Large Scale Parameter Meta-Optimization of Metaheuristic Optimization Algorithms with HeuristicLab Hive

Christoph Neumüller, Andreas Scheibenpflug, Stefan Wagner,
Andreas Beham, Michael Affenzeller

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

c.neumueller@gmail.com, {ascheibe, swagner, abeham, maffenze}@heuristiclab.com

Abstract— **In the recent decades many different metaheuristic algorithms have been developed and applied to various problems. According to the *no free lunch* theorem no single algorithm exists that can solve all problems better than all other algorithms. This is one of the reasons why metaheuristic algorithms often have parameters which allow them to change their behavior in a certain range. However, finding good parameter values is not trivial and requires human expertise as well as time. The search for optimal parameter values can be seen as an optimization problem itself which can be solved by a metaheuristic optimization algorithm (*meta-optimization*). In this paper the authors present the meta-optimization implementation for the heuristic optimization environment HeuristicLab. Because meta-optimization is extremely runtime intensive, a distributed computation infrastructure, HeuristicLab Hive, is used and will be described in this paper as well. To demonstrate the effectiveness of the implementation, a number of parameter optimization experiments are performed and analyzed.**

I. Motivation

Most metaheuristic algorithms have a number of behavioral parameters which affect their performance. For most algorithms, there exist established default values for these parameters which are commonly used. However parameter values for an algorithm might work well on one problem instance but not so well on another. Researchers are often in the situation that they need to tune the parameter values for a new problem instance. Unfortunately, finding the best parameter values is not a trivial task and it is difficult to understand the effect of every parameter. Additionally, parameters may not be independent of each other. The change to one parameter can change the effect of other parameters which makes the problem even more complex. Although metaheuristic algorithms exist for decades, a parameterless algorithm which performs well on all problems has not been found yet. According to the *no free lunch* theorem by Wolpert and Macready [1] it is in fact impossible to find such an algorithm which performs better than all other algorithms on all problems. Therefore, for a metaheuristic it is beneficial to have parameters and therefore the ability to adapt to the problem.

The problem of finding the optimal parameters for an algorithm can be seen as an optimization problem. A human expert is able to anticipate good parameters through experience, however novice users of metaheuristics will have to resort to trial-and-error to obtain parameters that achieve the desired results. Depending on the problem this may be a task that takes some time. The idea is to use an optimization algorithm as a meta-level optimizer to find the optimal parameter values of another algorithm. This concept is called *parameter meta-optimization* (PMO). There has been research in the area of PMO in the past, but in most of these approaches, specialized implementations were used for the meta-level algorithms which were not exchangeable. In the following, a flexible and extendable implementation of the PMO concept for the optimization environment HeuristicLab[1](HL) [2] is presented. As the execution of meta-optimization algorithms is highly runtime intensive, the distributed parallelization environment of HeuristicLab, called Hive, is used for the experiments which is also described in this paper.

II. HeuristicLab Hive

HeuristicLab Hive is an elastic, scalable and secure infrastructure for distributed computation. It is part of the open source optimization environment HeuristicLab implemented in C# using version 4.0 of the Microsoft .NET framework. Hive consists of a

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

server with a database and a number of computation *slaves*. A user can upload a job to the Hive server via a web service interface. Then the server distributes the jobs among the available slaves and retrieves the results after they are finished. The following list shows the most important aspects and features of HeuristicLab Hive:

Generic Usage: Though the main purpose of HeuristicLab Hive is to execute HL algorithms, it is designed to be completely independent from HL. Any type of job can be executed on Hive. A job is a .NET object which implements a certain interface. Hive offers a web service to add, control and remove jobs.

Elasticity: Slaves can be added and removed at any time, even while they are calculating jobs. Therefore, jobs have the ability to be paused and persisted. The advantage of an elastic system is to be able to add resources at peak demand times and remove them if they are needed otherwise. There are no automatic snapshots of job results, but a user can pause, modify, and resume a single job any time.

Heterogeneity: Slaves with different properties and constraints are supported. CPU performance and available memory can differ in every slave. Job scheduling respects these constraints. Hive is also independent of the network infrastructure. A slave can be directly connected through a fast high-speed link or it can be far away in a secured company network. The only requirement for slaves is to have the Microsoft .NET 4.0 framework installed.

