ABSTRACT

In this paper we present AOAB, the Automated Optimization Algorithm Benchmarking system. AOAB can be used to automatically conduct experiments with numerical optimization algorithms by applying them to different benchmarks with different parameter settings. Based on the results, AOAB can automatically perform comparisons between different algorithms and settings. It can aid the researcher to identify trends for good parameter settings and to find which algorithms are suitable for which type of problem.

We introduce the system structure of AOAB (the server and the graphical client interface), define the way in which optimizers and benchmark functions can be implemented for the use in AOAB, and conduct an illustrative example experiment with our system: a comparison between Random Search and two Hill Climbers.

Categories and Subject Descriptors
F.2.1 [ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY]: Numerical Algorithms and Problems—complexity measures, performance measures; G.1.6 [NUMERICAL ANALYSIS]: Optimization—Global optimization; C.4 [PERFORMANCE OF SYSTEMS]: Measurement techniques

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO'10, July 7–11, 2010, Portland, Oregon, USA.
Copyright 2010 ACM 978-1-4503-0073-5/10/07 ...$10.00.
It is clear that this procedure is time consuming, error-prone, cumbersome, and -- basically -- has nothing to do with algorithm design. Since the later is considered to be a much more interesting activity, thorough benchmarking unfortunately is often neglected in favor of parameter/feature tuning, fiddling, and minimalistic tests.\footnote{The first author admits that he suffered from this disease in his first years as PhD student.}

In this paper, we propose and discuss the realization of a system which relieves the researcher from most of the tasks involved in testing her algorithm, AOAB -- the Automated Optimization Algorithm Benchmarking system. It aims to support sustainable and thorough algorithm testing and comparing. Specifically, AOAB addresses the following issues:

1. It automates logging and makes it completely transparent.
2. It requires the researcher to deal with \textit{almost no} code additional to the plain algorithm implementation, no libraries or additional files.
3. It automates running experiment series for different configuration parameters of an optimization algorithm and/or comparisons between different algorithms.
4. It automates the experiment evaluation, trend analysis, statistical algorithm comparisons, and drawing of graphics.
5. It provides a convenient way for long-term data storage.
6. It provides a graphical user interface which allows uploading new algorithms and benchmark functions to the system, to configure new benchmarks, and to start and to evaluate experiments.\footnote{This feature makes it also suitable for teaching purposes.}
7. The system is inherently distributed and able to make use of an arbitrary number of computers for performing the experiments. The distribution of computation is transparent to the user and requires no interaction.
8. The system is able to deal with different programming languages and platforms (\texttt{Java}, \texttt{C++}, \texttt{Matlab}, \ldots) and can easily be extended for new languages and platforms.

The remainder of this paper is structured as follows. In Section 2 we give an overview on work related to our AOAB project whose system structure is discussed in detail in Section 3. Implementers of optimization algorithms and benchmark functions have to adhere to the simple criteria outlined in Section 4 in order to utilize AOAB. Section 5 shortly discusses some initial experiments and results obtained with our system. We conclude the paper in Section 6 with a summary and a discussion of the planned future work.

2. Related Work

The question “Which optimization algorithms is best?” has been answered by Wolpert and Macready \cite{39}: None. The performance of all optimization algorithms is the same when averaged over all possible optimization problems. Therefore, the question has to be refined to “Which of the optimization algorithm is good for a certain class of problems?”.

Tackling such questions on a theoretical basis is complicated and not always possible. Benchmarking, i.e., applying the algorithms to several clearly specified example problems, becomes the technique of choice when comparing optimization method. In the area of numerical optimization, the two most notable events related to this topic are the Black Box Optimization Benchmarking (BBOB) workshop \cite{16, 17} located at GECCO and the regularly reoccurring competitions \cite{19, 22, 23, 34–36} at the IEEE CEC.

In the latter, the benchmark functions are defined in a document and implementations are provided for several programming languages. In the 2010 challenge \cite{36}, for example, the goal is to find algorithms suitable for large-scale numerical optimization. A set of twenty benchmark functions comprising separable, non-separable, and \textit{m}-separable functions is provided as benchmark along with implementations in \texttt{Matlab} and \texttt{Java}. The contestants in the competition themselves are responsible for logging and evaluating their data according to predefined, fixed criteria.

