From benchmarking optimization heuristics to dynamic algorithm configuration

Vermetten, D.L.

Chapter 2

Preliminaries

In this chapter, we provide background information on several key aspects related to benchmarking and designing optimization algorithms. In particular, we outline many of the techniques we use throughout the remainder of the thesis to evaluate, analyze and configure optimization heuristics.

2.1 Iterative Optimization Heuristics

Throughout this thesis, we study optimization (minimization unless stated otherwise) of problems of the type $f \colon \mathcal{S} \to \mathbb{R}$, i.e., we assume our problem to be a single-objective, real-valued objective function (i.e., problems for which the quality of possible solutions is rated by real numbers), defined over a search space $\mathcal{S}$. While some of the methods discussed are independent of $\mathcal{S}$ (it can be discrete or continuous, constrained or unconstrained), we consider the case where $\mathcal{S} \subset \mathbb{R}^d$ unless stated otherwise. Generally, when dealing with benchmark problems, we assume $\mathcal{S} = [lb_i, ub_i]^d$, which we refer to as a *box-constrained* problem where lb_i and ub_i are the lower and upper bound of the i-th decision variable respectively.

These problems are often considered to be *black-box* problems: we don't assume any information about the underlying structure of the problem. For this type of problem, the only way to get information about it is to evaluate the quality of points $\mathbf{x} \in \mathcal{S}$. This type of problem is common in many real-world problems, e.g. when dealing with simulations of complex processes or physical experiments.

For some problems, some intermediate information about the problem structure is available, in which case we speak of *gray-box* optimization. This can be the case when

Algorithm 1 Blueprint of an iterative optimization heuristic (IOH) optimizing a function $f : \mathcal{S} \to \mathbb{R}$.

procedure IOH
  $t \leftarrow 0$ ▷ iteration counter
  $\mathcal{H}^{(0)} \leftarrow \emptyset$ ▷ search history information
  choose a distribution $\Lambda^{(0)}$ on $\mathbb{N}$ ▷ distribution of the number of samples
  while termination criterion not met **do**
    $t \leftarrow t + 1$
    sample $\lambda^{(t)} \sim \Lambda^{(t-1)}$ ▷ number of points to be evaluated
    Based on $\mathcal{H}^{(t-1)}$, choose a distribution $D^{(t)}$ on $\mathcal{S}^{\lambda(t)}$
    sample $\left(\mathbf{x}^{(t,1)}, \ldots, \mathbf{x}^{(t,\lambda(t))}\right) \sim D^{(t)}$ ▷ candidate generation
    evaluate $f\left(\mathbf{x}^{(t,1)}\right), \ldots, f\left(\mathbf{x}^{(t,\lambda(t))}\right)$ ▷ function evaluation
    choose $\mathcal{H}^{(t)}$ and $\Lambda^{(t)}$ ▷ information update
  end while
end procedure

information about variable interactions can be extracted from the problem formulation [259]. To ease notation, we will still consider these problems as *black-box* in this thesis. We emphasize that the sampling-based optimization algorithms studied in this thesis can be competitive even when the problem f is explicitly known. In pseudo-Boolean optimization, the low auto-correlation binary sequence (LABS) problem is a good example of such a problem that can be defined in two lines, but for which the best-known solvers are sampling-based [182].

The class of algorithms that we are interested in are *Iterative Optimization Heuristics* (IOHs). IOHs are entirely sampling-based, i.e., they sample the search space $\mathcal{S}$ and use the function values $f(x)$ of the evaluated samples x to guide the search. Algorithm 1 provides a blueprint for IOHs. Classical examples for IOHs are local search algorithms (this class includes Simulated Annealing [128] and Threshold Accepting [66] as two prominent examples), genetic and evolutionary algorithms [71], Bayesian Optimization and related global optimization algorithms [113], Estimation of Distribution algorithms [139], and Ant Colony Optimization algorithms [64]. In Section 2.3, we discuss the algorithms which this thesis focuses on.