Time schedules: It is possible to define time schedules for slaves and groups of slaves. A time schedule defines when a slave is allowed to calculate and when it is not. This is particularly useful for using office computers at night which would be unused otherwise.

Plugin-Independency: Slaves are lightweight applications and do not contain the assemblies needed to execute a job. When a job is uploaded, the files and assemblies that are needed to execute the job are automatically discovered and also uploaded. Assemblies and files are grouped in *plugins* which are versioned. Hive takes care of distributing, storing, and caching those files efficiently. The possibility to submit a job with versioned plugins makes Hive independent of already deployed plugins and eliminates the need to update slaves.

Security: Hive puts emphasis on security. Users can control on which slaves their jobs should be computed. Hive ensures confidentiality and integrity of all communication by using X.509 certificates and message level encryption. Since a user can upload custom assemblies to the system, it is crucial that on a slave each job is executed in its own sandbox. The sandbox limits the permissions of the jobs so that they have e.g. no access to the file system.

In the following the main components of HeuristicLab Hive are described:

Server: The server API is exposed as a *Windows Communication Foundation* web service which is hosted in *Microsoft Internet Information Server 7.5*. It offers methods to upload jobs and plugins, to fetch state information about jobs and to download results. It also exposes methods to control and administer the system, including tasks such as grouping slaves or analyzing system performance. The server also decides, if a slave gets another job or not depending on its schedule, free resources and the assignment of resource groups for the jobs.

Slave: Slaves are executed as Windows services. The reason for this choice is that the slave can run no matter which user is logged in and even if no user is logged in. The slave sends periodic messages to the server, reporting which jobs are calculated and how much resources (CPU cores, memory) are available. The server responds with a list of messages containing the next actions for the slave. Communication between the server and its slaves is always initiated by the slaves to avoid problems which may be caused by NAT (network address translation) and firewalls. Figure 1 shows an exemplary infrastructure with slaves deployed on various different locations behind firewalls.

Client: The HL client for Hive features a user interface for uploading and controlling jobs as well as an administrator console. It allows creating and arranging slave groups and observing the slaves' current state. The administrator console also enables a user to define a schedule for slaves or groups of slaves.

Jobs executed with Hive typically contain different parameterizations of one or more algorithms applied to one or more problems. Often each configuration is repeatedly applied to obtain the mean and variance of the optimization results. Theses jobs do no require communication during their execution, therefore, the speed-up scales very well with the number of available slaves. Naturally, there exists a certain overhead in the distribution of the job, as well as the required plugins, but especially for long-running jobs this overhead does not play a significant role. As is described by Gustafson's law [3] the speed-up increases with increasing job sizes.

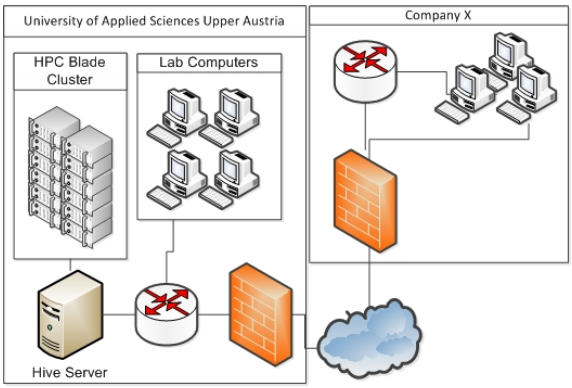

Fig. 1. Exemplary Hive slave deployment

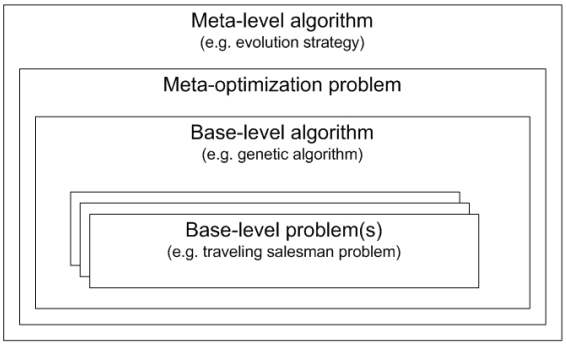

Fig. 2. Meta-optimization concept

III. Meta-Optimization

A. Existing Approaches

Meta-optimization is a technique where an optimization algorithm is applied as a meta-level optimizer to another optimization algorithm. The algorithm which solves the parameter optimization problem is called *meta-level* algorithm, whereas the algorithm which is optimized is called *base-level* algorithm. Figure 2 illustrates the structure of this idea.