BBOB, on the other hand, provides a more comprehensive framework for benchmarking: \texttt{COO}ping Continuous Optimisers (COCO) \cite{16, 17}. It again includes the benchmark functions, but already provides logging and evaluation utilities, thus reducing the development overhead for users. Many concepts of COCO are similar to AOAB. There are, however, also striking differences. AOAB, for instance, targets experimental series (currently in full factorial design \cite{4, 41}) for analyzing the influence of the parameter settings of the optimization algorithms. The goal of AOAB is to provide comprehensive data for not only comparing optimizers but for analyzing their behavior. Also, AOAB is not bound to a certain benchmark structure. While COCO requires a certain effort to change the benchmark, in AOAB a benchmark is simply a set of functions and arbitrary new benchmarks can be defined and new functions can be uploaded at any time. Besides helping researchers to compare and analyze their algorithms, AOAB features a second use case: students who are delving into the matter of optimization algorithms may use it as an easy framework to implement them and in order to explore their behavior.

A very general tool for testing and benchmarking optimization is the Optimization Algorithm Toolkit (OAT) implemented in \texttt{Java} by Brownlee \cite{6, 7} in 2007. Whereas OAT supports arbitrary problem domains, it also requires the researcher to delve much deeper into the system itself, mainly due to its generality and the integration into a graphical user interface. Unlike AOAB, OAT cannot support different programming languages and does not allow for automatic factorial experiments in terms of optimization algorithms and their parameters. It is designed to be used on a single computer only and does not support distribution of the work load. In terms of continuous optimization algorithm benchmarking, the highest supported dimension of the search space seems to be 2.

3. System Structure

The structure of our system is illustrated in Figure 1. It consists of a client with a graphical user interface, the AOAB
server, the database (solely accessed by the server), and an arbitrary number of worker machines.

3.1 The AOAB Client and User-provided Data

The client with its graphical user interface allows the user to upload new algorithm and objective function implementations to the AOAB server. In both cases, the user first selects the source files and then specifies for which programming language and platform they are intended. Algorithms as well as objective functions can be tagged with parameters. An implementation of a certain Genetic Algorithm, for instance, may be characterized with the parameters populationSize (positive integer), crossoverRate, and mutationRate (floating point numbers). In case of uploading an objective function, parameters defining the limits of the search space \((\text{min}X, \text{max}X)\) as well as its dimension may be declared along with their default values.

With the GUI the user can furthermore compose new benchmarks consisting of a selection of objective functions, the maximum number of function evaluations to be used in the optimization process, and possible settings for their parameter values. In a benchmark definition, parameters of the functions (for instance dimension) may be overridden. A function once uploaded may be part of multiple different benchmarks. The Sphere function, for instance, is used in [16] with \(\text{dimension} \in \{2, 3, 5, 10, 20, 40\}\) and in [40], \(\text{dimension} = 30\), amongst others.

To each benchmark-to-function relation, an interesting point set (IPS) is furthermore assigned. An IPS consists of definitions of interesting values and interesting function evaluations (IFEs), combining both the vertical-cut and horizontal-cut views discussed in [16]. The former are objective values which, when surpassed, should lead to the collection of one data point by the logging code. The IFEs, on the other hand, are function evaluation numbers which, too, should lead to the collection of a point when exhausted by the optimizer. The explicit IPS definition allows collecting data according to the (often more or less arbitrarily chosen) criteria used by the original benchmark designers. Yet, we would recommend having the IPS contain around 1000 entries in order to collect enough data for being able to sufficiently characterize the behavior of an optimization algorithm. Due to the automatic removal of redundant points, space in the database is preserved.

For starting an experiment, the user selects an algorithm implementation and a benchmark.\(^3\) Additionally, she may specify value sets for each parameter of the algorithm or benchmark. For the aforementioned GA example, she may be interested in carrying out experiments with \(\text{populationSize} \in \{50, 100, 200\}\) and \(\text{mutationRate} \in \{0.01, 0.05\}\). Maybe she is less interested in the crossover rate and, by not specifying a setting for it, simply uses the default value. She now can upload this configuration to the AOAB server which is then to deal with all tasks involved in carrying out the experiments.

After the experiment has been finished, the user can evaluate the results. Like in the COCO/BBOB scenario [16, 17], a pdf document is produced containing various evaluations and diagrams. The difference is that we aim to perform an automated statistical comparison between different algorithms and configurations. If the user chooses the \(\text{populationSizes} \) and \(\text{mutationRates} \) as in the GA example, this may result in six different configurations.\(^4\) The goal of our system is to discover which configuration is the best as well as trends (such as, hypothetically, “large values of populationSize are good” or “the results get worse with rising dimension”), but also \(\text{when} \) these trends set in.