2.2 Benchmark Problems: BBOB

This thesis is focused on continuous, single-objective, noiseless black-box optimization. As such, we make extensive use of the Black-Box Optimization Benchmarking (BBOB) suite. Originally developed as part of the COCO platform [95], BBOB is one of

the most used benchmark suites for continuous optimization. This suite contains 24 functions, which can be separated into five core classes based on their global properties, which are listed in Table 2.1.

While the suite is originally intended to be used for unconstrained optimization, in practice however, black box optimization functions like this are often considered to be box-constrained [8], in the case of BBOB with domain $[-5, 5]^d$.

Table 2.1: Classification of the noiseless BBOB functions based on their properties (multi-modality, global structure, separability, variable scaling, homogeneity, basin-sizes, global to local contrast). Predefined groups are separated by horizontal lines [96]. Table taken from [164].

	Function	multim.	gl.-struc.	separ.	scaling	homog.	basins	gl.-loc.
1	Sphere	none	none	high	none	high	none	none
2	Ellipsoidal separable	none	none	high	high	high	none	none
3	Rastrigin separable	high	strong	none	low	high	low	low
4	Bueche-Rastrigin	high	strong	high	low	high	med.	low
5	Linear Slope	none	none	high	none	high	none	none
6	Attractive Sector	none	none	high	low	med.	none	none
7	Step Ellipsoidal	none	none	high	low	high	none	none
8	Rosenbrock	low	none	none	none	med.	low	low
9	Rosenbrock rotated	low	none	none	none	med.	low	low
10	Ellipsoidal high-cond.	none	none	none	high	high	none	none
11	Discus	none	none	none	high	high	none	none
12	Bent Cigar	none	none	none	high	high	none	none
13	Sharp Ridge	none	none	none	low	med.	none	none
14	Different Powers	none	none	none	low	med.	none	none
15	Rastrigin multi-modal	high	strong	none	low	high	low	low
16	Weierstrass	high	med.	none	med.	high	med.	low
17	Schaffer F7	high	med.	none	low	med.	med.	high
18	Schaffer F7 mod. ill-cond.	high	med.	none	high	med.	med.	high
19	Griewank-Rosenbrock	high	strong	none	none	high	low	low
20	Schwefel	med.	deceptive	none	none	high	low	low
21	Gallagher 101 Peaks	med.	none	none	med.	high	med.	low
22	Gallagher 21 Peaks	low	none	none	med.	high	med.	med.
23	Katsuura	high	none	none	none	high	low	low
24	Lunacek bi-Rastrigin	high	weak	none	low	high	low	low

For each BBOB function, arbitrarily many problem instances can be generated by applying transformations to both the search space and the objective values [96] – such mechanism is implemented internally in BBOB and controlled via a unique identifier (usually referred to as instance identifier or IID) which defines the applied transformations. For most functions, the search space transformation is made up of rotations and translations (moving the optimum, usually uniformly in $[-4, 4]^d$). Since the objective values are also transformed (through shifting the function values), the performance measures used are generally relative to the global optimum value to allow for comparison of performance between instances, typically in a logarithmic scale.

While this instance generation method is certainly useful for many applications, it has not been without critique. In particular, the stability of low-level features under the used transformations might not be guaranteed [175]. In Section 3.2, we analyze the features of the BBOB instance generation mechanism in more detail.

2.3 Core Algorithms

For continuous optimization, there are several popular algorithm families that have been shown to be effective in the decades since they have been introduced. In this thesis, we focus on two evolutionary algorithms in particular: Differential Evolution (DE) [219] and Covariance Matrix Adaptation Evolution Strategies (CMA-ES) [98]. In addition to these two families, we often make use of a wider set of algorithms, many of them accessed via the Nevergrad framework [200], which are briefly introduced as well.