Seeing the parameter optimization problem as an optimization problem on its own is not a new idea. There are various previous works in this area [4], [5], [6], [7], [8], [9], [10], [11] which have some common characteristics. In most previous approaches to meta-optimization the implementation is tailored to only one or a few algorithm types. On the meta-level, specialized variants of existing optimization algorithms were implemented. The advantage is that they can be optimized to achieve good results with a few number of evaluations, however it is difficult to reproduce the results. When no broadly known and well-documented algorithms are used, it becomes merely impossible to re-implement an algorithm that is only briefly described in a publication. Most meta-optimization approaches are custom implementations with command line interfaces, application programmer interfaces (APIs), or rudimentary graphical user interfaces. This reduces usability for people who were not involved in the development. Many approaches use binary encoding for parameter values, mainly due to performance advantages [12], [13], [14], [15], however modern metaheuristics often use more natural encodings which represent the problem in a more direct way. Some authors did use parallelization concepts in their approaches, but most of them just mentioned that parallel and distributed computation is highly suitable for meta-optimization. Most approaches do only optimize against one optimization goal, which is the average best quality achieved by the base-level algorithm. Other goals such as robustness and effort are only considered by few [10].

B. HeuristicLab Meta-Optimization

The PMO approach described in this paper tries to improve on some of the previously mentioned drawbacks of available solutions. In the following some of the most important characteristics are outlined.

Due to the generic implementation it is not necessary to apply any adaptations to the meta-level and the base-level algorithm. Basically, any algorithm implemented in HeuristicLab can be used to optimize any other algorithm. This allows using the power of existing algorithms and new algorithms on the meta-level in the future. Another characteristic is a multi-objective fitness function. This allows to

- optimize for finding the best settings,
- optimize for maximum robustness or
- optimize for minimum effort.

Another aspect of the HL PMO implementation is that it doesn't use binary encoding for parameter values but an encoding which represents the parameters in a more natural way. There are boolean parameters, numeric parameters, and configuration parameters that use elements of a given unordered set. For this representation it offers exchangeable operators for manipulating parameter values in solution candidates. An additional property is that the user is able to decide which parameters of an algorithm should be optimized. Further, it is possible to define in which range each parameter value should be optimized. In that way a user can use his expertise and narrow search ranges to reduce the size of the search space. The PMO implementation in HeuristicLab is designed in such a way that parallel and distributed execution of the meta-level algorithm is feasible. To conclude with one of HL's most important concepts, the implementation offers a rich and easy to use graphical user interface. It allows even non-experienced users to get started quickly.

C. Solution Encoding

In the following a brief overview of the chosen solution encoding is given. In HeuristicLab algorithms and problems define certain parameters, some of these may be operators which may again define parameters. To give an example, a genetic algorithm

has a parameter for its population size as well as a parameter for the selection operator. The selection parameter could be tournament selection, which is an operator and has again a parameter for the group size. Therefore a composite tree structure of parameters emerges. Because the goal of PMO is to optimize these parameters, the PMO solution encoding has to mirror this parameter tree structure. For PMO, the representation of a solution candidate is a tree of concrete parameter values for a parameterizable object. An additional requirement for PMO is that the way how each parameter should be optimized needs to be configurable. It should be possible to define which parameters of a parameterizable object should be optimized and for each of these parameters different configuration options should be available, depending on the type of the parameter.

The solution representation for PMO in HL is called *parameter configuration tree*. Each parameter of an object (e.g. an algorithm, problem or operator) has a corresponding parameter configuration. The parameter configuration contains information about the parameter such as the name, the allowed data types, a list of possible values and a reference to the currently selected value. Possible values for a parameter can be a single value, multiple values and a range of values with an upper and a lower bound. Ranges support sampling random values or enumerating all values in the range for a given step size. Because parameter values can be parameterized themselves, there also exists a parameter configuration that contains a collection of parameter configurations, which corresponds to the parameters of the value. With the parameter configuration tree the user can decide which and how a parameter should be optimized. Every parameter configuration also offers functionality for randomization, crossover and mutation which is used by evolutionary optimization algorithms.

