Introduction to Optimization: Benchmarking

September 21, 2018

TC2 - Optimisation

Université Paris-Saclay, Orsay, France

Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

1	Mon, 17.9.2018 Thu, 20.9.2018	Monday's lecture: introduction, example problems, problem types groups defined via wiki everybody went (actively!) through the Getting Started part of github.com/numbbo/coco
2	Fri, 21.9.2018	lecture "Benchmarking", final adjustments of groups everybody can run and postprocess the example experiment (~1h for final questions/help during the lecture)
3	Fri, 28.9.2018	lecture "Introduction to Continuous Optimization"
4	Fri, 5.10.2018	lecture "Gradient-Based Algorithms"
5	Fri, 12.10.2018	lecture "Stochastic Algorithms and DFO"
6	Fri, 19.10.2018	lecture "Discrete Optimization I: graphs, greedy algos, dyn. progr." deadline for submitting data sets
	Wed, 24.10.2018	deadline for paper submission
7	Fri, 26.10.2018	final lecture "Discrete Optimization II: dyn. progr., B&B, heuristics"
	29.10.-2.11.2018	vacation aka learning for the exams
	Thu, 8.11.2018 / Fri, 9.11.2018	oral presentations (individual time slots)
	Fri, 16.11.2018	written exam

All deadlines:
23:59pm Paris time

Course Overview

1	Mon, 17.9.2018 Thu, 20.9.2018	Monday's lecture: introduction, example problems, problem types groups defined via wiki everybody went (actively!) through the Getting Started part of github.com/numbbo/coco ❷ remaining part difficulties in cont. opt.
2	Fri, 21.9.2018	❸ today's lecture "Benchmarking", ❶ final adjustments of groups everybody can run and postprocess the example experiment (❹ ~1h for final questions/help during the lecture)
3	Fri, 28.9.2018	lecture "Introduction to Continuous Optimization"
4	Fri, 5.10.2018	lecture "Gradient-Based Algorithms"
5	Fri, 12.10.2018	lecture "Stochastic Algorithms and DFO"
6	Fri, 19.10.2018	lecture "Discrete Optimization I: graphs, greedy algos, dyn. progr." deadline for submitting data sets
7	Wed, 24.10.2018 Fri, 26.10.2018	deadline for paper submission final lecture "Discrete Optimization II: dyn. progr., B&B, heuristics"
	29.10.-2.11.2018	vacation aka learning for the exams
	Thu, 8.11.2018 / Fri, 9.11.2018	oral presentations (individual time slots)
	Fri, 16.11.2018	written exam

All deadlines:
23:59pm Paris time

❷ Problem Difficulties in Continuous Optimization

What Makes a Function Difficult to Solve?

- dimensionality

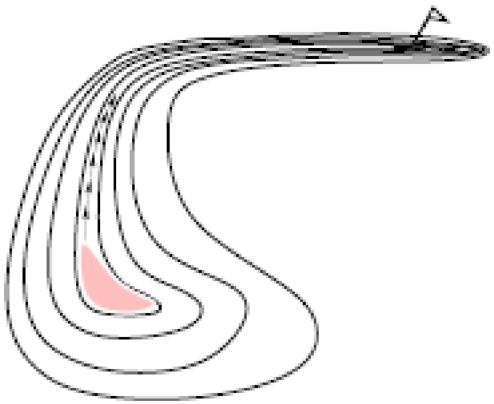
(considerably) larger than three

- non-separability

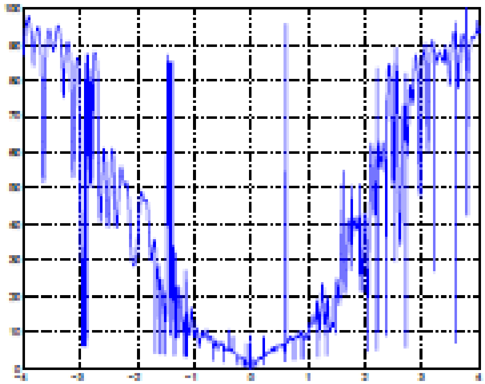
dependencies between the objective variables

- ill-conditioning
- ruggedness

non-smooth, discontinuous, multimodal, and/or noisy function



a narrow ridge



cut from 3D example, solvable with an evolution strategy

Curse of Dimensionality

- The term *Curse of dimensionality* (Richard Bellman) refers to problems caused by the rapid increase in volume associated with adding extra dimensions to a (mathematical) space.

- Example: Consider placing 100 points onto a real interval, say $[0,1]$. To get similar coverage, in terms of distance between adjacent points, of the 10-dimensional space $[0,1]^{10}$ would require $100^{10} = 10^{20}$ points. The original 100 points appear now as isolated points in a vast empty space.

- Consequently, a search policy (e.g. exhaustive search) that is valuable in small dimensions might be useless in moderate or large dimensional search spaces.

