and co-authored many other publications in the last six years that present theoretical and practical work mainly in the area of evolutionary algorithms (2 books, 2
book chapters, 10 journal articles, more than 50 peer-reviewed conference papers). HeuristicLab has been used as development and test environment for most of these publications which highlights its suitability and usability. A complete list of the author’s publications is presented in the CV at the end of this thesis.

1.3 Structure and Content

This thesis is structured as follows: In Chapter 2 a brief introduction to heuristic optimization is given. Several trajectory-based and population-based metaheuristics are outlined that represent state-of-the-art algorithms of the domain. In Chapter 3 the main user groups of heuristic optimization software systems are identified and their requirements are analyzed. Chapter 4 presents a selection of some existing frameworks for heuristic optimization and compares them with respect to the requirements analysis. Based on the strengths and weaknesses of these systems, the architecture and design of a new heuristic optimization environment called HeuristicLab is discussed in Chapter 5. Thereby, a main focus is put on the presentation of a flexible and generic algorithm model. In order to show the flexibility and the benefits of the proposed algorithm model, Chapter 6 outlines how different metaheuristics can be modeled. Finally, Chapter 7 summarizes the main characteristics of HeuristicLab and concludes the thesis by suggesting further improvements and additional research topics.
Chapter 2

Heuristic Optimization

An optimization problem can be described mathematically in the following way: Given an objective function \( f : S \rightarrow Q \) from some solution space \( S \) to a linearly ordered set \( Q \), find a solution \( x_0 \in S \) so that the following equation holds \( \forall x \in S : f(x_0) \geq f(x) \). In this case the problem is called a maximization problem.

Analogically, in a minimization problem the goal is to find such an \( x_0 \) that the value of the objective function becomes minimal. In many cases, the solution \( x_0 \) has to satisfy several constraints given as a set of equalities or inequalities. Depending on the characteristics of the objective function and the constraints, several types of optimization problems such as linear optimization, non-linear optimization, discrete optimization, or continuous optimization can be distinguished [74, 146]. Furthermore, not only a single objective function but a set of possibly conflicting objective functions might be considered. In this case the problem belongs to the class of multi-objective optimization problems [159].

If we take a look at all these different types of optimization problems in terms of complexity theory, only a few of them fall into the category of \( P \) problems (i.e., they can be solved in polynomial time using a deterministic Turing machine). For example, the Simplex method developed by George Dantzig in 1947 [49] can be used to solve optimization problems in which both, the objective function and all constraints, are linear (linear optimization). However, if the objective function or the constraints become non-linear or if some or all decision variables of the objective function are restricted to discrete values, efficient mathematical optimization techniques can only be applied in a few special cases [177]. In the general case, such discrete or non-linear optimization problems are \( NP \) problems and can be solved in polynomial time only if using a non-deterministic Turing machine. As a consequence, no efficient (polynomial time) algorithm is known until today for solving these problems exactly [137].

Many practical and theoretical optimization problems in various domains belong to the class of discrete or non-linear optimization problems. Additionally, the objective function is often highly multimodal and the cardinality of the solution space grows exponentially with the problem size. For example, in the
CHAPTER 2. HEURISTIC OPTIMIZATION

field of combinatorial optimization and operations research such problems occur
frequently (e.g., job scheduling, route planning, bin packing). If a global opti-
mal solution for one of these problems has to be found, enumeration algorithms
might be used that explore the whole solution space and evaluate each possible
solution (explicit enumeration). Backtracking algorithms are typical representa-
tives of such explicit or complete enumeration methods [47, 140]. However, as
this approach is not feasible in terms of required runtime even for small problem
instances, additional strategies are often used to limit the number of solutions
that have to be evaluated (implicit enumeration). Typical representatives are
tree search algorithms such as branch and bound which use a bound function to
estimate the solution quality of a partial solution that is evaluated [138]. If the
estimated quality is worse than some lower bound (for example the best solution
found so far), further expansion of the solution can be stopped. In this way
the size of the solution space is reduced by excluding regions that only contain
suboptimal solutions. Furthermore, integer linear programming (ILP) represents
specialized and efficient algorithms for the class of linear but discrete optimization
problems. For example, cutting plane algorithms, LP-based branch and bound,
branch and cut, branch and price, Lagrangian relaxation, and column generation
can be used to solve large instances of ILP problems [154, 188]. Additionally,
constraint programming (CP) [120] is an effective approach for modeling and
solving combinatorial optimization problems. In this context, tree search algo-
rithms and powerful pruning known as constraint propagation are used to find a
valid assignment for all variables of an optimization problem.

In many real-world optimization scenarios global optimality is often not the
most relevant property though. In a dynamic and quickly changing environment,
it is usually more reasonable to get a good solution quickly than to spend con-
siderably more resources to get a global optimal solution. Heuristic optimization
algorithms offer a reasonable tradeoff between computational effort and solution
quality in that case. In contrast to other optimization techniques, they construct
or improve solutions in some guided and intelligent way. Thereby only a small
portion of the solution space is investigated which requires less computational ef-
fort. As a consequence, heuristic algorithms cannot guarantee that they will find
a global optimal solution; even worse, they cannot even decide, if some solution
is a globally optimal solution or not. However, for many optimization problems
these drawbacks are not that critical.

2.1 Taxonomy of Heuristic Algorithms

In the last decades the domain of heuristic optimization has been an active field
of research. Empowered by the enormous increase of computational power of
computer hardware, it has become possible to apply heuristic algorithms to many
different and more and more complex optimization problems. As a result, a huge
variety of algorithms has been developed and discussed in the literature (see for example [190, 127]). Several taxonomies have been introduced to categorize these algorithms focusing on different properties. One frequently used categorization distinguishes between algorithms which iteratively construct a single high quality solution (construction heuristics) and algorithms which improve existing solutions step by step (improvement heuristics).

Another approach separates in problem-specific and problem-independent algorithms. Problem-specific heuristics focus on a particular optimization problem and are able to provide high quality solutions for instances of that problem in an efficient way. As an example, the Lin-Kernighan algorithm [117] for solving the traveling salesman problem can be mentioned. On the contrary, problem-independent algorithms are abstracted from a concrete optimization problem and provide solution strategies that can be applied to many different problems. Of course, they also contain problem-specific steps such as the evaluation of solutions, the generation of initial solutions, or the move from one solution to another, but these operations are encapsulated in a few problem-specific operators. In order to apply a problem-independent algorithm to an optimization problem, only these operators have to be adapted. All other parts of the algorithm stay the same and can be reused. Consequently, problem-independent heuristics often are not as efficient as problem-specific ones, but this is the price which has to be paid for a higher level of abstraction. From a theoretical point of view, problem-independent heuristics are especially interesting as enhancements of the problem-independent parts lead to an immediate benefit for all kinds of problems the algorithm is applied to. Furthermore, problem-independent heuristics are also important from a practical point of view, as they can be used as black box solvers and can be applied to new problems easily. Due to their characteristic of providing a high level search strategy to guide the use of problem-specific operators, problem-independent heuristics are also often referred to as metaheuristics [80, 83].

When developing new metaheuristics, nature often comes into play as a source of inspiration. For example, the foraging behavior of birds and ants was used as a metaphor for the development of ant colony optimization and particle swarm optimization. As another example, the evolution of species can be mentioned which inspired the development of metaheuristics called evolutionary algorithms. Therefore, one more property to describe heuristic optimization algorithms is whether they are inspired by a natural archetype or not.

Finally, metaheuristics can be categorized by the number of solutions they work on in each iteration. On the one hand, trajectory-based algorithms consider only a single element of the solution space at a time. They jump from one spot in the solution space to another, usually by sampling a new solution from the neighborhood of the current one, and try to reach promising regions in the solution space. The neighborhood of a solution is thereby defined by a neighborhood structure which is a function $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ from the solution space
This distinction between trajectory-based and population-based metaheuristics can also be found in [34] and is used in the following sections to introduce several representatives of typical metaheuristic algorithms (cf. [35]). More details on metaheuristics and their application to different problems can also be found in [56]. Furthermore, the topic of hybrid metaheuristics is also briefly touched at the end of this chapter.

### 2.1.1 Trajectory-Based Metaheuristics

#### Iterative Improvement Local Search

Iterative improvement local search (IILS), also known as greedy local search or hill climbing, represents the basic form of local search algorithms. It starts with some initial solution $s$ which is normally generated randomly. In each iteration a better solution is picked from the neighborhood of the current solution as long as a better solution can be found or some other termination criterion such as consumed CPU time or the number of evaluated solutions is not satisfied. The selection of a better solution can be done in two different ways: First improvement IILS picks the first better solution found in the neighborhood whereas best improvement IILS chooses the best solution found in the whole neighborhood. Listing 2.1 shows the IILS algorithm in pseudo-code.

```
1  s ← generate initial solution
2  WHILE termination criterion not met DO BEGIN
3      s ← choose better solution ∈ $\mathcal{N}(s)$
4  END WHILE
5  RETURN s
```

Listing 2.1: Iterative improvement local search (adapted from [35])

IILS always heads towards the next optimum located near the initial solution depending on the used neighborhood structure. As no diversification strategy is used, it has no possibility to escape from this optimum again. In the case of highly multimodal solution spaces this behavior is a severe drawback, as the found optimum is in most cases a local optimum of rather poor quality. This problem is also referred to as the problem of getting stuck in a local optimum and has led to the development of more advanced algorithms which incorporate
diversification steps to enable the search to explore other regions of the solution space.

**Simulated Annealing**

As stated in [35], simulated annealing (SA) is commonly said to be the oldest among the metaheuristics and one of the first algorithms that contained an explicit strategy to escape from local optima. It was inspired by the annealing process of metal and glass which assume a low energy configuration when cooled down, and is therefore a representative of a nature-inspired optimization algorithm. Its origins go back to the field of statistical mechanics and the Metropolis algorithm published in 1953 [124]. In 1983 Scott Kirkpatrick, Charles D. Gellart and Mario P. Vecchi generalized this algorithm, introduced the name “simulated annealing” and applied it to problems of computer design and the traveling salesman problem [106].

SA also starts with an initial solution $s$ which can be created randomly or using some heuristic construction rule. Similarly to first improvement IILS, a solution $s'$ is randomly selected from the neighborhood $\mathcal{N}$ of the current solution in each iteration. If this solution is better, it is accepted and replaces the current solution. However, if $s'$ is worse, it is not discarded immediately but is also accepted with a probability $P(s'|T_k, s) = e^{-|f(s') - f(s)|/T_k}$ depending on the actual temperature parameter $T_k$ and the quality difference. Listing 2.2 shows a pseudocode representation of the algorithm.

```
1 s ← generate initial solution
2 k ← 0
3 T_k ← initial temperature
4
5 WHILE termination criterion not met DO BEGIN
6     s' ← choose solution ∈ $\mathcal{N}(s)$ randomly
7     IF $s'$ is better than s THEN BEGIN
8         s ← s'
9     END
10 ELSE BEGIN
11            s ← s' with probability $P(s'|T_k, s) = e^{-|f(s') - f(s)|/T_k}$
12     END IF
13     T_{k+1} ← adapt temperature $T_k$
14     k ← k + 1
15 END WHILE
16 RETURN best solution found
```

Listing 2.2: Simulated annealing (adapted from [35])
Due to this stochastic acceptance criterion the temperature parameter $T_k$ can be used to balance diversification and intensification of the search. At the beginning the temperature should be high to enable the algorithm to easily escape from local optima. As the algorithm proceeds, the temperature has to be reduced step by step to focus the search on a promising region of the solution space such that it will converge eventually. The way how the temperature is decreased over time is defined by the cooling scheme. Frequently used cooling schemes include linear, geometric or logarithmic cooling. However, also more complex cooling strategies have been proposed that are not necessarily monotonous and for example suggest reheating phases to diversify the search again from time to time [119, 136]. The choice of an appropriate initial temperature and cooling scheme is crucial for the performance of the algorithm and therefore has to be adapted for each problem instance to which SA is applied.

Tabu Search

In 1986 Fred Glover introduced tabu search (TS) in [80]. In contrast to SA, TS is a memory-based method that uses the search history to navigate in the solution space and to prevent stagnation in local optima. Especially in the field of combinatorial optimization TS is considered to be one of the most successful metaheuristics. A detailed description of the algorithm and various applications can be found in [84].

Listing 2.3 outlines a basic TS algorithm. TS can be considered as best improvement IILS with an additional extension to prevent cycles. Without this modification the algorithm would always jump between a local optimum and the best solution in the neighborhood of this local optimum (next best solution). To prevent this behavior and to force the exploration of new areas of the solution space after reaching an optimum, TS uses a short term memory also called tabu list which stores solutions visited in the past in a FIFO list. In each iteration the tabu list is used to generate the allowed set of neighbors $N_a(s)$ by removing all solutions from the neighborhood of the current solution which have been visited in the last iterations. Then the best solution of the allowed set is chosen as new solution and the tabu list is updated. As storing complete solutions is too inefficient for many applications, the tabu list often contains only the solution components involved in a move. Also multiple tabu lists, one for each solution component, might be used. Finally the algorithm stops, if some termination criterion is met (e.g., execution time or number of evaluated solutions) or all neighboring solutions are tabu and the set of allowed solutions is empty.

The size of the tabu list (tabu tenure) represents an important parameter to influence intensification and diversification of the search. A shorter tabu list leads to a more intensive exploitation of a smaller area of the solution space, whereas a longer tabu list forces the algorithm to go to other regions of the solution space more quickly. Selecting an appropriate tabu tenure is a critical step and
2.1. TAXONOMY OF HEURISTIC ALGORITHMS

determines the success of the search. Therefore, several approaches have been discussed to adapt the length of the tabu list automatically during the execution of the algorithm [81, 162, 21]. However, as a longer tabu list might have the effect that a promising region of the solution space is not fully explored, additional aspiration criteria are frequently used which overrule the tabu condition of a solution. For example, if a solution is found in the neighborhood that is tabu but better than the best solution found so far, this solution is also included in the allowed set.

Besides the tabu list as a short term memory, additional memories can also be added in order to control the search process on a higher level. For example, an intermediate term memory can be used to keep track of promising regions of the solution space and to restart the algorithm with a shorter tabu list to perform an intensified search there. Another approach suggests a long term memory to store the frequency of solutions in order to penalize solutions that have been considered already. Additional information on these advanced TS concepts is summarized in [84].

Variable Neighborhood Search

Pierre Hansen and Nenad Mladenović presented variable neighborhood search (VNS) in 2001 [91]. As a solution that is locally optimal with respect to some neighborhood structure might not be locally optimal concerning another neighborhood structure, their idea was to use different neighborhood structures in a single run to diversify the search.

As outlined in Listing 2.4, the basic procedure of VNS works as follows: At first, \( k_{\text{max}} \) different neighborhood structures are defined and the initial solution \( s \) is created either randomly or using a construction heuristic. Then, the algorithm loops until some termination criterion is fulfilled. Each iteration consists of three

\begin{verbatim}
1 s ← generate initial solution
2 TL ← empty tabu list
3 WHILE termination criterion not met DO BEGIN
4 \( \mathcal{N}_a(s) \leftarrow \{ s' \in \mathcal{N}(s) \mid s' \text{ is not tabu or satisfies an aspiration condition} \} \)
5 \( s' \leftarrow \text{best solution} \in \mathcal{N}_a(s) \)
6 update TL with \( s \) and \( s' \)
7 s ← s'
8 END WHILE
9 RETURN best solution found

Listing 2.3: Tabu search (adapted from [35])
\end{verbatim}
different phases: In the shaking phase, a new solution $s'$ is randomly generated by applying one of the neighborhood structures to $s$. This solution serves as a starting point to perform a local search procedure (for example an IILS) which is not restricted to the neighborhood structures $N_k$ but might even use another neighborhood (local search phase). In the last phase (move phase), the result of the local search $s''$ is compared with the original solution $s$. If it is better, $s$ is replaced and the neighborhood index is set back to 1. If it is worse though, $s''$ is discarded and the neighborhood index is incremented by one. Therefore, the next neighborhood structure will be used in the next iteration.

```plaintext
1 select set of neighborhood structures $N_k$ where $k = 1, \ldots, k_{\text{max}}$
2 $s \leftarrow$ generate initial solution
3 WHILE termination criterion not met DO BEGIN
4   $k \leftarrow 1$
5   WHILE $k < k_{\text{max}}$ DO BEGIN
6     $s' \leftarrow$ choose solution $\in N_k(s)$ randomly // shaking phase
7     $s'' \leftarrow$ perform local search on $s'$ // local search phase
8     IF $s''$ is better than $s$ THEN BEGIN // move phase
9       $s \leftarrow s''$
10       $k \leftarrow 1$
11     END ELSE BEGIN
12       $k \leftarrow k + 1$
13     END IF
14   END WHILE
15 END WHILE
16 RETURN best solution found
```

Listing 2.4: Variable neighborhood search (adapted from [35])

Basically, VNS is a local search algorithm which uses different neighborhood structures to escape from a local optimum. The purpose of the shaking phase is to pick a solution in some neighborhood of the current optimum that might serve as a good starting point for another run of the local search procedure. If the picked solution is in the attraction basin of another local optimum, the search jumps from one local optimum to the next. However, it is important that the neighborhood structures used in the shaking phase are not too large and do not provide solutions too far away from the current solution. Otherwise, VNS becomes similar to a simple multi-start local search with random starting points.
2.1. TAXONOMY OF HEURISTIC ALGORITHMS

Iterated Local Search

Iterated local search (ILS) [121, 160, 118] is a very general metaheuristic that offers several degrees of freedom. As shown in Listing 2.5, ILS starts with an initial solution on which a local search procedure is applied leading to a (local) optimum \( \hat{s} \). In each iteration a new starting point \( s' \) is calculated by perturbing the current optimum \( \hat{s} \). On that solution local search is applied again which results in another optimum \( \hat{s}' \). Finally, the new optimum replaces the old one depending on some acceptance criterion. As in both steps, perturbation and acceptance, the history of the search might be used, ILS is another version of memory-based metaheuristics.

\[
\begin{align*}
1 & \quad s \leftarrow \text{generate initial solution} \\
2 & \quad \hat{s} \leftarrow \text{perform local search on } s \\
3 \quad & \text{WHILE terminate criterion not met DO BEGIN} \\
4 & \quad s' \leftarrow \text{perturbate } \hat{s} \\
5 & \quad \hat{s}' \leftarrow \text{perform local search on } s' \\
6 & \quad \text{IF acceptance criterion is satisfied THEN BEGIN} \\
7 & \quad \quad \hat{s} \leftarrow \hat{s}' \\
8 & \quad \text{END IF} \\
9 & \quad \text{END WHILE} \\
10 & \quad \text{RETURN best solution found}
\end{align*}
\]

Listing 2.5: Iterated local search (adapted from [35])

Obviously, ILS follows a trajectory of local optima \( \hat{s}_1, \ldots, \hat{s}_n \). Thereby the choice of the perturbation scheme is crucial: On the one hand, the new starting point should be located outside the attraction basin of the current local optimum; on the other hand, if perturbation is too strong, ILS behaves like a random multi-start local search. Therefore, the strength of the perturbation has to be selected according to the tackled problem instance. More sophisticated versions of ILS also suggest a variable perturbation procedure that adapts its strength dynamically in order to have a good balance of diversification and intensification of the search.

Due to its very generic formulation, ILS can be seen as a high level definition of trajectory-based metaheuristics. Other algorithms such as VNS or SA can be described as special cases of ILS.

2.1.2 Population-Based Metaheuristics

Evolutionary Algorithms

All metaheuristic algorithms inspired by the Darwinian principle of “survival of the fittest” [50] and the process of evolution are denominated as evolutionary al-
algorithms (EAs). In general, EAs mimic the natural process of species adapting to
the environment and simulate this concept to solve combinatorial or continuous
optimization problems. The foundations of EAs date back to the 1960s and 1970s.
In that time several slightly different algorithms were proposed, but fundamen-
tally all of them followed similar ideas. The most prominent variants of EAs are
evolution strategies (ES), developed by Ingo Rechenberg in Germany [147, 148],
and genetic algorithms (GAs), introduced by John H. Holland and his students
in the USA [93, 88]. Although the ES and GA community competed heavily in
the early days of EAs, in the recent years effort can be noticed to unite both
approaches and to develop a unified and general model for evolutionary compu-
tation [53].

\[
\begin{align*}
1 & P \leftarrow \text{generate initial population} \\
2 & \text{evaluate } P \\
3 & \text{WHILE termination criterion not met DO BEGIN} \\
4 & \quad P_{\text{parents}} \leftarrow \text{select solutions from } P \\
5 & \quad P_{\text{children}} \leftarrow \text{recombine and/or mutate } P_{\text{parents}} \\
6 & \quad \text{evaluate } P_{\text{children}} \\
7 & \quad P \leftarrow \text{select from } P \text{ and } P_{\text{children}} \\
8 & \text{END WHILE} \\
9 & \text{RETURN best solution found}
\end{align*}
\]  

Listing 2.6: Evolutionary algorithm

Listing 2.6 shows the basic procedure of EAs in a very generalized form. In
each iteration three different steps are applied on a set of solutions usually called
the parent population:

- **Selection**
  First, a selection method is used to pick solutions from the current popu-
lation that should be modified. If the selection probability of a solution is
proportional to the solution quality, which is usually the case for GAs, this
steps plays an important role for directing the search process.

- **Modification**
  Second, the selected solutions are manipulated. In general, two different
modification concepts can be applied either separately or in a combined
way. On the one hand, solution components of two or more solutions can
be combined to create a new valid and hopefully better solution (crossover).
On the other hand, some kind of local modification can be used to change
single solutions (mutation).
• **Replacement**
  Third, a set of solutions has to be selected from the modified solutions and perhaps also from the current population to build a new generation of parent solutions for the next iteration. Again, if the solution quality is used as a decision criterion whether a solution should become a member of the next generation or not, replacement is also an important factor for navigating the search through the solution space.

The general EA model described above offers many degrees of freedom to the algorithm designer. How selection, modification, and replacement are implemented depends on the type of EA as well as on the tackled problem. This is probably one of the main success factors of EAs, as they can be easily adapted to particular optimization problems. As a result, many different EA derivatives have been proposed for multiple optimization problems and solution representations. In this context genetic programming (GP) [110] should also be mentioned as a prominent example which uses GAs to evolve computer programs represented as tree structures.

Furthermore, other search strategies can be integrated as well to build hybrid heuristic optimization algorithms. For example, memetic algorithms represent EAs which additionally use some kind of local search method in the modification step [129]. A comprehensive discussion of different EA variants and many application areas can be found in several publications such as [22, 23, 65, 92, 71].

**Scatter Search**

Scatter search (SS) and its generalized form called path relinking were developed by Fred Glover in the late 1990s [82, 85]. As described in [87], SS basically consists of five methods:

• **Diversification Generation Method**
  The diversification generation method aims to create a set of solutions as different to an existing set of solutions as possible.

• **Improvement Method**
  The improvement method tries to improve a solution usually by applying some kind of local search procedure.

• **Reference Set Update Method**
  This method is used to build and maintain the reference set containing the “best” solutions found so far. The notion of best is thereby not limited to the solution quality. Other quality criteria such as the diversity of solutions are also taken into account.

• **Subset Generation Method**
  The subset generation method operates on the reference set and selects
subsets for creating combined solutions. The most common method is to generate all pairs of reference solutions.

- **Solution Combination Method**

Finally, the solution combination method takes the generated subsets of the reference set and combines all solutions of each subset in order to create new solutions. This method is similar to the crossover operator used in EAs.