D. Fitness Function

The fitness function of a PMO solution candidate consists of the following components:

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

These components represent conflicting objectives. For example it is expected that, e.g. an iteration based optimization algorithm is able to deliver better results with a higher number of iterations. Certainly the results will not worsen if the best quality is remembered. If the maximum number of iterations was therefore set to be optimized, solution quality would be expected to improve as the iterations increase, but the effort would become worse the more solutions are evaluated in this configuration. A better configuration could be found that reaches the same quality, but in less iterations and therefore with less evaluated solutions.

There are two options to tackle multi-objective problems. The simplest one is to turn it into a single-objective problem by computing a weighted sum of the objective values. Another option would be to obtain a pareto front [16] of the objectives. A number of algorithms have emerged to perform such kind of multi-objective optimization as for example the NSGA-II [17]. However, for the sake of simplicity, in this implementation a weighted average was chosen. Another requirement is that optimal parameter values for multiple problem instances should be found. Therefore, PMO allows a user to add multiple problem instances for the base-level algorithm. However, different problem instances might yield quality values in different dimensions. An example would be to optimize a GA for three Griewank [18] test-functions in the dimensions 5, 50 and 500. A GA with a given parameterization might result in the quality values 0.17, 4.64 and 170.84. Using a simple arithmetic mean for the fitness function would overweight the third instance a lot. It is the goal to find settings which are suited for each instance equally well. To tackle this issue, normalization has to be applied on all results of each run (q, r, e). The reference values for normalization are the best values from the first generation (q_0, r_0, e_0). Furthermore each objective needs to be weighted (w_q, w_r, w_e). The quality (Q) of a solution candidate (c) for m base-level problems is thereby defined as:

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

E. Operators

Algorithms in HL are abstracted from the optimization problem and the problem encoding. Algorithms are represented as a sequence of steps including certain operators that manipulate the solution. The problem, and especially the encoding in HL implement and these operators while the problem exposes them to the algorithm. Among these encoding-specific operators are those that create solutions, crossover-, and mutation-operators for population-based algorithms as well as move-operators for trajectory-based algorithms. Until now only operators for population-based evolutionary algorithms have been implemented for the parameter configuration tree encoding. The solution creator is responsible for creating a random solution candidate. This operator is mainly used to initialize an initial generation of randomized individuals. The solution creator for PMO needs one initial parameter configuration tree which can be generated from

a given algorithm and a problem.

The evaluation operator applies the fitness function on a solution candidate in order to compute its quality. In the case of PMO, the base-level algorithm needs to be parameterized and executed n times for each base-level problem. The *PMOEvaluator* therefore creates an instance of the base-level algorithm for each repetition and for each base-level problem instance. Then each of the base-level algorithm instances is executed and after all runs have finished, the results of each run are collected. For each base-level problem instance the average quality, the standard deviation of the qualities, and the average number of evaluated solutions is computed. The last step in the evaluation is the normalization of these values. The average of the normalized fitness measures represents the final quality value for one solution candidate. The evaluation of solution candidates is the computationally most intense part of meta-optimization. Fortunately, the solution evaluations are independent from each other, which makes parallelization a viable option. In HL, parallel evaluation of solution candidates is possible either locally with multiple threads or distributed using HL Hive. For this paper, HeuristicLab Hive was used to overcome the runtime requirements.

The mutation operation in evolutionary algorithms is supposed to manipulate one solution candidate. There are two operators for selecting nodes of the tree that should be manipulated. The *ParameterConfigurationOnePositionManipulator* selects one parameter randomly while the *ParameterConfigurationAllPositionsManipulator* selects all parameters. After the element nodes that should be mutated are selected, type-specific manipulation operators are applied on each of them:

- **Mutation of Parameter Configurations**: When a parameter configuration is mutated, a new current value is selected randomly from the list of available value configurations. The selection probability is equally distributed.
- **Mutation of Boolean Values**: Since boolean values can only have two different values (*true*, *false*), one of these values is randomly chosen by the mutation operator.
- **Mutation of Numeric Values**: For integer and double values mutation is done by sampling from a uniform or normal distribution.

