HeuristicLab 3.3: A unified approach to metaheuristic optimization

Stefan Wagner, Andreas Beham, Gabriel Kronberger,
Michael Kommenda, Erik Pitzer, Monika Kofler, Stefan Vonolfen,
Stephan Winkler, Viktoria Dorfer, Michael Affenzeller

Josef Ressel-Centre HEUREKA! for Heuristic Optimization
School of Informatics, Communications and Media - Hagenberg
Upper Austria University of Applied Sciences

{swagner, abeham, gkronber, mkommend, epitzer, mkofler,
svonolve, swinkler, vdorfer, maffenze}@heuristiclab.com

Abstract

The awareness of heuristic methods as optimization tools and their, in comparison to exact algorithms, quick and simple application has expanded to many different domains in the recent history. In the course of these developments the user base of heuristic optimization methods has also grown from mathematicians and computer scientists to practitioners in virtually every field. To facilitate the application of heuristic optimizers in domains where no computer-scientist has gone before, a number of more or less advanced software frameworks exists. In this paper the authors introduce a new version of their software environment HeuristicLab which aims to provide a comprehensive solution for algorithm development, testing, analysis and generally the optimization of complex problems.

1 Motivation

Researchers of different domains often speak different languages and certainly in almost every interdisciplinary project one quickly reaches the point where this difference becomes obvious and where the cooperating partners learn to develop a common language. The application of heuristic optimization is an inherently interdisciplinary task as the real world problem is often in a different domain which makes this kind of research particularly challenging from an optimization expert's point of view: Optimization knowledge is extended by learning about the domain. On the other hand often the domain experts themselves want to get more involved in the practical aspects of optimization theory and make use of that knowledge in the future.

In the opinion of the authors, this interplay and cooperation between experts of different domains demands for appropriate software support. With HeuristicLab the authors have created not only a framework but a complete environment that integrates the problem aspects and the algorithmic details. It is important to have on the one hand a rich environment with lots of ready to use modules and on the other hand a graphical user interface (GUI) for visualizing the aspects in each domain. Also it is clear that collaboration through software requires developing parts of the same software and so the foundation of HeuristicLab is a flexible plugin-based architecture that allows independent development and flexible combination of various parts.

Another source of motivation and also often inspiration of HeuristicLab is the task of teaching heuristic optimization techniques to students. A ready to use environment which features testing, exploring and playing with algorithms, problems and algorithm design attracts the students' interest in the topic. Optimization concepts such as local search, local minima and motivation for finding solutions to complex problems can be easily explained with practical examples. Students are more eager to explore the field of heuristic optimization. They can be motivated by competing against each other in searching for well working combinations of algorithms, problems and their parameters in a comfortable GUI.

Generally speaking, it is the motivation of the authors to help bringing people of different domains together, provide them with a unified problem solving environment and to create an enjoying experience when teaching heuristic optimization methods. The result of this motivation is the software environment HeuristicLab.

In the following a very brief overview of some of the existing frameworks and environments for heuristic optimization is given. Then the core concepts of HeuristicLab and its modeling approach are explained and a case study of algorithm modeling is presented. Using the HeuristicLab algorithm model a simple local search algorithm is described and it is shown how to extend it to tabu search. Finally, the benefits and drawbacks of HeuristicLab are discussed and some conclusions and topics of future work are stated.

2 Overview of heuristic optimization environments

There are several frameworks for heuristic optimization freely available on the internet. It would be beyond the scope of this publication to list and review them all, so just a short overview is given. For a more complete comparison and more details to each of them refer to [1], [2], [3], and the respective reference given for each framework.

2.1 HotFrame

HotFrame (Heuristic OpTimization FRAMEwork) is an object-oriented framework developed in C++ by Andreas Fink and his colleagues at the University of Hamburg [4]. 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. The authors of HotFrame are aware that "[..]there is no way to apply the framework [for new applications] without having a good command of C++ (including templates)[..]"[1].

2.2 Evolving Objects (EO)

EO 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 [5]. 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. On top of EO there also exist several other frameworks such as PardisEO [6] which provides parallel and distributed execution of algorithms.

2.3 Open BEAGLE

Similarly to EO, also the Open BEAGLE [7] project aims at implementing a C++ framework for evolutionary computation. It is mainly driven by Christian Gagné and Marc Parizeau at the Computer Vision and Systems Laboratory of Laval University in Canada.