As outlined in Listing 2.7 these methods are used as follows: At first, an initial set of solutions is created using some heuristic method and the initial reference set is selected. Note that the diversification generation method might also be used to get a highly diverse set of initial solutions. Then, in each iteration of the algorithm, an intensification phase and a diversification phase are executed. The intensification phase (inner loop) is repeated as long as the reference set changes (i.e., better solutions are found). The subset generation, combination, and improvement methods are applied to create new solutions; then the reference set is updated if possible. By this means, the regions of the solution space defined by the reference solutions are exploited as much as possible. After the reference set has converged, diversification takes place by applying the diversification generation method on the reference set and choosing a new reference set again.

```
1 P ← create initial solutions
2 P_{ref} ← choose reference set from P
3 
4 WHILE termination criterion not met DO BEGIN
5 REPEAT
6 P_{sub} ← select subsets from P_{ref}
7 P_{comb} ← apply recombination on P_{sub}
8 P_{comb} ← apply improvement on P_{comb}
9 P_{ref} ← update reference set with P_{comb}
10 UNTIL P_{ref} converges
11 
12 P ← apply diversification generation on P_{ref}
13 P_{ref} ← choose reference set from P
14 END WHILE
15 
16 RETURN best solution found
```

Listing 2.7: Scatter search (adapted from [85, 35])

Originally, SS was designed to optimize solutions encoded as points in the Euclidean space. However, in the recent years increasing interest in this meta-heuristic concept also showed how it can be applied to other problems such as
linear ordering, route planning, graph coloring, or multi-objective optimization (see for example [87, 86]).

### Ant Colony Optimization

Ant colony optimization (ACO) [61, 59, 63] is a general term for metaheuristic algorithms inspired by the foraging behavior of ants. A single ant has very limited perceptual and intellectual capabilities. However, an ant colony is able to perform an implicit route optimization to find shortest paths between the nest and food sources. This is achieved by collective communication based on a chemical substance called pheromone which is dropped by each ant while walking around. The amount of emitted pheromone depends on the quality and quantity of food carried back to the nest and of course on the number of ants moving over a place. As ants prefer to walk where the pheromone concentration is high, shorter trails to rich food sources are preferred. ACO mimics this concept to solve combinatorial optimization problems.

Contrarily to EAs or SS, ACO belongs to the category of construction heuristics. The construction of solutions is based on a probabilistic model called the pheromone model which represents a probability distribution over the solution space. In ACO algorithms a so-called construction graph is used whose vertices represent solution components and the edges describe valid transitions for adding a solution component to a partial solution. Consequently, every solution is equivalent to a path connecting its components in the construction graph. Each edge is weighted by a pheromone value that gives the probability for selecting the corresponding transition. In every iteration each virtual ant starts with an empty partial solution on a randomly chosen vertex of the construction graph. Step by step it assembles a solution by selecting transitions and moving through the graph. Additionally, in most ACO implementations the probability of a move is also biased by a heuristic value depending on the quality of the target solution component. This allows for additional parameters to weight the influence of pheromone and heuristic information.

After all ants have finished constructing a solution, the pheromone values are updated. This step is crucial to balance intensification and diversification of the search. On the one hand, pheromone has to be added to the edges of one or more high quality solutions of the current or even former iterations. This leads to a stronger focus on promising regions in the solution space. On the other hand, to prevent the algorithm from converging too quickly, pheromone has to be removed again which is also known as pheromone evaporation. The choice of an appropriate pheromone update rule is therefore an important step and often depends on the characteristics of the problem the algorithm is applied on.

Finally, daemon actions might be used to implement actions that cannot be done by single ants. For example, in hybrid ACO algorithms a local search procedure is applied to the constructed solutions in that step. Furthermore, a
daemons can also be used to collect information for the pheromone update rule in order to bias the search from a global point of view.

The basic ACO procedure consisting of solution construction, pheromone update, and optional daemon actions is shown in Listing 2.8. Note that these three steps do not necessarily have to be executed in that order; it is up to the algorithm designer to decide how they are scheduled and synchronized. Beside the ant system, which was the first ACO algorithm proposed by Marco Dorigo in his PhD thesis in 1992 [58], several other algorithms have been presented that differ in the way how transitions are selected in the construction graph and how pheromone values are updated (see for example [60, 161, 33]). Additional information on the principles of ACO and various fields of applications can be found in [62, 63].

Listing 2.8: Ant colony optimization (adapted from [32])

```
1 WHILE termination criterion not met DO BEGIN
2 ant based solution construction
3 pheromone update
4 optional demon actions
5 END WHILE
6
7 RETURN best solution found
```

Particle Swarm Optimization

Particle swarm optimization (PSO) [104, 68] is a metaheuristic inspired by the foraging behavior of bird flocks and was introduced by James Kennedy and Russel C. Eberhardt in 1995.

As outlined in Listing 2.9, PSO starts with a randomly generated initial population or swarm. For each solution a velocity vector \( v_s \) is initialized and the solution is memorized as its personally best solution \( p_{best}^s \). As long as the termination criterion is not met, each solution is moved in every iteration. Therefore, its velocity vector \( v_s \) is updated with respect to its personally best solution \( p_{best}^s \) and the globally best solution \( g_{best} \) found so far. Then the particle is moved in the solution space depending on its velocity vector and the personally best solution is updated, if a better solution has been found. Finally, if the whole swarm has found a better solution, \( g_{best} \) is updated as well.

Updating the velocity vector of each solution is an important step regarding the performance of the algorithm. Both movement directions, towards the personally best and the globally best solution, are multiplied by a random variable uniformly distributed between 0 and 1 to introduce non-determinism. Additionally, they are weighted by two values called cognitive and social parameter. On
Listing 2.9: Particle swarm optimization

```
1 \( P \leftarrow \) generate initial population randomly
2 evaluate \( P \)
3 \textbf{FOREACH} \( s \) IN \( P \) \textbf{DO BEGIN}
4 \( v_s \leftarrow \) initialize velocity vector
5 \( p_s^{\text{best}} \leftarrow s \)
6 \textbf{END FOREACH}
7 \( g^{\text{best}} \leftarrow \) best solution of \( P \)
8 \textbf{WHILE} termination criterion not met \textbf{DO BEGIN}
9 \textbf{FOREACH} \( s \) IN \( P \) \textbf{DO BEGIN}
10 \( v_s \leftarrow \) update velocity vector
11 move \( s \) by \( v_s \)
12 evaluate \( s \)
13 \textbf{IF} \( s \) is better than \( p_s^{\text{best}} \) \textbf{THEN BEGIN}
14 \( p_s^{\text{best}} \leftarrow s \)
15 \textbf{END IF}
16 \textbf{END FOREACH}
17 \textbf{IF} \( \exists s \in P \) that is better than \( g^{\text{best}} \) \textbf{THEN BEGIN}
18 \( g^{\text{best}} \leftarrow \) best solution of \( P \)
19 \textbf{END IF}
20 \textbf{END WHILE}
21 \textbf{RETURN} \( g^{\text{best}} \)
```

the one hand, a high value of the cognitive parameter puts the focus on local exploration of the solution space around each particle (diversification). On the other hand, a high value of the social parameter attracts the whole swarm to the region of the globally best solution found so far (intensification). Therefore, these two weighting parameters have to be set carefully and with respect to the problem instance in order to obtain a good balance of intensification and diversification.

Due to the notion of a velocity vector moving the solutions through the solution space, PSO is particularly suitable for tackling problems that can be encoded as \( n \)-dimensional real vectors.

Further Metaheuristics

Apart from the typical representatives of trajectory-based and population-based metaheuristics presented above, several other algorithms have been proposed that also belong to either of these two categories. Some of them are listed in the following, even though this list is not complete as the research community focusing on
metaheuristic search is very active and new algorithms are frequently published. Additional information about these algorithms can be found in the referenced publications.

- Greedy Randomized Adaptive Search Procedure (GRASP) [69, 139]
- Guided Local Search (GLS) [166, 167]
- Very Large-Scale Neighborhood Search (VLSN) [18, 44]
- Variable Depth Search (VDS) [105, 117, 95]
- Estimation of Distribution Algorithms (EDA) [126, 125]
- Evolutionary Programming (EP) [72, 73]
- Artificial Immune Systems (AIS) [51, 52]

### 2.1.3 Hybrid Metaheuristics

In the last years so-called hybrid metaheuristics have become more and more popular that do not strictly stick to one specific metaheuristic approach. In many complex real-world optimization scenarios these algorithms are able to outperform classical metaheuristics by exploiting the advantages of different concepts. However, the development of an effective hybrid approach is in general a difficult task and highly depends on the tackled optimization problem. Various ways of hybridization have been discussed in the literature and a comprehensive overview can be found in [144].

Basically, three different categories of hybrid algorithms can be distinguished: Metaheuristics can be hybridized with each other, metaheuristics can be combined with problem-specific algorithms, or metaheuristics can be used together with other optimization algorithms as for example exact optimization techniques or neural networks.

A typical strategy that belongs to the first category is embedding metaheuristic algorithms in order to improve solutions during the search process of another metaheuristic. For example, memetic algorithms [128, 129] follow this approach: They combine evolutionary algorithms with trajectory-based metaheuristics to optimize some or all solutions of the population; a local search algorithm is thereby used to intensify the search and to focus the surrounding EA on local optima.

Regarding the second category, metaheuristics can be combined with problem-specific construction heuristics that are used to obtain good starting points for the search (cf. GRASP [69, 139]) or to transform indirect or incomplete representations into complete and feasible solutions (decoder-based algorithms). The
latter strategy is frequently found in algorithms solving combinatorial optimization problems, if a permutation-based solution encoding is used [103]. For example, when solving scheduling problems, an order of jobs can be represented as a permutation; in this case, a schedule builder such as the Giffler-Thompson algorithm [79] has to be applied to create concrete schedules which can be evaluated with respect to some objective function [189, 38].

Last but not least, the combination of metaheuristics with exact optimization techniques is a typical example of an approach that belongs to the last category [66, 143, 145]: On the one hand, exact optimization techniques such as ILP or CP can be used within metaheuristics to reduce the solution space, to efficiently search large neighborhoods, to merge solutions, or to gain additional information to guide the search by solving a relaxation of the tackled problem. On the other hand, many exact optimization techniques rely on good bounds to restrict the area of the solution space that has to be examined; therefore, metaheuristics can be used to quickly obtain good and feasible solutions [152].
Chapter 3

Requirements Analysis

Developing a generic software system for heuristic optimization is a challenging task. The variety of heuristic optimization paradigms and the multitude of application domains (cf. Chapter 2) make it difficult for software engineers to build a system that is on the one hand flexible enough and on the other hand provides a large amount of reusable components. Additionally, the users of a heuristic optimization software system are also very heterogeneous concerning their individual skills and demands. As a consequence, an analysis of requirements is essential so that a heuristic optimization software system will meet the users’ needs.

When considering literature on optimization software systems for different heuristic algorithms, some requirements are repeatedly stated by researchers. For example in [76], Christian Gagné and Marc Parizeau define the following six genericity criteria to qualify evolutionary computation frameworks: generic representation, generic fitness, generic operations, generic evolutionary model, parameters management, and configurable output. Quite similar ideas can also be found in [100, 163, 102, 172].

Although most of these aspects reflect important user demands, none of these publications sketch a clear picture of the system’s target users groups. As a consequence, without a precise picture of the users it is hard to determine whether the list of requirements is complete or some relevant aspects have been forgotten. Thus, before thinking about and defining requirements, it is necessary to identify all users.

3.1 User Groups

In general, users of a heuristic optimization system can be categorized into three often overlapping groups: practitioners, trying to solve real-world optimization problems with classical or advanced heuristics; heuristic optimization experts, analyzing, hybridizing and developing advanced algorithms; and students, trying to learn about and work with heuristic optimization algorithms and problems. As
a result, we have three groups of users called practitioners, experts and students.
Their views on the area of heuristic optimization and their individual needs are
described in detail in the following.

3.1.1 Practitioners

Practitioners are people who have encountered some difficult (often NP-hard) optimization problem and who want to get a solution for that problem. Hard optimization problems can be found in almost every domain (for example in engineering, medicine, economics, computer science, production, or even in arts), so this group is huge and very heterogeneous. Due to that heterogeneity, it is not possible to list all the domains where heuristic optimization algorithms have already been successfully applied or even to think of all possible domains in which they might be applied successfully in the future. Therefore, further refinement and categorization of the members of this user group is omitted.

Seen from an abstract point of view, practitioners work in a domain usually not related to heuristic optimization or software engineering. They normally have very little knowledge of heuristic algorithms but a profound and deep knowledge of the problem itself, its boundary conditions and its domain. This results in a highly problem-oriented way of thinking; in this context heuristic optimization algorithms are merely used as black box solvers to get a solution.

Usually, practitioners want to get a satisfactory solution to their problem as quickly as possible. Each second spent on computation means that some real-world system is running in a probably sub-optimal state. To them, time is money, and thus their number one concern is performance. Take for example a scheduling problem in a production plant manufacturing some heavy machinery. The logisticians of this company are interested in an optimal schedule of operations in order to minimize tardiness of orders and to keep customers happy. Any time a machine breaks down, a new order is accepted, or the available capacity changes, a new schedule is needed at once. Each minute production is continued without following an optimized schedule may lead to the wrong operations being chosen for production. This may finally result in a higher tardiness on some high priority orders causing penalties and a severe loss of money.

Parallelism and scalability are of almost equal importance. In our fictive production company the heuristic optimization software system used to compute optimized schedules should be able to provide equally good results, even if business is going well and there are twice as many production orders to be scheduled. A simple equation should hold: More computing power should either lead to better results or to the possibility to solve larger problems. Therefore, it has to be possible to enhance the optimization system with some additional hardware to obtain better performance.

Next, practitioners require a high level of genericity. Due to the heterogeneous nature of domains in which optimization problems might arise, a software sys-
system has to support easy integration of new problems. Usually this integration is
done by implementing problem-specific objective functions, custom solution repre-
sentations, and a generic way to introduce new operations on these solutions.
For example, new solution manipulation operators might have to be developed,
respecting some constraints a feasible solution has to satisfy.

Another important aspect is the integration of a heuristic optimization soft-
ware system. Usually an optimization system is not a stand-alone application.
Data defining a problem instance is provided by other existing software systems
and solutions have to be passed on to other applications for further processing.
For this reason, in most real-world scenarios heuristic optimization software sys-
tems have to be integrated into a complex network of existing IT infrastructure.
Well-defined interfaces and technology for inter-system communication and cou-
ping are necessary.

Finally, due to the highly problem-oriented focus of practitioners, they should
not have to deal with the internals of algorithms. After a problem has been de-
efined and represented in a heuristic optimization software system, the system
should provide a comprehensive set of classical and advanced optimization algo-
rithms. These algorithms can be evaluated on the concrete problem at hand and
a best performing one can be chosen as a black box solver for live operation on
real-world data.

3.1.2 Experts

Experts are researchers focusing on heuristic optimization algorithm engineer-
ing. Their aim is to enhance existing algorithms or develop new ones for various
kinds of problems. Following the concept of metaheuristics, especially problem-
independent modifications of algorithms are of major interest. By this means
different kinds of optimization problems can benefit from such improvements. As
a result and in contrast to practitioners, experts consider algorithms as white
box solvers. For them, concrete optimization problems are less important and
are used as case studies to show the advantageous properties of a new algorithmic
concept, such as robustness, scalability, and performance in terms of solution
quality and runtime. In many cases, problem instances of well-known bench-
mark optimization problems, as for example the traveling salesman problem or
n-dimensional real-valued test functions, are used. Ideally, a comprehensive set
of benchmark problems is provided out of the box by a heuristic optimization
software system.

Due to the focus on algorithms, one main concern of experts is genericity. A
heuristic optimization software system should offer abilities to integrate new al-
gorithmical concepts easily. There should be as few restrictions in the underlying
algorithm model as possible, enabling the incorporation of techniques stemming
from different areas such as evolutionary algorithms, neighborhood-based search
or swarm systems (hybridization), the flexible modification of existing algorithms,
or the development of new ones. The sequence of operations applied to one or more solutions during algorithm execution, in other words the algorithm model, should be freely configurable. Furthermore, experts, similarly to practitioners, demand the integration of new operations, solution representations and objective functions.

One main task of experts is testing algorithms on different kinds of problems, as empirical evaluation is necessary to analyze properties of a new algorithm. Thus, automation of test case execution and statistical analysis play an important role. Thereby, performance in terms of execution time is usually just of secondary importance, as time constraints are not that financially critical as they are for practitioners.

In order to get some hints for algorithm improvements, experts have to use various tools to obtain a thorough understanding of the internal mechanisms and functionality of an algorithm. Since many heuristic optimization algorithms are very sensitive to parameters values, a generic way of parameter management is of great value, owing to the fact that various parameters have to be adjusted from test run to test run. Furthermore, stepwise algorithm execution and customizable output are the basis for any in-depth algorithm analysis, in order to get a clearer picture of how an algorithm is performing and what effect was caused by some parameter adjustment or structural modification.

Finally, replicability and persistence have to be mentioned: Each algorithm run has to be reproducible for the sake of later reference and analysis. A persistence mechanism is thus an essential means by which a run can be saved at any time during its execution and can be restored later on.

3.1.3 Students

Students entering the area of heuristic optimization are users that are located between the two user groups described above. In the beginning, they experiment with various heuristic optimization techniques and try to solve well-known benchmark problems. Therefore, a comprehensive set of classical heuristic optimization algorithms and benchmark problems should be provided by a heuristic optimization software system.

During several experiments and algorithm runs, students gain more and more insight into the internal functionality of algorithms and the interdependency between diversification and intensification of the search process. As the mystery of heuristic optimization is slowly unraveled, their view of algorithms changes from black box to white box. Hence, requirements relevant to experts - such as genericity, parameter management, automation, or customizable output - become more and more important to these users as well.

Additionally, when using heuristic optimization techniques and a heuristic optimization software system for the first time, an easy-to-use and intuitive application programming interface (API) is helpful to reduce the necessary learning
3.2. MAIN REQUIREMENTS

Effort. Even more, a graphical user interface (GUI) is extremely advantageous, so that students do not have to worry about peculiarities of programming languages and frameworks. Instead, with a GUI they are able to concentrate on the behavior of algorithms on an abstract level. Especially, studying charts and logs of the internal state of an algorithm during its execution is very effective to gain a deeper understanding of algorithm dynamics.

3.2 Main Requirements

Based on the analysis of the three user groups, the following characteristics of heuristic optimization software systems can be defined which represent the main requirements in the context of this thesis\(^1\) (some of these requirements can be found in similar form in [101, 163, 102, 172]):

- **Automation**
  As heuristic algorithms are per se non-deterministic, comparison and evaluation of different algorithms requires extensive empirical tests. Therefore, a heuristic optimization software system should provide functionality for experiment planning, automated algorithm execution, and statistical analysis. In this context, the integration of existing frameworks for software testing might be a reasonable approach.

- **Customizable Output**
  In a real-world scenario heuristic optimization never is an end in itself. To enable further processing of results with other applications, the user has to customize the output format of a heuristic algorithm. Furthermore, user defined output is also required by experts to visualize the internal mechanisms of algorithms by logging internal states such as distribution of solutions in the solution space, similarity of solutions, or stagnation of the search.

- **Generic Algorithm Model**
  In order to represent different kinds of heuristic optimization algorithms, the main algorithm model has to be flexible and customizable. It should not be dedicated or limited to any specific heuristic optimization paradigm. Especially, this aspect should also be kept in mind in terms of naming of classes and methods, so that users are not irritated or misled by the API.

- **Generic Operators**
  Related to the demand for a generic algorithm model, the operations applied by a heuristic algorithm should be generic as well. The user has to be able

---

\(^1\)The requirements are listed alphabetically; the order does not reflect the importance of each requirement.
to implement either problem-specific or generic operations in an easy and intuitive way. There has to be a standardized interface for all operations and a uniform way how data is represented, accessed, and manipulated.

- **Generic Objective Functions**
  To enable integration of custom optimization problems, a generic concept of objective functions has to be available. Users should be able to add custom methods for quality evaluation easily. The quality evaluation mechanism has to be based on a clearly defined interface, so that all other operations depending on quality values - such as selection or heuristic manipulation operations - do not have to take care of how the quality of a solution is calculated in detail. In that context, working with the quality of a solution has to be abstracted from the concrete quality representation. For example, there should be no difference for selection operations whether the optimization problem is single-objective or multi-objective or a minimization or maximization problem. Therefore, a generic way of comparing two solutions is necessary.

- **Generic Solution Representations**
  As users need to integrate custom optimization problems, not only generic objective functions but also a generic way of solution representation is required. Ideally, the user should be able to assemble a custom solution representation by combining different standard data representations such as single values, arrays, matrixes or enumerations of different data types. As an alternative, solution representations using complex custom data structures independent of any predefined ones should also be supported. This requirement has a strong impact on the requirement of generic operators, as crossover or manipulation operators have to work on solutions directly.

- **Graphical User Interface**
  To pave the way to heuristic optimization for users not so familiar with software development or specific programming languages, a heuristic optimization software system needs to be equipped with a graphical user interface (GUI). Users should be able to modify or develop algorithms without depending on any specific development environment. Furthermore, a GUI is also very helpful for experimenting and rapid prototyping, as algorithms can be visualized seamlessly directly within the system.

- **Integration**
  After their development, heuristic optimization algorithms for real-world optimization problems have to be integrated into some existing information technology landscape. Hence, a heuristic optimization software system should be modular to be able to integrate just the parts required in a custom scenario. Generic communication protocols for the system and
its environment are necessary for passing new optimization tasks into and getting results out of the system.

- **Learning Effort**
  Users of a heuristic optimization software system should be able to start to work with the system quickly. Only little knowledge of programming languages and just basic skills in programming and software development should be necessary. The API of a heuristic optimization software system should therefore be intuitive, easy to understand, and should follow common design practices. Additionally, a high level user interface should be provided to decouple algorithm and problem engineering from software development.

- **Parallelism**
  A heuristic optimization software system should be scalable in terms of computing power. Using additional computing resources should either enable the user to solve larger problems, or to achieve better solution quality. Consequently, exploitation of parallelism is an important success factor. A heuristic optimization software system has to offer a seamless integration of parallelism for development and execution of parallel algorithms. Ideally, the user just has to define which parts of an algorithm should be executed in parallel, without having to think about how parallelization is finally done. Due to the duality of high-performance computing systems (shared memory multi-core CPUs versus distributed memory cluster or grid systems) parallelization concepts for both architectures should be provided.

- **Parameter Management**
  Many heuristic optimization algorithms offer several parameters to influence their behavior. As performance of most algorithms is very sensitive in terms of parameter values, users need to run an algorithm many times to find an optimal configuration. Consequently, a generic parameter management facility is required to change parameters without needing to modify any program code or to recompile operators or, even worse, the whole system.

- **Performance**
  Heuristic optimization applications are usually time-critical. Many objective functions of real-world optimization problems as well as heuristic optimization algorithms are very expensive in terms of execution time. Thus a heuristic optimization software system should support runtime efficient implementation and execution of algorithms.

- **Predefined Algorithms and Problems**
  To enable solving of optimization problems or comparison of algorithms out of the box, a heuristic optimization software system has to provide a broad spectrum of predefined algorithms and problems. Especially, it should be
possible to use parts of existing algorithms and problems as a basis for further development or hybridization.

- **Replicability and Persistence**
  As experimental evaluation is a substantial task in heuristic algorithm development, test runs of algorithms have to be reproducible. Users should be able to save an algorithm and to restore it later on. The software system should therefore also enable stopping and saving algorithms during execution at any time. In that context, random number generators have to be handled with care, depending on whether an algorithm should be replayed with the same or a different random number sequence.

As a summary of this chapter Figure 3.1 shows a chart stating the opinion of the author concerning the relative importance of each requirement to the three user groups.

![Figure 3.1: Relative importance of requirements for each user group](image)
Chapter 4

Existing Software

Modern concepts of software engineering such as object-oriented or component-oriented programming represent the state of the art for creating complex software systems by providing a high level of code reuse, good maintainability and a high degree of flexibility and extensibility (see for example [123, 99, 48, 89]). However, such approaches are not yet established on a broad basis in the area of heuristic optimization, as this field is much younger than classical domains of software systems (e.g., word processing, calculation, image processing, or integrated development environments). Most systems for heuristic optimization are one man projects and are developed by researchers or students to realize one or a few algorithms for solving a specific problem. Naturally, when a software system is developed mainly for personal use or a very small, well-known and personally connected user group, software quality aspects such as reusability, flexibility, genericity, documentation and a clean design are not the primer concern of developers. As a consequence, seen from a software engineering point of view, in most cases these applications still suffer from a quite low level of maturity.

