Model Driven Rapid Prototyping of Heuristic Optimization Algorithms

Stefan Wagner, Gabriel Kronberger, Andreas Beham, Stephan Winkler, Michael Affenzeller

Heuristic and Evolutionary Algorithms Laboratory
School of Informatics, Communications and Media - Hagenberg
Upper Austria University of Applied Sciences
Softwarepark 11, A-4232 Hagenberg, Austria
{swagner,gkronber,abeham,swinkler,maffenze}@heuristiclab.com

Abstract. In this paper the authors describe a model driven approach for the development of heuristic optimization algorithms. Based on a generic algorithm model, several operators are presented which can be used as algorithm building blocks. In combination with a graphical user interface, this approach provides an interactive and declarative way of engineering complex optimization heuristics. By this means, it also enables users with little programming experience to develop, tune, test, and analyze heuristic optimization techniques.

1 Introduction

Since the beginning of the 1990s heuristic optimization is a very active field of research. Many different algorithms have been developed that were applied to optimization problems of numerous domains. Thereby, especially the development and application of metaheuristic algorithms is a favored approach, as such algorithms can be easily reused for many different problems [2].

However, according to the No Free Lunch theorem for heuristic search (see for example [5]), no single heuristic optimization algorithm is able to outperform all other algorithms for all possible problems. This simple fact led to a magnitude of different heuristic optimization paradigms, which often adapt strategies found in nature to solve technical optimization problems (e.g., evolutionary algorithms, simulated annealing, ant colony optimization, particle swarm optimization). All these algorithms have specific characteristics and are suitable for different problem scenarios and solution space topologies.

As a consequence, selecting a suitable algorithm for a given optimization problem is a cumbersome task. Extensive empirical testing is required to compare different optimization paradigms and to select appropriate parameter values. Furthermore, in many cases of hard real-world optimization problems it is not

The work described in this paper was done within HEUREKA!, the Josef Ressel centre for heuristic optimization sponsored by the Austrian Research Promotion Agency (FFG). Visit http://heureka.heuristiclab.com for more details.

enough to tune parameters in order to achieve satisfactory results. Approaches of different paradigms have to be combined, leading to hybrid algorithms or even new classes of metaheuristic optimization techniques.

For this reason, heuristic optimization software systems have to support rapid prototyping and testing of algorithms. By providing a set of ready-to-use components and by supporting data analysis and visualization, the process of assembling heuristic optimization algorithms can be simplified. However, for combining these components profound knowledge of the underlying framework and good programming skills are required in most cases. Therefore, algorithm development is usually restricted to software engineers who have to bridge the gap between heuristic optimization experts and experts in the problem domains.

In order to overcome this drawback, the authors propose a model driven approach to heuristic algorithm engineering. Based on a generic algorithm model developed by the authors in a previous work [4], several operators for representing heuristic optimization algorithms are discussed in this paper. In contrast to similar ideas in the heuristic optimization community (for example EASEA [1]), modeling is thereby abstracted from a concrete programming language as far as possible. Additionally, by providing a graphical user interface, development and testing of heuristic optimization algorithms becomes possible even for users with little experience or interest in programming and software development.

2 Generic Algorithm Model

In [4] the authors introduced a generic model for representing arbitrary algorithms. In this model algorithms are described as operator graphs, whereby each operator represents a single operation. The data manipulated by an algorithm is structured in hierarchical data structures called scopes. When an algorithm is executed by an engine, its operators are applied on the scopes in order to access and manipulate variables or sub-scopes. For additional details about the model and for a detailed description of its benefits with respect to the representation of parallel algorithms, the reader is referred to [4].

3 Modeling Heuristic Optimization Algorithms

Based on the generic algorithm model, several operators are presented in the following which can be used as building blocks for defining heuristic optimization algorithms.

3.1 Basic Operators

VariableInjector: Each algorithm execution starts with applying an initial operator on an empty global scope. Therefore, an operator is required for adding user-defined variables to a scope. This task is fulfilled by the operator *VariableInjector* which can be used for example to add global variables such as the

population or neighborhood size, the mutation rate, the tabu tenure, the maximum number of iterations, or the random number generator. Each succeeding operator will then be able to access these values.