2.3.1 CMA-ES

One of the most commonly used types of evolutionary algorithms is the CMA-ES [98]. As with most ES, its sampling is based on a d-dimensional normal distribution, from which we can sample as follows:

$$\mathbf{y}^{(t+1,i)} = \mathcal{N}(\mathbf{0}, \mathbf{C}^{(t)}) \tag{2.1}$$

$$\mathbf{x}^{(t+1,i)} = \mathbf{m}^{(t)} + \sigma^{(t)}\mathbf{y}^{(t+1,i)} \tag{2.2}$$

Where $\mathbf{m}$ is the center of mass, $\mathbf{C}$ is the covariance matrix, σ is the step-size and i is the index of the individual in the population. The CMA-ES adapts these parameters based on the relative success of the previously sampled points. For the center of mass, an intermediate recombination strategy is used: the new mean is placed according to a weighted sum of the μ best individuals from the previous generation as follows:

$$\mathbf{m}^{(t+1)} = \mathbf{m}^{(t)} + \sum_{i=0}^{\mu} w_i(\mathbf{x}^{(t+1,i)} - \mathbf{m}^{(t)}) \tag{2.3}$$

Several variants for the recombination weights w_i have been proposed, but the most common variants use exponential decaying weights.

To update the covariance matrix, the CMA-ES incorporates not just information from the current step, but a weighted combination of information collected in the search so far, in the form of the rank-μ update:

$$\mathbf{C}^{(t+1)} = (1 - c_\mu)\mathbf{C}^{(t)} + c_\mu \sum_{i=0}^{\mu} \mathbf{y}^{(t+1,i)}(\mathbf{y}^{(t+1,i)})^T \tag{2.4}$$

Incorporating this previous information allows for a more robust estimation of the

covariance matrix which is most likely to lead to improving steps. However, the used weighted sum causes signs to be lost, which have to be reintroduced to preserve directionality. This is done in the form of the rank-one update, which utilizes the evolution path $\mathbf{p}_c$, which is an aggregation of the search path (selected steps) of the population through a number of successive generations. In practice, $\mathbf{p}_c$ is computed as an exponential moving average of the mean of the search distribution:

$$\mathbf{p}_c^{(t+1)} = (1 - c_c)\mathbf{p}_c^{(t)} + \sqrt{c_c(2 - c_c)\mu_{\text{eff}}}\frac{\mathbf{m}^{(t+1)} - \mathbf{m}^{(t)}}{\sigma^{(t)}} \tag{2.5}$$

Where c_c is the learning rate for the exponential smoothing of $\mathbf{p}_c$, and μ_{eff} denotes the variance effective selection mass.

The full update of the covariance matrix then reads:

$$\mathbf{C}^{(t+1)} = (1 - c_1 - c_\mu \sum_{i=0}^{\mu} w_i)\ \mathbf{C}^{(t)} \tag{2.6}$$

$$+ c_1 \mathbf{p}_c^{(t+1)} (\mathbf{p}_c^{(t+1)})^T \qquad \text{rank-one update} \tag{2.7}$$

$$+ c_\mu \sum_{i=0}^{\mu} w_i \mathbf{y}^{(t+1,i)} (\mathbf{y}^{(t+1,i)})^T \qquad \text{rank-}\mu\text{ update} \tag{2.8}$$

Finally, the stepsize σ has to be adapted. This is one of the most commonly modified aspects of the CMA-ES procedure, since the covariance adaptation procedure described above does not directly produce a corresponding stepsize update. However, implementations of CMA-ES usually employ the Cumulative Step length Adaptation (CSA), which also employs a form of evolution path: the conjugate evolution path $\mathbf{p}_\sigma^{(t+1)}$, which is constructed via:

$$\mathbf{p}_\sigma^{(t+1)} = (1 - c_\sigma)\mathbf{p}_\sigma^{(t)} + \sqrt{c_\sigma(2 - c_\sigma)\mu_{\text{eff}}}\mathbf{C}^{(t)^{-\frac{1}{2}}}\frac{\mathbf{m}^{(t+1)} - \mathbf{m}^{(t)}}{\sigma^{(t)}} \tag{2.9}$$

Here, the inverse square root of the covariance matrix is used, which can be calculated via eigendecomposition. However, this inversion is still computationally expensive, and thus it is recommended to only update this inverse every $\max(1, \lfloor 1/(10n(c_1 + c\mu)) \rfloor)$ generations [90].