In the last years and with the ongoing success of heuristic algorithms in scientific as well as commercial areas, the heuristic optimization community started to be aware of this situation. Advantages of well designed, powerful, flexible and ready-to-use heuristic optimization frameworks were discussed in several publications [151, 116, 163, 101, 172, 76], identifying similar requirements as described in the previous chapter. Furthermore, some research groups started to head for these goals and began redesigning existing or developing new heuristic optimization software systems which were promoted as flexible and powerful white or even black box frameworks, available and useable for a broad group of users in the scientific as well as in the commercial domain. In comparison to the systems available before, main advantages of these frameworks are on the one hand a wide range of ready-to-use classical algorithms, solution representations, manipulation operators, and benchmark problems which make it easy to start experimenting and comparing various concepts. On the other hand, a high degree of flexibility due to a clean object-oriented design makes it easy for users to implement custom
CHAPTER 4. EXISTING SOFTWARE

extensions such as specific optimization problems or algorithmic ideas.

One of the most challenging tasks in the development of such a general purpose heuristic optimization framework is the definition of an object model representing arbitrary heuristic optimization paradigms. On the one hand, this model has to be flexible and extensible to a very high degree so that users can integrate non-standard algorithms that often do not fit into existing paradigms exactly. On the other hand, the model should be very fine-grained so that a broad spectrum of existing classical algorithms can be represented as algorithm modules. Then, these modules can serve as building blocks to realize different algorithm variations or completely new algorithms with a high amount of reusable code.

Consequently, the question is on which level of abstraction such a model should be defined. A high level of abstraction leads to large building blocks and a very flexible system. A lower level of abstraction supports reusability by providing many small building blocks, but the structure of algorithms has to be predefined more strictly in that case which reduces flexibility. As a consequence, these two requirements are contradictory to some degree.

Taking a look at several existing frameworks for heuristic optimization, it can be seen that this question has been answered in quite different ways. In the following sections some of the (in the opinion of the author) most relevant and mature heuristic optimization software systems available today are described\(^1\).

At the end of this chapter these frameworks are finally compared and analyzed with respect to the requirements stated in the previous chapter, showing that none of them is able to satisfy all of them completely.

4.1 Heuristic Optimization Frameworks

4.1.1 Templar

Templar\(^2\) [100, 101] is a C++ framework for implementing arbitrary search techniques to solve combinatorial optimization problems. It was developed by Martin Jones and his colleagues at the University of East Anglia from 1995 to 2000.

Templar decomposes heuristic search algorithms into three main parts: algorithms, problems, and solution representations. Hence, the framework provides three abstract base classes called TrEngine, TrProblem, and TrRepresentation from which concrete algorithms, problems and representations have to be inherited. Additionally, the framework includes several concrete implementations such as TrPermutation, SAEngine, GAEngine, or TSP. Furthermore, TrEngine and TrProblem have a common base class TrComponent which contains functionality for defining parameters, managing status values, or naming instances. The relationships of all these classes is shown schematically in Figure 4.1.

\(^1\)Additional descriptions and comparisons of frameworks can be found in [151, 164, 178, 76].

\(^2\)http://www.sys.uea.ac.uk/~templar
Communication between algorithms and problems is realized on the level of base classes. Therefore, each algorithm does not have any knowledge on which optimization problem it is applied and which kind of solution encoding is used; vice versa a TrProblem instance does not know anything about the concrete engine that is used to solve it. This architecture leads to a clean separation between problem-specific and problem-independent parts and supports code reuse, as an algorithm can be used to solve any kind of problem.

Problem-specific operations, such as evaluating or manipulating a solution, performing a neighborhood move, or recombining two solutions, are realized by so-called abilities. Abilities are objects that are added to TrProblem or TrRepresentation instances and represent basic functionality that can be queried and used by engines. Templar provides a catalog of pre-defined abilities that represent fundamental building blocks required for many heuristic optimization algorithms. However, the list of abilities can also be extended on demand, if some new ability is required. Thus, abilities represent one important way of abstraction in Templar. As abilities are loaded and updated at run-time, Templar provides a dynamic block-box oriented way of code reuse in contrast to polymorphic compile-time approaches of other frameworks.

One major focus in the design and implementation of Templar is put on the parallel and distributed execution of engines. In order to utilize multi-core or
CHAPTER 4. EXISTING SOFTWARE

multi-CPU systems, each engine is executed in an own thread and can be distributed on remote hosts. Communication between engines is realized according to the message passing interface (MPI) standard. As multiple engines can work on a single TrProblem instance and can exchange information using asynchronous messages, Templar enables the development of cooperative and hybrid heuristic search techniques.

Finally, Templar also provides a simple graphical user interface built on top of the framework. By using this application, users can easily instantiate and link arbitrary TrEngine and TrProblem instances and observe the progress of the search.

4.1.2 HotFrame

HotFrame\(^3\), developed by Andreas Fink and his colleagues at the University of Hamburg [70], is an object-oriented framework for heuristic search applications implemented in C++. It is based on a comprehensive domain analysis of meta-heuristic search with a strong focus on local search techniques (simulated annealing, tabu search, iterated local search). Nevertheless, HotFrame is not restricted to that kind of algorithms. The classes provided in HotFrame realize a complex but very flexibly algorithm model which can be used to represent other search strategies as well (e.g., evolutionary algorithms or candidate list strategies).

The main design goals of HotFrame are flexibility and run-time efficiency. Therefore, HotFrame relies on inheritance, polymorphism, generic programming, and static type variation; C++ templates are used excessively. It provides several generic classes representing algorithm building blocks that have to be parameterized with problem-specific types such as solution spaces or neighborhoods. As a consequence of generic programming and C++ templates, applications based on HotFrame are of a more static nature compared to other frameworks, as generic classes are instantiated by the compiler and cannot be modified dynamically at runtime. For example, if a user wants to change the neighborhood structure used in an algorithm, the application has to be compiled again.

As the algorithm model of HotFrame is very fine-grained to improve flexibility of the framework, a lot of generic classes have to be parameterized to build an application. This is a complex task and requires a profound knowledge of the framework. As an alternative, a generator application can be used for configuration that is implemented in Java\(^\text{TM}\) and provides a GUI. It produces customized source code adapted to the selected metaheuristic and problem type. However, additional graphical user interfaces to visualize or steer the search process are not provided.

\(^3\)http://www1.uni-hamburg.de/IWI/hotframe/hotframe.html
4.1. HEURISTIC OPTIMIZATION FRAMEWORKS

4.1.3 Evolving Objects (EO)

Evolving Objects (EO)\(^4\) is the name of a C++ class library for evolutionary computation originally developed by Juan J. Merelo and his team at the University of Granada. Since 1999 the development of EO is led by Maarten Keijzer and Marc Schoenauer who redesigned the library from scratch [102]. Today, the latest stable release is EO 1.0.1 which has been published at the beginning of 2007.

The main design goal of EO is to enable evolutionary optimization techniques for any kind of objects which provide a quality value and manipulation operators. To achieve this goal, templates and generic programming are used to a large extent. The type of objects that should be evolved has to be passed as a template parameter to all components of an algorithm. Besides, EO also provides a large spectrum of built-in representations and operators.

EO uses the concept of functors which is inspired by the C++ standard template library (STL). Each basic operation of an algorithm is represented as a parameterless, unary, binary, or \(n\)-ary functor which can also be combined to define more complex behavior. However, as instantiation and combination of many different functors is a cumbersome task, EO additionally contains a basic algorithm class called \textit{EasyEA} that can be used as a generic frame for implementing evolutionary algorithms.

Furthermore, convenient parameter management is integrated into the framework. Algorithm parameters, such as population size or mutation rate, are read from configuration files as strings. Therefore, these parameters can be adapted for each run without recompiling the application. Additionally, there is a persistence mechanism which can be used to store the current state of an algorithm at specific points during its execution (checkpoints). Thus, the execution of an algorithm can be interrupted and restarted at a later point.

In general, EO is a very powerful and mature framework. It provides a flexible class hierarchy and a comprehensive set of representations and operators that can be used straight away. However, due to the excessive use of generic programming and the STL, it is also very complex and quite hard to learn. In order to adapt the core algorithm model to implement new algorithmic ideas, a profound knowledge of the framework is indispensible.

Some other projects should be mentioned as well that are related to EO and follow interesting ideas:

First, EASEA\(^5\) [45] is a high-level programming language to specify evolutionary algorithms. Its code contains domain-specific constructs as well as blocks of code in C/C++ style. Different code generators can be used to transfer EASEA code into source code for specific EA libraries (e.g., EO or GAlib). Therefore, EASEA offers a library-independent way for describing EAs.

\(^4\)http://eodev.sourceforge.net
\(^5\)http://sourceforge.net/projects/easea
Second, GUIDE\textsuperscript{6} \cite{guide} implements a graphical user interface for defining EAs. It is built on top of EASEA and enables users with few programming skills to build an EA, generate the corresponding EASEA code, and transform it into compilable source code.

Third, ParadisEO\textsuperscript{7} \cite{paradiseo1, paradiseo2} is a framework built on top of EO that aims at trajectory-based and population-based metaheuristics for single-objective as well as multi-objective optimization. As one of its main features, ParadisEO uses MPI for parallel and distributed execution of metaheuristics in a cluster or grid environment.

### 4.1.4 Open BEAGLE

Similarly to EO\textsuperscript{8}, also the Open BEAGLE project aims at implementing a C++ framework for evolutionary computation. It is mainly driven by Christian Gagné and Marc Parizeau \cite{beagle1, beagle2} at the Computer Vision and Systems Laboratory of Laval University in Canada.

Open BEAGLE is also based on the STL. Additionally, it uses a class library called Portable Agile C++ Classes (PACC), mainly developed by Marc Parizeau. PACC provides a comprehensive set of utility classes for XML parsing, matrix and vector operations, network communication, threading, or generation of SVG graphics. As Figure 4.2 shows, the architecture of Open BEAGLE is divided into three layers.

![Figure 4.2: Architecture of the Open BEAGLE framework (adapted from [76])](image)

At the lowest level, the object-oriented foundations contain utility classes not related to evolutionary computation. Their main purpose is to provide a convenient infrastructure for application development and to overcome some of the drawbacks of C++. Its design is inspired by other state-of-the-art application development platforms such as Java\textsuperscript{TM} or the Microsoft\textsuperscript{®} .NET environment.

\textsuperscript{6}http://guide.gforge.inria.fr
\textsuperscript{7}http://paradiseo.gforge.inria.fr
\textsuperscript{8}http://beagle.gel.ulaval.ca
In this layer an abstract class called `Object` is implemented which serves as a root class for all other Open BEAGLE classes and contains abstract method declarations for persisting objects as XML files, object comparison, and cloning. Furthermore, automated memory management and exception handling are included.

In the next layer, a generic framework for evolutionary computation is implemented. It contains a generic hierarchical data model (for representing populations, sub-populations, individuals, and genotypes), the evolution system (which is responsible for state and parameter management, logging, and pseudo-random number generation), and evolvers. Especially evolvers are a central concept in Open BEAGLE; they represent a generic and flexible algorithm model and contain several operators that are applied on a sub-population iteratively. Furthermore, a special kind of operators, called breeder operators, is available to allow processing of sub-populations on the level of single solutions.

On top of the generic EC framework, specialized frameworks for different flavors of EAs are realized as independent modules. In these modules, specific representations and operators are provided. Additionally, the co-evolution framework implements mechanisms for simultaneous evolution of many species of individuals.

One of the most interesting features of Open BEAGLE is the use of XML configuration files. These configuration files contain not only parameter values of an algorithm but also define which operators should be applied in the evolver. By this means, the structure of algorithms can be changed by manipulating configuration files instead of recompiling the application. Therefore, Open BEAGLE has a far more dynamic nature than other frameworks such as EO or HotFrame.

Finally, the development of Open BEAGLE has also led to two other spin-off projects. On the one hand, Distributed BEAGLE\(^9\) [77, 64] implements a master-slave architecture for parallel and distributed algorithm execution. On the other hand, BEAGLE Visualizer provides a web interface for creating pretty printed and human readable reports from Open BEAGLE output files. It is the first step towards a graphical user interface; however, an interactive GUI that allows dynamic configuration of algorithms and on-the-fly analysis during execution is still missing.

### 4.1.5 ECJ

ECJ\(^{10}\) is a Java\(^{TM}\)-based evolutionary computation research system and is developed by Sean Luke and his colleagues at the Evolutionary Computation Laboratory of George Mason University in Virginia, USA. It is probably the most popular public EC system coded in Java\(^{TM}\) and reached a high degree of maturity.

---

\(^9\)http://beagle.gel.ulaval.ca/distributed  
\(^{10}\)http://www.cs.gmu.edu/~eclab/projects/ecj
since its birth in 1997.

One of the most interesting aspects of ECJ is that it is implemented in Java™ and not in C++. Therefore, ECJ can use a powerful platform and a broad spectrum of existing APIs and third party components (e.g., for charting, network communication, threading, or report generation). However, this aspect is very controversially discussed in the heuristic optimization community. On the one hand, a feature-rich and state-of-the-art application development platform enables fast and efficient programming, easy integration of external modules, and more comfort for developers. On the other hand, the price for these advantages has to be paid in terms of memory and execution time, as a bytecode-based platform requires more resources.

The basic design of ECJ is quite similar to Open BEAGLE (both systems obviously influenced each other in the last years). It provides a generic and flexible algorithm model based on operators. Similarly to Open BEAGLE, ECJ also uses configuration files to specify parameter values as well as the structure of algorithms. A comprehensive set of operators and solution encodings for many different kinds of EAs is already implemented in ECJ and can be used out of the box.

ECJ also provides mechanisms for distributing algorithms in a network using a master-slave model for parallel fitness evaluation or execution of coarse-grained parallel genetic algorithms. Furthermore, it has a basic GUI for loading and manipulating configuration files, executing algorithms, and visualizing results.

4.1.6 EasyLocal++

The EasyLocal++ framework was developed by Andrea Schaerf and Luca Di Gaspero at the University of Udine [54, 55]. It is an object-oriented C++ framework for developing local search applications.

As shown in Figure 4.3, the core of EasyLocal++ consists of several classes which are hierarchically structured into four layers. At the lowest level basic data classes encapsulate all problem-specific data structures which are the problem representation (Input), solutions (Output), states of the search space (State), and attributes of solution transitions (Moves). The next level provides local search features called helpers. These include for example different neighborhood structures which can be explored by selecting and executing moves, or the evaluation of an objective function. Based on top of the helpers, runners represent the algorithmic core of the framework and implement concrete local search metaheuristics. Finally, solvers are located at the highest level of abstraction. In contrast to runners, solvers represent local search strategies which might use different simple local search procedures. For example, the token-ring solver iteratively applies several runners, always starting from the best solution found by the previous one.

\footnote{http://tabu.diegm.uniud.it/EasyLocal++}
4.2 Analysis and Comparison

Users can extend EasyLocal++ in two different ways. On the one hand, to incorporate a new optimization problem, problem-specific data classes have to be implemented in the basic data layer. These data classes are passed as template parameters to classes on the higher levels which have to be instantiated with problem-specific types. On the other hand, helpers, runners, and solvers contain virtual functions that act as variation points and can be redefined in inherited classes. Therefore, users can also integrate new neighborhood structures or search strategies easily by inheriting from existing framework classes and overwriting the corresponding methods. In this context, it should be mentioned that only the four basic data structures - Input, Output, State, and Move - are used as template parameters. Compared to other frameworks which rely on generic programming more intensively (e.g., HotFrame or EO), this moderate use of templates makes it a bit easier for inexperienced users to learn, adapt and apply the framework.

EasyLocal++ also supports batch execution of optimization runs. It provides a scripting language called EXPSPEC that can be used to define test scenarios and to execute multiple runs automatically. This is a very helpful feature to compare and analyze different metaheuristics, even though automatic distribution and parallel execution of test runs is not implemented so far.

4.2 Analysis and Comparison

As a summary of this chapter, Table 4.1 shows a comparison of the heuristic optimization frameworks described in the previous sections with respect to the requirements defined in Chapter 3. A bullet (●) indicates that a requirement is fulfilled, a circle (○) marks requirements which are somehow addressed but other frameworks provide a better or more comprehensive solution, and a dot (·) shows requirements which are not considered in a framework.

All of the discussed frameworks have reached a high degree of maturity,
Table 4.1: Comparison of existing heuristic optimization software systems

<table>
<thead>
<tr>
<th>Feature</th>
<th>Templar</th>
<th>HotFrame</th>
<th>EO</th>
<th>Open BEAGLE</th>
<th>ECJ</th>
<th>EasyLocal++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automation</td>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Customizable Output</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Generic Algorithm Model</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Generic Operators</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Generic Objective Functions</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Generic Solution Representations</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Graphical User Interface</td>
<td>◦</td>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Integration</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Learning Effort</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Parallelism</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Parameter Management</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Performance</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Predefined Algorithms and Problems</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
<tr>
<td>Replicability and Persistence</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>◦</td>
<td>•</td>
</tr>
</tbody>
</table>

adopted interesting and promising concepts, and provide useful features. How- ever, in each of them focus is put on some specific aspects. In the following paragraphs, the main highlights and drawbacks of each framework are summarized.

Tempal provides a scripting language for automatic batch execution of algo- rithm runs. Furthermore, it concentrates on parallel and distributed execution of algorithms which enables the development and efficient execution of hybrid or cooperative search strategies. Additionally, a GUI is provided to configure algo- rithms as well as to visualize the progress of a run. Although, Templar’s algorithm model is very generic and can represent any heuristic optimization paradigm, it is rather clumsy. There are no additional layers in between coarse-grained algo- rithm or problem classes and fine-grained abilities. Therefore, compared to other frameworks it provides less structure to guide algorithm developers.

HotFrame puts a strong emphasis on performance and runtime efficiency. Generic programming and static type variation are used to assemble algorithms at compilation time in order to reduce virtual method calls. As a consequence, the produced algorithms are of quite static nature and can hardly be manipulated at runtime. This leads to disadvantages regarding flexible algorithm configura- tion, management of parameters, and integration into other applications. Its fine-grained, generic, and comprehensive algorithm model is very complex and imposes a severe learning effort on algorithm developers. Furthermore, paral-
Similarly to HotFrame, EO also uses generic programming and static type variation excessively to increase performance which yields akin difficulties. Its algorithm model is dedicated to EAs, even though its operator concept is very flexible and might be generalized to fit for any kind of heuristic optimization algorithm. Additionally, it contains a data model to abstract the notion of solution quality and uses configuration files to store parameters of an algorithm. Several spin-off projects provide extensions for GUIs and parallelism; however, these are not integrated into the framework. This is why they contain some design flaws and work-arounds which could have been avoided, if the framework had been designed for these purposes right from the start.

The algorithm model of Open BEAGLE is also limited to the domain of evolutionary computation. As in the case of EO and ECJ, Open BEAGLE provides generic operators that are combined to complex algorithms. In contrast to EO, assembling of algorithms is not done at compile-time using templates, but at runtime by building complex object graphs. This reduces performance a little bit, but on the other side it enables dynamic configuration of algorithms. The parameter values as well as the structure of an algorithm are specified in an XML-based configuration file. A GUI and parallelism are not considered in the core framework, though a master-slave architecture for distributed and parallel execution of algorithms is developed in a spin-off project called Distributed BEAGLE.

The design of ECJ is very similar to Open BEAGLE. As ECJ is developed in Java™, it can utilize a comprehensive runtime environment and also provides more comfort and less learning effort for developers. Furthermore, it enables an easy and clean integration into other Java™-based applications. However, a bytecode-based platform has drawbacks concerning memory and runtime consumption, although these drawbacks should become less severe due to the development and optimization of JIT compilers. ECJ also includes a basic GUI for algorithm configuration and analysis. Additionally, support for parallel and distributed execution is embedded into the core framework.

EasyLocal++ enables batch execution of multiple tests by its scripting language ExpSpec. It has a clearly structured and intuitive architecture which significantly reduces learning effort. However, its algorithm model is dedicated to local search strategies. As a consequence, the integration of population-based metaheuristics is rather difficult; furthermore, it lacks a GUI and support for parallelism.

Taking a look at the horizontal and vertical rows of Table 4.1, it is obvious that all requirements are fulfilled in at least some of the frameworks. Nevertheless, none of the systems described above is able to satisfy all of them. This indicates that time is ripe for consolidation. The lessons learned in the development of the different frameworks as well as their beneficial features should be incorporated
into new heuristic optimization software environments. This is - of course - the main goal of the work described in this thesis.
Chapter 5

HeuristicLab Architecture

As pointed out in the previous chapter, none of the existing heuristic optimization software systems fulfill all requirements identified in Chapter 3 to a fully satisfactory extent. Therefore, the author has worked on the development of an advanced generic and flexible environment for heuristic optimization called HeuristicLab. In this process several of the systems described in Chapter 4 supplied constructive ideas and a good basis for further considerations. HeuristicLab has continuously evolved in the last seven years and three versions have been developed until now which are referred to as HeuristicLab 1.1, HeuristicLab 2.0, and HeuristicLab 3.0. In the following, the previous versions are briefly covered and especially the newest version, HeuristicLab 3.0, is presented in detail. Furthermore, all three versions are analyzed and compared with other heuristic optimization software systems regarding the requirements discussed in Chapter 3. Finally, several research projects and applications are outlined at the end of this chapter to give some examples how HeuristicLab has been used so far.

5.1 HeuristicLab 1.1

The development of HeuristicLab 1.1\(^1\) [168, 172] started in 2002 as a programming project at the Johannes Kepler University Linz, Austria. The main goal of the project was to develop a generic, extensible and paradigm-independent environment for heuristic optimization that can be used by researchers in scientific and industrial projects to develop and compare new optimization algorithms and by students in lectures and exercises.

Microsoft\(^\text{®}.\) .NET and the C# programming language were chosen as the development platform for HeuristicLab 1.1. Reasons for this decision were that HeuristicLab 1.1 had a strong focus on a graphical user interface (GUI) right from the start to enhance usability and to provide a shallow learning curve especially for students. Consequently, a powerful GUI framework was required that is well

\(^1\)http://www.heuristiclab.com
integrated into the runtime environment and provides an authentic look and feel of applications. Concerning these aspects, back in 2002 the Microsoft® .NET platform provided a more promising approach than other alternatives such as Java™ or C++\(^2\). Other aspects as for example platform independence were of minor importance as the author was always focused on the Windows® operating system.

Similarly to some other heuristic optimization frameworks such as the Templar framework described in Chapter 4, the main idea of the HeuristicLab 1.1 architecture is to provide a clear separation of problem-specific and problem-independent parts. On the one hand, a user should be able to develop a new heuristic optimization algorithm and to test and compare it with several existing optimization (benchmark) problems. On the other hand, a new problem should be easy to integrate and to solve with existing algorithms. By this means, this concept leads to a significant level of code reuse, as heuristic optimization algorithms can be used without any modification to solve new optimization problem and vice versa.

In order to realize this concept, HeuristicLab 1.1 offers two abstract base classes called Algorithm and Problem from which every new extension, either optimization algorithm or problem, has to be inherited. Furthermore, another base class Solution represents data entities that are created and evaluated by problems and manipulated by algorithms. Any specific solution encoding has to be inherited from that class. On top of these basic framework classes, the HeuristicLab 1.1 GUI layer is located. It provides two more base classes for visualizing algorithms (AlgorithmForm) and problems (ProblemForm). These classes represent forms in terms of the Microsoft® .NET WinForms framework and are presented in the GUI. As each algorithm is executed in its own thread, transport objects called Results are used to inform an AlgorithmForm about the progress of its algorithm by propagating values such as the actual number of evaluated solutions or the current solution quality. To support easy integration of new algorithms and problems, HeuristicLab 1.1 additionally is equipped with a plugin mechanism enabling users to add custom extensions without knowing or even having access to the whole source code. As a summary of the HeuristicLab 1.1 core architecture, Figure 5.1 schematically shows all these basic classes and their interactions.