Definition (Separable Problem)

A function f is separable if

$$\underset{(x_1,\ldots,x_n)}{\operatorname{argmin}} f(x_1, \ldots, x_n) = \left(\underset{x_1}{\operatorname{argmin}} f(x_1, \ldots), \ldots, \underset{x_n}{\operatorname{argmin}} f(\ldots, x_n)\right)$$

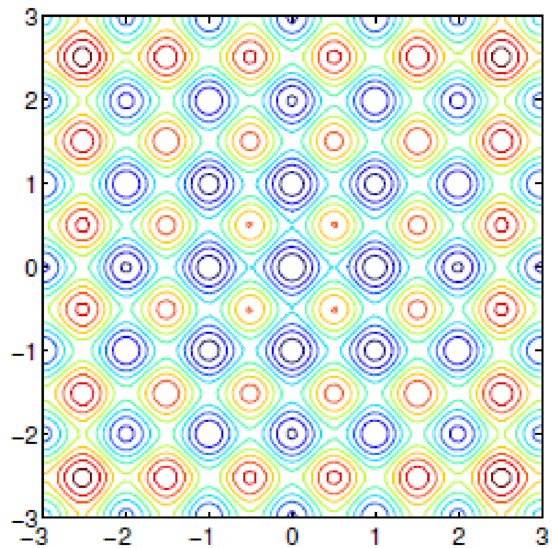
$\Longrightarrow$ *it follows that f can be optimized in a sequence of n independent 1-D optimization processes*

Example:

Additively decomposable functions

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} f_i(x_i)$$

Rastrigin function



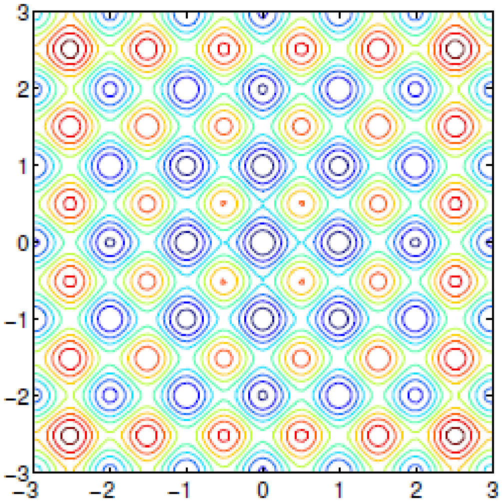
Non-Separable Problems

Building a non-separable problem from a separable one [1,2]

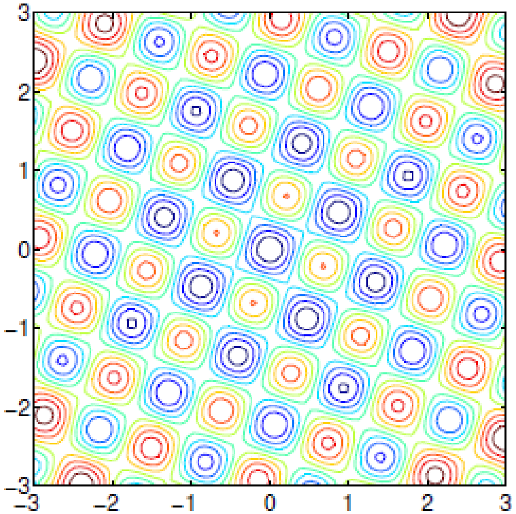
Rotating the coordinate system

- $f: \boldsymbol{x} \longmapsto f(\boldsymbol{x})$ separable
- $f: \boldsymbol{x} \longmapsto f(R\boldsymbol{x})$ non-separable

R rotation matrix



$\xrightarrow{R}$



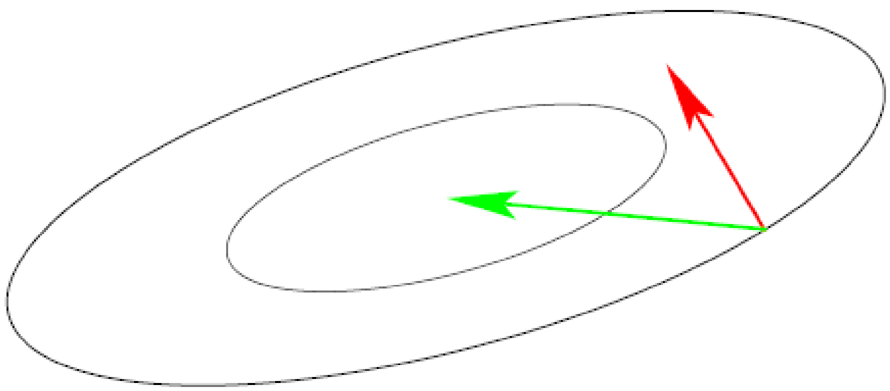
[1] N. Hansen, A. Ostermeier, A. Gawelczyk (1995). "On the adaptation of arbitrary normal mutation distributions in evolution strategies: The generating set adaptation". Sixth ICGA, pp. 57-64, Morgan Kaufmann

[2] R. Salomon (1996). "Reevaluating Genetic Algorithm Performance under Coordinate Rotation of Benchmark Functions; A survey of some theoretical and practical aspects of genetic algorithms." BioSystems, 39(3):263-278

Consider the convex-quadratic function

$$f(\boldsymbol{x}) = \frac{1}{2}(\boldsymbol{x}-\boldsymbol{x}^*)^T H(\boldsymbol{x}-\boldsymbol{x}^*) = \frac{1}{2}\sum_i h_{i,i}x_i^2 + \frac{1}{2}\sum_{i,j} h_{i,j}x_i x_j$$

H is Hessian matrix of f and symmetric positive definite



gradient direction $-f'(x)^T$

Newton direction $-H^{-1}f'(x)^T$

Ill-conditioning means squeezed level sets (high curvature). Condition number equals nine here. Condition numbers up to 10^{10} are not unusual in real-world problems.