One algorithm, for instance, may initially quickly detect a good area in the search space but then prematurely converge. Another one may proceed much slower but finally find better solutions. With the collected data, it is possible to make statements regarding such issues automatically. We will provide a plugin architecture allowing adding, removing, and selecting evaluation modules in the GUI. This architecture will allow us to, for instance, include the evaluation modules provided by COCO [16, 17] into AOAB.

3.2 The AOAB Workers

An AOAB worker is a process which occupies exactly one processor. It is programmed to do jobs automatically according to functions it is assigned. These functions are implemented in any language that the AOAB server supports and are uploaded to the AOAB server by the user. The user may upload the same function to multiple workers at the same time, or even multiple functions to the same worker.

When a worker receives a job from the AOAB server, it executes the task and sends the results back to the server. The AOAB server then aggregates the results and presents them to the user in a graphical user interface.

Notice that algorithm/function upload, benchmark specification, experiment starting, and evaluation may all be performed by different users.

\(^3\)Currently, only full factorial designs [4, 41] are supported. Later, support for other methods such as fractional factorial designs [56] will be added.
virtual CPU on a computer and is able to request and perform jobs from the AOAB server. It simply runs the procedure illustrated in Figure 2 in a loop. Every worker is characterized by its capabilities. A worker running on a Windows machine may, for instance, have detected a Matlab and a Visual C++ installation on startup while failing to find the Java JDK needed to compile Java programs. As a consequence, it is able to process Matlab and C/Visual C++ jobs, but none which require Java or depend on a Linux platform.

A job request (w1 in Figure 2) sent to the AOAB job service contains these capabilities and is either (w2) answered negatively or with a job. A job is defined by all parameter settings of the run to be executed and two zip archives containing the algorithm and benchmark function sources. The worker will then copy the source files into a temporary directory (w3) and create an additional file representing the configuration. In the C programming language, this file would be a header configuration.h containing one #define for each parameter setting (such as #define populationSize 100, #define selectionAlgorithm 0,...).

Furthermore, the benchmark functions are not used as uploaded but wrapped into code which performs in-memory logging. The worker then compiles the sources to a program (w5) which it subsequently executes. This process is the running the optimization algorithm extended with logging code. When it finishes, it sends all the collected data as a SOAP message to the server. The worker can now delete the temporary directory (w9) and request the next job.

3.3 The AOAB Server and System Robustness

The AOAB server is the center of the system and provides its functionality as web services via a SOAP [25] interface. It is responsible for storing and retrieving experimental results, optimization algorithm, and objective function implementations. It determines which runs have to be performed when an experiment is started by inferring the configurations, the platform and programming language from the parameter settings and the selected algorithm implementation. It furthermore manages a job queue for these runs for the workers.

One interesting feature of all components of AOAB (except for the database, obviously) is that they are stateless. The AOAB server, for instance, keeps all information regarding the experiments and the job queue in the database. If a worker requests a job, the priority of the returned job in the queue will be decreased, but the job itself is not removed from the queue until a corresponding result has been received and successfully stored in the database. Hence, if either the AOAB server or the worker crash (or both), no experimental data or experiment job will be lost. Instead, the worst that can happen is that the job is not executed until all other jobs in the queue have been processed. Furthermore, the AOAB server does not keep track of the available workers or job assignments. Instead, a worker ready to process a job will request one. Therefore, the number of workers can arbitrarily change at any given time, be it by new computers becoming available or crashes. If an uploaded algorithm implementation is faulty, the server may detect that the number of workers involved in the related jobs is suspiciously increasing. Since the AOAB server also does not keep track of connected clients, the same robustness is provided in this area as well.

4. Source Formats

The core idea of our experimenting approach is that the AOAB workers assemble the programs to be executed on the machine in which they run. This involves placing the sources of the optimization algorithm and the benchmark function to be used in the current run into one temporary directory and to automatically generate some additional source code for logging and providing the configuration parameters. This method has the advantage that issues like data representation, different numeric representations, and processor optimization on different computers are automatically solved. The overhead caused by compiling the programs is relevant only for optimization problems with low dimensions and/or low numbers of maximum allowed function evaluations. In order to be compatible with this concept, algorithm and function implementers have to adhere to a few simple constraints. These restrictions are discussed in the following two sections on basis of examples in the Java language which, however, easily carry over to other languages such as C, C#, Matlab, or Python.