As algorithms and problems are loosely coupled to be able to exchange both parts at will, communication between algorithms and problems is realized by delegates (i.e., types representing methods). An algorithm defines method interfaces it expects in order to be able to do its work (e.g., evaluation operators, manipula-

\(^2\)It has to be mentioned that today a very powerful and flexible GUI framework and a comfortable development environment is also available on the Java™ side in form of the Eclipse™ IDE and the Eclipse Rich Client Platform. So in fact concerning GUI support the choice of an appropriate runtime environment would not be that easy today, as both solutions, Java™ and Microsoft® .NET, are well developed and reached a high degree of maturity.
5.1. HEURISTICLAB 1.1

Figure 5.1: Interaction of HeuristicLab 1.1 classes

On the other side, a problem provides implementations of these interfaces. If implementations of all required delegates are available, a problem can be solved by an algorithm. Linking between delegates (interfaces) and delegate implementations (operators) is done dynamically at runtime using code attributes and reflection. The whole application does not have to be compiled again when integrating new algorithms or problems.

Beside this basic object model for representing arbitrary heuristic optimization algorithms and problems, HeuristicLab 1.1 also includes another front-end for batch execution of algorithms called TestBench. In the HeuristicLab 1.1 GUI multiple algorithm and problem configurations can be saved in a comma-separated text format (CSV) and can be executed in batch mode using the TestBench. This feature is especially useful for large scale experiments to compare different heuristic optimization algorithms. Furthermore, another sub-project called HeuristicLab Grid [169, 170] offers distributed and parallel batch execution of algorithm runs in a client-server architecture.

In the past, HeuristicLab 1.1 has been intensively and successfully used by the research group of Michael Affenzeller and the author in many research projects as well as in several heuristic optimization lectures. A broad spectrum of more than 40 plugins providing different heuristic optimization algorithms and problems has been developed. For example, various genetic algorithm variants, genetic programming, evolution strategies, simulated annealing, particle swarm optimization, tabu search, and scatter search are available. Furthermore, several heuristic optimization (benchmark) problems - for example, the traveling salesman problem, vehicle routing, $n$-dimensional real-valued test functions, the Boolean satisfiability problem, scheduling problems, or symbolic regression - are provided as plugins. An exemplary screenshot of the HeuristicLab 1.1 GUI is shown in Figure 5.2.
5.2 HeuristicLab 2.0

Although HeuristicLab 1.1 was extensively used in several research projects and was continuously extended with new algorithm and problem plugins, the author and the other members of the research group identified a few drawbacks of HeuristicLab 1.1 during its development and productive use. The most important ones of these issues are listed and discussed in the following:

- **Monolithic Plugins**
  As HeuristicLab 1.1 provides a very high level of abstraction by reducing heuristic optimization to two main base classes (Algorithm and Problem), no specific APIs for particular optimization paradigms are available. For example, if a new variant of genetic algorithms is developed that differs from an existing algorithm in just some small aspects (for example using two different selection schemes for selecting solutions for reproduction [173]), the whole algorithm has to be provided in a separate and independent plugin. As more and more algorithms were added to HeuristicLab 1.1, this situation led to a severe amount of code duplication and to a significant reduction of maintainability.
5.2. **HEURISTICLAB 2.0**

- **Strict Separation of Algorithms and Problems**
  HeuristicLab 1.1 requires strict separation of algorithms and problems and loose coupling between these two parts based on delegates. This approach makes it rather complicated to integrate heuristic optimization algorithms such as tabu search or hybrid algorithms that contain problem-specific concepts. A tighter interaction between algorithms and problems should be possible on demand.

- **Interruption, Saving and Restoring of Algorithms**
  The internal state of an algorithm during its execution cannot be persisted in HeuristicLab 1.1. As a consequence, it is not possible to interrupt, save and restore an algorithm during its execution. For example, if an algorithm has to be stopped because computation resources are temporarily required for some other task, the whole run has to be aborted and cannot be continued later on. As algorithm runs might take quite a long time in several application scenarios of heuristic optimization, this behavior turned out to be quite uncomfortable for users.

- **Comprehensive Programming Skills**
  Due to the high level of abstraction, comprehensive programming skills are required especially for developing new heuristic optimization algorithms; each new algorithm plugin has to be developed from scratch. There is only little support for developers in terms of more specialized APIs supporting particular heuristic optimization paradigms. Furthermore, also the HeuristicLab API has to be known to a large extent. It is not possible to assemble algorithms in the GUI dynamically at runtime by defining a sequence of operations without having to use a development environment for compiling a new plugin.

As a result of these insights, the author decided in 2005 to redesign and extend the core architecture and the object model of HeuristicLab. Based on the Microsoft® .NET 2.0 platform a prototype was developed (HeuristicLab 2.0) that realized a more fine-grained way of representing algorithms.

In HeuristicLab 2.0 so-called workbenches represent the basic entities of each heuristic optimization experiment. A workbench contains three main items in form of an algorithm, a problem and a solution representation.

In contrast to HeuristicLab 1.1, solution representations are not directly integrated into problems anymore, but are treated as independent objects. As a consequence, manipulation concepts do not have to be provided by each problem but can be shared, if solutions to a problem are represented in the same way.

Algorithms are no longer represented as single classes inherited from an abstract base class. Instead, each algorithm consists of several operators. Each operator works on either one or more solutions and represents a basic operation (e.g., manipulating or evaluating a solution, selecting solutions out of a solution...
set, or reuniting different solution sets). Furthermore, an operator may contain and execute other operators, leading to a hierarchical tree structure. By this means, the development of high level operators is possible which represent more complex operations and can be specialized with more specific operators on demand. For example, a solution processing operator can be defined that iterates over a solution set and executes an operator on all contained solutions (for example an evaluation operator). Another example is a mutation operator that executes a manipulation operator on a solution with some predefined probability. This concept leads to a fine-grained representation of algorithms and to better code reuse. Additionally, the strict separation between problem-specific and problem-independent parts is softened, as operators are able to access problem-specific information via the workbench.

Figure 5.3: Screenshot of HeuristicLab 2.0

The main idea of this enhanced algorithm model is to shift algorithm engineering from the developer to the user level. Complex heuristic optimization algorithms can be built by combining different operators in the GUI of HeuristicLab 2.0 (see for example Figure 5.3). This aspect is especially important to support users who are not so well familiar with programming and software development (as for example many practitioners who are experts in some problem
domain but not in software engineering). Furthermore, it also enables rapid prototyping and evaluation of new algorithmic concepts, as an algorithm does not have to be implemented and compiled as a plugin.

In order to enable interrupting, saving and restoring of algorithms at any time during execution, a 2-phase commit strategy has been realized. If an algorithm is stopped during its execution, the actual program flow is located somewhere in the depths of the operator tree. As operators also may have local status variables, it has to be assured that the whole operator tree is left in a consistent state, so that execution can be continued, if the algorithm is restored or restarted again. The actual iteration that has not been finished yet has to be rolled back. Consequently, each operator has to keep a local copy of its status variables to be able to restore the last save state of the last completed (and committed) iteration. At the end of an iteration (i.e., a single execution of the top level algorithm of a workbench), a commit is propagated through the whole operator tree indicating that the actual state can be taken for granted.

All these concepts described above were implemented in the HeuristicLab 2.0 prototype and were evaluated in several research projects showing the advantages of the new architecture in terms of code reuse, algorithm development time and flexibility. An overview of these applications can be found at the end of this chapter.

5.3 HeuristicLab 3.0

Even though HeuristicLab 2.0 contains fundamental improvements compared to version 1.1 and has been extensively used in research projects as well as in lectures on heuristic optimization, major problems regarding its operator model emerged: For example, due to the local status variables stored in operators and due to the nested execution of operators, the implementation of parallel algorithms turned out to be difficult. Moreover, the 2-phase commit strategy caused a severe overhead concerning the development of new operators and the required memory.

Therefore, it seemed reasonable to develop a new version called HeuristicLab 3.0 (HL3) completely from scratch to overcome limitations due to architectural and design decisions of previous versions. Although this decision led to starting over the design and development process again, it offered the essential possibility to build a thoroughly consistent heuristic optimization software system by picking up ideas of preliminary projects, integrating novel concepts, and always keeping learned lessons in mind. In the following, the architecture of HL3 and its object and algorithm model are presented in detail (cf. [174, 175]).
5.3.1 Plugin Infrastructure

Introduction to the Concept of Plugins

In software engineering modularity is considered as an import software quality aspect [123, 99, 48]. Application developers have to deal with unclear and changing requirements, new feature requests and short development times. Hence, building software systems structured into clearly separated modules that communicate using well-defined interfaces is an essential success factor: On the one hand, modular software systems encourage code reuse, as generic modules can be used in other projects; on the other hand, modularity simplifies reacting on change and extension requests, as modifications are just affecting a few parts and in most cases it is not necessary to change (or even know) an application as a whole.

Taking a look at the evolution of software development methodologies, the trend towards modular application development is obvious (see [89, 149] for a comprehensive overview): In the middle of the last century applications were developed as monolithic programs, then procedural and later on object-oriented programming was introduced, and today component-oriented or aspect-oriented development of complex software systems is state-of-the-art. In each new software development paradigm additional concepts have been introduced to enhance clear structuring and separation of concerns (e.g., procedures, objects, components, and aspects). Additionally, communication between modules has been specified on a more and more abstract level (e.g., procedure signatures, interfaces, and pointcut descriptions) to reduce tightness of coupling and to encourage exchangeability. However, all these approaches of decomposition are used at the level of source code. After compilation, the modules of an application are still often linked to a few monolithic executables or libraries. Thus, when a module has to be modified, changes in source code are clearly separated; nevertheless, the whole application has to be re-compiled or at least re-linked. As a consequence, it is hard to extend software systems with new modules that are for example developed by third parties.

To overcome this problem, the concept of plugins or addins was developed. A plugin is a software module that adds new functionality to an existing application after the application has been compiled and deployed. In other words, plugins enable modularity not only at the level of source code but also at the level of object or byte code, as plugins can be developed, compiled, and deployed independently of the main application.

In the middle of the 1970s the EDT$^{\text{TM}}$ text editor was one of the first applications that came up with the concept of plugins. EDT$^{\text{TM}}$ offered the ability to start programs directly out of the editor and to let these programs access its editor buffer [1]. At the end of the 80s and the beginning of the 90s several other software systems provided plugin mechanisms, as flexible and independent
extension of applications turned out to be a very powerful and useful approach. Today, extensibility by plugins can be found in many mainstream applications such as graphics software, e-mail clients, web browsers, development environments, or media players, enabling users to integrate additional functionality, for example filters, codecs, or rendering engines. However, in most of these applications plugins are still treated as a subordinate topic. They are used to integrate very specific functionality at a few designated points; still, the main application is realized as a set of a few monolithic executables and libraries.

The concept of plugins can be used in a more extensive way though. Due to dynamic loading techniques offered in modern application frameworks such as Java™ or Microsoft® .NET, a trend towards software systems can be observed in the last few years that use plugins as main architectural pattern. In these systems the main application is reduced to a basic plugin management infrastructure that provides plugin localization, on demand loading, and object instantiation. All other parts of an application are implemented as plugins. Communication and collaboration of plugins is based on extension points. An extension point can be used in a plugin to provide an interface for other plugins to integrate additional functionality. In other words, plugins can define extension points or provide extensions which leads to a hierarchical application structure. By this approach, very flexible applications can be built, as an application’s functionality is determined just by its plugins. If a comprehensive set of plugins is available, a huge variety of applications can be developed easily by selecting, combining and deploying appropriate plugins.

**Examples of Plugin Frameworks**

The OSGi™ Alliance offers a well-developed and standardized architecture for plugin-based and service-oriented software systems called the OSGi™ Service Platform Specification [133, 131]. The OSGi™ architecture is structured in several layers which are built on top of the Java™ runtime environment and provide mechanisms required for application building (e.g., enhanced class loading, module management, or communication and cooperation between modules). The application itself (also called bundle) can be situated on any layer depending on the required functionality. Figure 5.4 shows the layout of the OSGi™ architecture. Successful implementations of the OSGi™ Service Platform Specification are documented in several publications (for example in [90]) as well as in open source and commercial software projects.

Beside the OSGi™ Service Platform, also the well-known Eclipse™ development platform and in particular the Eclipse™ Rich Client Platform intensively use the concept of plugins [25, 24, 122]. Additionally, other popular applications - for example, the Mozilla® web browser [155] or the NetBeans® environment - follow similar ideas.

However, the idea of plugin-oriented application development is not restricted
to the Java™ platform. A powerful and flexible plugin infrastructure can be realized with any kind of application development environment which offers basic features such as dynamic loading and reflection. Several publications show how to implement plugin-based applications using the Microsoft®.NET common language runtime and underpin the worth of this approach [134, 135, 94, 187]. Furthermore, the Razor project of Mark Belles has to be mentioned which goes beyond scientific analysis and prototypical implementation and focuses on developing a plugin framework suitable for productive use\(^3\).

Benefits of Plugin-Based Heuristic Optimization Software Systems

In Chapter 3 genericity has been identified as an essential quality criterion of heuristic optimization software systems. In order to enable integration of custom optimization problems and algorithms, main parts of the system such as objective functions, operators, or solution encodings have to be exchangeable. A high degree of modularity is required and consequently a plugin-based architecture is reasonable for a heuristic optimization software system [176].

Besides modularity and genericity, a plugin-based approach has some additional benefits in the domain of heuristic optimization: As users can add custom modules after compilation and deployment of the application, no access to the source code of any other plugin or the main application, nor re-compiling or re-linking is required. Therefore, closed source deployment of the base system and its plugins is possible. This aspect is especially important for third parties which are members of the practitioners user group. For example, a software company developing applications for heuristic production planning or logistics optimization develops sophisticated solution representations and manipulation operators. These concepts are their unique selling points and represent business critical knowledge. Thus, selling this software together with the source code usually is not an option. However, if the used heuristic optimization software system

\(^3\)http://www.codeproject.com/KB/architecture/razorpt1.aspx
provides a plugin infrastructure, additional plugins do not have to be distributed open source. The company can sell their plugins using a closed source license model without restricting extensibility of the system.

A plugin-based approach leads to a win-win situation for all involved stakeholders: Software companies can develop and sell new plugins adding functionality to an existing application. Due to these enhancements, the software system becomes more attractive for users, which is a win situation for the software manufacturer of the base system. As the number of potential users and therefore the market is growing, more requirements come up, encouraging the development of even more additional plugins. In Figure 5.5 this economic cycle of plugin-based software systems is shown.

![Figure 5.5: Economic cycle of plugin-based software systems](image)

HeuristicLab Plugin Infrastructure

Motivated by the benefits of plugin-based software systems especially in the case of heuristic optimization, plugins are used in HL3 as architectural paradigm as in the versions HeuristicLab 1.1 and 2.0. In contrast to previous versions, a lightweight plugin concept is implemented in HL3 by keeping the coupling between plugins very simple: Collaboration between plugins is described by interfaces. The plugin management mechanism contains a discovery service that can be used to retrieve all types implementing an interface required by the developer. It takes care of locating all installed plugins, scanning for types, and instantiating objects. As a result, building extensible applications is just as easy as defining appropriate interfaces (contracts) and using the discovery service to retrieve all objects fulfilling a contract. Interfaces that can be used for plugin coupling do not have to be marked by any specific declaration.

Meta-information has to be provided by each plugin to supply the plugin management system with further details. For example, when installing, loading,
updating, or removing a plugin, the plugin infrastructure has to know which files belong to the plugin and which dependencies of other plugins exist. With this information the plugin infrastructure can automatically install other required plugins, or disable and remove dependent plugins, if a base plugin is removed. Hence, it is guaranteed that the system is always in a consistent state. In the HL3 plugin infrastructure all plugin-related data is stored in the source files together with the plugin code. Plugin meta-information is expressed using code annotations instead of separate configuration files, keeping the configuration of the plugin system simple and clean.

Additionally, a special kind of plugin is necessary: Some designated plugins, called application plugins, have to be able to take over the main application flow. Application plugins have to provide a main method and usually they also offer a GUI. Due to these application plugins the HeuristicLab plugin infrastructure leads to a flexible hierarchical system structure. It is possible to have several different front-ends (applications) within a single system. Figure 5.6 shows this structure in a schematic way.

![Layered structure of the HL3 plugin infrastructure](image)

Figure 5.6: Layered structure of the HL3 plugin infrastructure

### 5.3.2 Object Model

Based on the plugin infrastructure, a generic object model is provided by HL3 which is described in detail in this section. Following the paradigm of object-oriented software development, the whole HL3 system is represented as a set of interacting objects. In a logical view these objects are structured in an object hierarchy using the principle of inheritance. Though, this logical structure is not related to the physical structure which is defined by the plugins the application consists of. Therefore, the object hierarchy is spanned over all plugins.

Similarly to the structure of object-oriented frameworks such as the Java™ or Microsoft® .NET runtime environment, a single class represents the root of
the object hierarchy. It has been decided to implement a custom root class and not to use an existing one of the runtime environment, as it is necessary to extend the root class with some additional properties and methods which usually are not available in the root classes of common runtime environments.

Due to the requirement of replicability and persistence, a heuristic optimization software system has to offer functionality to stop an algorithm, save it (in a file, database or any other kind of media), restore it, and continue it at any later point in time. Hence, all objects currently alive in the system have to offer some kind of persistence functionality (also known as serialization). The persistence mechanism has to be able to handle circular object references, so that arbitrarily complex object graphs can be stored without any difficulties. To simplify communication with other systems and to enable integration into an existing IT environment, a generic way of data representation such as XML or relational databases should be used for storing objects. Additionally, all objects should offer cloning functionality which is also determined by the replicability and persistence requirement. Seen from a technical point of view, cloning is very similar to persistence, as cloning an object graph is nothing else than persisting it in memory.

A graphical user interface is the second requirement which has to be considered in the design of the basic object model. The user of a heuristic optimization software system should be able to view and access all objects currently available at any time, for example to inspect current quality values, to view some charts, or to take a look at a custom representation of the best solution found so far. Nevertheless, the representation of data in a graphical user interface is a very time consuming task and therefore critical concerning performance. Consequently, visualizing all objects per default is not an option. The user has to be able to decide which objects should be presented in the GUI, and to open or close views of these objects as needed. Therefore, a strict separation between objects and views is required.

Objects need to offer functionality to create new views, but must not have any references to these views after creation. To realize this functionality, event-driven programming is a suitable approach. If the state of an objects changes and its representation has to be updated, events are used to notify all views. In object-oriented programming this design is known as the observer pattern [78] and is derived from model-view-controller. Model-view-controller (MVC) is an architectural pattern which was described by Trygve Reenskaug for the first time when working together with the Smalltalk group at Xerox PARC in 1979. It suggests the separation into a model (containing the data), views (representing the model) and a controller for handling user interactions (events). By this means, it is assured that the view is decoupled from the model and the control logic and

\[4\text{Additionally, a graphical user interface is also very helpful for reducing the learning effort required when starting to work with the system.}\]
that it can be exchanged easily. A comprehensive description of MVC is given by Krasner and Pope in [111].

In HL3 these concepts are used for object visualization. In addition, a strict decoupling of objects and views simplifies the development of complex views which represent compound objects. A view of a complex object containing several other objects can be composed by adding views of the contained objects. Figure 5.7 outlines this structure in a schematic way and Figure 5.8 shows a screenshot of the HL3 GUI.

![Figure 5.7: Compound views representing complex objects](image)

Finally, as a last aspect of the basic HL3 object model, constraints have to be considered. In real-world optimization scenarios solutions often have to satisfy several constraints to be valid. In many cases basic solution representations such as permutations or vectors of real numbers are too generic and consequently manipulation operators frequently create invalid solutions. As an alternative, specialized solution representations can be developed that take all constraints into account which represent specific aspects of the optimization problem. However, in this case all operations working on solutions (e.g., manipulation, neighborhood, or crossover operators) have to be adapted as well in order to create valid solutions. In addition to customized solution representations and operators, constraints should therefore be handled separately from solution representations. By this means, basic solution representations and operators can be used in combination with operators that realize different ways to deal with invalid solutions such as penalties added to a solution’s quality value or repair strategies.

Constraints do not only have to be considered for solution representations though. Several constraints such as ranges of parameter values or valid combinations of operators also apply to heuristic optimization algorithms. Therefore, it is reasonable to implement constraint checking directly in the root class of all
objects. In HL3 each object can be specialized by adding constraints which the object has to satisfy to be valid. The definition of a constraint is separated from the object itself, resulting in a modular representation and allowing reuse of constraints for different objects. Additionally, as constraints are also represented as objects, they might have constraints as well.

As a summary, the basic properties of each HL3 object can be defined as follows:

- **Persistence**
  Objects can be persisted in order to save and restore themselves and all contained objects.

- **Cloning**
  Objects provide a cloning mechanism to create copies of themselves.

- **Visualization**
  Views can be created for each object to represent it in a graphical user interface.
• **Constraints**

Constraints can be added to objects to specify specific properties an object has to have.

In Figure 5.9 the structure of HL3 objects defined by the basic object model is shown.

![Figure 5.9: HL3 object model](image)

### 5.3.3 Algorithm Model

In the previous section, the HL3 object model has been described. It is the basis for implementing arbitrary objects interacting in HL3. In this section the focus is now on using this object model to represent heuristic optimization algorithms and problems.

As stated in the requirements analysis in Chapter 3, the algorithm model of a heuristic optimization software system has to be very generic. Users have to be able to implement custom solution representations and objective functions and to realize individual optimization algorithms. However, the area of heuristic optimization algorithms and problems is very heterogeneous. As discussed in Chapter 2 many different phenomena (e.g., evolution, hill climbing, foraging, or cooling of matter) were used as a source of inspiration leading to a broad spectrum of algorithms. These algorithms were applied in various problem domains including medicine, finance, engineering, economics, biology, chemistry, and many more.

Furthermore, due to the increase of computational power, many new problem domains are opened up for heuristic algorithms. Today, problems are solved for which using a (meta-)heuristic algorithm was unthinkable a few years ago because of unacceptable execution times. Every year new paradigms of heuristic
optimization are introduced, new hybrid algorithms combining concepts of different optimization techniques are developed, and new optimization problems are solved. Therefore, diversity of algorithms and problems in the area of heuristic optimization is growing steadily and it can be expected that this trend will continue within the next years. Hence, developing an algorithm model capable of representing all these different cases is quite challenging.

For software engineers this reveals an interesting problem: On the one hand, a uniform and generic algorithm model is necessary to be able to represent all different optimization paradigms. Even more, the model has to be flexible enough to realize new algorithms that are not even known today and probably will be invented in the future. On the other hand, heterogeneity in the field of heuristic optimization makes it very difficult to develop such a generic model, as the different concepts and paradigms of heuristic optimization can hardly be unified. Although, some efforts were made in the scientific community to develop common models for subareas of heuristic optimization (e.g., for evolutionary algorithms as described in [53]), still there is no common theoretical model of heuristic optimization algorithms in general.

To solve this challenging task, radical abstraction and meta-modeling are essential success factors. As it is impossible to foresee which kinds of algorithms and problems will be implemented in a heuristic optimization software system, abstraction has to be shifted one level higher. Instead of developing a model from the viewpoint of heuristic optimization, the algorithm model has to be able to represent any kind of algorithm in any domain. By this means, the model turns into an algorithm meta-model that enables users to quickly build customized models that exactly fit to their needs.

When developing such a generic and domain-independent algorithm model, one has to start with taking a closer look at algorithms in general. In literature several definitions for an algorithm can be found. One of the popular ones is the following given by Knuth in [107]:

An algorithm is a finite, definite, effective procedure, with some output.

Additionally, Knuth states that an algorithm has to have the following five properties:

- **Finiteness**
  An algorithm must always terminate after a finite number of steps.

- **Definiteness**
  Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
• **Input**
  An algorithm has none, one or more input quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects.

• **Output**
  An algorithm has one or more output quantities which have a specified relation to the inputs. These outputs are computed during the execution of the algorithm.

• **Effectiveness**
  All of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a human using paper and pencil.

Interpreting this definition from an object-oriented point of view, an algorithm is an interaction of three parts: It is a sequence of steps (operations, instructions, statements) describing manipulation of data (input and output variables) that is executed by a machine (or human). Therefore, these three components (data, operators, and execution) are the core of the HL3 algorithm model and are described in detail in the following.