If $H \approx I$ (small condition number of H) first order information (e.g. the gradient) is sufficient. Otherwise second order information (estimation of H^{-1}) information necessary.

Different Notions of Optimum

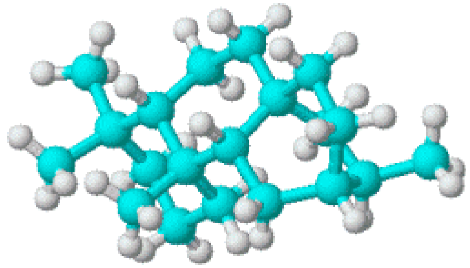
Unconstrained case

- local vs. global
  - local minimum $\boldsymbol{x}^*$: ∃ a neighborhood V of $\boldsymbol{x}^*$ such that $\forall \boldsymbol{x} \in \mathrm{V}: f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*)$
  - global minimum: $\forall \boldsymbol{x} \in \Omega: f(\boldsymbol{x}) \geq f(\boldsymbol{x}^*)$
- strict local minimum if the inequality is strict

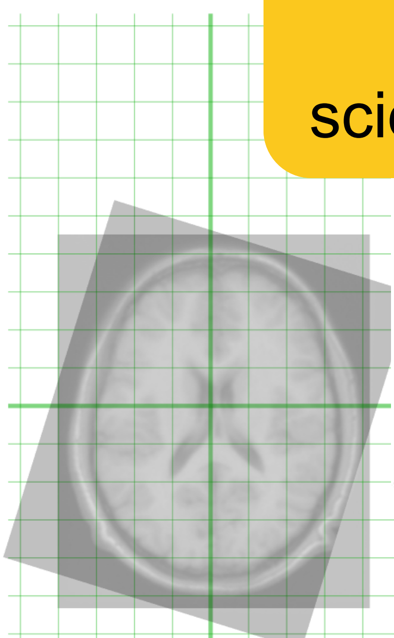
Constrained case

- a bit more involved
- hence, later in the lecture ☺

❸ Benchmarking Optimization Algorithms



challenging optimization problems
appear in many
scientific, technological and industrial domains



Practical (Numerical) Blackbox Optimization

Given:

derivatives not available or not useful

Not clear:

which of the many algorithms should I use on my problem?

Numerical Blackbox Optimizers

Deterministic algorithms

Quasi-Newton with estimation of gradient (**BFGS**) [Broyden et al. 1970]
Simplex downhill [Nelder & Mead 1965]
Pattern search [Hooke and Jeeves 1961]
Trust-region methods (NEWUOA, BOBYQA) [Powell 2006, 2009]

Stochastic (randomized) search methods

Evolutionary Algorithms (continuous domain)

- Differential Evolution [Storn & Price 1997]
- Particle Swarm Optimization [Kennedy & Eberhart 1995]
- **Evolution Strategies, CMA-ES** [Rechenberg 1965, Hansen & Ostermeier 2001]
- Estimation of Distribution Algorithms (EDAs) [Larrañaga, Lozano, 2002]
- Cross Entropy Method (same as EDA) [Rubinstein, Kroese, 2004]
- Genetic Algorithms [Holland 1975, Goldberg 1989]

Simulated annealing [Kirkpatrick et al. 1983]
Simultaneous perturbation stochastic approx. (SPSA) [Spall 2000]

Numerical Blackbox Optimizers

Deterministic algorithms

Quasi-Newton with estimation of gradient (**BFGS**) [Broyden et al. 1970]
Simplex downhill [Nelder & Mead 1965]
Pattern search [Hooke and Jeeves 1961]
Trust-region methods (NEWUOA, BOBYQA) [Powell 2006, 2009]

choice typically not immediately clear although practitioners have knowledge about which difficulties their problem has (e.g. multi-modality, non-separability, ...)