Open BEAGLE uses XML configuration files that 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

[1] `http://www1.uni-hamburg.de/IWI/hotframe/`, April 26th, 2010

can be changed by manipulating configuration files instead of recompiling the application. However, an interactive GUI that allows dynamic configuration of algorithms and on-the-fly analysis during execution is still missing.

2.4 ECJ

ECJ[2] is a Java-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.

The basic design of ECJ is quite similar to Open BEAGLE. 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. It has a basic GUI for loading and manipulating configuration files, executing algorithms, and visualizing results.

2.5 Summary

Several frameworks do not make more than basic use of a GUI, and in many cases algorithm development strictly requires knowledge about a certain programming language and its features. Open BEAGLE and ECJ already allow designing algorithms without programming, but still at a basic level. HeuristicLab tries to go much further in this direction and provides automation and analysis features that would go beyond the scope of this paper.

3 HeuristicLab 3.3

In the course of the development of HeuristicLab, from the first major version to the most recent version 3.3, the architecture has gone through several major changes. It is written in C# and uses the Microsoft .NET framework. The windows platform has historically provided a well rounded base for GUI applications, and a rich GUI experience is one of the requirements of HeuristicLab. Similar to many other frameworks the separation of algorithms and problems was one of the initial core concepts, but over the development from version 1.0 to 3.3 even finer levels of separation broken down into basic operations have been introduced. An algorithm thus has transformed from a block of code that has to be compiled to a customizable graph of operations that can be specified and modified even down to very detailed levels in the GUI itself. The experimentation possibilities allow defining, executing, and analyzing hundreds of runs of algorithms with different parameter settings in one place.

[2] `http://cs.gmu.edu/~eclab/projects/ecj/`, April 26th, 2010

3.1 Software environment

HeuristicLab 3.3 builds upon a plugin architecture so that it can be extended without developers having to touch the original source code [8].

Plugins are used as architectural paradigm. A lightweight plugin concept is implemented in HeuristicLab 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.

Every HeuristicLab feature, even the Optimizer user interface, is available as a plugin. It is possible to easily derive completely customized GUIs tailored to very specific applications or bundle specific plugins together to packages, so that users can be given exactly the functionality they need.

3.2 Algorithm model

The core of HeuristicLab is its algorithm model [9]. It is very easy to understand on an abstract level, but naturally reveals more complexity the deeper one descends into it.

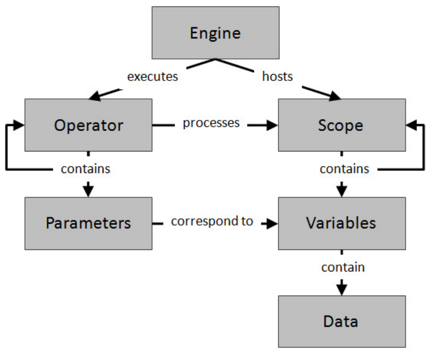

Figure 1: HeuristicLab 3.3 algorithm model

However, luckily for most users, understanding more than the abstract level of the core language is not necessary for applying heuristic optimization.

The algorithm model splits into three distinct models itself: A data model, an operator model, and an execution model which shall be explained in more detail.

In the **data model** each value is represented as an object that can be *saved, restored and viewed*. Standard data types such as integers, doubles, strings, or arrays that do not offer these properties are wrapped.

These values are linked to a name by storing them in a **variable**. 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.

Finally, these scopes can be hierarchically organized as trees. A scope may have several so-called sub-scopes, much like a metaheuristic population has many solutions which again might have several components.

The next part of the algorithm model considers the representation of single steps or statements in form of the **operator model**.

Each algorithm is a sequence of clearly defined, unambiguous and executable instructions. In the HeuristicLab 3.3 algorithm model these atomic building blocks are called **operators** and are also represented as HeuristicLab 3.3 objects. Operators fulfill two major tasks: On the one hand, they are applied on scopes to access and manipulate variables and subscopes. On the other hand, operators define which operators are executed next.

Regarding the manipulation of variables, the approach used in the HeuristicLab 3.3 operator model is similar to formal and actual parameters of procedures. Each operator defines **parameters** for each variable it expects and possibly manipulates. By this means, operators are able to encapsulate functionality in an abstract way: A formal name is defined for a parameter which is used inside the operator. But after instantiating a concrete operator in the GUI and adding it to an algorithm at run time, the user defines the actual name of the parameter. The original code of the operator does not have to be modified. The implementation of the operator is decoupled from concrete variables and can be reused.