**Data Model**

Data values are represented as objects according to the HL3 object model. Therefore, each value can be persisted, viewed, or restricted with constraints. Standard data types such as integers, doubles, strings, or arrays that do not offer these properties have to be wrapped in HL3 objects.

In imperative programming, variables are used to represent data values that are manipulated in an algorithm. Variables link a data value with a (human readable) name and (optionally) a data type, so that they can be referenced in statements. Adapting this concept in the HL3 data model, a variable object is a key-value-pair storing a name (represented as a sequence of characters) and a value (an arbitrary HL3 object). The data type of a variable is not fixed explicitly but is given by the type of the contained value.

In a typical heuristic optimization algorithm a lot of different data values and therefore variables are used. Hence, in addition to data values and variables, another kind of objects called scopes is required to store an arbitrary number of variables. To access a variable in a scope, the variable name is used as an identifier. Thus, a variable name has to be unique in each scope the variable is contained.

Hierarchical structures are very common in heuristic optimization algorithms. For example, in an evolutionary algorithm an environment contains several populations, each population contains individuals (solutions) and these solutions may
consist of different solution parts. Moreover, hierarchical structures are not only suitable for heuristic optimization. In many algorithms complex data structures (also called compound data types) are assembled from simple ones. As a result, it is reasonable to combine scopes in a hierarchical way to represent different layers of abstraction. Each scope may contain any number of sub-scopes which leads to an \( n \)-ary tree structure. For example, a scope representing a set of solutions (population) contains other scopes that represent a single solution each.

As operators are applied on scopes to access and manipulate data values or sub-scopes (as described in the next section), the hierarchical structure of scopes also has another benefit: The interface of operators does not have to be changed according to the layer an operator is applied on. For example, selection operators and manipulation operators are both applied on a single scope. However, in the case of selection this scope represents a set of solutions and in the case of manipulation it stands for a single solution. Therefore, users are able to create as many abstraction layers as required, but existing operators do not have to be modified. Especially in the case of parallel algorithms, this aspect is very helpful and will be discussed in detail later on.

The hierarchy of scopes is also taken into account when accessing variables. If a variable is not found in a scope, looking for the variable is dedicated to the parent scope of the current scope. The lookup is continued as long as the variable is not found and as long as there is another parent scope left (i.e., until the root scope is reached). Each variable is therefore “visible” in all sub-scopes of the scope that contains it. However, if another variable with the same name is added to one of the sub-scopes, the original variable is hidden due to the lookup procedure\(^5\).

**Operator Model**

According to the definition of an algorithm, steps are the next part of algorithms that have to be considered. Each algorithm is a sequence of clearly defined, unambiguous and executable instructions. In the HL3 algorithm model these atomic building blocks of algorithms are called operators and are represented as objects. In analogy to imperative programming languages, operators can be seen as statements that represent instructions or procedure calls.

Operators fulfill two major tasks: On the one hand, they are applied on scopes to access and manipulate variables and sub-scopes. On the other hand, operators define which operators are executed next.

Regarding the manipulation of variables, the approach used in the HL3 operator model is similar to formal and actual parameters of procedures. Each operator defines which variables it expects and how these variables are used (i.e., if they are added to a scope, read, written, or deleted from a scope). Inside an

\(^5\)This behavior is very similar to blocks in procedural programming languages. That is the reason why the name “scope” has been chosen.
operator formal variable names are used to reference variables. However, formal
names do not correspond to real variable names, as they are not known at the
development time of an operator.

After adding an operator to an algorithm at run time, a mapping between
formal variable names and actual variable names has to be defined by the user.
When an operator is executed and accesses a variable, the variable lookup mech-
anism automatically translates the formal name into the actual variable name
which is then used to retrieve the variable from the scope the operator is applied
on.

In order to define variables and to store formal and actual names, each op-
erator has to provide meta-information about its variables. In the HL3 operator
model this information is represented by variable information objects which con-
tain static information defined at development time. They describe a variable
by giving its formal name, data type, kind, and a human-readable description.
Additionally, users can access variable information at run time to set dynamic
information such as the actual variable name. In analogy to procedure calls,
variable information objects can therefore be interpreted as parameter lists of
operators.

By this means, operators are able to encapsulate functionality in an abstract
way: For example, a simple increment operator contains a single variable infor-
mation object indicating that the operator manipulates an integer variable. A
formal name is defined for that variable and inside the operator this name is
used to implement the increment operation. After instantiating an increment
operator and adding it to an algorithm at run time, the user defines the concrete
name of the variable which should be incremented. This actual name is also
set in the variable information. However, the original code of the operator (i.e.,
the increment operation) does not have to be modified, regardless which variable
is actually incremented. The implementation of the operator is decoupled from
concrete variables. Therefore, the operator can be reused easily to increment any
integer variable.

Regarding their second major purpose, operators define the execution flow of
an algorithm. Each operator may contain references to other operators which
represent the static structure of an algorithm. When an operator is executed, it
can decide which of its sub-operators have to be executed next. Hence, complex
algorithms are built by combining operators.

Control operators can be implemented that do not manipulate data, but dy-
namically define the execution flow. For example, executing a sequence of op-
erators can be realized by an operator that returns all its sub-operators to be
executed next. Another example is a branch operator that chooses one of its sub-
operators depending on the value of a variable in the scope the branch operator
is applied on (cf. if- or switch-statements).

In contrast to scopes, operators are not combined hierarchically (although
contained operators are called sub-operators), but represent a graph. An operator
used in an upper level of an algorithm can be added as a sub-operator in a lower level again. Thus, operator references may contain cycles. In combination with sequences and branches, these cycles can be used to build loops; for example, a do-while-loop is realized as a sequence operator which contains a branch operator as its last sub-operator. This branch operator has a reference back to the sequence operator that defines the branch to be executed, if the value of the checked variable is true (see Chapter 6 for a detailed description of control operators).

As described above, classical concepts of programming such as sequences, branches, loops, or recursion can be represented in the operator model. Therefore, the HL3 algorithm model is capable of representing any algorithm that can be described in imperative programming languages.

Execution Model

The execution of algorithms is the last aspect which has to be defined in the HL3 algorithm model. Algorithms are represented as operator graphs and executed step-by-step on virtual machines called engines. In each iteration an engine performs an operation, i.e., it applies an operator to a scope. Therefore, an operation is a tuple containing an operator and a scope. Before executing an algorithm, each engine is initialized with a single operation containing the initial operator and an empty global scope.

As the execution flow of an algorithm is dynamically defined by its operators, each operator may return one or more operations that have to be executed next. Consequently, engines have to keep track of all operations that wait for execution. These pending operations are kept in a stack. In each iteration, an engines pops the next operation from the top of its stack, executes the operator on the scope, and pushes all returned successor operations back on the stack again in reversed order\(^6\). By this means, engines perform a depth-first expansion of operators. Listing 5.1 states the main loop of engines in pseudo-code.

```
1 clear global scope  // remove all variables and sub-scopes
2 clear operations stack
3 push initial operation // initial operator and global scope
4
5 WHILE NOT operations stack is empty DO BEGIN
6 pop next operation
7 apply operator on scope
8 push successor operations // in reverse order
9 END WHILE
```

Listing 5.1: Main loop of HL3 engines

\(^6\)Reversing the order of operations is necessary to preserve the execution sequence, as a stack is a last-in-first-out queue.
Finally, as a summary of the HL3 algorithm model and its three sub-models (data model, operator model, execution model), Figure 5.10 shows the structure of all components.

![Figure 5.10: HL3 algorithm model](image)

### 5.3.4 Parallelism

As already pointed out in Chapter 3, short execution time is very important in many real-world applications of heuristic optimization. In order to increase performance, concepts of parallel and distributed computing are used frequently to utilize multiple cores or even computers and to distribute the work load. In the area of parallel heuristic optimization several models of parallelization have been developed which reflect different strategies. In general, there are two main approaches:

On the one hand, calculating the quality of solutions can be parallelized. In many optimization problems, evaluating a solution requires much more runtime than the execution of solution manipulation operators. As an example consider heuristic optimization for production planning or logistics: In that case, evaluating a solution is done by building a schedule of all jobs or vehicles (for example by using the Giffler-Thompson scheduling algorithm [79]), whereas a solution manipulation operator just needs to change permutations. As another example,

---

This depends on the solution encoding, but variations of permutation-based encoding are frequently used for combinatorial optimization problems and have been successfully applied in
5.3. HEURISTICLAB 3.0

heuristic optimization of data representation or simulation models can be mentioned. In these applications, evaluating a solution means executing the whole model (i.e., performing the simulation or checking the quality of the model for all training data). In both examples (and there are many more), evaluating a single solution is independent of all other solutions. Therefore, quality calculation can be easily executed in parallel which is called *global parallelization* [19]. However, the heuristic algorithm performing the optimization still is a sequential one.

On the other hand, heuristic optimization algorithms can also be parallelized directly [19, 43]. By splitting solution candidates into distinct sets, an algorithm can work on these sets independently and in parallel. For example, parallel multi-start heuristics are simple representatives of that idea. In a parallel multi-start heuristic algorithm, multiple optimization runs are executed with different initial solutions in order to achieve larger coverage of the solution space. Nevertheless, no information is exchanged between these runs until all of them are finished and the best solution is determined. In more complex algorithms (e.g., coarse- or fine-grained parallel genetic algorithms), information is additionally exchanged from time to time to keep the search process alive and to support diversification of the search. Hence, population-based heuristic optimization algorithms are especially well suited for this kind of parallelization.

Consequently, a heuristic optimization software system should support parallelization for all these kinds of algorithms. However, developing a generic model for parallel algorithms is a tricky task and still an active field of research, especially encouraged by the boom of multi-core CPUs. Throughout the software engineering community different concepts of parallelization are discussed and developed, but none of them have achieved a high level of acceptance so far. In the opinion of the author, reasons for this situation are the following ones: First, not enough time is spent on parallel concepts in software engineering education. As a consequence, software developers are often very well skilled in sequential programming, but it is hard for them to solve problems using parallel paradigms. Second, existing frameworks for developing parallel algorithms, such as MPI, OpenMP™, or mere threading, are quite difficult to use and require a large learning effort. Thus, software based on these frameworks is often quite error-prone. Finally, most state-of-the-art software development environments lack of a comfortable integration of parallelization concepts.

To overcome these difficulties in a heuristic optimization software system, parallelization has to be considered in the algorithm model. It has to provide sequential as well as parallel blocks, so that all different kinds of parallel algorithms can be represented. Furthermore, the definition of parallel parts in an algorithm has to be abstracted from parallel execution. By this means, users of a heuristic optimization software system can focus on algorithm development and do not have to rack their brains on how parallelism is actually implemented.

---

many applications.
In the HL3 algorithm model presented in the previous section, data, operators and algorithm execution have been separated strictly. Consequently, parallelism can be integrated by grouping operations into sets that might be executed in parallel. As an operator may return several operations that should be executed next, it can mark the successor operations as a parallel group. These operations are then considered to be independent of each other and the engine is able to decide which kind of parallel execution should be used. How this parallelization is actually done just depends on the engine and is not defined by the algorithm.

For example, one engine can be developed that does not consider parallelism at all and executes an algorithm strictly in a sequential way. Such an engine is especially helpful for testing algorithms before they are really executed in parallel. Another engine might use multiple threads to execute operations of a parallel group in order to utilize multi-core CPUs; an even more sophisticated engine might distribute parallel operations to several nodes in a network, either by using a client-server-based or a peer-to-peer based approach. Furthermore, meta-engines can be provided that use other engines to realize hybrid parallelization by distributing operations to different cluster nodes and using shared-memory parallelization on each node. By this means, the user can specify the parallelization concept used for executing parallel algorithms by choosing an appropriate engine. The definition of an algorithm is not influenced by that decision.

Based on this parallelization concept of the HL3 algorithm model, special control operators can be developed for parallel processing. For example, an operator similar to the sequence operator can be used for parallel execution of operators. In contrast to the sequence operator, this operator just has to mark its successor operations as a parallel group (for a detailed description of parallel control operators see Chapter 6).

Furthermore, the hierarchical structure of scopes enables data partitioning in an intuitive way. A parallel sub-scopes processor only has to return a parallel group of operations which contains an operation for each of its sub-operators to be executed on one of the sub-scopes of the current scope. Therefore, parallelization can be applied on any level of scopes which enables the definition of global, fine- or coarse-grained parallel heuristic algorithms.

5.3.5 Layers of User Interaction

Working with a heuristic optimization software system on the level of a generic algorithm model such as the one described above offers a high degree of flexibility. Therefore, this level of user interaction is very suitable for experts who focus on algorithm development. However, dealing with algorithms on such a low level of abstraction is not practical for practitioners or students. Practitioners require a comprehensive set of predefined algorithms which can be used as black box optimizers right away. Similarly, predefined algorithms and also predefined problems are equally important for students, so that they can easily
start to experiment with different algorithms and problems and to learn about heuristic optimization. Consequently, a heuristic optimization software system has to provide several layers of user interaction that reflect different degrees of detail [20].

Based on the algorithm model, such layers of user interaction are provided in HL3. Predefined algorithms are realized as specific engines that contain entire operator graphs representing algorithms such as evolutionary algorithms, simulated annealing, hill climbing, tabu search, or particle swarm optimization. By using these engines, users can work with heuristic optimization algorithms right away and do not have to worry about how an algorithm is represented in detail. They only have to specify problem-specific parts (objective function, solution encoding, solution manipulation) or choose one of the predefined optimization problems. Additionally, custom views can be provided for these engines that reveal only the parameters and outputs the user is interested in. By this means, the complexity of applying heuristic optimization algorithms is significantly reduced.

Between the top layer of predefined solvers and the algorithm model, arbitrary other layers can be specified that offer algorithm building blocks in different degrees of detail. For example, generic models of specific heuristic optimization paradigms, as for example evolutionary algorithms or local search algorithms, are useful for experimenting with these algorithms. These models also hide the full complexity and flexibility of the algorithm model from users who therefore can solely concentrate on the optimization paradigm. Especially in that case a graphical user interface is very important to provide suitable representations of algorithms, for example flow charts or domain-specific languages.

In Figure 5.11 the layered structure of user interaction in HL3 is shown schematically. Users are free to decide how much flexibility they require and on which level of abstraction they want to work. However, as all layers are based on the algorithm model, users can decrease the level of abstraction step by step, if additional flexibility is necessary in order to modify an algorithm.

```

Figure 5.11: Layers of user interaction in HL3
```
5.4 Analysis and Comparison

A comparison of HeuristicLab and some existing frameworks discussed in Chapter 4 concerning the requirements defined in Chapter 3 is shown in Table 5.1. A bullet (●) indicates that a requirement is fulfilled, a circle (○) marks requirements which are somehow addressed but other frameworks provide a better or more comprehensive solution, and a dot (·) shows requirements which are not considered in a framework (cf. Table 4.1).

Table 5.1: Evaluation and comparison of HeuristicLab

<table>
<thead>
<tr>
<th></th>
<th>Templar</th>
<th>HotFrame</th>
<th>EO</th>
<th>Open BEAGLE</th>
<th>ECJ</th>
<th>EasyLocal++</th>
<th>HeuristicLab 1.1</th>
<th>HeuristicLab 2.0</th>
<th>HeuristicLab 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automation</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>Customizable Output</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Generic Algorithm Model</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Generic Operators</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Generic Objective Functions</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Generic Solution Representations</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Graphical User Interface</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Integration</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Learning Effort</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Parallelism</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Parameter Management</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Performance</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Predefined Algorithms and Problems</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Replicability and Persistence</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

HeuristicLab 1.1 provides a paradigm-independent algorithm model which is very similar to the model of Templar, although it lacks a clean separation of optimization problems and solution representations. Furthermore, algorithm runs cannot be interrupted, saved, and restarted. HeuristicLab 1.1 is developed in C# and uses the Microsoft® .NET environment, which leads to similar benefits and drawbacks as those already discussed for ECJ in Chapter 4. It provides a plugin mechanism that enables dynamic extension, web-based deployment, and clean integration into other applications. A broad spectrum of plugins for trajectory-based as well as population-based metaheuristics and many different optimization problems have been implemented. Furthermore, a bunch of plugins developed in the HeuristicLab Grid project enables parallel and distributed batch execution. Additionally, GUI components are integrated into the core framework and enable graphical and interactive configuration and analysis of algorithms.
HeuristicLab 2.0 was designed to overcome some of the drawbacks of the previous version. The main purpose was to combine a generic and fine-grained algorithm model with a GUI to enable dynamic and interactive prototyping of algorithms. Algorithm developers can use the GUI to define, execute, and analyze arbitrary complex search strategies and do not have to be experienced programmers. Furthermore, HeuristicLab 2.0 also offers support for replicability and persistence, as algorithm runs can be aborted, saved, loaded and continued. However, HeuristicLab 2.0 is a research prototype and has never been officially released. Therefore, it still contains some drawbacks concerning performance, stability, usability, learning effort, and documentation. Additionally, the implementation of parallel metaheuristics is difficult due to the nested execution of operators and the local status variables which are used in many operators.

Finally, HL3 fulfills almost all requirements identified so far. Just performance remains as the last requirement that is still an open issue. Even though HL3 enables parallel execution of algorithms and can therefore utilize multi-core CPUs or clusters, the runtime performance of sequential algorithms is worse compared to other frameworks. Reasons for this drawback are the very flexible design of HL3 and the dynamic representation of algorithms. As algorithms are defined as operator graphs, an engine has to traverse this graph and execute operators step by step. Of course, this requires more resources in terms of runtime and memory as is the case when algorithms are implemented as static blocks of code. In order to get a clear picture of this effect, some runtime tests have been carried out which are discussed in the following.

In Table 5.2 the absolute runtime of a standard genetic algorithm (SGA) implemented in HeuristicLab 1.1 and HeuristicLab 3.0 is shown. The SGA was applied on instances of the traveling salesman problem (TSP) of different sizes (130, 1379, and 3038 cities) taken from the TSPLIB\(^8\) [150]. Solutions were encoded as permutations using the path representation (for example, the permutation \((1 \ 3 \ 2)\) represents the tour that goes from city 1 to city 3, then to city 2 and finally back to city 1 again). Typical crossover and mutation operators for permutations were applied that are well-known in the literature (see for example [114]). All parameter settings of the SGA are listed in Table 5.3. Each value shown in Table 5.2 represents the average absolute runtime in seconds of five test runs and the average relative runtime difference of HeuristicLab 1.1 and HeuristicLab 3.0, respectively. All tests were executed on a workstation with an Intel\(^\text{®}\) dual-core CPU (2.33 GHz per core) and 4 GB RAM running Microsoft\(^\text{®}\) Windows Vista\(^\text{®}\) Enterprise SP1 (32 bit). Additionally, the SGA was also enhanced with global parallelization in HeuristicLab 3.0 to outline the effects of parallel execution; in this case, solutions were evaluated in parallel using two threads.

As it can be seen in Table 5.2, the relative runtime difference becomes less for TSP instances of larger size. If more effort has to be put on the evaluation of

---

\(^8\)http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95
### Table 5.2: Runtime of HeuristicLab 3.0 compared to HeuristicLab 1.1 (in seconds)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>HeuristicLab 1.1</th>
<th>HeuristicLab 3.0 (sequential)</th>
<th>HeuristicLab 3.0 (parallel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch130</td>
<td>0.44</td>
<td>4.74 (+986.24%)</td>
<td>4.00 (+817.43%)</td>
</tr>
<tr>
<td>nrw1379</td>
<td>9.19</td>
<td>13.95 (+51.81%)</td>
<td>9.93 (+8.08%)</td>
</tr>
<tr>
<td>pcb3038</td>
<td>21.23</td>
<td>25.85 (+21.74%)</td>
<td>17.74 (-16.42%)</td>
</tr>
</tbody>
</table>

### Table 5.3: SGA parameter settings used in the runtime tests

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generations</td>
<td>1000</td>
</tr>
<tr>
<td>Population size</td>
<td>100</td>
</tr>
<tr>
<td>Selection</td>
<td>Tournament</td>
</tr>
<tr>
<td>Tournament group size</td>
<td>2</td>
</tr>
<tr>
<td>Crossover</td>
<td>Order crossover</td>
</tr>
<tr>
<td>Mutation</td>
<td>2-opt</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>5%</td>
</tr>
<tr>
<td>Replacement</td>
<td>Generational</td>
</tr>
</tbody>
</table>

solutions, the overhead of the HeuristicLab 3.0 algorithm model is less critical. This is in fact an important aspect, as most resources are consumed for the evaluation of the objective function in many real-world optimization applications. Therefore, the benefits of HeuristicLab 3.0 in terms of flexibility, genericity and extensibility should outweigh the additional overhead. Furthermore, the efficiency of HeuristicLab 3.0 can be easily improved by executing operators in parallel.

### 5.5 Applications of HeuristicLab

Since 2002 all versions of HeuristicLab have been extensively used in the research group “Heuristic and Evolutionary Algorithms Laboratory (HEAL)” of Michael Affenzeller and the author for the development of enhanced evolutionary algorithms as well as in several research projects and lectures. The broad spectrum of these applications highlights the flexibility and suitability of HeuristicLab for the analysis, development, test, and productive use of metaheuristics. In this section the applications of HeuristicLab and some of the resulting publications are summarized. A comprehensive description of the research activities of the group can also be found in the book “Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications” [17] written by Michael Affenzeller, Stephan Winkler, Stefan Wagner, and Andreas Beham.
Analysis and Development of Enhanced Evolutionary Algorithms

Already HeuristicLab 1.1 served as a fundamental basis for the analysis and development of enhanced evolutionary algorithms. Michael Affenzeller and the author focused on the analysis of genetic algorithms and on the effects of selection and selection pressure with respect to genetic diversity and solution quality [7, 6, 168, 171, 14]. They developed a new selection concept for evolutionary algorithms called offspring selection (OS) [5, 9, 11, 10, 12, 16] which was inspired by the concept of birth surplus adapted from evolution strategies. The main idea of OS is to compare the quality of child solutions with the quality of their parents. Selection pressure is controlled in a self-adaptive way, as only those children are added to the next generation that are able to outperform their parents. As long as a predefined ratio of successful offspring is not reached, new children are created repeatedly. When applying OS, the genetic information is assembled in a more homogeneous way, premature convergence is delayed, and considerably better solutions are reached in the end. Furthermore, OS also provides an intuitive termination criterion for evolutionary algorithms, as convergence is detected, if there are not enough successful offspring even after an upper bound for the selection pressure has been reached.

Based on OS, a new hybrid coarse-grained parallel evolutionary algorithm (SASEGASA) was presented [4, 8]. SASEGASA combines OS with concepts of simulated annealing and introduces a new dynamical migration scheme. In order to combine the genetic information distributed in the subpopulations, the number of subpopulations is decreased in every migration phase after convergence has been detected. As the total number of solutions stays constant, the subpopulations grow larger and larger and finally become a single panmictic population. In addition to SASEGASA, another enhanced evolutionary algorithm called RAPGA [13, 17] was also developed which contains a single population and uses OS to adjust the population size in a self-adaptive way.

The benefits of these algorithms have been shown for benchmark optimization problems such as the TSP or real-valued test functions as well as in several research projects tackling real-world optimization problems (see below). In this context the flexible design of HeuristicLab 1.1 and the separation of problem-dependent and problem-independent parts turned out to be very effective, as it was easy to add new optimization problems and to reuse the OS, SASEGASA, or RAPGA plugin. Furthermore, in order to demonstrate the effects of OS and the other algorithms, the graphical user interface of HeuristicLab was very helpful, as it provided an intuitive and immediate way to visualize and analyze the processes inside an algorithm as for example the propagation of alleles or the change of genetic diversity. Moreover, in HeuristicLab 2.0 and 3.0 OS and SASEGASA have been decomposed and implemented as a set of operators which provides even more flexibility in terms of hybridization and analysis.
Optimization of Fungus Growth Media

In cooperation with Stefan Hutwimmer from the University of Innsbruck the author worked on the optimization of growth media for the entomopathogenic fungus *Metarhizium anisopliae*. Based on HeuristicLab 2.0 a genetic algorithm was used to find optimal configurations of synthetic growth media in order to increase the conidium production. Concerning metaheuristic optimization the challenging part was to develop a custom solution representation describing the configuration of growth media as well as the implementation of corresponding crossover and manipulation operators. HeuristicLab 2.0 provided comprehensive support for this task, as the existing standard solution representation for vectors of real numbers could be reused for building a complex solution representation containing a set of vectors. This approach had the additional benefit that custom manipulation operators could be defined in the GUI by combining existing operators which significantly reduced the required development time. Furthermore, an external and human-based fitness evaluation had to be applied, as each medium configuration had to be tested by growing *Metarhizium anisopliae* in the laboratory and analyzing its conidium production. Each of these experiments took several days which led to a very long runtime of the genetic algorithm. Therefore, the persistence mechanism of HeuristicLab was important to be able to save an algorithm each time after a new generation of configurations had been created and to reload and continue it after the evaluation of the growth media had been finished. A detailed description of this work can be found in two articles which have been published in the Journal of Applied Microbiology [96, 97].