To update σ, the length of the conjugate evolution path is compared with its expected length under random selection, i.e.: $\mathbb{E}||\mathcal{N}(\mathbf{0}, \mathbf{I})||$. If the evolution path is too short, this means that single steps are not making enough progress and cancel each other out, thus the step-size should be decreased. If the evolution path is long,

the progress made in previous steps is correlated, and thus could have been made with fewer steps, and requires an increase in step-size. With an additional dampening parameter d_σ, this allows $\sigma^{(t+1)}$ to be computed as:

$$\sigma^{(t+1)} = \sigma^{(t)} \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\mathbf{p}_\sigma^{(t+1)})}{\mathbb{E}||\mathcal{N}(\mathbf{0}, \mathbf{I})||} - 1\right)\right) \tag{2.10}$$

Throughout the years, many other step-size adaptation strategies have been proposed [72, 91]. In fact, given its popularity, a wide range of modifications and additional components for the CMA-ES have been published [3, 254, 155, 5]. This explosion of algorithm variants has been the catalyst for the development of modular versions of CMA-ES, starting with modEA [234, 232], which transformed into modCMA [51] and will be discussed in Chapter 4.1.2.

2.3.2 DE

As opposed to CMA-ES, DE does not make use of normal distributions for its candidate generation at all [219]. Instead, DE relies on differences between individuals in its population in the creation of its offspring. In the context of DE, each individual in the population undergoes crossover with a mutant individual, which is in turn created by adding one or more difference vectors to a reference vector. The most basic mutation operator is *rand/1*, where a mutant vector is created as follows:

$$\mathbf{v} = \mathbf{x}_1 + F \cdot (\mathbf{x}_2 - \mathbf{x}_3) \tag{2.11}$$

Where $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ are three random (distinct from $\mathbf{x}$ and each other) individuals from the population, and F is the mutation strength. This dependence on differences within the current population enables DE to implicitly scale its mutation without explicitly relying on a step-size mechanism.

Based on the created mutant vector, crossover is performed with the current individual $\mathbf{x}$ as follows:

$$\mathbf{x}'_i \leftarrow \begin{cases} \mathbf{v}_i & \text{if } \mathcal{U}(0,1) \leq Cr \text{ or } i = i_{rand} \\ \mathbf{x}_i & \text{otherwise} \end{cases} \tag{2.12}$$

Where Cr is the crossover rate, $i \in \{0 \dots d\}$ and i_{rand} is a randomly selected index to ensure at least one component from the mutant vector is used. After crossover, the created *trial* solution competes directly against its parent to determine which

individual is added to the next generation.

The structure of DE is modular by design, with the mutation and crossover operators functioning completely separately. This has led to the proposal of many different versions of these operators [266, 73, 106, 49, 37], in addition to a wide range of adaptation mechanisms for their corresponding parameters [224, 266]. In Chapter 4.1.1 we propose a modular version of DE that incorporates a large selection of these modifications.

2.3.3 Other Common Algorithms

PSO Particle Swarm Optimization (PSO) [123] is population-based method where individuals are moved in the search space based on a velocity that is updated according to an individual's search history and the history of its connected individuals.

BFGS Broyden-Fletcher-Goldfarb-Shannon (BFGS) [29, 77, 85, 212] is a Quasi-Newton method that approximates the Jacobian or the Hessian instead of actually computing them.

MLSL Multi-Level Single Linkage (MLSL) [147, 183] is an algorithm that combines global search phases based on clustering with more focused, local search procedures.

Cobyla Constrained optimization by linear approximation (Cobyla) [191] is an algorithm that iteratively optimizes linear approximations of the objective function using trust regions.

EMNA Estimation of Multivariate Normal Algorithm (EMNA) [140] is an estimation of distribution algorithm that uses a multivariate normal distribution.

NGopt An algorithm selection wizard implemented in the Nevergrad platform that selects the optimizer to use based on high-level features such as available budget and problem dimensionality [166].

2.4 Performance Indicators

Since the iterative optimization heuristics only query the objective function through sampling, we use function evaluations as the basis for most performance measures. While this is different from many other disciplines in which solutions are generated

constructively, it allows for much easier ways to compare performance than measures such as CPU time, since function evaluation counts are independent of the used hardware [92].