4.1 Implementing Benchmark Functions

Listing 1 contains an example implementation of the simple sphere function in Java for AOAB.

In our opinion, such an application structure is extremely useful for larger experiments for two reasons: On one hand, it allows the user to utilize all resources in a lab or cluster whenever they become (and for as long as they stay) available. On the other hand, it is often the case that one computer or run fails for an arbitrary reason which, in many systems, would force the user to deal with the experiment-running code or logging data in order to isolate and restart the run [16]. With our system structure, such things become unnecessary and are handled by the system automatically.

4.2 Implementing Optimization Algorithms

The restrictions imposed on implementing algorithms are similar. Listing 2 provides an example implementation of

```java
public class Sphere {
    public static double compute(double[] x) {
        int s;
        double s, t;
        s = 0d;
        for (i = Configuration.dimension; (--i) >= 0; ) {
            t = x[i];
            s += (t * t);
        }
        return s;
    }
}
```

Listing 1: The implementation of the sphere function in Java for AOAB.

```java
# define selectionAlgorithm 0
# define populationSize 100

import java.util.Random;
import java.util.Properties;

public class Sphere {
    public static double compute(double[] x) {
        int s;
        double s, t;
        s = 0d;
        for (i = Configuration.dimension; (--i) >= 0; ) {
            t = x[i];
            s += (t * t);
        }
        return s;
    }
}
```

Listing 2: The implementation of the sphere function in C/C++ for AOAB.
the random sampling [1, 5] algorithm in Java. Here, the following constraints have to be considered: (a) The objective function is assumed to be performed in a class called Function and has exactly the character mentioned in Section 4.1 (see line 21 of Listing 2). (b) Additionally, this class has two more functions init and done which have to be invoked before and after the optimization process, respectively (lines 9 and 23). (c) All configuration values such as, for example, the populationSize of an EA, but also the boundaries and dimension of the search space (minX, maxX, dimension), the maximum number of allowed function evaluations, maxFEs (lines 12 to 15), reside as constants in the class Configuration. Since the algorithm uploader who specifies the parameters of the algorithm implementation usually is also the implementer, adhering to the rules regarding the use of parameters should be natural.

5. Initial Experiments

Major parts of AOAB are still in implementation stage. However, we are already able to conduct experiments and evaluations automatically. In the following we will show some proof-of-concept tests. It should be noted that these experiments are merely performed in order to provide illustrative examples for the capabilities of our system and we did not intend to provide competition benchmark or comparison data of optimization algorithms.

5.1 Setup

We compared three different algorithms: plain random sampling (RS) [1, 5] and two non-adaptive hill climbers (HC-G, HC-C). The random sampling algorithm RS uses all function evaluations (FEs) granted to it to create (independent) random points in the search space via uniform sampling. It remembers the best candidate solution discovered and returns it as result when all FE s are exhausted. A hill climber starts at a random point p in the search space and randomly creates neighbors q around this point. As soon as it discovers a better solution (f(q) < f(p)), it transmutes to it (p ← q).

For sampling the neighborhood, our hill climbers therefore add (independent) random values to each element of the p. In case of HC-G, the Gaussian distribution with standard deviation 1 is used. HC-C generates Cauchy-distributed random numbers via inverse transformation sampling of uniformly distributed random numbers (see Fig. 3.m).

AOAB automatically applied C++ implementations of these algorithms to the first eight functions from the benchmark used in the 2010 Competition on Large-Scale Global Optimization [36] with dimension ∈ {200, 300, 400} and maxFEs = 300000. As prescribed in [36], 25 runs were executed per configuration. Here we present an excerpt of the results. Besides tables with statistics about the performance of optimizers, AOAB can produce elaborate graphical output. In Section 5, we present some of the diagrams AOAB created as evaluation of the experiment5.

5.2 Performance Diagrams

In [36], the first benchmark function (f1) is a shifted ellipse. The graphics Fig. 3.a to Fig. 3.e illustrate the performance of the different algorithms when applied to f1 in terms of best objective value discovered until a certain function evaluation (both axes are logarithmically scaled). During the experiments, AOAB has recorded data according to the interesting point set. For a specific FE, it can thus compute the median, minimum, maximum, and the quantiles (5%, 25%, 75%, and 95%) of the performance of these independent runs. We choose such a representation over an arithmetic mean ±2×stddev plot since we assume that the performances of the runs may not be normally distributed. Fig. 3.d to Fig. 3.f sketch the same information for the fourth benchmark function (f4) from [36], a rotated and shifted version of the elliptic function. Here, only the FE axis is logarithmically scaled.