- **Evolution Strategies, CMA-ES** [Rechenberg 1965, Hansen & Ostermeier 2001]
- Estimation of Distribution Algorithms (EDAs) [Larrañaga, Lozano, 2002]
- Cross Entropy Method (same as EDA) [Rubinstein, Kroese, 2004]
- Genetic Algorithms [Holland 1975, Goldberg 1989]

Simulated annealing [Kirkpatrick et al. 1983]
Simultaneous perturbation stochastic approx. (SPSA) [Spall 2000]

Need: Benchmarking

- understanding of algorithms
- algorithm selection
- putting algorithms to a standardized test
  - simplify judgement
  - simplify comparison
  - regression test under algorithm changes

Kind of everybody has to do it (and it is tedious):

- choosing (and implementing) problems, performance measures, visualization, stat. tests, ...
- running a set of algorithms

that's where COCO comes into play

Comparing Continuous Optimizers Platform

`https://github.com/numbbo/coco`

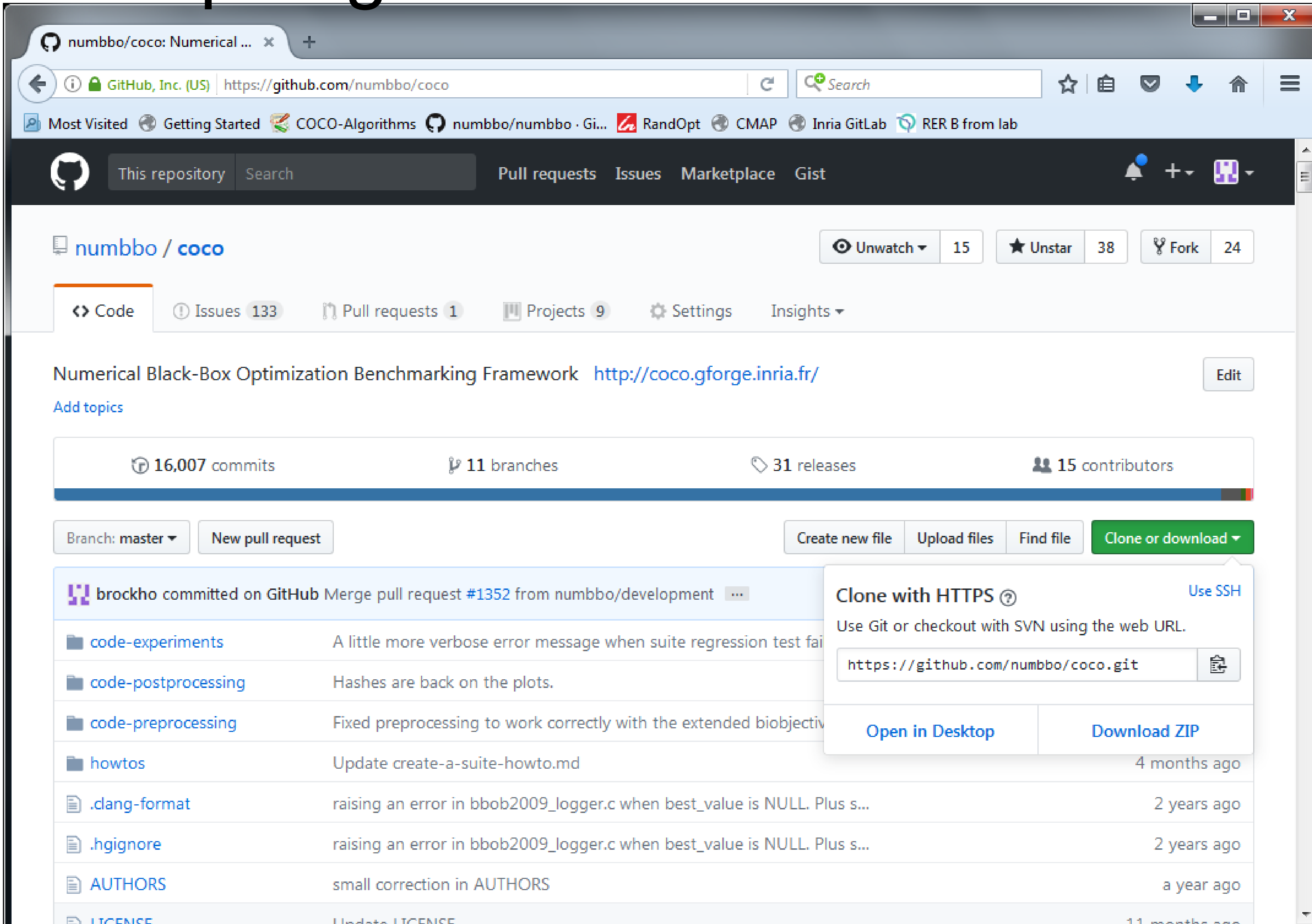
automatized benchmarking

benchmarking is non-trivial

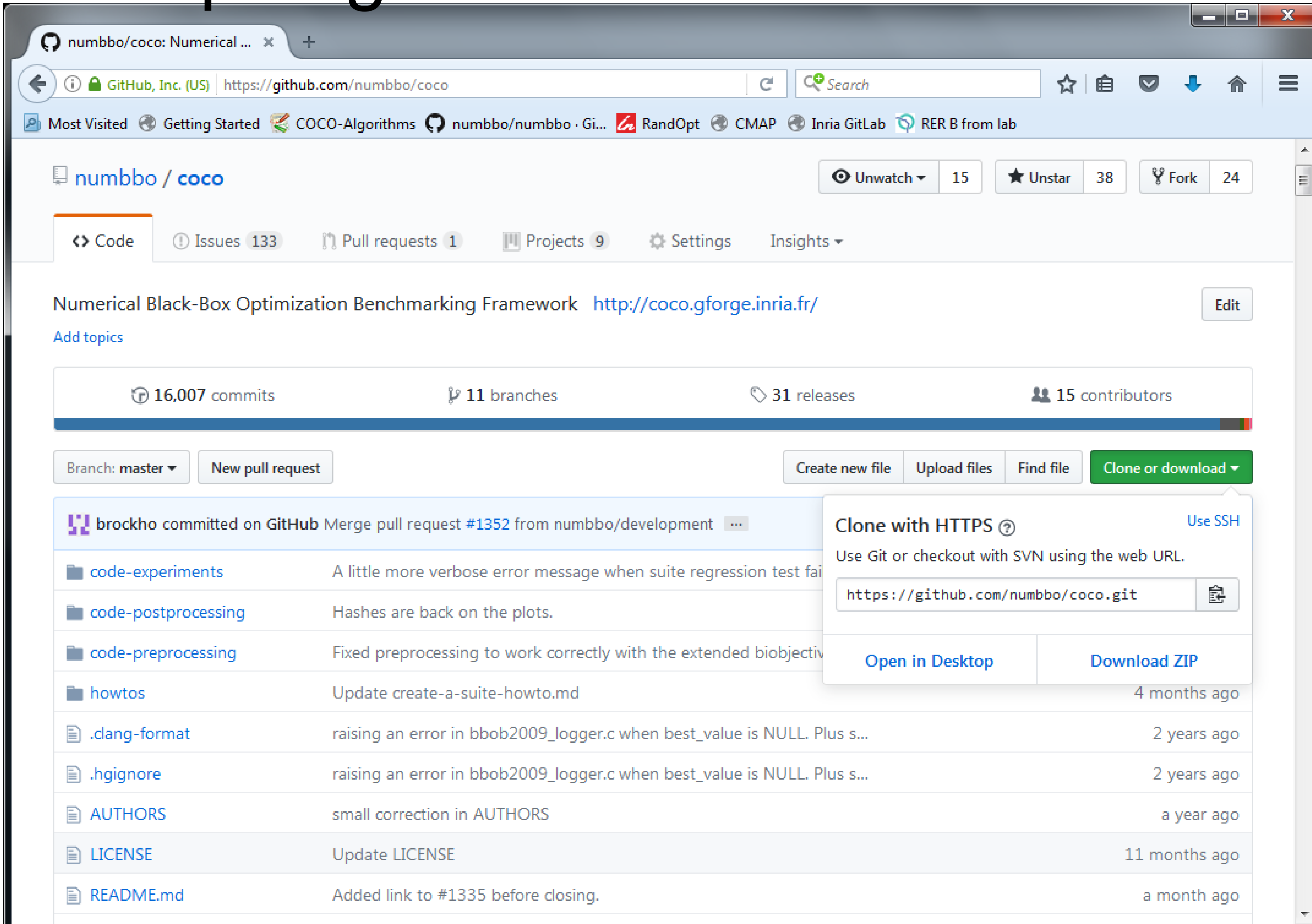
hence, COCO implements a
reasonable, well-founded, and
well-documented
pre-chosen methodology

How to benchmark algorithms with COCO?