Generally, we have two axes in which we can consider performance. The first is to consider a *fixed-budget* setting: we have a limited number of function evaluations available and want to find the best possible solution within this budget. This is orthogonal to the *fixed-target* setting, where we have a given quality target and want to reach this threshold with the fewest possible function evaluations.

As discussed earlier, many state-of-the-art IOHs are randomized in nature, therefore yielding random performance traces even when the underlying problem f is deterministic. The performance space is spanned by the number of evaluations, by the quality of the assessed solutions, and by the probability that the algorithm has found within a given budget of function evaluations a solution that is at least as good as a given quality threshold.

Basic Notation To define the performance measures used in this thesis, we use the following notation.

- $\mathcal{F}$ denotes the set of problems under consideration. Each problem (or problem instance, depending on the context) $f \in \mathcal{F}$ is assumed to be a function $f : \mathcal{S} \to \mathbb{R}$. The *dimensionality* of $\mathcal{S}$ is denoted by d. We often consider scalable functions that are defined for several or all dimensions $d \in \mathbb{N}$. In such cases, we make the dimension explicit.

- $\mathcal{A} = \{A_1, A_2, \ldots\}$ is the set of algorithms under consideration. $\mathcal{A}$ can be finite or infinite. Often, $\mathcal{A}$ is a configurable meta-algorithmic framework, which allows users to specify parameters such as the degree of parallelism, the intensity of the local perturbations, the memory size, the use (or not) of recombination operators, etc.

- We denote by r the number of independent runs of an algorithm $A \in \mathcal{A}$ on problem $f \in \mathcal{F}$ in dimension d.

- $T(A, f, d, B, v, i) \in \mathbb{N} \cup \{\infty\}$ is a *fixed-target measure*. It denotes the number of function evaluations that algorithm A performed, in its i-th run and when minimizing the d-dimensional variant of problem f, to find a solution x satisfying $f(x) \leq v$. When A did not succeed in finding such a solution within the maximal allocated budget B, $T(A, f, d, B, v, i)$ is set to ∞. Several ways to deal with

such failures are considered in the literature, as we shall discuss in the next paragraphs.

- Similar to the above, $V(A, f, d, t, i) \in \mathbb{R}$ is a *fixed-budget measure.* It denotes the function value of the best solution that algorithm A evaluated within the first t evaluations of its i-th run, when minimizing the d-dimensional variant of problem f.

Descriptive Statistics We next recall some basic descriptive statistics.

- The average function value over r runs given a budget value t is simply

$$\bar{V}(t) = \bar{V}(A, f, d, t) = \frac{1}{r} \sum_{i=1}^{r} V(A, f, d, t, i).$$

  As we do with all other measures, we omit explicit mention of A, f, and d when they are clear from the context.

- The *Penalized Average Runtime* (PAR-c score, where $c \geq 1$ is the penalty factor) for a given target value v is defined as

$$\text{PAR-c}(v) = \text{PAR-c}(A, f, d, B, v) = \frac{1}{r} \sum_{i=1}^{r} \min \{T(A, f, d, B, v, i), cB\}, \quad (2.13)$$

  i.e., the PAR-c score is identical to the sample mean when all runs successfully identified a solution of quality at least v within the given budget B, whereas non-successful runs are counted as cB. In IOHanalyzer, we typically study the PAR-1 score, which, in abuse of notation, we also refer to as the mean runtime.

- Apart from mean values, we are often interested in quantiles, and in particular in the *sample median* of the r values $\{T(A, f, d, B, v, i)\}_{i=1}^{r}$ and $\{V(A, f, d, t, i)\}_{i=1}^{r}$, respectively.

- We also study the *sample standard deviation* of the running times and function values, respectively.

- The *empirical success rate* is the fraction of runs in which algorithm A reached the given target v within the maximal number B of allowed function evaluations.

That is, in the case of a minimization problem,

$$\widehat{p}_s = \widehat{p}_s(A, f, d, B, v) \tag{2.14}$$

$$= \frac{1}{r} \sum_{i=1}^{r} \mathbb{1}(V(A, f, d, B, i) < v) \tag{2.15}$$

$$= \frac{1}{r} \sum_{i=1}^{r} \mathbb{1}(T(A, f, d, B, v, i) < \infty), \tag{2.16}$$

where $\mathbb{1}(\mathcal{E})$ is the characteristic function of the event $\mathcal{E}$.