Manipulation Operators: Manipulation operators are basic operators that change the variables of a scope. For example, a *Counter* operator can be used for incrementing variables. Furthermore, custom manipulations operators can be defined which represent problem-specific or encoding-specific manipulation operations.

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 the type of comparison (e.g., less, equal, greater or equal). After retrieving both variable values from the scope, it creates a new Boolean variable containing the result of the comparison and adds it to the scope.

SubScopesCreater: The operator *SubScopesCreator* is used to extend the scope tree. It expects an input variable specifying how many new sub-scopes should be appended to scope it is applied on.

3.2 Control Operators

Operators have to define which operations an engine has to execute next (i.e., which operators are applied on which scopes). Therefore, the execution flow of an algorithm can be defined using control operators that do not manipulate variables or scopes but return successor operations. In the following, basic control operators are presented to model sequences and branches. Any other control structure known from classical programming languages (for example switch statements or loops) can be realized by combining these operators.

SequentialProcessor: *SequentialProcessor* represents a sequence of operations. It tells the engine to apply all its sub-operators on the current scope.

UniformSequentialSubScopesProcessor: This operator can be used to navigate through the hierarchy levels of the scope tree. It tells the engine to apply its sub-operator on each sub-scope of the current scope. As each solution is represented as a scope, this operator can for example be used for evaluating or manipulating a set of solutions.

SequentialSubScopesProcessor: *SequentialSubScopesProcessor* can also be used to apply operators to sub-scopes. However, in contrast to the UniformSequentialSubScopesProcessor, it provides a sub-operator for each sub-scope which enables 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, it tells the engine to apply either its first sub-operator (true branch) or its second sub-operator (false branch) on the current scope.

3.3 Selection and Reduction

Seen from an abstract point of view, a large group of heuristic optimization algorithms (improvement heuristics) follows a common strategy: In an initialization step one or more solutions are generated either randomly or using construction heuristics. Then these solutions are 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.

As each solution is represented as a scope and scopes are hierarchically organized, selection and replacement operations can be realized in a straight forward 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. 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. Depending on the type of the reduction operator, this reunification step may also include elimination of some sub-scopes that are no longer required. The general principle of selection and reduction operators is schematically shown in Figure 1.

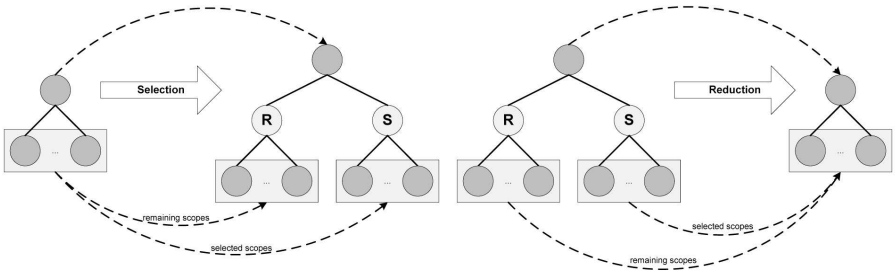

Fig. 1. General principle of selection and reduction operators

According to this simple principle of selection and reduction of solutions, a set of selection and reduction operators can be defined which can be used as a

basis for realizing complex selection and replacement schemes. These operators are described in the following.

Selection Operators: The most trivial form of selection operators are the two operators *LeftSelector* and *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, a *RandomSelector* can be used which additionally expects a random number generator 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, several classical quality-based selection schemes well-known from the area of evolutionary algorithms can be realized as well, as for example fitness proportional selection (*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.

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* reunites both scope groups by merging all sub-scopes.

Sorting Operators: Many 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 subscopes 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 ways of ordering can be implemented.

3.4 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.

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.

The two operators *CombinedOperator* and *OperatorExtractor* are responsible for modularization of operator graphs and are described in detail in the following. An important aspect is that 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 algorithm model, but depends only on its users.

CombinedOperator: A *CombinedOperator* contains a whole operator graph. When executed it tells the engine to apply the initial operator of its graph on the current scope. 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, the user has 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 also considered as data, 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 required manipulation and evaluation operator, but the contained op-

erator 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 containing an operator. The operator looks for that variable in the scope tree recursively and tells the engine to apply the contained operator on the current scope.