Such graphics already provide some initial insight into the behavior of optimization algorithms. Here, it becomes for instance clear that even trivial hill climbers already behave very differently from random sampling. The RS algorithm initially finds some improvements and later needs exponentially increasing time to discover better solutions. The hill climbers, on the other hand, seem to need some time to discover the “right direction” towards the optimum of the (unimodal) objective function and then are able to improve very quickly for some time. Since the algorithms are non-adaptive and sample the neighborhood with constant variance/scale, this improvement ends at some point in time and converges to the RS behavior.

5.3 Statistical Test Result Diagrams

An interesting question in optimization research is not only whether a certain algorithm is better than another one, but also when it becomes better (or worse, in the opposite case). With graphics of the types used in Fig. 3.g to Fig. 3.l, we want to show how AOAB can help to find answers to this question as well. In most research papers, the authors compare the average results of their algorithms with the average results of some other algorithm in order to determine which is better. However, such a comparison is more or less useless. If we assume that the objective values are continuous and the algorithms do not end up in similar local optima, 5The graphics have been slightly beautified/zoomed for presentation in this paper and due to the development/non-beautified state of the system.

Listing 2: The implementation of a random sampling [1, 5] algorithm in Java for AOAB.

```java
public class RandomSampling {
    public static void main(String[] params) {
        final double[] d;
        int i, j;
        final Random r;
        double v, f;
        Function.init();
        r = new Random();
        v = (Configuration.maxX - Configuration.minX);
        d = new double[Configuration.dimension];
        for (i = Configuration.maxFEs; (--i) > 0; ) {
            for (j = d.length; (--j) >= 0; ) {
                d[i] = Configuration.minX +
                (v * r.nextDouble());
                //random sampling [1, 5] ignores the fitness
                f = Function.compute(d);
            }
            Function.done();
        }
    }
}
Fig. 3.a: Performance of RS on $f_1$, dimension = 200.

Fig. 3.b: Performance of HC-G on $f_1$, dimension = 200.

Fig. 3.c: Performance of HC-C on $f_1$, dimension = 200.

Fig. 3.d: Performance of RS on $f_4$, dimension = 400.

Fig. 3.e: Performance of HC-G on $f_4$, dimension = 400.

Fig. 3.f: Performance of HC-C on $f_4$, dimension = 400.

Fig. 3.g: Comparison of the optimizers on $f_7$, dimension = 200.

Fig. 3.h: Comparison of the optimizers on $f_7$, dimension = 300.

Fig. 3.i: Comparison of the optimizers on $f_7$, dimension = 400.

Fig. 3.j: The total test wins in all functions dimension = 200.

Fig. 3.k: The total test wins in all functions dimension = 300.

Fig. 3.l: The total test wins in all functions dimension = 400.

Histogram of samples from the two random number generators used in the two hill climbing algorithms:
The interval $(-7, 7)$ has uniformly been divided into 1000 baskets and each generator has been sampled $1e8$ times. The y-axis illustrates the fraction of samples which fell into the baskets. HC-G uses the Gaussian random numbers $R_G \sim N(0,1)$. In HC-C, Cauchy-distributed random $R_C$ numbers are used, which have been obtained by applying inverse transformation sampling to uniformly distributed random numbers $u$, i.e., by computing $\tan(u)$.

Fig. 3.m: The different distributions used in the hill climbers: $R_G$ in HC-G and $R_C$ in HC-C.

Figure 3: Some graphics produced by AOAB (and manually post-processed and rearranged for presentation in this paper).
there will always be differences in the mean performance. Whether these differences are significant or not, however, is not a priori clear. It is thus necessary compare the results using statistical tests. The best practice may be the application of non-parametric tests since these make the least assumptions about the distribution of the run performance. AOAB here uses the Mann-Whitney U-test Mann and Whitney [24], Siegel and Castellan Jr. [32] in a two-tailed version with a significance level of 2%.

However, AOAB does not only apply the test at the end of the experiment, but at each recorded time step. This means that we can determine when and for how long a certain algorithm is significantly better than another one and when we cannot say that both behave differently without making an error with high probability. In Fig. 3.g to Fig. 3.l, we joined all information given by the three algorithms over the function evaluations for dimension ∈ {200, 300, 400} for f2 (the Single-group Shifted m-dimensional Schwefel’s Problem 1.2 combined with the Sphere function [36]). All three diagrams exhibit the same behavior: in the comparison between HC-C and RS (red line), initially there is no significant difference. For the next approximate 70 FEs, the random sampling finds improvements more quickly than the hill climber. When its fast convergence phase is reached, HC-C then outperforms the random search significantly. The relation between HC-G and RS looks quite similar.