The execution of algorithms is the last aspect which has to be defined in the HeuristicLab 3.3 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. 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. These pending operations are kept in a stack. In each iteration, an engine pops the next operation from the top of its stack, executes the operator on the scope, and pushes returned successor operations back on the stack.

A graphical summary of these basic concepts of the HeuristicLab algorithm model is shown in Figure 1.

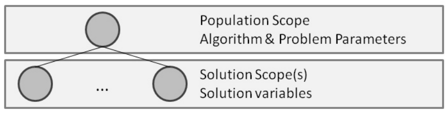

Figure 2: Basic scope structure for most population-based algorithms

3.3 Basic optimization concepts

With the algorithm model in mind, it will now be shown how to realize basic operations that often occur in heuristic optimization algorithms [10]. The hierarchical structure of scopes is used to define a population, a solution and several operations that modify the scope structure to perform the required operations. The base structure for a population-based algorithm such as in this example is given in Figure 2.

For this example consider a simple genetic algorithm that has a population of solutions and **selects** among these solutions those that should be crossed. After selection, the children are **crossed** which results in two populations of the same size: The old parent population and the new child population. Finally, after mutating some children with a given probability, the parent population is replaced, i.e., it is **reduced** to the child population. Figures 3 and 4 show how these operations transform the scope structure.

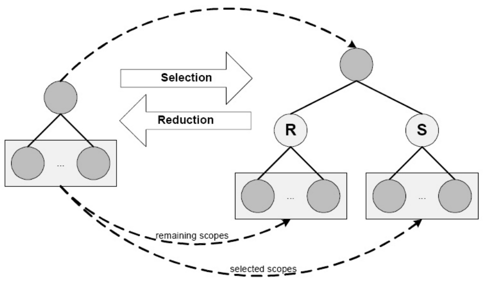

Figure 3: General principle of selection and reduction operators

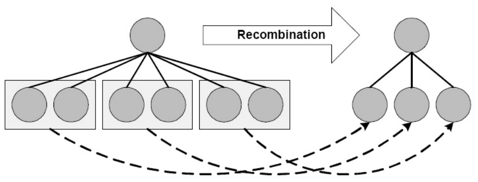

Figure 4: General principle of crossover operators

3.4 Modeling toolbox

There are many different operators implemented in HeuristicLab 3.3. They can be put in any order to model the behavior of heuristic optimization algorithms. In this section some of the basic and most common operators are introduced.

Some of these operators implement behavior known from many programming languages such as for example branches. In HeuristicLab 3.3 there are two types of branches: A *ConditionalBranch*, and a *StochasticBranch*. The ConditionalBranch will apply either the operator that is connected to its TrueBranch point or the one connected to its FalseBranch point depending on the value of a Boolean variable. It is often used together with a *Comparator* that compares two variables for $<, \leq, =, \geq$, or $>$ and stores the result of the comparison as a Boolean variable. The StochasticBranch applies the operator connected to its FirstBranch point with probability p and the operator connected to its SecondBranch point with probability $1 - p$. Additionally there is a *StochasticMultiBranch* when there are more than two branches that should be executed with a given, but not necessarily equal, probability.

Operators performing iterations over subscopes are also often used. For this purpose there exist a *SubScopesProcessor* and a *UniformSubScopesProcessor*. The later one applies a single operator on each sub-scope, while the first one applies a different operator on each sub-scope. This is also one of the points where parallelism can be introduced easily as the operations on each sub-scope are in most cases independent of each other and therefor can be executed concurrently. Naturally, there is also an operator to create sub-scopes (*SubScopesCreator*) as well as to remove a single or all sub-scopes (*SubScopesRemover*). Finally there is also a *SubScopesSorter* which rearranges the order of the sub-scopes given the value of a certain variable.

Loops in HeuristicLab 3.3 are modeled through the successors of operators. When a certain operator returns one of its preceding operators as next operator to be executed, a loop has been created. Naturally, one should make sure there is an exit to the loop, so typically a branch is used at the beginning or end of the loop that loops while a certain condition is given.

Finally, simple counting operators exist as well so that one can increase the number of generations or iterations that have been performed. Although of course there are many more and rich operators available, this is a good base to start designing algorithms.

3.5 Modularization concept

Reusability of algorithm modules or algorithm parts is another concern in HeuristicLab 3.3. Users do not have to start modeling every algorithm from scratch, but can **reuse** parts that have already been modeled. Additionally, it is possible to model an algorithm generally and to decide later on for which problem and with which operators the algorithm should be **parameterized**. In HeuristicLab 3.3 reusing parts is achieved through the concept of *CombinedOperators* which are operators that contain an operator graph and thus encapsulate possibly complex functionality as a single operator. Basically, any algorithm given in HeuristicLab is such a CombinedOperator.