Expected Running Time An alternative to the PAR-c score is the *expected running time (ERT)*. ERT assumes independent restarts of the algorithm whenever it did not succeed in finding a solution of quality at least v within the allocated budget B. Practically, this corresponds to sampling indices $i \in \{1, \ldots, r\}$ (i.i.d. uniform sampling with replacement) until hitting an index i with a corresponding value $T(A, f, d, B, v, i) < \infty$. The running time would then have been $mB + T(A, f, d, B, v, i)$, where m is the number of sampled indices of unsuccessful runs. The average running time of such a hypothetically restarted algorithm is then estimated as

$$\begin{aligned} \mathrm{ERT}(A, f, d, B, v) &= \frac{\sum_{i=1}^{r} \min\{T(A, f, d, B, v, i), B\}}{r\widehat{p}_s} \\ &= \frac{\sum_{i=1}^{r} \min\{T(A, f, d, B, v, i), B\}}{\sum_{i=1}^{r} \mathbb{1}(T(A, f, d, B, v, i) < \infty)}. \end{aligned} \tag{2.17}$$

Note that ERT can take an infinite value when none of the runs was successful in identifying a solution of quality better than v.

Cumulative Distribution Functions For the fixed-target and fixed-budget analysis, we can also estimate probability density (mass) functions and compute *empirical cumulative distribution functions* (ECDFs). For the fixed-budget function value, its probability density function is estimated via the well-known *Kernel Density Estimation* (KDE) method [100], which approximates the density function by a superposition of kernel functions (e.g., Gaussian functions with a fixed width) centred at each data point. Intuitively, a set of crowded data points would lead to a very peaky empirical density due to massive superpositions of the kernel, while a set of distant points can only generate a relatively flat curve. For the fixed-target running time (an integer-

valued random variable), we estimate its probability mass function by treating it as a real value and applying the KDE method. For a set $\{T(A, f, d, v, i)\}_{i=1}^{r}$ of fixed-target running times, its ECDF is defined as the fraction of runs that successfully found a solution of quality at least as good as v within a budget of at most t function evaluations. That is,

$$\mathrm{ECDF}(A, f, d, v, t) = \frac{1}{r} \sum_{i=1}^{r} \mathbb{1}(T(A, f, d, v, i) \leq t).$$

ECDF values are most typically used in aggregated form. We use the following two aggregations:

- The aggregation over a set $\mathcal{V}$ of *target values*:

$$\mathrm{ECDF}(A, f, d, \mathcal{V}, t) = \frac{1}{r|\mathcal{V}|} \sum_{v \in \mathcal{V}} \sum_{i=1}^{r} \mathbb{1}(T(A, f, d, v, i) \leq t), \tag{2.18}$$

  i.e., the fraction of (run,target value) pairs (i, v) for which algorithm A has identified a solution of quality at least v within a budget of at most t function evaluations.

- Given a set of functions $\mathcal{F}$ and a mapping $\mathcal{V}\colon \mathcal{F} \to 2^{\mathbb{R}}$ that specifies the target values to consider for each function, the ECDF can be further aggregated by the following definition:

$$\mathrm{ECDF}(A, \mathcal{F}, d, \mathcal{V}, t) = \frac{1}{r \sum_{f \in \mathcal{F}} |\mathcal{V}(f)|} \sum_{f \in \mathcal{F}} \sum_{v \in \mathcal{V}(f)} \sum_{i=1}^{r} \mathbb{1}(T(A, f, d, v, i) \leq t). \tag{2.19}$$

For the BBOB suite in particular, we have a default setting for the set of targets to consider: $\mathcal{V} = \{10^{2-\frac{x}{5}}\}_{x \in 0 \ldots 51}$.

Area Over/Under the ECDF (AOC/AUC) The ECDF can be further aggregated by taking the area under the ECDF (AUC), which results in a measure of *anytime performance* over the included (function, target) pairs included in the ECDF [93]. By taking the difference between the maximal budget and the AUC, we get the area over the ECDF (AOC) instead, which is useful in two ways. First, it allows us to stick to minimization, simplifying visualizations. Second, the AOC is an approximation of the geometric average running time. As such, we often make use of (normalized) AOC in this thesis.