Data-Based Identification of Non-Linear Systems

The data-based identification of non-linear systems is one of the main research areas of HEAL [15]. In this context, the work of Stephan Winkler has to be mentioned, as he developed a very flexible and feature-rich application for system identification using genetic programming based on HeuristicLab 2.0. In his work the operator-based algorithm model of HeuristicLab 2.0 has shown its potential, as it enabled an easy integration of OS into genetic programming as well as the hybridization of genetic programming with other concepts such as pruning or model parameter optimization using evolution strategies. Moreover, numerous operators have been implemented which enable an extensive analysis of the genetic programming process by visualizing for example the similarity of models, the dynamics in a population, the impact of input variables, or the diversity of functions and terminals. The effectiveness of OS-based genetic programming for solving data analysis problems in HeuristicLab 2.0 has been shown for classification, regression, and time-series problems using benchmark and real-world data sets [186, 181, 183, 182, 180, 185, 184].

Furthermore, Gabriel Kronberger has ported main parts of the genetic pro-
gramming framework of HeuristicLab 2.0 to HeuristicLab 3.0. In his work, he concentrates on the cooperative application of multiple data analysis agents to be able to analyze large data sets not only with respect to a single target variable, but with respect to arbitrary predictive models describing the input/output relationships contained in a data set [113]. Thereby, the support for parallel and distributed execution of algorithms provided by HeuristicLab 3.0 is essential, as many genetic programming runs have to be executed for each data set.

Production Optimization

Another research focus of HEAL covers the optimization of complex production systems. In several research projects the members of the research group applied metaheuristics to improve schedules in order to minimize the total tardiness of jobs or the required setup costs. In this context, genetic algorithms (with and without OS), SASEGASA, iterated local search, and tabu search have been applied on benchmark problem instances of the job shop scheduling problem as well as on real-world data of production plants [38, 39, 37, 40, 156]. Additionally, the genetic programming frameworks of HeuristicLab 2.0 and 3.0 have been used to generate complex and human-competitive dispatching rules in order to build schedules for highly volatile systems [29, 109].

In these projects all three versions of HeuristicLab have been used as development and test environment. Similarly to the case of data-based structure identification, the modular architecture of HeuristicLab had remarkable benefits: On the one hand, HeuristicLab provided a set of ready to use algorithms, solution representations, and operators which were applied and compared regarding their suitability to solve scheduling problems; on the other hand, problem-specific operators such as the Giffler-Thompson scheduling algorithm, enhanced crossover operators for job shop scheduling problems, or graph-based neighborhood structures were easy to implement due to the operator model used to represent algorithms. Furthermore, HeuristicLab 3.0 has also been deployed in the involved companies which shows that HeuristicLab is ready for productive use and can be embedded into existing enterprise resource planning systems.

Simulation-Based Optimization

As a third area of interest the researchers of HEAL concentrate on optimization problems that do not provide an explicit objective function but use a simulation model to evaluate the quality of a solution. In this regard, evolutionary algorithms represent a powerful and effective approach to optimize the input parameters of simulation models [3]. In such applications, heuristic optimization software systems have to face two essential challenges: On the one hand, a single simulation run usually requires considerable runtime; therefore, evaluation has to be done in parallel in order to be able to evaluate a reasonable number of solutions. On the
other hand, interoperability with other applications is crucial, as the simulation model is usually not implemented in the heuristic optimization software system but in some kind of simulation framework.

In recent publications, it was shown that HeuristicLab 3.0 is able to handle these challenges. As parallelism is integrated into the algorithm model and the uniform representation of data in scopes enables the development of generic operators for communicating with other frameworks, simulation-based optimization applications can be realized straight away. For example, Andreas Beham discussed the application of evolution strategies to deterministic and stochastic simulation-based optimization problems [27, 28]. He developed plugins for HeuristicLab 3.0 which provide a generic interface for combining HeuristicLab with arbitrary simulation frameworks such as AnyLogic® or Simulink®. In this context, arbitrary communication protocols using different channels can be defined interactively in the graphical user interface. Furthermore, Monika Kofler used HeuristicLab 3.0 to apply genetic algorithms on a simulation model for medical ultrasound transducers [108].

**Lectures and Theses**

Finally, HeuristicLab has not only been used in research and industrial projects but also in various lectures on heuristic optimization given by Michael Affenzeller and the author at the Johannes Kepler University in Linz and the School of Informatics, Communications and Media of the Upper Austria University of Applied Sciences in Hagenberg. In these lectures the graphical user interface and the variety of available classical metaheuristics and benchmark optimization problems increased the learning success significantly and enabled students to experiment with different algorithms immediately. Furthermore, all versions of HeuristicLab have also been used in several master theses supervised by Michael Affenzeller and the author [36, 179, 112, 158, 30, 98, 141, 26, 31, 57, 142, 157]. In these diploma projects the plugin architecture, the intuitive API, and especially the flexible algorithm model of HeuristicLab 3.0 turned out to be very well suited for students to implement applications focusing on different theoretical and practical aspects.
Chapter 6
Algorithm Modeling

Based on the HL3 architecture introduced in Chapter 5, it is shown in this chapter how heuristic optimization algorithms can be represented in the HL3 algorithm model. As a proof of concept, operators are presented in the following sections that are used to build some typical examples of standard heuristic optimization algorithms. Furthermore, these operators serve as algorithm building blocks for successively defining more complex parallel and hybrid metaheuristics, showing the flexibility and genericity of the framework.

6.1 Basic Operators

First of all, simple operators are discussed that perform typical tasks required in every kind of algorithm.

EmptyOperator

The EmptyOperator is the most simple form of an operator. It represents an operator that does nothing and can be compared to an empty statement in classical programming languages.

To get an idea of how an operator implementation looks like in HL3, Listing 6.1 shows its implementation. Each operator is inherited from the abstract base class OperatorBase. This base class takes care of aspects that are identical for all operators (e.g., storing of sub-operators and variable information, persistence and cloning, and events to propagate changes). Note that the most important method of each operator is Apply which is called by an engine to execute the operator. The scope that the operator is to be executed on is passed as a parameter. Apply may return an object implementing IOperation which represents the successor operations (one or more operators and scopes). If the operator has no successor operations, null is returned.
CHAPTER 6. ALGORITHM MODELING

public class EmptyOperator : OperatorBase {
    public EmptyOperator () {
    }

    public override IOperation Apply(IScope scope) {
        return null;
    }
}

Listing 6.1: EmptyOperator

VariableInjector

Each algorithm execution starts with an initial operator being executed on an empty global scope. Therefore, a mechanism is required to add user-defined variables to a scope. For example, at the beginning of each execution of an algorithm some initialization is required. Global parameters (e.g., population or neighborhood size, mutation rate, tabu tenure, or maximum number of iterations) have to be added to the global scope, so that each succeeding operator will be able to access these values. This task is fulfilled by the operator VariableInjector. On its execution, a VariableInjector clones variables that have been defined by the user and adds them to the scope the operator is applied on.

RandomInjector

As most heuristic optimization algorithms are stochastic processes, uniformly distributed high quality pseudo-random numbers are required. For creating random numbers a single pseudo-random number generator (PRNG) should be used to enable replicability of runs. By setting the PRNG’s random seed the produced random number sequence is always identical for each run; it is therefore useful to inject a single PRNG into the global scope. Although PRNGs are also represented as data objects and consequently can be stored in variables and injected using a VariableInjector, it has been decided to implement a custom operator RandomInjector for this task. Reasons are that it can be specified for a RandomInjector whether to initialize the injected PRNG with a fixed random seed to replay a run or to use an arbitrary seed to get varying runs.

Counter

Counter is a basic operator for incrementing integer variables. It increases the value of a variable in a scope by one. As an example, this operator can be used for counting the number of evaluated solutions or to increment the actual iteration number.
6.1. BASIC OPERATORS

As the Counter operator is the first operator that changes a variable, Listing 6.2 shows its implementation to give an impression how the manipulation of variables in a scope is done. In the constructor a new variable information object is added which declares that the operator expects an integer variable holding the value that should be increased. This variable is an input and an output variable as the old value is read and the new value is written back. In the Apply method the actual value of the variable is retrieved by feeding the formal name and the actual scope into the generic GetVariableValue method. This method translates the formal variable name into the actual name (which has been specified by the user in the GUI) and looks for the variable in the actual scope. The third parameter of GetVariableValue states, if a recursive variable lookup should be performed when the variable is not found in the scope; i.e., if the scope tree should be traversed upwards looking for the variable until the variable is found or no other scope is available. Finally, the value is incremented and null is returned as there are no more successor operations.

```csharp
public class Counter : OperatorBase {
    public Counter() {
        AddVariableInfo(new VariableInfo(
            "Value", // formal name
            "Counter value", // description
            typeof(IntData), // data type
            VariableKind.In | VariableKind.Out // kind
        ));
    }

    public override IOperation Apply(IScope scope) {
        IntData value = GetVariableValue<IntData>("Value", scope, true);
        value.Data = value.Data + 1;
        return null;
    }
}
```

Listing 6.2: Counter

Comparator

The Comparator operator is responsible for comparing the values of two variables. It expects two input variables which should be compared and a comparison operation specifying which type of comparison should be applied (e.g., less, equal, greater or equal). After retrieving both variable values and comparing them,
Comparator creates a new Boolean variable containing the result of the comparison and writes it back into the scope.

**SubScopesCreator**

The operator *SubScopesCreator* is used to extend the scope tree. It expects an input variable specifying how many new and empty sub-scopes should be appended to the current scope. For example, this operator can be used for creating a new scope for each solution in a population.

### 6.2 Control Operators

As already discussed in the previous chapter, operators of the HL3 architecture are able to decide which operations are executed next. Therefore, the execution flow of an algorithm can be defined using control operators that do not manipulate any variables but return successor operations. In this section basic control operators are presented to model sequences and branches. Any other control structure known from classical programming languages (as for example switch statements, do-while loops, repeat-until loops) can be realized by combining these operators. Furthermore, some special operators are available to model parallelism.

**SequentialProcessor**

*SequentialProcessor* represents a sequence of operations that should be applied on the current scope. It returns an operation for each of its sub-operators and the scope it is applied on. Listing 6.3 shows the implementation.

```csharp
public class SequentialProcessor : OperatorBase {
    public override IOperation Apply(IScope scope) {
        CompositeOperation next = new CompositeOperation();
        for (int i = 0; i < SubOperators.Count; i++)
            next.AddOperation(new AtomicOperation(SubOperators[i], scope));
        return next;
    }
}
```

Listing 6.3: SequentialProcessor

A set of successor operations is represented using a *CompositeOperation* object which implements the IOperation marker interface and contains arbitrarily many other operations (i.e., objects implementing IOperation). In contrast, objects of
the type \textit{AtomicOperation} are simple operations containing just one operator and one scope. For each sub-operator an \textit{AtomicOperation} is created and added to the \textit{CompositeOperation}.\footnote{Note that operations are implemented using the composite pattern [78].}

\section*{ParallelProcessor}

If a group of operators should not be executed sequentially but in parallel, the operator \textit{ParallelProcessor} can be used. Very similarly to the SequentialProcessor it returns an operation of each of its sub-operators and the current scope, but additionally this group is marked as a parallel group. This is simply done by setting the \textit{ExecuteInParallel} property of \textit{CompositeOperation} as shown in Listing 6.4.

\begin{lstlisting}[language=C#]
public class ParallelProcessor : OperatorBase {
    public override IOperation Apply(IScope scope) {
        CompositeOperation next = new CompositeOperation();

        // contained operations might be executed in parallel
        next.ExecuteInParallel = true;

        for (int i = 0; i < SubOperators.Count; i++)
            next.AddOperation(new AtomicOperation(SubOperators[i], scope));

        return next;
    }
}
\end{lstlisting}

\textit{Listing 6.4: ParallelProcessor}

\section*{UniformSequentialSubScopesProcessor and UniformParallelSubScopesProcessor}

Another important kind of control operators is necessary to navigate through the hierarchy levels of the scope tree (i.e., to apply an operator on each sub-scope of the current scope). The \textit{UniformSequentialSubScopesProcessor} fulfills this task by returning an operation for each sub-scope and its first sub-operator. Analogically to the SequentialProcessor, also a parallel version called \textit{UniformParallelSubScopesProcessor} is available which defines that processing of all sub-scopes can be done in parallel leading to a single program multiple data style of parallel processing.
SequentialSubScopesProcessor and ParallelSubScopesProcessor

As a generalization of UniformSequentialSubScopesProcessor and UniformParallelSubScopesProcessor the two operators SequentialSubScopesProcessor and ParallelSubScopesProcessor return an operation not only for the first sub-operator and every sub-scope, but pair sub-operators and sub-scopes together. For each sub-scope there has to be a sub-operator which is executed on its corresponding sub-scope to enable individual processing of all sub-scopes.

ConditionalBranch

The operator ConditionalBranch can be used to model simple binary branches. It retrieves a Boolean input variable from the scope tree. Depending on the value of this variable, an operation containing the actual scope and either the first (true branch) or the second sub-operator (false branch) is returned.

Note that do-while or repeat-until loops can be constructed without any other specific loop operator by combining the operators ConditionalBranch and SequentialProcessor. Figure 6.1 shows an operator graph for these two loop structures.

![Operator graphs representing a do-while and a repeat-until loop](image)

StochasticBranch

In heuristic optimization algorithms it is a common pattern to execute operations with a certain probability (for example mutation of individuals in evolutionary algorithms or post-optimization heuristics in hybrid algorithms). Of course, this
6.3. EXEMPLARY PROBLEM REPRESENTATION

could be realized by using an operator injecting a random number into a scope in combination with the Comparer and ConditionalBranch operators. However, for convenience reasons the StochasticBranch operator performs this task in one step. It expects a double variable as an input specifying the probability and a PRNG. When applying the operator a new random number between 0 and 1 is generated and compared with the probability value. If the random number is smaller, the true branch (first sub-operator) or otherwise the false branch (second sub-operator) is chosen.

6.3 Exemplary Problem Representation

After defining the set of generic operators which build the basis for every heuristic optimization algorithm, the next step is to integrate an optimization problem and to focus on problem-specific aspects such as solution encoding, quality evaluation and manipulation operators. As this chapter should be a proof of concept showing that different typical heuristic optimization algorithms from various paradigms can be modeled within the HL3 algorithm model, it seems reasonable to choose a well-known and frequently used benchmark problem that can be attacked with a multitude of algorithms. One problem fulfilling this requirement is the traveling salesman problem (TSP).

The TSP is an NP-hard combinatorial optimization problem. It is one of the most basic forms of route planning problems and its objective can be stated as follows [115]:

*Given a finite number of cities and given the travel distance between each pair of them, find the shortest tour visiting all cities and returning to the starting point.*

From a mathematical point of view the TSP can be described as the problem to find a Hamiltonian cycle with a minimum total weight for the graph built by the cities (vertices) and the distances between them (weighted edges). A precise formal definition of the TSP as well as a broad overview of its different variants and of algorithms used to solve it is given in [115].

In fact, the TSP has already been discussed for a long period of time. Its first appearance in literature goes back to 1831 [165]. As the TSP is NP-hard, no efficient way is known to solve high dimensional TSP instances until today. Furthermore, the TSP is not only an academical benchmark problem but also plays an important role in lots of different practical fields, from the problem of finding an optimal way to drill circuit boards to the minimization of setup costs by optimizing the execution sequence of jobs on a machine. A detailed description of TSP applications in various areas is given in [153]. These facts illustrate why the TSP has been so intensively studied in the past and was attacked with various exact and heuristic optimization techniques.
6.3.1 Solution Encoding

For applying heuristic optimization algorithms to the TSP an appropriate solution encoding is required to represent tours. In literature various suggestions can be found ranging from specific encodings specially designed for a particular algorithm to more general ones such as adjacency lists or permutations. One of the frequently used representations is the so-called path representation. In the path representation a TSP tour is encoded as permutation of length $n$, where $n$ stands for the number of cities. The sequence of numbers in the permutation represents the sequence of cities in the tour. For example, the solution $(1\ 5\ 3\ 4\ 2)$ represents the tour that goes from city 1 to city 5 to city 3 to city 4 to city 2 and finally back to city 1 again. The advantages of path representation are on the one hand its high efficiency concerning the calculation of the total tour length and on the other hand the richness of different operators suitable for manipulating permutations. A comprehensive overview of these operators and of other solution encodings suitable for the TSP is provided in [114].

However, permutations are a frequently used solution encoding not only in the case of the TSP. In each problem where a sequence of elements has to be selected from a finite set, this sequence can be easily represented as a permutation. Therefore, permutation-based solution encoding is also suitable for many other combinatorial optimization problems as for example job scheduling. This fact is another reason why the TSP and path representation have been chosen to demonstrate how to represent an optimization problem using HL3 operators. In the next paragraphs a set of operators for creating and manipulating permutations is introduced that can be reused for any other problem for which permutation-based solution encoding is reasonable. Besides, several other solution encodings and the corresponding initialization and manipulation operators are also implemented in HL3 (e.g., binary encoding, vectors of real values, or vectors of integer values).

RandomPermutationGenerator

First of all, a solution initialization operator is required that adds a new random permutation of given length to a scope. RandomPermutationGenerator expects the permutation length and a PRNG as input variables and adds a new output variable containing the generated permutation into the scope it is applied an. To generate random permutations the Fisher-Yates shuffle (also known as Knuth shuffle) [67] is used which has linear runtime complexity.

Manipulation Operators

Several operators for manipulating permutations are implemented in HL3. For example, among them the operators Swap2Manipulator (swapping two randomly chosen numbers), InversionManipulator (inverting a sub-sequence; also known as...
2-opt), *ScrambleManipulator* (randomly scrambling a sub-sequence), or *TranslocationManipulator* (shifting a sub-sequence) are available. All these operators expect an input output variable containing the permutation and manipulate this variable when they are applied on a scope.

**Recombination Operators**

Especially in the case of evolutionary algorithms, recombination (crossover) operators are required as a special kind of manipulation operators. A recombination operator does not manipulate a solution directly but creates new solutions by combining two or more parent solutions; in most cases two parent solutions are crossed to create a single child solution. As in the HL3 algorithm model each solution is represented by a scope, recombination operators therefore do not only manipulate variables but create new scopes. Each recombination operator is applied one level above the scopes representing concrete solutions. It removes all sub-scopes from the current scope, groups and crosses them and adds the new child scopes to the current scope. Figure 6.2 shows this procedure in a schematic way.

![Figure 6.2: General principle of recombination operators](image)

HL3 also offers a broad spectrum of operators for crossing permutations such as *OrderCrossover*, *CyclicCrossover*, *EdgeRecombinationCrossover*, *PartiallyMatchedCrossover*, or *MaximalPreservativeCrossover* (for a comprehensive explanation of these operators see for example [114]). All these operators contain a variable information object storing the actual name of the variable containing the permutation in each scope. This name is used to retrieve permutations from the parent scopes and to store resulting permutations in the child scopes.
6.3.2 Quality Evaluation

After defining the way solutions are encoded, the next step is to implement an objective function. The quality of a TSP solution is defined as the total length of the tour, i.e., the total sum of the travel distances or costs between all passed cities. In general, TSP instances are categorized into two groups: In symmetric TSP instances the travel distance between city $A$ and city $B$ is the same as the travel distance between city $B$ and $A$, whereas in asymmetric instances these two values might differ. One reason why the TSP is frequently used as a benchmark problem for comparing heuristic algorithms is the large variety of problem instances for which the quality of the global optimal solution (or at least a very good bound) has been determined exactly and proven mathematically. One collection of such benchmark instances can be found in the TSPLIB [150] which is an embracing collection of different route planning problems collected by the group around Gerhard Reinelt at the University of Heidelberg. It provides a lot of symmetric and asymmetric TSP benchmark problem instances with one proven optimal (or at least best known) solution\(^2\).

According to the HL3 algorithm model the objective function is also realized as an operator. Besides the operator calculating the actual solution quality, another operator is required for providing additional data representing the TSP instance. These operators are explained in detail in the next sections.

**TSPInjector**

*TSPInjector* is a special kind of VariableInjector adding variables to a scope that represents a symmetric TSP instance. As we are focusing on TSP benchmark instances taken from TSPLIB, the user has to provide a TSP description file in the TSPLIB file format. When applied on a scope, TSPInjector adds an integer variable representing the number of cities and a two-dimensional integer matrix for the city coordinates. Optionally, the best known quality value might also be added to the scope, if it is provided by the user. TSPInjector is typically applied in an initialization step at the beginning of an algorithm injecting TSP data into the global scope.

**RoundedEuclideanPathTSPEvaluator**

The most frequent way to realize an objective function for the TSP is provided by the *RoundedEuclideanPathTSPEvaluator*. It expects two input variables, one for the coordinates matrix and another one containing a permutation which encodes a solution in path representation. The operator uses the rounded Euclidean distance measure to calculate the total traveling costs and adds the solution.

---

\(^2\)Additional information and updates for the best (known) solutions can be found on [http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95](http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95).
6.4. SELECTION AND REDUCTION

quality as a double variable into the current scope. Due to the recursive variable lookup strategy, the coordinates matrix does not have to be contained in the current scope representing a solution but might be located in a scope somewhere above (usually in the global scope).

**TSPRoundedEuclideanDistanceMatrixInjector and DistanceMatrixPathTSPEvaluator**

Using the RoundedEuclideanPathTSPEvaluator for evaluating solutions of symmetric TSP instances is not the most efficient way, as the traveling distance between two cities is calculated again and again every time a solution is evaluated. Another more suitable way is calculating a distance matrix for all cities once at the beginning of the algorithm and using this distance matrix to get the traveling costs between two cities when evaluating a solution. This approach is realized by the two operators `TSPRoundedEuclideanDistanceMatrixInjector` and `DistanceMatrixPathTSPEvaluator`. The first one calculates the distance matrix using the rounded Euclidean distance measure again and can be applied after the TSPInjector in an initialization phase of an algorithm to inject the distance matrix into the global scope. The second one retrieves the distance matrix using recursive variable lookup to calculate the quality of a solution in path representation; thus computing the distance measurement function again is omitted.

As shown by these two evaluation operators for the TSP, a benefit of the HL3 algorithm model is the possibility to provide various quality functions of a problem as different operators. The general problem representation - for example the TSPInjector - does not have to be changed in any way. Additionally, specialized evaluation operators can also be implemented; for example, another evaluation operator might perform incremental evaluation which is frequently used in local search algorithms in order to improve the efficiency of evaluation.

6.4 Selection and Reduction

Seen from an abstract point of view, a large group of heuristic optimization algorithms called improvement heuristics follows a common strategy: In an initialization step one or more solutions are generated either randomly or using construction heuristics. These solutions are then iteratively manipulated in order to navigate through the solution space and to reach promising regions. In this process manipulated solutions are usually compared with existing ones to control the movement in the solution space depending on solution qualities. Selection splits solutions into different groups either by copying or moving them from one group to another; replacement merges solutions into a single group again and overwrites the ones that should not be considered anymore.
In the HL3 algorithm model each solution is represented as a scope and scopes are organized in a hierarchical structure. Therefore, these two operations, selection and replacement, can be realized in a straightforward way: On the one hand, selection operators split sub-scopes of a scope into two groups by introducing a new hierarchical layer of two sub-scopes in between, one representing the group of remaining solutions and one holding the selected ones as shown in Figure 6.3. Thereby solutions are either copied or moved depending on the type of the selection operator. On the other hand, reduction operators represent the reverse operation. A reduction operator removes the two sub-scopes again and reunites the contained sub-scopes as shown in Figure 6.4. Depending on the type of the reduction operator this reunification step may also include elimination of some sub-scopes that are no longer required.