Parameterizing such parts is then possible through special operators called *Placeholders*. These operators fetch another operator, probably given as parameter of the algorithm, and execute that one in place of itself. This is useful when e.g. a mutation should occur at a certain point in the algorithm, but the algorithm designer uses a Placeholder and thus creates an option in the algorithm that can be parameterized with probably many different mutation operators.

4 Examples of metaheuristics

4.1 Local search

Figure 5 shows the operator graph of a simple local search algorithm. It starts at the *VariableCreator* operator which writes several variables with initial values into the scope. It then proceeds applying several operators to each sub-scope (*UniformSubScopesProcessor*). This local search module expects one or more solutions to be present as sub-scopes below the current scope. Below the solution scope it then generates some moves according to a specified neighborhood using the *MoveGenerator*. Each of these moves is put in a sub-scope. So afterwards another *UniformSubScopesProcessor* applies the *MoveEvaluator* on each move. This will inject a variable describing the fitness of each move. Finally the best move is selected using the *BestSelector*. As per the selection principle shown above, this creates two sub-scopes with all the remaining moves to the left and in this case the single selected move in a right branch. All other moves are then forgotten, as the *RightReducer* throws away the left and thus remaining sub-scopes, and the best move is applied if and only if (*ConditionalBranch*) it is better than the best solution found so far (*QualityComparator*). It then counts the iterations and loops again until it reaches a maximum amount of iterations. With just a little understanding of the basic operators like SubScopesProcessor, Selection, and Reduction it is very easy to read what an algorithm will do and evaluate its correctness. Additionally, all operators have descriptions, and so even encountering a new operator would not present much of a problem to the user.

4.2 Tabu search

When this algorithm should be extended to a tabu search, the following parts have to be added: a tabu list, a check whether a move is tabu, an operator that sets a move tabu, and an additional stop criterion when the neighborhood is empty as all moves are tabu. Figure 6 shows the operator graph with the

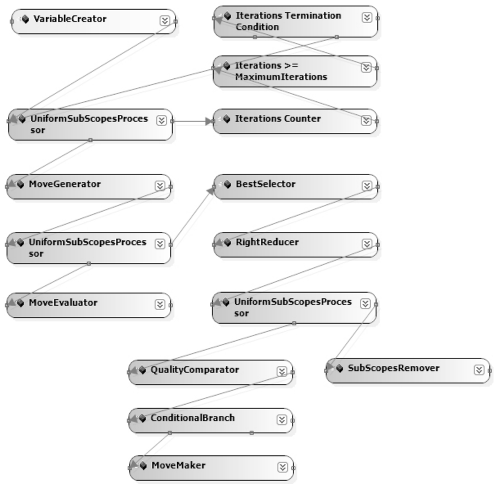

Figure 5: Operator graph of a local search main loop as modeled in HeuristicLab 3.3

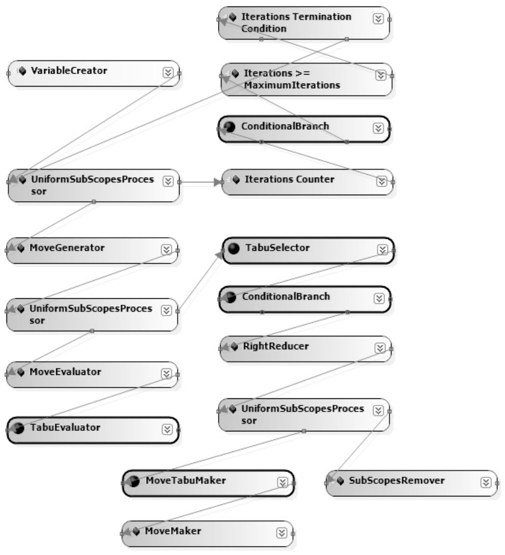

Figure 6: Operator graph of a tabu search main loop derived from the local search given above

changes marked. The first thing that changed is that the *MoveEvaluator* is followed by a *TabuEvaluator* that checks if the given move is tabu or not. Next, the *BestSelector* is replaced by a *TabuSelector* which does not select the best move, but the best move that is not tabu. The tabu list as such is a list element (or a more complex data structure) that is simply added to the *VariableCreator* and does not necessarily change the operator graph. Also it is visible that the quality comparison of the local search has been replaced by a *TabuMaker*, an operator that declares a move tabu. Finally, the termination criterion is added, once after the selection of the next move to prevent performing any moves and once before continuing with the main loop.