Area Over the Convergence Curve (AOCC) The AOCC is an anytime performance measure, which is equivalent to the area under the cumulative distribution curve (AUC) given infinite targets for the construction of the ECDF [78]. This measure is thus slightly more precise than the AUC, and does not require the selection of the targets to use. Instead, we only need to define the lower (f_l) and upper (f_u) bounds for the function values, as well as an optional scaling function $\mathcal{S}_f$.

$$AOCC(A, f, d, B) = \frac{1}{B \cdot r} \sum_{i=1}^{r} \sum_{t=1}^{B} \left(1 - \frac{clip((\mathcal{S}_f(V(A, f, d, t, i))), f_l, f_u) - f_l}{f_u - f_l}\right)$$

, where the *clip* function caps the function value to the range $[f_l, f_u]$. To remain consistent with the values used for AUC, and analysis of results on BBOB in general, we generally use 10^2 and 10^{-8} as the bounds for our function values, and perform a log-scaling after subtracting the global optimum before calculating the AOCC. We thus calculate the normalized AOCC as follows:

$$AOCC(A, f, d, B) = \frac{1}{B \cdot r} \sum_{i=1}^{r} \sum_{t=1}^{B} \left(1 - \frac{clip((\log_{10}(V(A, f, d, t, i) - f(x*)), -8, 2) + 8}{10}\right)$$

2.5 Exploratory Landscape Analysis

Since the performance of optimization algorithms relies on the problems which are being solved, it is important to focus our analysis not only on the algorithm, but also to look at the structure of the problems themselves. In our black-box context, we can not rely on any outside information about the problem, and thus we can only gain insights through sampling. The field of Exploratory Landscape Analysis [164] (ELA) aims to use information obtained through sampling to estimate the complexity of an optimization problem, by capturing its topology or landscape characteristics. More precisely, the high-level landscape characteristics of optimization problems, such as multi-modality, global structure and separability, are numerically quantified through classes of manually designed low-level features. These landscape characteristics, commonly referred to as ELA features, can be cheaply computed based on a Design of Experiments (DoE), consisting of some samples and their corresponding objective values.

In recent years, ELA has gained increasing attention in the *landscape-aware algorithm selection problem (ASP)* tasks, where the correlation between landscape characteristics and optimization algorithm performances has been intensively researched.

In fact, previous works have revealed that ELA features are indeed informative in explaining algorithm behaviors and can be exploited to reliably predict algorithm performances, e.g., using machine learning approaches [18, 107, 126]. Apart from ASP tasks, ELA has shown promising potential in other application domains, for instance, classification of the BBOB functions [204] and instance space analysis of different benchmark problem sets [216].

One of the most popular implementations of ELA is the Flacco package [127] (and its Python version pFlacco [192]), which incorporates several hundred features. We focus on the set of features which do not require additional sampling after the DoE. In particular, we consider features from six different classes: y-distribution, level set, meta-model, local search, curvature and convexity [164, 165]. While we are fully aware that these ELA features are highly sensitive to sample size [175] and sampling strategy [202, 253], they remain the best tool available at the moment for analyzing the low-level features of black-box problems.

2.6 Reproducibility

Reproducibility is an important aspect in any scientific domain, and evolutionary computation is no different [151]. In this field, many new algorithms are proposed or benchmarked without accompanying code being made public. At best, this hinders the usage of the method by other researchers, but more often it is combined with an underspecification of the algorithm, leading to ambiguities in its working principles.

To ensure the reproducibility of the results presented in this thesis, most publications discussed are linked to a repository on Zenodo. While the individual references to these repositories have been removed from this thesis, they can be found under the Zenodo profile "Diederick Vermetten".[1] In addition to these repositories, many of the datasets discussed throughout this thesis can be accessed directly on the IOHanalyzer website.[2]

[1] https://tinyurl.com/39v642nw
[2] https://iohanalyzer.liacs.nl