The crossover operation is supposed to combine the genes of two solution candidates to create a new one. For PMO this means to combine two parameter configuration trees. Each node in the trees is crossed with the corresponding node of the other tree. The concrete crossover operation depends on the data type of the node:

- **Crossover of Parameter Configurations**: When two parameter configurations are crossed, the index of the currently selected value configuration is chosen from one of the parents randomly.
- **Crossover of Boolean Values**: The boolean value of either of the two parent solution candidates is chosen randomly.
- **Crossover of Numeric Values**: There are several crossover operators implemented. For example a *DiscreteValueCrossover* takes the value of one of the parents randomly. Another example is an *AverageValueCrossover* which computes the average of the values of both parents.

Another important operator is the solution cache which keeps a history of previously evaluated solutions. Its purpose is to avoid the evaluation of solutions which have already been evaluated. An additional option offered by the HeuristicLab PMO is exhaustive search. In some scenarios it might as well be interesting to explore the whole search space. Of course, this is only possible, if a small amount of parameters and narrow search ranges are explored. Therefore a special operator was created that is able to enumerate all possible combinations of parameter values of a parameter configuration tree.

Parameter Name	Value
Algorithm	GA
Maximum generations	100
Population size	30
Mutation probability	10%
Elites	1
Selection	Proportional
Mutation	OnePosition
Mutation (values)	NormalValue
Crossover	NormalValueOX
Evaluation repetitions	6
Quality weight	1.0
Robustness weight	0.0
Effort weight	0.0

TABLE I
PARAMETERS OF THE META-LEVEL ALGORITHM (m_1) FOR S_1

IV. Experimental Results

In the following, two test scenarios are described and the results of the experiments are shown.

A. Varying Problem Dimensions

In scenario S_1 the parameters of a GA are optimized. Multiple different Griewank test-functions are used as base-level problems. As a meta-level optimizer, a GA is applied. The goal of this scenario is to find out, how the optimal parameter values differ when different problem dimensions of test-functions are used. Table I shows the parameter configuration for the meta-level GA. The rather small population size and maximum generations are related to the immense runtime requirements. A repetition-count of six has been chosen as a trade-off between high runtime demand and high evaluation accuracy. The average quality is used as the only optimization objective in this scenario as the goal is to optimize for optimal parameters.

Parameter Name	Values
Algorithm	GA
Maximum generations	1'000
Population size	100
Mutation probability	0%–100%:1%
Elites	0–100:1
Selection	6 different operators e.g. LinearRank, Proportional, Random, Tournament
Mutation	7 different operators e.g. Breeder, MichalewiczNonUniformAllPositions, UniformOnePosition, no mutation
Crossover	11 different operators e.g. Average, Discrete, Heuristic, Local, RandomConvex, SinglePoint

TABLE II
PARAMETER CONFIGURATION (c_1) OF THE BASE-LEVEL ALGORITHM FOR S_1

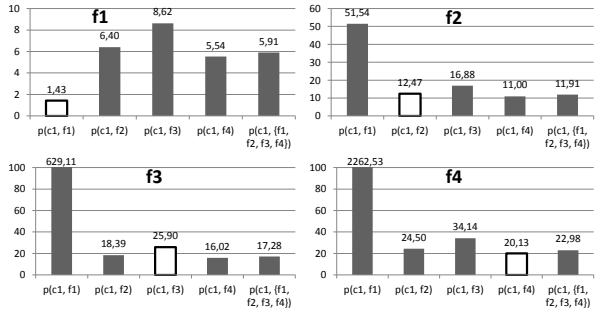

Fig. 3. Average qualities achieved by different parameterizations for the problems f_1 to f_4. Each parameterization (p) was repeated 10 times on each base-level problem.

The parameter configuration (c_1) of the base-level GA is shown in Table II. The population size and the number of maximum generations are fixed for this scenario in order to keep the runtime approximately equal for all solution candidates. When ranges are specified, the number after the colon represents the step size. For parameters not shown in Table II concrete values have been chosen to reduce the search space and simplify the problem.

As a base-level problem the Griewank function [18] was used in the dimensions 500 f_1, 1.000 f_2, 1.500 (f_3) and 2.000 (f_4). One meta-optimization run was performed for each base-level problem f_1–f_4 as well as for the combined problem instance set $\{f_1 f_2 f_3 f_4\}$. To validate whether each parameterization is really optimal for the problem set it has been optimized for, cross-testing was performed for all results. In these cross-tests, each parameterization was applied to a different base-level problem. Figure 3 shows that $p(c_1, f_1)$ performs significantly better on f_1 (as expected) than on the other problems, while $p(c_1, f_2)$ to $p(c_1, f_5)$ perform almost equally well on f_2 to f_5, but not so well on f_1.