4 Interactive Algorithm Engineering

All operators discussed in the previous section have been implemented by the authors in the HeuristicLab[1] optimization environment [3]. In combination with the graphical user interface of HeuristicLab (see Figure 2), this enables a declarative and interactive definition of complex heuristic optimization algorithms. By this means, also users with little or no programming experience can easily develop, tune, test, and analyze sequential and parallel metaheuristic optimization techniques.

Therefore, the generic algorithm model and the operators discussed in this paper provide a basis for bridging the gap between heuristic optimization experts, practitioners of different application domains, and software engineers. On the one hand, experts can use basic combined operators representing 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 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.

5 Conclusion

Based on the generic algorithm model described in [4], several operators have been discussed in this paper that can be used to define heuristic optimization algorithms. Due to the page restriction of this contribution, problem specific concepts such as different solution encodings or evaluation and manipulation operators have not been covered in detail. However, these aspects will be described in subsequent publications and it will be shown that they can also be realized easily as additional data elements and operators. Furthermore, concrete case studies will also be presented in the future to demonstrate how different

[1] `http://www.heuristiclab.com`

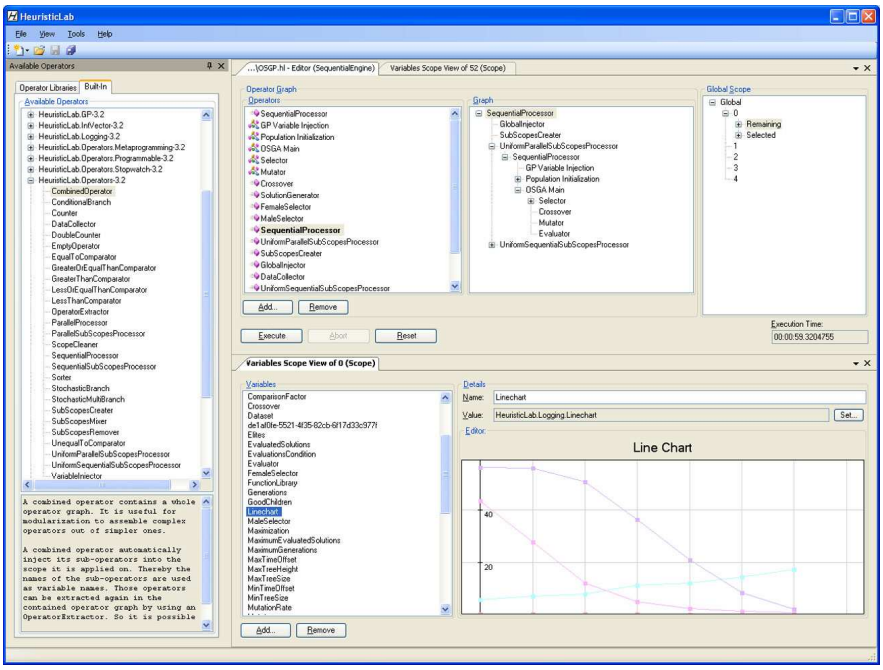

Fig. 2. Screenshot of HeuristicLab 3.0

metaheuristic algorithms such as genetic algorithms or simulated annealing can be modeled.

References

1. P. Collet, E. Lutton, M. Schoenauer, and J. Louchet. Take it EASEA. In *Parallel Problem Solving from Nature PPSN VI*, volume 1917 of *Lecture Notes in Computer Science*, pages 891–901. Springer, 2000.
2. K. F. Doerner, M. Gendreau, P. Greistorfer, W. Gutjahr, R. F. Hartl, and M. Reimann, editors. *Metaheuristics: Progress in Complex Systems Optimization*. Operations Research/Computer Science Interfaces Series. Springer, 2007.
3. S. Wagner. *Heuristic Optimization Software Systems - Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*. PhD thesis, Johannes Kepler University, Linz, Austria, 2009.
4. S. Wagner, G. Kronberger, A. Beham, S. Winkler, and M. Affenzeller. Modeling of heuristic optimization algorithms. In A. Bruzzone, F. Longo, M. A. Piera, R. M. Aguilar, and C. Frydman, editors, *Proceedings of the 20th European Modeling and Simulation Symposium*, pages 106–111. DIPTEM University of Genova, 2008.
5. D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.