5 Benefits and Drawbacks

Up to now, the only major drawback of HeuristicLab 3.3 is runtime performance. Naturally, live updating of the GUI and the flexible algorithm model result in some runtime overhead and the authors are aware that their optimization environment is not the fastest. It is never possible to beat a framework that is targeted and succeeds at achieving best runtime performance, but likewise such a framework will never beat the flexibility and comfort of HeuristicLab. There are several frameworks available, some are optimized for runtime performance, but none of these frameworks offer such a rich GUI experience as HeuristicLab, at least as far as it is known by the authors.

In the opinion of the authors, the problem of runtime performance can be mitigated in two ways: First, as the problem size increases, the operators spend more time executing their code and thus benefit more and more from the speed of the runtime environment itself and the compiler optimizations present therein. And second, any algorithm designed, tested, and analyzed in HeuristicLab can be implemented quickly for performance critical applications later on.

Unfortunately, topics such as execution of algorithms and results analysis could not be covered in the scope of this paper. It shall be noted, that each execution of an algorithm will produce a run object which contains all the parameters and results such as solution quality,

quality progress charts, problem specific visualization, etc. Many such runs form a run collection which can be filtered and graphically analyzed as bubble chart. Statistical analysis is not yet part of HeuristicLab 3.3, but the run collection can also be viewed in a table which can easily be copied to familiar programs such as Excel® for further analysis.

6 Conclusions and future work

The authors have described the next major version of their optimization environment HeuristicLab, the algorithm model and have shown in a case study how to design and extend algorithms. They will continue to enhance HeuristicLab and extend the already powerful library with more benchmark problems, and even more algorithms described in the literature. The software framework is released under the GNU General Public License (GPL). The source code as well as a binary version of the environment can be downloaded from the website[3]. Interested readers may also find screenshots and other development related material there.

Acknowledgements

The work described in this paper was done within the Josef Ressel-Centre HEUREKA! for Heuristic Optimization sponsored by the Austrian Research Promotion Agency (FFG).

[3]`http://dev.heuristiclab.com`

References

[1] S. Voß and D. L. Woodruff, Eds., *Optimization Software Class Libraries.* Kluwer, 2002.

[2] C. Gagné and M. Parizeau, "Genericity in evolutionary computation software tools: Principles and case-study," *International Journal on Artificial Intelligence Tools*, vol. 15, no. 2, pp. 173–194, 2006.

[3] S. Wagner, "Heuristic optimization software systems - Modeling of heuristic optimization algorithms in the HeuristicLab software environment," Ph.D. dissertation, Johannes Kepler University, Linz, Austria, 2009.

[4] A. Fink and S. Voß, "HotFrame: A heuristic optimization framework," in *Optimization Software Class Libraries.* Kluwer, 2002, pp. 81–154.

[5] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer, "Evolving Objects: A general purpose evolutionary computation library," in *EA-01, Evolution Artificielle, 5th International Concerence in Evolutionary Algorithms*, 2001, pp. 231–242.

[6] S. Cahon, N. Melab, and E.-G. Talbi, "ParadisEO: A framework for reusable design of parallel and distributed metaheuristics," *Journal of Heuristics*, vol. 10, no. 3, pp. 357–380, 2004.

[7] C. Gagné and M. Parizeau, "Open BEAGLE: A new C++ evolutionary computation framework," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO) 2002.* Morgan Kaufmann, 2002, p. 888.

[8] S. Wagner, S. Winkler, R. Braune, G. Kronberger, A. Beham, and M. Affenzeller, "Benefits of plugin-based heuristic optimization software systems," in *Computer Aided Systems Theory - EUROCAST 2007*, ser. Lecture Notes in Computer Science, vol. 4739. Springer, 2007, pp. 747–754.

[9] S. Wagner, G. Kronberger, A. Beham, S. Winkler, and M. Affenzeller, "Modeling of heuristic optimization algorithms," in *Proceedings of the 20th European Modeling and Simulation Symposium.* DIPTEM University of Genova, 2008, pp. 106–111.

[10] S. Wagner et al., "Model driven rapid prototyping of heuristic optimization algorithms," in *Computer Aided Systems Theory - EUROCAST 2009*, ser. Lecture Notes in Computer Science, vol. 5717. Springer, 2009, pp. 729–736.