More interesting is the comparison between HC-G and HC-C: After neither of the two being better than the other in the startup phase, the Cauchy-distributed neighborhood sampling picks up pace faster and for some time, outperforms HC-G significantly. However, for all settings of the parameter dimension, after around 1000 FEs the situation turns around and HC-G suddenly finds the better solutions. For this behavior, we believe the reason is as follows: The Cauchy distribution applied in our algorithm has a heavier tail than the Gaussian distribution used (see Fig. 3.m). The latter is thus more likely to sample points closer to its center which, presumably, leads to shorter search steps. The HC-G algorithm therefore likely needs more search steps when approaching the optimum. However, once a certain proximity to the optimum is reached, the higher probability of HC-G to sample the close neighborhood allows it to find improvements for a longer time than HC-C can. These assumptions are mainly based on the fact that both algorithms are non-adaptive. Sampling parallel to the axes in [15] may play a role in the initially higher speed of HC-C as well.

In Fig. 3.j to Fig. 3.l, we joined all information given by all tests in the different benchmark functions for the FE steps to score values. For every comparison an optimizer significantly wins in a step, it receives 1 point. In the eight functions, it could therefore score at most 16 points. The total score sum will always be between 0 and 24 for each step. A high score indicates better performance. The three comparison graphs in our example illustrate that in the beginning, none of our two hill climbers beats random sampling. During the first 1000 steps, HC-C seems to be the algorithm of choice. If more FEs are available, HC-G is the most promising approach in average in the benchmark.

With AOAB, such conclusions can not only be drawn by comparing algorithms, but also by comparing different parameter settings for the same algorithm. If the distribution used for sampling was a parameter choice in a hill climber, for example, the experiment we performed would indicate that switching between Cauchy and Gaussian distributions may be a good idea at some point in time (in an otherwise non-adaptive algorithm, that is).

6. Conclusion and Future Work
In this paper, we presented the architecture of an Automated Optimization Algorithm Benchmarking tool called AOAB. We outlined its ability of running efficient and distributed experimental series. It can not only be used for comparing algorithms, but also for gaining insight into the effects of algorithm parameter setting. We conduct a study on how the behavior of different optimizers changes when the scale of the problems increases. By doing so, we hope to complement existing theoretical work with practical experimental results and, at the same time, open new perspectives in understanding of optimization methods in a way similar to Section 5.3.

Another goal we strive to attain with AOAB in the midterm future is to federate research on optimization algorithms: A researcher may use her own AOAB server for her experiments. Additionally, a research group may also maintain a central AOAB server. If a researcher has finished a series of experiments and analysis and found an interesting algorithm, she may press a button in her local GUI to upload the algorithm and experiments from her local machine to the groups AOAB server. From this server, in turn, the data may be forwarded to the AOAB server hosted at our institute. Of course, this should also work the other way around, i. e., it should be possible to download benchmarks, functions, and algorithm implementations from a central server to the locally running instances. With such a federation, it would be easy to aggregate and keep track on a giant pool of performance data and to provide automatically updated websites featuring e.g. top-ten lists of algorithms for certain benchmarks.

7. REFERENCES


@inproceedings{WNT2010AOABAOAB,
    title = {AOAB -- Automated Optimization Algorithm Benchmarking},
    author = {Thomas Weise and Li Niu and Ke Tang},
    booktitle = {Black Box Optimization Benchmarking 2010 (BBOB 2010) at Genetic and Evolutionary Computation Conference (GECCO-2010) Companion Publication},
    month = jul # {7--11},
    page = {1479--1486},
    year = {2010},
    location = {Portland Marriott Downtown Waterfront Hotel, Portland, OR, USA},
    publisher = {ACM SIGEVO},
    doi = {10.1145/1830761.1830763},
    isbn = {978-1-4503-0073-5},
}