	$p(c_1, f_1)$	$p(c_2, f_1)$
Elites	1	10
Crossover	BlendAlpha	BlendAlpha
Mutation	SelfAdaptiveNormalAllPos	Breeder
Mutation probability	27%	24%
Selection	Tournament	LinearRank
Tournament group size	5	
Generations	100	100
CPU time (days)	23	66.7
Avg. qualities	1.3952	6.6×10^{-10}

TABLE III
SOLUTIONS OF THE META-OPTIMIZATION RUNS FOR S_1 AND S_2

B. Varying Generations

To use more realistic settings, scenario S_1 has been repeated with the same parameter configurations but with 10.000 instead of 1.000 generations. This scenario S_2 shows similar results in the cross tests as the ones presented with scenario S_1, though the optimal found parameters differ between S_1 and S_2. Table III shows the parameters found by the PMO for S_1 and S_2 for the f_1 problem. This shows that changing the number of generations also influences the performance of other parameters.

To further validate the results, the parameter settings of S_1 and S_2 were cross-tested. These cross tests show that the parameter values from S_1 clearly outperform the parameter values of S_2 for 1'000 generations at every problem. The opposite is the case for 10'000 generations. For further analysis, the quality charts for f_1 and 10'000 generations are shown with the settings from S_1 (Figure 4) and S_2 (Figure 5). In Figure 4 the quality improves significantly until generation 1'000, but then it stagnates and does not improve anymore. In contrast, the quality chart in Figure 5 shows slower convergence, but it does not suffer from stagnation at all. The quality improves in almost every iteration until the last generation is reached.

V. CONCLUSIONS AND FUTURE WORK

The experiments in Scenario 1 have shown that the optimal parameters can differ significantly when the same test-function problem with different dimensions is used. In Scenario 2, very interesting parameters with extremely high mutation probabilities and a high number of elites were identified as best parameter settings. It indicates that the best parameters can be far off the default and commonly used settings. In the comparison of scenario 1 and 2, the effect of using different numbers of iterations on the parameters was analyzed. It was shown that the parameter settings are valid because switching them

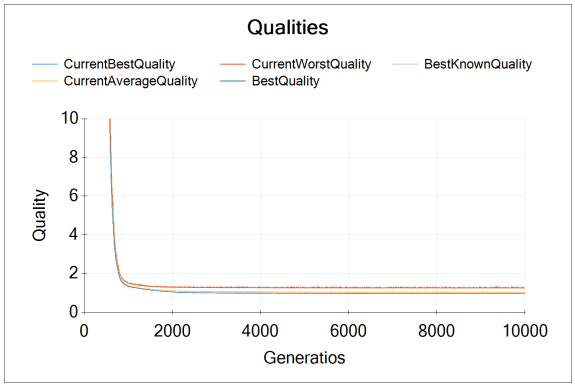

Fig. 4. Shows the quality history of a GA run for f_1 over 10'000 generations. The settings that were used in this run were optimized for 1'000 generations.

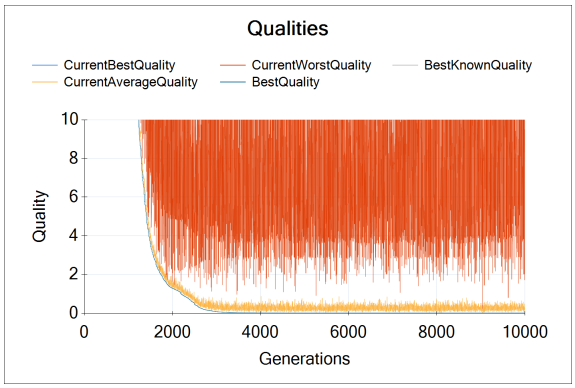

Fig. 5. Shows the quality history of a GA run for f_1 over 10'000 generations. The settings that were used in this run were optimized for 10'000 generations.

between scenarios led to worse results. Concluding, the approach of using a meta-level algorithm to find the optimal parameters has proven to work very well on some problems. It is possible to find parameter value combinations that are very different from commonly used settings. This functionality comes at the cost of huge runtime demands.