![Figure 6.3: General principle of selection operators](image)

Following this simple principle of selection and reduction of solutions, HL3 provides a set of predefined selection and reduction operators that can be used as a basis for realizing complex selection and replacement schemes. These operators are described in detail in the following sections.

### 6.4.1 Selection Operators

The most trivial form of selection operators are the two operators \textit{LeftSelector} and \textit{RightSelector} which select sub-scopes either starting from the leftmost or the rightmost sub-scope. If the sub-scopes are ordered for example with respect to
solution quality, these operators can be used to select the best or the worst solutions of a group. If random selection of sub-scopes is required, RandomSelector can be used which additionally expects a PRNG as an input variable.

In order to realize more sophisticated ways of selection, ConditionalSelector can be used which selects sub-scopes depending on the value of a Boolean variable contained in each sub-scope. This operator can be combined with a selection preprocessing step to inject this Boolean variable into each scope depending on some other conditions.

Furthermore, HL3 also offers a set of classical quality-based selection schemes well-known from the area of evolutionary algorithms, as for example fitness proportional selection optionally supporting windowing (ProportionalSelector), linear rank selection (LinearRankSelector), or tournament selection with variable tournament group sizes (TournamentSelector). Additionally, other individual selection schemes can be integrated easily by implementing custom selection operators.

### 6.4.2 Reduction Operators

Corresponding reverse operations to LeftSelector and RightSelector are provided by the two reduction operators LeftReducer and RightReducer. Both operators do not reunite sub-scopes but discard either the scope group containing the selected or the group containing the remaining scopes. LeftReducer performs a reduction
to the left and picks the scopes contained in the left sub-scope (remaining scopes) and RightReducer does the same with the right sub-scopes (selected scopes). Additionally, another reduction operator called MergingReducer is implemented that reunites both scope groups by merging all sub-scopes.

6.4.3 Sorting Operators

Most selection operators consider solution quality as the main property affecting selection. However, as there are many different ways how solution qualities can be represented, selection operators should be abstracted from quality values as much as possible. For the operators which just expect an ordering of sub-scopes and do not need to consider exact quality values (for example best selection, worst selection, linear rank selection, tournament selection), operators are required for reordering sub-scopes regarding some property. SubScopesSorter is a representative of this class of operators and reorders sub-scopes depending on the value of a double variable contained in each scope which usually represents the quality value. Additionally, in other cases, as for example multi-objective optimization problems, custom sorting operators realizing other forms of ordering can be implemented.

6.5 Modularity

All operators introduced so far represent simple functionality required as a basis for building complex heuristic optimization algorithms. However, working directly with these simple operators can become quite cumbersome for users. Even for simple algorithms, such as a hill climber or a canonical genetic algorithm, operator graphs are quite large and complex as the level of abstraction of these operators is rather low. As a consequence, developing algorithms is a complex and error-prone task and it happens quite easily that users get lost in the depths of operator graphs.

To overcome these difficulties, a concept of modularization is needed. Users have to be able to define new operators fulfilling more complex tasks by combining already existing operators, either simple or of course also combined ones. For example, it is reasonable to define an operator for picking the best $n$ solutions out of a set of solutions or one for processing all solutions of a set with an evaluation operator. These operations are very common in different kinds of heuristic optimization algorithms, so it is not suitable to define them again and again when creating a new algorithm. Instead, using combined operators enables reuse of complex operators in different algorithms.

In HL3 the two operators CombinedOperator and OperatorExtractor are available for modularization of operator graphs that are described in detail in the next sections. An important aspect is that the framework itself does not provide any
predefined combined operators. Combined operators can be created by users. Therefore, it is possible for users to develop own or share existing operator libraries containing various combined operators for specific tasks. By this means the level of abstraction is not determined by the framework, but depends only on its users. On the one hand, experts can use basic combined operators for fundamental concepts required in a broad spectrum of heuristic optimization algorithms or can just use the provided simple operators for developing complex algorithms; they are able to work on a low level of abstraction to benefit from genericity and flexibility to a large extend. On the other hand, practitioners or students can use complex combined operators representing whole heuristic optimization algorithms as black box solvers to attack problems of their interest. However, if necessary each user can take a glance at the definition of a combined operator in order to explore the internal functionality of an algorithm. This concept of operator modularization satisfies the individual needs of the different user groups discussed in Chapter 3 and represents a fundamental aspect of the different layers of user interaction already described in Chapter 5.

**CombinedOperator**

A CombinedOperator contains a whole operator graph. When executed it returns an operation containing the initial operator of its graph and the current scope it was applied on. Therefore, the whole operator graph is executed by the engine subsequently.

In order to have a clearly defined interface and to enable parameterization of combined operators, variable information objects can be added dynamically by the user to declare which variables are read, manipulated or produced by the operators contained in the graph. However, parameterization is not restricted to data elements. As operators are considered as data in the HL3 architecture, operators can be injected into the scope tree that are used somewhere in the operator graph which enables a functional style of programming. For example, a combined operator might be needed that encapsulates manipulation and evaluation of all solutions in a solution set. Such an operator can be defined by using an UniformSequentialSubScopesProcessor to iterate over all solutions (sub-scopes) in a set (scope) applying a SequentialProcessor on each solution. The sequential processor contains three sub-operators, one for manipulating solutions, one for evaluating them, and one for counting the number of evaluated solutions. However, the two operators for manipulating and evaluating a solution do not have to be defined in the operator graph directly but can be retrieved from the scope tree. Consequently, the combined operator can be parameterized with the re-

---

3Several combined operators for different purposes are deployed together with HL3 for an easy introduction into the environment and to be able to use several standard heuristic optimization algorithms out of the box. However, these operator libraries are not hard-coded but are provided as HeuristicLab files that can be adapted by any user.
required manipulation and evaluation operator, but the contained operator graph
does not have to be changed in any way. In other words, combined operators
enable the definition of reusable algorithm building blocks that are independent
of specific optimization problems.

**OperatorExtractor**

Another operator called *OperatorExtractor* can be used to retrieve operators from
the scope tree. OperatorExtractor is a placeholder that can be added anywhere in
an operator graph of a combined operator and expects an input variable contain-
ing an operator. The operator looks for that variable in the scope tree recursively
and returns the contained operator and the current scope as its successor opera-
tion.

### 6.6 Greedy Local Search

As a first example of heuristic optimization algorithms that can be built using the
basic operators described in the previous sections, a greedy local search algorithm
is considered (cf. IILS as described in Chapter 2). Greedy local search improve-
ment heuristics navigate through the solution space by picking better solutions
from the neighborhood of existing ones. Thereby the current solution becomes
better and better until no improvement can be achieved anymore (i.e., until the
algorithm converges to an optimum) or any other termination criterion has been
reached (e.g., the maximum number of evaluated solutions). Initially, any solu-
tion from the solution space can be selected as a starting point (usually the initial
solution is generated randomly). The neighborhood of a solution is defined by
the manipulation operation used to change solutions. For selecting the next so-
lution several strategies can be considered which are heuristics themselves. The
two most common ones are first improvement and best improvement. The former
picks the first solution found in the neighborhood that is better than the current
solution; the latter picks the best solution found in the entire neighborhood that
outperforms the current solution. Listing 6.5 shows a first improvement greedy
local search in pseudo-code which is also often called hill climbing.

Representing this heuristic optimization algorithm in the HL3 algorithm
model for solving the TSP is a quite straightforward task. The user just has
to define an operator graph containing all required operators for creating solu-
tions, evaluating and manipulating them, selecting the better of two solutions
and so on. However, instead of building a single operator graph which would be
quite large even for such a simple algorithm, it is reasonable to decompose the
algorithm into combined operators. Several steps in this simple algorithm, such
as creating new solutions, manipulating and evaluating solutions, or picking *n*
best solutions, are also required in many other heuristic optimization algorithms
and can therefore be reused. In the following sections some combined operators representing basic solution processing, selection and reduction operations are described that are used as algorithm building blocks not only for representing the first improvement hill climber but also for other algorithms that will be described later on.

In this context, one main strength of HL3 has to be highlighted: These combined operators are not hard-coded in the framework but are provided as XML files which can be loaded, inspected and manipulated in the GUI by any user. Combined operators represent a higher level of abstraction that is defined completely on the user side. Therefore, if a user requires some extensions or modifications for building a specific algorithm, they can be easily adapted without changing any code of the core framework.

**SequentialSolutionsProcessor**

Two of the most fundamental operations in heuristic optimization algorithms are manipulating and evaluating all elements of a solution set. These tasks can be easily represented in a combined operator called `SequentialSolutionsProcessor`. `SequentialSolutionsProcessor` expects three variables: An operator representing the manipulation that should be applied on each solution (`Processor`), an operator representing the objective function (`Evaluator`), and an integer variable for counting the number of evaluated solutions (`EvaluatedSolutions`). To iterate over all solutions in the set (i.e., all sub-scopes of the current scope) a UniformSequentialSubScopesProcessor is used that applies a SequentialProcessor on each sub-scope. This SequentialProcessor contains three operators, two operator extractors for extracting Processor and Evaluator from the scope tree and a Counter
for incrementing EvaluatedSolutions. Figure 6.5 shows the corresponding operator graph.

![Operator graph of SequentialSolutionsProcessor](image)

**Figure 6.5: Operator graph of SequentialSolutionsProcessor**

### SelectedSolutionsManipulator

Based on the SequentialSolutionsProcessor a more complex combined operator `SelectedSolutionsManipulator` is defined. This operator selects some solutions from a solution set and manipulates all selected solutions using a SequentialSolutionsProcessor. Obviously, this operator is the first one in which a combined operator is reused. SelectedSolutionsManipulator can be parameterized with an operator that should be used for selection (`Selector`), with another one for manipulating solutions (`Manipulator`), one more for solution evaluation (`Evaluator`), and finally with a variable representing the number of evaluated solutions (`EvaluatedSolutions`). Note that the last three parameters are required by SequentialSolutionProcessor and are just passed through.

As the selector adds an intermediate hierarchy level in the scope tree, a SequentialSubScopesProcessor has to be used to process both solution sub-sets which contain the remaining solutions and the selected ones. However, as no operators have to be applied on the remaining solutions, an EmptyOperator is used as the first sub-operator. Further details of SelectedSolutionsManipulator are shown in Figure 6.6.

### SolutionsTruncator

The combined operator `SolutionsTruncator` can be used to truncate a solution set in order to discard all supernumerous solutions. The desired cardinality of the solution set is defined by the parameter `Solutions`. As shown in Figure 6.7,
a LeftSelector is applied first that selects the given number of solutions without replacement and creates two distinct sets. A RightReducer is used afterwards which discards the left sub-scope containing the remaining solutions and moves the selected ones stored in the right sub-scope back one level up in the scope tree.

SolutionInitializer

Another combined operator called SolutionInitializer encapsulates the task of creating new solutions. It expects an input variable representing the number of solutions to create (Solutions), two operators, one for adding a solution’s data into a scope (Generator) and one for evaluating each new solution (Evaluator), and another input variable for counting the number of evaluated solutions (EvaluatedSolutions). After all new sub-scopes have been created, a SequentialSolutionsProcessor is used once more to iterate over all sub-scopes and to initialize
and evaluate every solution, using Generator as Processor for the SequentialSolutionsProcessor. Finally, a Sorter is applied for ordering all solutions by their quality value. Figure 6.8 shows the operator graph of SolutionInitializer.

![Operator graph of SolutionInitializer](image)

**HillClimber**

Finally, a first improvement greedy local search can be assembled using the operators introduced in the previous sections. The entire algorithm is represented in form of a combined operator (**HillClimber**) to support later reuse especially in the context of parallel multi-start heuristics or hybrid genetic algorithms, where a local search strategy is used to post-optimize a fraction of the population in each iteration.

**HillClimber** expects two input variables representing the operators used for solution manipulation (**Manipulator**) and evaluation (**Evaluator**), one variable containing the maximum number of solutions that should be evaluated (**MaxEvaluatedSolutions**), and the variable representing the quality value of each solution (**Quality**). The algorithm’s main loop is realized in a similar way as the loop patterns described earlier in this chapter. A Comparator is used to compare the actual number of evaluated solutions with the maximum value and produces a Boolean variable holding the result of the comparison (**TerminationCondition**). This value is used by a ConditionalBranch to decide which operator should be executed next. If not enough solutions have been evaluated yet, ConditionalBranch
Figure 6.9: Operator graph of HillClimber

applies a SequentialProcessor representing a single iteration of the algorithm: At first, a SelectedSolutionManipulator is used to select the actual solution with replacement (i.e., the current solution is duplicated) and applies the manipulation and evaluation operator on the selected solution. Then, both solutions, the original one and the manipulated one, are joined by using a MergingReducer. To determine which of the two solutions is the better one and should therefore be the next solution, a Sorter is used for sorting all solutions contained in the current scope (i.e., all sub-scopes of the current scope) by their quality value. The better solution is moved to the first position of the current scope’s sub-scopes and by applying a SolutionTruncator the remaining worse solution is finally discarded. Figure 6.9 shows the detailed operator graph of this procedure.

6.7 Simulated Annealing

As greedy local search algorithms head for the optimum located in the attraction basin of the initial solution, they severely suffer from the problem of getting stuck in a local optimum. To overcome this problem many heuristic optimization algorithms use additional strategies to support diversification of the search (a detailed description of these approaches has already been given in Chapter 2). One of these algorithms is simulated annealing (SA) which additionally introduces an acceptance probability. If a worse solution is selected in the solution space, it is
accepted with some probability depending on the quality difference of the actual (better) and the new (worse) solution and on a parameter called temperature; to be more precise, the higher the temperature and the smaller the quality difference, the more likely it is that a worse solution is accepted. In each iteration the temperature is continuously decreased leading to a lower and lower acceptance probability so that the algorithm converges to an optimum in the end. A detailed description of SA is given in Listing 6.6.

```
s ← new random solution // starting point
evaluate s
s_{best} ← s // best solution found so far
i ← 1 // number of evaluated solutions
t ← t_i // initial temperature

WHILE i ≤ maxSolutions DO BEGIN
  s' ← manipulate s // get solution from neighborhood
  evaluate s'
i ← i + 1
q ← quality difference of s and s'
IF s' is better than s THEN BEGIN
  s ← s'
  IF s is better than s_{best} THEN BEGIN
    s_{best} ← s
  END IF
END IF
  t ← t_i // calculate next temperature
END WHILE
RETURN s_{best}
```

Listing 6.6: Simulated annealing

As SA is obviously still quite similar to the greedy local search algorithm presented in the previous section, the HillClimber combined operator can be considered as a template for building a new combined operator SimulatedAnnealing. Just two additional parameters are required: On the one hand, InitialTemperature provides the initial temperature value; on the other hand, an operator TemperatureManipulator is necessary for calculating the new temperature value in every iteration (cooling scheme). For retrieving and applying this operator an OperatorExtractor is used at the end of each iteration. Note that HL3 offers a set of operators for performing perpetual manipulation of a variable value such
as addition of or multiplication with a constant factor. These operators can be used to parameterize the SimulatedAnnealing combined operator to achieve the desired cooling scheme.

To realize the enhanced acceptance criterion, the Sorter used in HillClimber has to be replaced by a more complex operator called SimulatedAnnealingAcceptor. If the quality of the new solution (second sub-scope) is better than the quality of the old one (first sub-scope), this operator just behaves like an ordinary Sorter. In the other case the order of sub-scopes is rearranged depending on the current acceptance probability. Therefore, SimulatedAnnealingAcceptor calculates the current acceptance probability depending on that quality difference of both solutions and on the current temperature and brings the second sub-scope to the front, if a random number between 0 and 1 is smaller than the probability value.

All other parts of HillClimber remain unchanged; this shows the power of modularity in the HL3 algorithm model again. The whole operator graph of the SimulatingAnnealing combined operator is shown in Figure 6.10 (for a better comparison with HillClimber changed parts are highlighted).
6.8 Evolution Strategies

The next example of heuristic optimization algorithms are evolution strategies (ES). Evolution strategies belong to the group of evolutionary algorithms and rely on mutation and deterministic selection as their main operators to drive the search. In a $(\mu + \lambda)$ ES, $\lambda$ children are selected randomly from a parent population of size $\mu$. After all these children have been mutated and evaluated, they are merged with all parents and the best $\mu$ solutions are selected as the next generation’s population (plus replacement). On the contrary, the $(\mu, \lambda)$ ES uses another replacement scheme (comma replacement) which discards the whole parent population and selects only the best $\mu$ solutions from the set of generated children. A pseudo-code description of both ES variants is shown in Listing 6.7.

```
1 create population pop0 of size $\mu$ // initial population
2 FOREACH s IN pop0 DO BEGIN
3   s ← new random solution
4   evaluate s
5 END FOREACH
6 i ← 1 // number of generations
7 WHILE $i \leq \text{maxGenerations}$ DO BEGIN
8   popi ← select $\lambda$ solutions of popi-1 randomly
9   FOREACH s IN popi DO BEGIN
10      mutate s
11      evaluate s
12 END FOREACH
13 IF plus replacement THEN BEGIN
14      copy all solutions of popi-1 into popi
15 END IF
16 popi ← select $\mu$ best solutions of popi
17 i ← i + 1
18 END WHILE
19 RETURN best solution of popi-1
```

Listing 6.7: Evolution strategy

An ES with both parameters, $\mu$ and $\lambda$, set to 1 behaves in the same way as a first improvement greedy local search algorithm. Thus, HillClimber serves also as a good starting point for building an EvolutionStrategy combined operator. It just has to be enhanced to be able to deal with a whole population of solutions instead of a single solution at a time. Therefore, two additional parameters $Mu$ and $Lambda$ are added. The LeftSelector used in HillClimber for creating a copy
of the current solution is replaced by a RandomSelector selecting $\lambda$ solutions from
the current population which are manipulated and evaluated using the Selected-
SolutionsManipulator. After manipulating all children, either a MergingReducer
(plus replacement) is used for merging parents and children or, if the comma
replacement should by used, it is replaced by a RightReducer discarding the left
sub-scope containing the parents. Then, SolutionsTruncator has to be applied
again with the parameter $\mu$ (instead of 1 in HillClimber) defining the number
of solutions that should remain.

As in an ES algorithm several solutions are created and evaluated in each it-
eration, the number of generations (iterations) is normally used as a termination
criterion instead of the number of evaluated solutions. Consequently, the param-
eter MaxEvaluatedSolutions is replaced by $\text{MaxGenerations}$ and another variable
$\text{CurrentGeneration}$ is introduced. These variables are used in the Comparator
to decide whether another iteration should be processed and in each iteration a
Counter is used for incrementing $\text{CurrentGeneration}$.

![Operator graph of EvolutionStrategy](image)

Figure 6.11: Operator graph of EvolutionStrategy

Obviously, just some minimal changes of HillClimber are required to realize a
simple plus or comma ES. The operator graph of the resulting EvolutionStrategy
combined operator is shown in Figure 6.11 (the parts that differ from HillClimber are highlighted again).

### 6.9 Genetic Algorithms

Finally, as a last example of standard heuristic optimization algorithms, genetic algorithms (GA) are considered. As also ES, genetic algorithms are a flavor of evolutionary algorithms and work on a population of solutions. The main differences between these two paradigms is that GAs use stochastic selection of parents (selection for reproduction) and crossover to combine the genetic material of several, in most cases two, parent solutions. Mutation is only applied with low probability as a background operator to introduce new genetic information for preventing premature convergence and to keep the search process alive. In the canonical genetic algorithm an entire population (generation) of children of the same size as the previous one is created in each iteration, completely replacing the parents (generational replacement). However, as it is convenient in many practical applications to have monotonous quality improvement, an additional concept called elitism is frequently used: A predefined number of best solutions (elites) is copied from the parent to the child generation so that at least the best solution found so far is not lost. Listing 6.8 shows the pseudo-code of such a standard genetic algorithm with elitism and generational replacement.

As GAs use two solution manipulation operations (crossover and mutation), the manipulation of selected solutions (i.e., SelectedSolutionsManipulator) has to be extended to incorporate crossover. Furthermore, generational replacement and elitism also represent a more complex replacement scheme that should be represented in a custom combined operator as well. In the following sections these operators are outlined in detail.

#### AdvancedSelectedSolutionsManipulator

In a GA recombination (crossover) has to be performed before selected solutions can be processed with a manipulation (mutation) operator. Therefore, SelectedSolutionsManipulator has to be extended to AdvancedSelectedSolutionsManipulator providing two additional parameters Preprocessor and Postprocessor. These parameters represent two operators that are applied on the super-scope of all selected solutions before and after manipulation. The resulting operator graph is shown in Figure 6.12. Note that the functionality of an ordinary SelectedSolutionsManipulator can be achieved by using an EmptyOperator for the

---

4Note that Postprocessor is not required for modeling a standard genetic algorithm. However, as post-processing is also used in many more advanced genetic algorithm concepts such as offspring selection or hybridization, this parameter is also introduced already.
6.9. GENETIC ALGORITHMS

Listing 6.8: Genetic algorithm with elitism and generational replacement

```
create population pop_0 // initial population

FOREACH s IN pop_0 DO BEGIN
    s ← new random solution
    evaluate s
END FOREACH

i ← 1 // number of generations

WHILE i ≤ maxGenerations DO BEGIN
    WHILE |pop_i| < |pop_{i-1}| DO BEGIN
        select parent_1 and parent_2 from pop_{i-1}
        child ← cross parent_1 and parent_2
        IF Random(0, 1) < mutationProbability THEN BEGIN
            mutate child
        END IF
        evaluate child
        add child to pop_i
    END WHILE
    replace elites worst solutions of pop_i by best solutions of pop_{i-1}
    i ← i + 1
END WHILE

RETURN best solution of pop_{i-1}
```

two new parameters; thus, each SelectedSolutionsManipulator used in the combined operators presented in the previous sections can be easily replaced by an AdvancedSelectedSolutionsManipulator.

**ElitesReplacement**

Merging the parent and child generations using generational replacement and elitism requires two steps: The parent generation has to be truncated to get the elite solutions (best solutions) and the same number of worst children has to be discarded in order to keep a constant population size.

In the *ElitesReplacement* combined operator these steps are realized by successive applications of selection and reduction operators. As the parent and the child generation are represented as two scopes, a SequentialSubScopesProcessor is used to process both solution sets independently. First, a LeftSelector is applied on the parents selecting *Elites* best solutions followed by a RightReducer picking just the selected solutions. Second, a RightSelector is applied on the children selecting the same number of worst solutions which are then thrown away
by a LeftReducer picking only the child solutions (remaining solutions) that have not been selected. Additionally, a Sorter is applied on the children before selection and reduction is applied, as the generated children are not ordered by their quality.
After both solutions sets, parents and children, have been truncated that way, a simple MergingReducer is used to merge both sets producing the next generation. Finally, the resulting solutions are also ordered according to their quality using the Sorter again so that the population is ready for the next GA iteration. For a graphical representation of ElitesReplacement consider Figure 6.13 showing the whole operator graph.

**GeneticAlgorithm**

Based on the two operators described above, a *GeneticAlgorithm* combined operator can now be assembled as shown in Figure 6.14. In each iteration an AdvancedSelectedSolutionsManipulator is used to create a population of children and ElitesReplacement is applied to merge parents and children creating the next generation. Beside the parameters already known from previous algorithms (Random, MaxGenerations, Evaluator, Quality), this operator expects some additional inputs:

- **Selector** represents the operator used to select parents for reproduction. In this way GeneticAlgorithm can be parameterized with different selection schemes. HL3 offers several common selection strategies such as fitness proportional selection (roulette wheel selection), linear rank selection, or tournament selection that can be used in addition to the standard selection operators (LeftSelector, RightSelector, RandomSelector).