The advancements in computing power in the recent years have made PMO feasible. Parallelization and distributed computing has been used to perform experiments. However, there is room for improvement in terms of runtime performance. Two ways to optimize runtime would be *racing* and *sharpening* [8]. When *racing* is used, promising solution candidates are evaluated more often than bad solution candidates. With *sharpening*, the number of repetitions is increased depending on the current generation. In this way, the solution evaluation is faster in the beginning of the optimization process and gets more and more precise towards the end.

An idea to simplify future development of PMO would be to provide benchmark problems for parameter settings. Such a benchmark problem could have a generated meta-fitness landscape. Evaluating a solution candidate would only require to lookup a value, so that the evaluation of solution candidates would become extremely fast. Of course, the fitness evaluation should underlie a stochastic distribution, just as real evaluations of parameter settings. This would make it much easier to tune the parameters of a meta-level optimizer.

Where to get HeuristicLab Hive

HL Hive is part of HeuristicLab since version 3.3.6. HeuristicLab can be downloaded from the official homepage[2]. HeuristicLab PMO is still in development and can be downloaded as an additional package[3].

The software described in this paper is licensed under the GNU General Public License[4].

Hardware Infrastructure

The hardware used for the experiments is a Dell Blade System. One blade has the following properties:

- CPU: 2x Intel Xeon E5420, 2.50 GHz, 4 cores
- Memory: 32 GB
- OS: Windows Server 2008R2 Enterprise 64-bit

All experiments described in this paper were executed in the Hive with 4 blades at a time. The execution of the experiments took place at an early stage of the Hive roll out. Hive currently consists of around 65 computers varying from blade computers to PC's from the computer labs of the University of Applied Sciences Upper Austria resulting in a total number of 150 CPU cores.

Acknowledgments

HeuristicLab Hive is a 3 years effort on which various students as well as members of the HEAL team worked on. The computational resources for the experiments were provided by the University of Applied Sciences Upper Austria. 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).

References

[1] David H. Wolpert and William G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

[2] Stefan Wagner, *Heuristic optimization software systems-Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*, Ph.D. thesis, Johannes Kepler University, Linz, Austria, 2009.

[3] John L. Gustafson, "Reevaluating amdahl's law," *Commun. ACM*, vol. 31, pp. 532–533, 1988.

[4] R.E. Mercer and J.R. Sampson, "Adaptive search using a reproductive metaplan," *Kybernetes*, vol. 7, no. 3, pp. 215–228, 1978.

[5] Darrell Whitley, "A free lunch proof for gray versus binary encodings," in *Proceedings of the Genetic and*

[2] http://dev.heuristiclab.com/download

[3] http://dev.heuristiclab.com/trac/hl/core/wiki/BranchesDailyBuilds

[4] http://www.gnu.org/licenses/gpl.txt

Evolutionary Computation Conference. 1999, vol. 1, pp. 726–733, Morgan Kaufmann.

[6] Thomas Bäck, "Parallel optimization of evolutionary algorithms," *Lecture Notes In Computer Science*, vol. 866, pp. 418–427, 1994.

[7] Michael Meissner, Michael Schmuker, and Gisbert Schneider, "Optimized particle swarm optimization (opso) and its application to artificial neural network training," *BMC Bioinformatics*, vol. 7, no. 1, pp. 125, 2006.

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

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

[10] Erik Magnus Hvass Pedersen, *Tuning & Simplifying Heuristical Optimization*, Ph.D. thesis, University of Southampton, 2010.

[11] Mihaela Iunescu, *Parameter Optimization of Genetic Algorithms by Means of Evolution Strategies in a Grid Environment*, Ph.D. thesis, Johannes Kepler Universität Linz, 2006.

[12] John Grefenstette, "Optimization of control parameters for genetic algorithms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 1, pp. 122–128, 1986.

[13] D.B. Fogel, L.J. Fogel, and J.W. Atmar, "Meta-evolutionary programming," in *Signals, Systems and Computers*. 1991, pp. 540–545, IEEE Computer Society Press.

[14] Thomas Bäck and Hans-Paul Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evolutionary Computation*, vol. 1, no. 1, pp. 1–23, 1993.

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

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

[17] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

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