- **Crossover** refers to the crossover operator applied in the pre-processing step of AdvancedSelectedSolutionsManipulator to create child solutions out of the selected parents.

- **Manipulator** represents the manipulation operator used for mutation. As mutation has to be applied with some predefined probability (*Mutation-Rate*), this operator is wrapped by a StochasticBranch that is passed as the manipulation operator (*Mutator*) to AdvancedSelectedSolutionsManipulator.

- **Elites** gives the number of elite solutions and is passed directly to ElitesReplacement in each iteration.

**SteadyStateReplacement**

To demonstrate the flexibility of the HL3 architecture, another common variant of GAs should also be considered. In a steady-state genetic algorithm solutions are not processed in a generational way, but a generated child is immediately
inserted into the actual population replacing some existing individual. One frequently used scheme to select the solution to be replaced is random but best replacement; i.e., a solution except the best one is selected randomly from the current population.

To realize such a steady-state GA only the replacement operator ElitesReplacement has to be exchanged whereas all other parts of the combined operator remain the same. The combined operator SteadyStateReplacement represents random but best replacement for a single child; the corresponding operator graph is shown in Figure 6.15. At first, a SequentialSubScopesProcessor is used to process the actual population using a LeftSelector to select and separate the best solution. Another SequentialSubScopesProcessor is used afterwards to select and discard a random solution out of the remaining individuals by applying a RandomSelector followed by a LeftReducer. Finally, the remaining solutions are reunited with the separated best solution by the first MergingReducer and are merged with the generated child using the second MergingReducer.

As shown by this example, successive applications of selection and reduction operators can be used to build more complex replacement schemes which can be integrated in GeneticAlgorithm to realize many different GA derivatives.
6.10 Hybrid Algorithms

In the previous sections several standard heuristic optimization algorithms were described and it was shown how different kinds of algorithms can be modeled by combining basic algorithmic building blocks represented as combined operators. In this section another important topic in the area of heuristic optimization is addressed, namely the combination of different optimization paradigms in a single algorithm.

One common approach is to hybridize global population-based optimization techniques with local search elements to post-optimize solutions in every generation to occupy niches (i.e., the search is intensified to exploit the current region of the solution space). Because of the modular and flexible definition of algorithms in HL3, the step towards a hybrid GA is not a difficult one. In AdvancedSelectedSolutionsManipulator the input parameter Postprocessor has already been introduced, although it was set to an EmptyOperator and was not used in GeneticAlgorithm. By removing the fixed assignment of an EmptyOperator and by declaring Postprocessor as an input parameter of GeneticAlgorithm, a possibility of integrating complex post-optimization procedures can be established.

Remember that the Postprocessor operator is executed on the super-scope of all generated children in each iteration of the GA. For example, when a \((\mu + \lambda)\) evolution strategy should be used for post-optimization of all children, the EvolutionStrategy combined operator can be passed as a Postprocessor to the GA. As another alternative even another GeneticAlgorithm with some different...
parameter settings can be used in the post-processing step. Thereby, actual variable names in the post-optimization algorithm have to be adapted to avoid name clashes with the surrounding GA due to identical input parameter names.

Furthermore, also local search algorithms working on single solutions can be used for hybridization. However, in that case another operator is required as an adapter from population-based to single solution algorithms to prepare the solution set of children. As all children (i.e., scopes) are located beneath a single super-scope in the population-based case, an intermediate layer has to be inserted first. Each child solution has to be moved to a separate scope so that combined operators - as for example HillClimber or SimulatedAnnealing - can be applied as they expect a scope containing just a single solution. This preparation step is implemented by an operator called IntermediateSubScopesCreator which can be considered as a special form of SubScopesCreator. The scope structure resulting from an application of IntermediateSubScopesCreator is shown in Figure 6.16.

![Figure 6.16: Scope manipulation of IntermediateSubScopesCreator](image)

After this preparation step, a simple UniformSequentialSubScopesProcessor can be used to apply for example HillClimber or SimulatedAnnealing on each intermediate scope optimizing the contained solution. Afterwards, a MergingReducer is required to bring all solutions into a single scope back again by removing the intermediate layer. In the end, all post-optimized solutions end up in a single solution set again and are ready for further processing by the surrounding algorithm. As the application of IntermediateSubScopesCreator and MergingReducer is of course independent of a concrete post-optimization algorithm, these steps can be encapsulated in a combined operator called LocalSearchHybridizationAdapter as shown in Figure 6.17.
6.11 Parallel Algorithms

It has already been discussed in Chapter 5 that incorporating parallelism in heuristic optimization algorithms is a common approach to enable execution on parallel computing systems and to distribute the work load. As parallelism is fundamentally anchored in the HL3 algorithm model, assembling parallel algorithms is as easy as replacing sequential control operators by parallel ones (ParallelSubScopesProcessor and UniformParallelSubScopesProcessor). It has to be kept in mind that these operators are used to mark blocks of operations that may be executed in parallel. The way how parallel execution of operations actually takes place is not defined by the operators but has to be implemented in corresponding engines, as for example the ThreadParallelEngine that enables parallel execution on multi-core workstations using multiple threads. This approach makes it very comfortable for users to develop parallel algorithms without having to worry about or implement the details of parallelization.

Depending on which parts of an algorithm are executed in parallel, two different parallelization approaches are used in the field of heuristic optimization: The first and more simple approach is just executing quality evaluation in parallel. As in many application domains solution evaluation is computationally far more expensive than solution manipulation, this approach is often very effective and still quite easy to implement (for example by using a master-slave architecture to distribute quality evaluations among a group of workers). Based on the HL3 algorithm model parallel quality evaluation can be added to an algorithm simply by using a UniformParallelSubScopesProcessor for iterating over a group of solutions and calculating their qualities. In combination with incrementing the number of evaluated solutions this functionality is represented in the ParallelEvaluator combined operator shown in Figure 6.18. Note that a special variant of Counter is required to avoid race conditions, as the same variable (EvaluatedSolutions) is accessed by each process.

The second and more complex approach is a heuristic optimization algorithm
that independently works on one or more solutions and therefore also performs solution manipulation in parallel. Due to the hierarchical structure of scopes such an algorithm can be realized following the same pattern in HL3. As scopes can be used to represent not only single solutions but also sets of solutions, sets of sets of solutions and so on (see Figure 6.19), UniformParallelSubScopesProcessor can also be used for parallel processing on any abstraction level.

For example, consider a simple parallel multi-start local search algorithm such as repeated executions of a hill climber or simulated annealing. As in the case of greedy local search the final solution always depends on the starting point of the search, repeating multiple runs is meaningful to increase the coverage of the solution space and to improve the solution quality. These runs are independent of each other, no communication is required in between and therefore they can be easily parallelized. In order to build for example such a parallel multi-start hill climber, an additional hierarchical layer has to be created in the scope tree using SubScopesCreator. The global scope is moved up to the super solution set level and contains several sub-scopes on the solution set level, each representing a sole hill climber run. By using a UniformParallelSubScopesProcessor the Hill-
6.12 ADVANCED CONCEPTS OF GENETIC ALGORITHMS

Climber combined operator can then be applied to every sub-scope in parallel\(^5\). Moreover, it is even possible to execute some of the hill climber runs with alternative parameter settings (for example with some other manipulation operator) by injecting different input variables also into the scopes on the solution set level which hide default values injected into the global scope.

Furthermore, parallelization is not limited to simple multi-start heuristics in HL3. More complex algorithms using some form of communication can be realized by applying operators on the parent scope above the scopes processed in parallel. By this means, the algorithm is structured in a parallel and a sequential part that are executed alternately. For example, a parallel GA implementing the island model can be realized by executing the GeneticAlgorithm combined operator in parallel on each population scope of the solution set level followed by a migration operator applied on the global scope that exchanges solutions from the different sub-scopes (populations). Several migration operators are already available in HL3 (e.g., \texttt{UnidirectionalRingMigrator} or \texttt{ScrambleMigrator}) that can be used together with selection and reduction operators to select and reintegrate solutions migrating from one subpopulation to another.

Finally, due to the flexible concept of the HL3 algorithm model, parallelization is also not limited to executing the same algorithm in parallel several times. By using ParallelSubScopesProcessor instead of UniformParallelSubScopesProcessor, an individual operator can be defined for each scope enabling parallel execution of completely different algorithms (parallel hybridization) as proposed in [19].

6.12 Advanced Concepts of Genetic Algorithms

As a last aspect dealt within this chapter and to conclude the topic of algorithm modeling, advanced concepts of GAs developed by Affenzeller and Wagner should also be considered to show how they can be realized in terms of the HL3 algorithm model.

Offspring selection is a generic and self-adaptive selection concept for genetic algorithms. It is inspired by evolution strategies and considers the quality of children in comparison to their parents. Roughly speaking, if a child is able to outperform its parents, it is called successful and is included in the next generation. By this means, offspring selection enables self-adaptive selection pressure steering, as it creates as many children as required in every generation to receive some predefined amount of successful offspring. Furthermore, offspring selection also enables a very intuitive termination criterion, as evolution can be stopped, if a defined upper limit of selection pressure is reached which indicates that no more successful offspring can be generated and premature convergence has occurred.

\(^5\)As in the case of hybridization, HillClimber can be reused as it is again.
Based on offspring selection, SASEGASA denotes an advanced parallel GA inspired by the island model. In SASEGASA each subpopulation is evolved using conventional operators well-known in GAs together with offspring selection, until maximum selection pressure is reached. Then, in contrast to classical migration in island GAs, a reunification scheme is used to reset borders between subpopulations reducing the number of subpopulations by 1. Consequently, solutions from different subpopulations are mixed step by step, until one huge panmictic population is left.

A detailed description of offspring selection and SASEGASA can be found in [8, 9, 2]. In order to incorporate both concepts into HL3, three operators are required that are described in the following.

**OffspringAnalyzer and OffspringSelector**

In order to realize offspring selection in the HL3 algorithm model two additional operators have to be provided. OffspringAnalyzer is responsible for deciding whether a created child was successful or not. This operator has to be executed twice in each iteration of the GA. At first, it is executed before crossover on the super-scope of all selected solutions to retrieve and store the quality values of all parents. Then, after crossover and mutation (i.e., after children have been created) it is executed on the super-scope of all solutions again to compare the quality values of the children with the saved qualities of their parents. Depending on the child’s quality and the actual comparison factor, a Boolean variable is injected into the child scope. If this Boolean flag is true, the quality of the child is higher than the current threshold and the child is considered as successful.

Based on the results of OffspringAnalyzer another operator called OffspringSelector has to be executed before replacement takes place. It separates generated children into successful and not successful ones depending on the Boolean value injected by OffspringAnalyzer. Additionally, it calculates the actual selection pressure and decides, if enough children have been created to build a new generation (either by reaching the desired success ratio or the maximum selection pressure). If not enough children are available yet, it saves the children already contained in the population scope, restores the original parent population and starts one more iteration of the GA creating more children. Otherwise, it retrieves all stored children and assembles the actual child population by mixing good and bad children concerning the value of success ratio. Finally, a classical replacement operator has to be used after OffspringSelector to mix parent and child solutions creating the next generation.

**SASEGASAREunificator**

To realized the SASEGASA reunification scheme a special operator is provided in HL3 called SASEGASAREunificator. Similarly to classical migration opera-
tors for example used in parallel island GAs (e.g., UnidirectionalRingMigrator or ScrambleMigrator), it is applied on the super-scope of all populations. At first, SASEGASAREunificator sequentially collects all solutions from every subpopulation. Then, the number of subpopulations is reduced by 1 and all solutions are separated into subpopulations again.
Chapter 7

Conclusion

The main goal of this thesis was to propose a new architecture for heuristic optimization software systems that consolidates existing frameworks and satisfies the requirements of three heterogeneous user groups, namely practitioners, heuristic optimization experts, and students. In order to achieve this goal, the author developed the HeuristicLab optimization environment which has continuously evolved since 2002. Three versions of HeuristicLab, referred to as HeuristicLab 1.1, HeuristicLab 2.0, and HeuristicLab 3.0, have been implemented which were discussed in this thesis. By incorporating beneficial features of existing frameworks as well as several novel concepts, especially the most recent version, HeuristicLab 3.0, represents a powerful and mature framework which can be used for the development, analysis, comparison, and productive application of heuristic optimization algorithms. The key innovations of HeuristicLab can be summarized as follows:

- **Plugin-Based Architecture**
  The concept of plugins is used as the main architectural pattern in HeuristicLab. In contrast to other monolithic frameworks, the HeuristicLab main application just provides a lightweight plugin infrastructure. All other parts are implemented as plugins and are loaded dynamically at runtime. This architecture offers a high degree of flexibility. Users can easily integrate custom extensions such as new optimization algorithms or problems by developing new plugins. They do not need to have access to all the source code of HeuristicLab or to recompile the whole application. Furthermore, the modular nature of the plugin-based architecture simplifies the integration into existing software environments, as only the plugins required in a specific optimization scenario have to be deployed.

- **Generic Algorithm Model**
  As there is no unified model for all different heuristic optimization techniques in general, a generic algorithm model is implemented in HeuristicLab that is not restricted to a specific heuristic optimization paradigm.
Any kind of algorithm can be represented. To achieve this level of flexibility, algorithms are not implemented as static blocks of code but are defined as operator graphs which are assembled dynamically at runtime. Users can define custom algorithms by combining basic operators for different solution encodings or optimization problems provided by several plugins, or they can add custom operators to integrate specific functionality. Consequently, not only standard trajectory-based or population-based heuristic optimization algorithms but also generic or problem-specific extensions as well as hybrid algorithms can be easily realized.

Additionally, users can also define combined operators that contain a complete operator graph. These operators encapsulate more complex functionality and can be reused when building algorithms. Therefore, different layers of abstraction can be introduced. For example, specific combined operators can be defined for tasks which are typically required in evolutionary algorithms such as selecting and recombining solutions or iterating over a population and applying manipulating operators stochastically. Furthermore, it is even possible to define combined operators representing an entire heuristic optimization algorithm which can be applied on different optimization problems as black box solvers. However, it has to be highlighted that combined operators are not implemented statically in the framework but are defined at runtime. Therefore, user-specific algorithm models can be realized which are dedicated to certain heuristic optimization paradigms and provide the required level of abstraction.

- **Graphical User Interface**
  As practitioners, students, and also in some cases heuristic optimization experts might not have comprehensive programming skills, a suitable user interface is required to define, execute, and analyze algorithms. Consequently, a graphical user interface (GUI) is tightly integrated in HeuristicLab. According to the model-view-controller pattern, each HeuristicLab object (e.g., operators, variables, or data values) can provide a view to present itself to the user. However, as graphical visualization of objects usually is a performance critical task, these views are shown and updated on demand. Furthermore, the GUI reduces the required learning effort significantly. Similarly to standard software products, it enables users to apply heuristic optimization algorithms immediately.

- **Parallelism**
  Last but not least parallel execution of algorithms is also respected in HeuristicLab. Dedicated control operators can be used to define parts of an algorithm that should be executed in parallel. These operators can be used anywhere in an algorithm which enables the definition of parallel heuristic optimization methods, as for example global, coarse-grained, or fine-grained
parallel GAs. However, how parallelization is actually done does not depend on the operators but is defined when executing an algorithm by choosing an appropriate execution engine. Several engines are provided in HeuristicLab to execute parallel algorithms for example using multiple threads on a multi-core CPU or multiple computers connected in a network.

Since 2002 HeuristicLab has been used in several projects of the research group founded by Michael Affenzeller and the author. Its suitability for the development, analysis and application of heuristic optimization algorithms in many different domains has been documented in numerous publications. However, the development process of HeuristicLab has not come to an end so far.

In the near future improving runtime efficiency is one of the main tasks which have to be considered. Thereby, one promising approach is to use generative programming techniques. It is planned to develop a code generator to transform operator graphs back into source code. In this way, the full potential and flexibility of the algorithm model can be utilized, but before an algorithm is executed it can be compiled into a single block of code which can be processed more efficiently. By this means, the algorithm model will turn into a domain-specific programming language that can be used to define and evaluate heuristic optimization algorithms very quickly.

Furthermore, as it is very easy to apply and compare different algorithms by using HeuristicLab, it can be quickly identified which heuristic algorithms and which corresponding parameter settings are effective for a certain optimization problem. In order to systematically analyze this information, there are concrete plans within the scope of the recently established research laboratory “Josef Ressel-Centre for Heuristic Optimization (Heureka!)”\(^1\) to store the results of all algorithm runs executed in HeuristicLab in a large database. The ultimate goal of this optimization knowledge base is to identify correlations between heuristic optimization algorithms and solution space characteristics of optimization problems. This information will provide essential clues for the selection of appropriate algorithms and will also encourage the development of new enhanced and hybrid heuristic optimization algorithms in order to solve problems for which no suitable algorithms are known yet.

\(^1\)http://heureka.heuristiclab.com
Bibliography


[78] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


[165] B. F. Voigt. *Der Handlungsreisende, wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein*. Von einem alten Commis Voyageur, 1831.


List of Tables

4.1  Comparison of existing heuristic optimization software systems . . 40

5.1  Evaluation and comparison of HeuristicLab . . . . . . . . . . . . . 68
5.2  Runtime of HeuristicLab 3.0 compared to HeuristicLab 1.1 (in sec-
    onds) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.3  SGA parameter settings used in the runtime tests . . . . . . . . 70
List of Figures

3.1 Relative importance of requirements for each user group 30
4.1 Basic class relationships of Templar (taken from [100]) 33
4.2 Architecture of the Open BEAGLE framework (adapted from [76]) 36
4.3 Levels of abstraction in EasyLocal++ (taken from [54]) 39
5.1 Interaction of HeuristicLab 1.1 classes 45
5.2 Screenshot of HeuristicLab 1.1 46
5.3 Screenshot of HeuristicLab 2.0 48
5.4 OSGiTMD architecture (taken from [132]) 52
5.5 Economic cycle of plugin-based software systems 53
5.6 Layered structure of the HL3 plugin infrastructure 54
5.7 Compound views representing complex objects 56
5.8 Screenshot of HeuristicLab 3.0 57
5.9 HL3 object model 58
5.10 HL3 algorithm model 64
5.11 Layers of user interaction in HL3 67
6.1 Operator graphs representing a do-while and a repeat-until loop 80
6.2 General principle of recombination operators 83
6.3 General principle of selection operators 86
6.4 General principle of reduction operators 87
6.5 Operator graph of SequentialSolutionsProcessor 92
6.6 Operator graph of SelectedSolutionsManipulator 93
6.7 Operator graph of SolutionsTruncator 93
6.8 Operator graph of SolutionInitializer 94
6.9 Operator graph of HillClimber 95
6.10 Operator graph of SimulatedAnnealing 97
6.11 Operator graph of EvolutionStrategy 99
6.12 Operator graph of AdvancedSelectedSolutionsManipulator 102
6.13 Operator graph of ElitesReplacement 102
6.14 Operator graph of GeneticAlgorithm 104
6.15 Operator graph of SteadyStateReplacement 105
6.16 Scope manipulation of IntermediateSubScopesCreator . . . . . . . 106
6.17 Operator graph of LocalSearchHybridizationAdapter . . . . . . . 107
6.18 Operator graph of ParallelEvaluator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
6.19 Different levels of parallelization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
## List of Listings

2.1 Iterative improvement local search (adapted from [35]) . . . . . . . 8  
2.2 Simulated annealing (adapted from [35]) . . . . . . . . . . . . . . 9  
2.3 Tabu search (adapted from [35]) . . . . . . . . . . . . . . . . . . . 11  
2.4 Variable neighborhood search (adapted from [35]) . . . . . . . . . 12  
2.5 Iterated local search (adapted from [35]) . . . . . . . . . . . . . . 13  
2.6 Evolutionary algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 14  
2.7 Scatter search (adapted from [85, 35]) . . . . . . . . . . . . . . . 16  
2.8 Ant colony optimization (adapted from [32]) . . . . . . . . . . . . 18  
2.9 Particle swarm optimization . . . . . . . . . . . . . . . . . . . . . . 19  
5.1 Main loop of HL3 engines . . . . . . . . . . . . . . . . . . . . . . . 63  
6.1 EmptyOperator . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76  
6.2 Counter . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77  
6.3 SequentialProcessor . . . . . . . . . . . . . . . . . . . . . . . . . 78  
6.4 ParallelProcessor . . . . . . . . . . . . . . . . . . . . . . . . . . . 79  
6.5 Greedy local search (first improvement hill climbing) . . . . . . . 91  
6.6 Simulated annealing . . . . . . . . . . . . . . . . . . . . . . . . . . 96  
6.7 Evolution strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . 98  
6.8 Genetic algorithm with elitism and generational replacement . . . 101
Curriculum Vitae

Personal Data

Name          Dipl.-Ing. Stefan Wagner
Position      Associate Professor for Software Project Engineering
              Heuristic and Evolutionary Algorithms Laboratory
              School of Informatics/Communications/Media
              Upper Austria University of Applied Sciences
              Softwarepark 11, A-4232 Hagenberg, Austria

Address       Orionstraße 21a, A-4030 Linz, Austria
E-mail        stefan.wagner@fh-hagenberg.at
Date of Birth January 23rd, 1979
Place of Birth Bad Ischl, Austria
Nationality   Austrian
Marital Status Unmarried

Education

1985 - 1989  Elementary school
            Volkschule Römerberg, Linz, Austria
1989 - 1993  Secondary school
            Realgymnasium Kollegium Alloisianum, Linz, Austria
1993 - 1997  High school
            Realgymnasium Kollegium Alloisianum, Linz, Austria
1997 - 1998  Bachelor studies in industrial mathematics
            Johannes Kepler University, Linz, Austria
1999 - 2004  Bachelor and master studies in computer science
            Johannes Kepler University, Linz, Austria
October 2004 Masters degree in computer science (with distinction)
2004 - 2009  PhD studies in engineering sciences
            Johannes Kepler University, Linz, Austria

Professional Career

1998 - 2000  Web developer
            Rosenbauer Int. AG, Leonding, Austria
1998 - 2002  Tutor for computer science
            Johannes Kepler University, Linz, Austria
since 2002  Head developer of HeuristicLab
(see http://www.heuristiclab.com)
since 2003  Member of Dr. Michael Affenzeller’s research group
“Heuristic and Evolutionary Algorithms Laboratory”
2003 - 2004  Junior researcher
“Strategic Project 4.3 - Identification of Nonlinear Structures”,
Linz Center of Competence in Mechatronics, Linz, Austria
2005  External lecturer
Upper Austria University of Applied Sciences, Hagenberg, Austria
since 2005  Associate professor for software project engineering
Upper Austria University of Applied Sciences, Hagenberg, Austria
2007 - 2009  Project manager and senior researcher
Research project “Emergency Mission Control Center (emc²)”
funded by the Austrian Research Promotion Agency (FFG)
2007 - 2009  Project manager and senior researcher
Research project “Intelligent Production Steering (IPS)”
funded by the Mechatronics Cluster Upper Austria
since 2008  Deputy head and senior researcher
Josef Ressel-Centre “HEUREKA! - Heuristic Optimization”
funded by the Austrian Research Promotion Agency (FFG)

Civilian Service
1998 - 1999  Fire brigade head quarters, Linz, Austria

Invited Talks
August 2008  Guest lecturer at the University of Technology, Sydney, Australia

Awards
2001  Scholarship for excellent merits in studies
Technical Faculty, Johannes Kepler University, Linz, Austria
2003  Scholarship for excellent merits in studies
Technical Faculty, Johannes Kepler University, Linz, Austria
2004  Best paper award of EMCSR 2004 for the contribution
2005  Best paper award of WMSCI 2005 for the contribution
“Solving Multiclass Classification Problems by Genetic Programming” in the session “Management Information Systems”

2005

Best paper award of WMSCI 2005 for the contribution

“SexualGA: Gender-Specific Selection for Genetic Algorithms”
in the session “Artificial Intelligence / Expert Systems”

2007

Best academic teacher award 2006/2007, third place

Department of Software Engineering, Upper Austria University of Applied Sciences, Hagenberg, Austria

Publications

Books


Book Chapters


Journal Articles


**Conference Papers**


Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, März 2009

Dipl.-Ing. Stefan Wagner