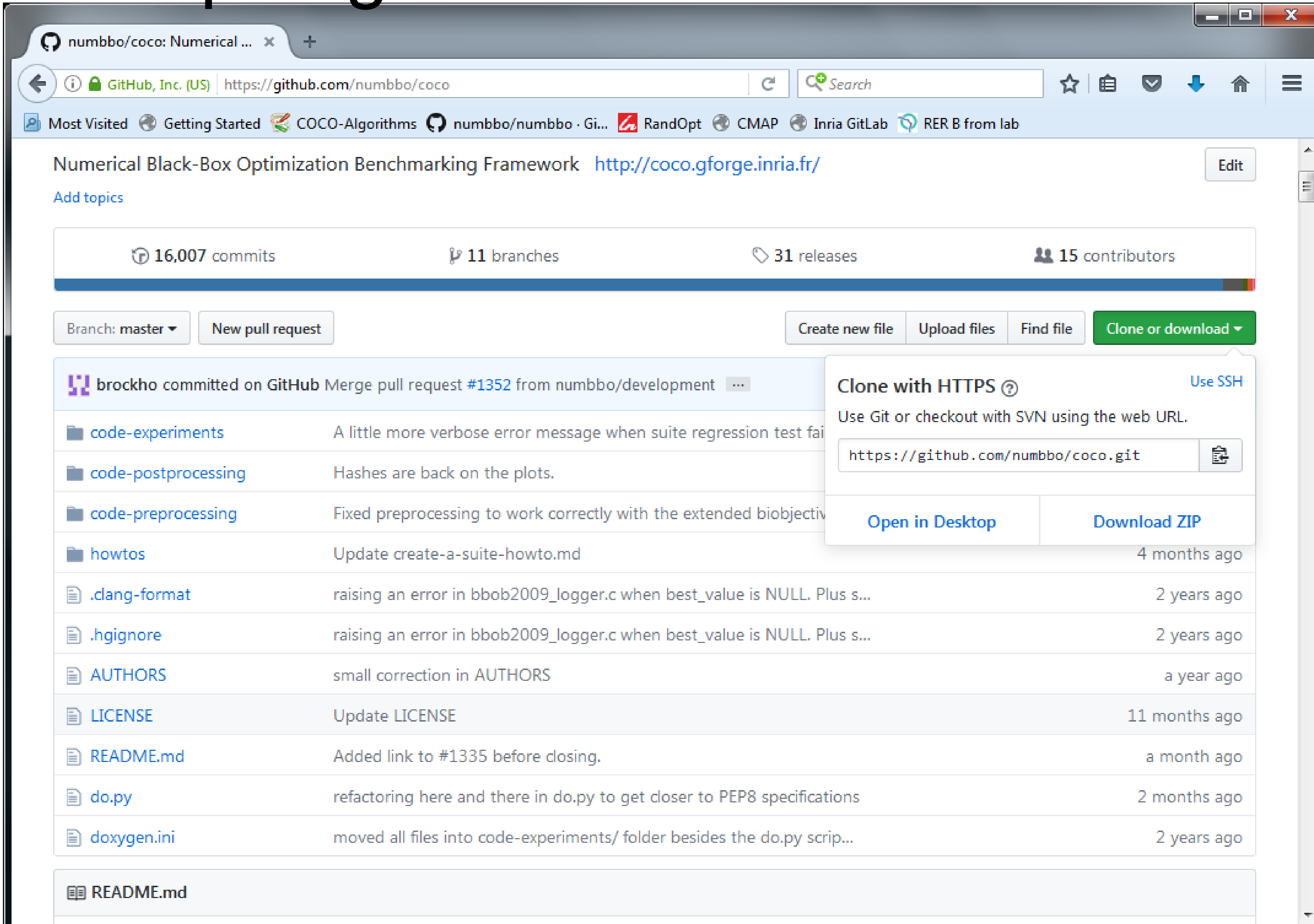
https://github.com/numbbo/coco



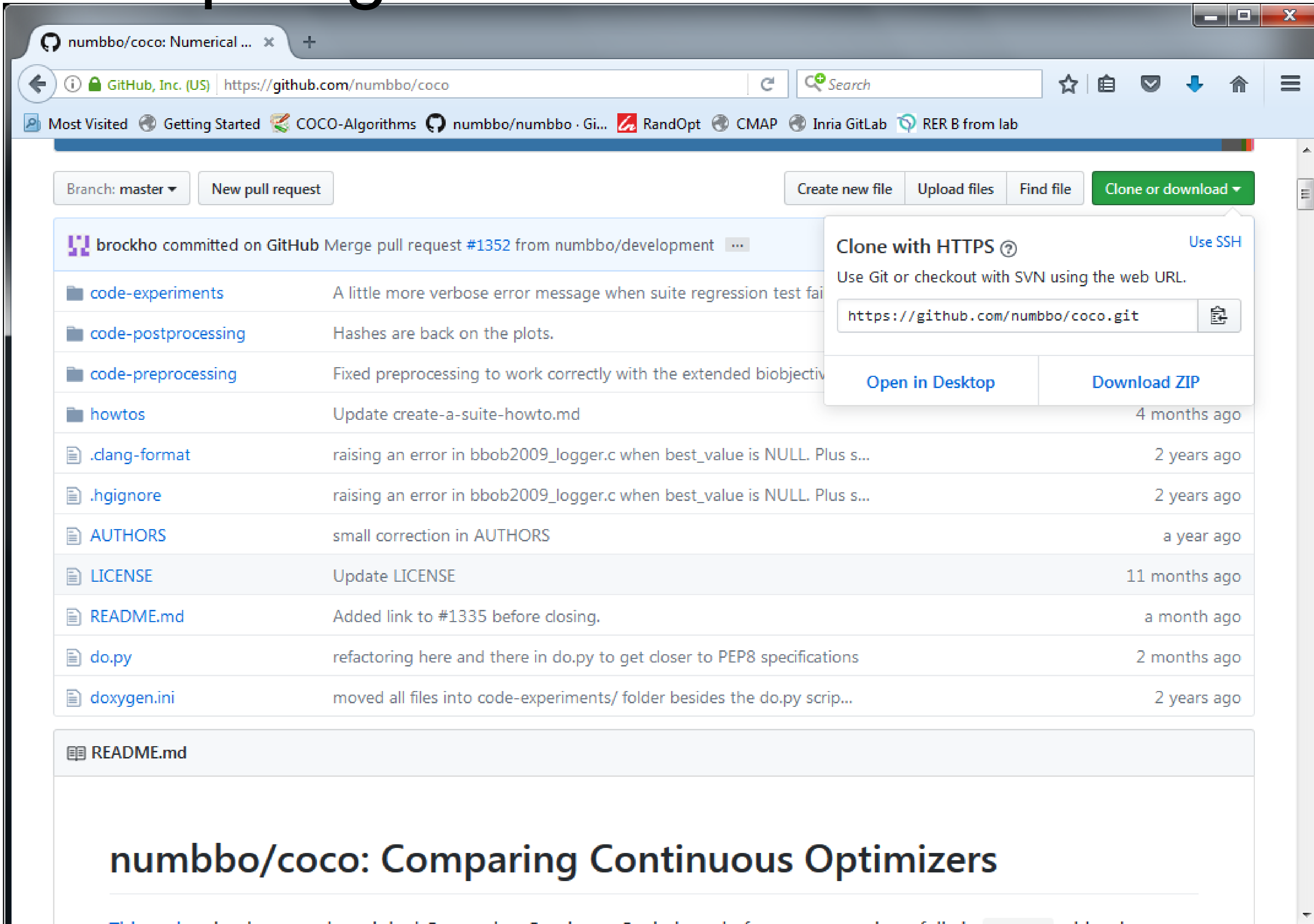
https://github.com/numbbo/coco



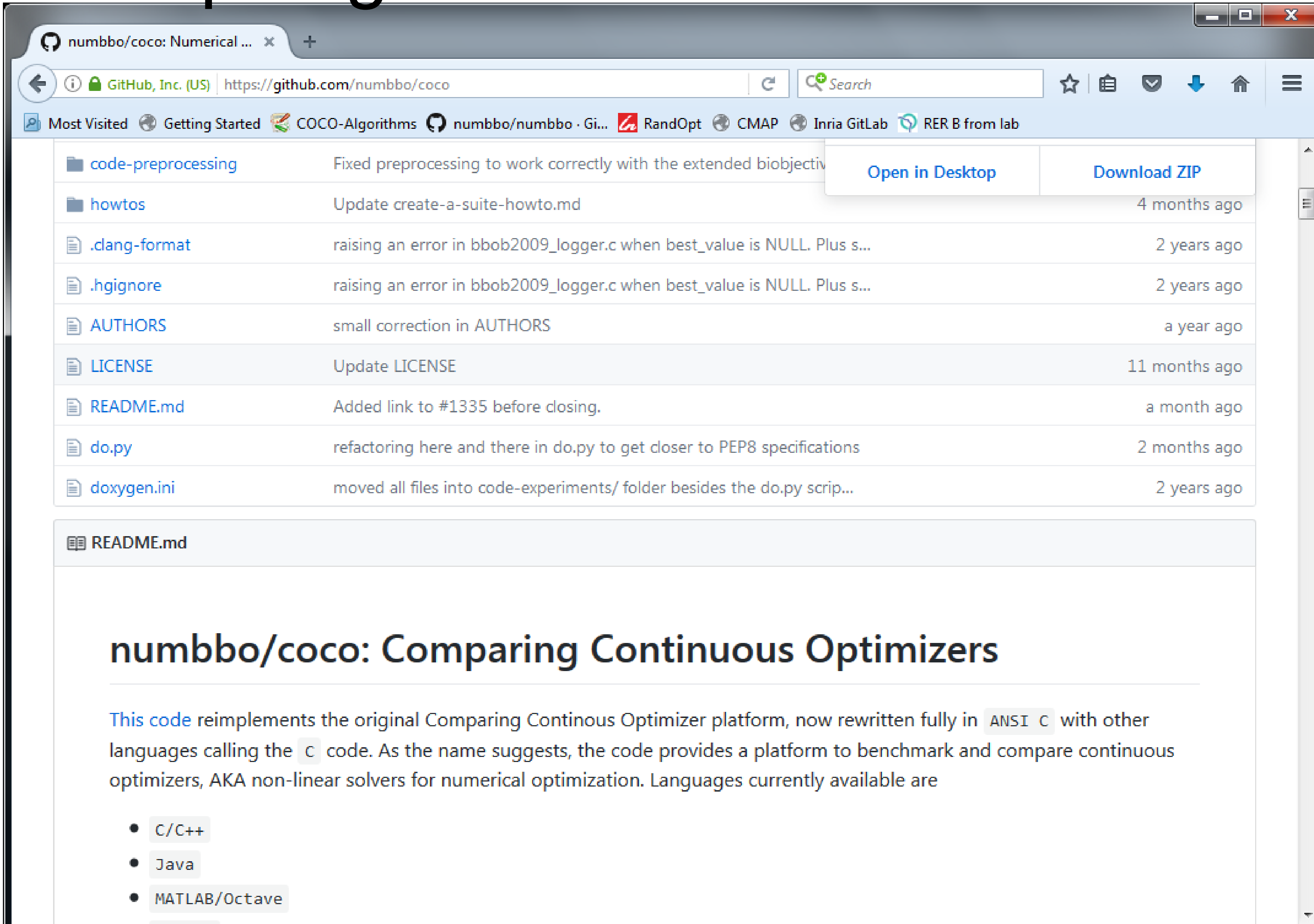
https://github.com/numbbo/coco



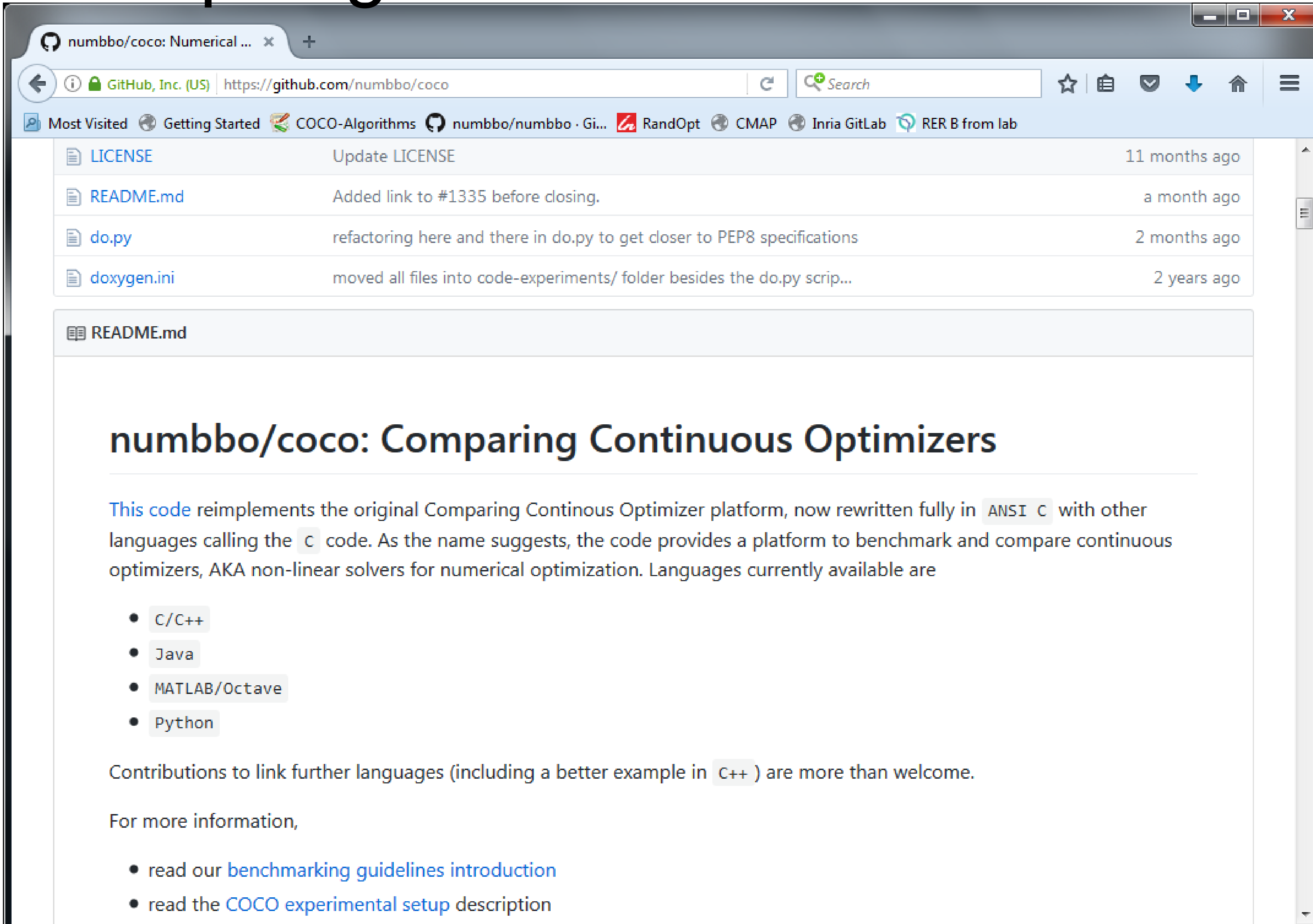
https://github.com/numbbo/coco



https://github.com/numbbo/coco



https://github.com/numbbo/coco



LICENSE	Update LICENSE	11 months ago
README.md	Added link to #1335 before closing.	a month ago
do.py	refactoring here and there in do.py to get closer to PEP8 specifications	2 months ago
doxygen.ini	moved all files into code-experiments/ folder besides the do.py scrip...	2 years ago

README.md

numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continous Optimizer platform, now rewritten fully in `ANSI C` with other languages calling the `C` code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers, AKA non-linear solvers for numerical optimization. Languages currently available are

- `C/C++`
- `Java`
- `MATLAB/Octave`
- `Python`

Contributions to link further languages (including a better example in `C++`) are more than welcome.

For more information,

- read our benchmarking guidelines introduction
- read the COCO experimental setup description

https://github.com/numbbo/coco

numbbo/coco: Comparing Continuous Optimizers

This code reimplements the original Comparing Continous Optimizer platform, now rewritten fully in `ANSI C` with other languages calling the `C` code. As the name suggests, the code provides a platform to benchmark and compare continuous optimizers, AKA non-linear solvers for numerical optimization. Languages currently available are

- `C/C++`
- `Java`
- `MATLAB/Octave`
- `Python`

Contributions to link further languages (including a better example in `C++`) are more than welcome.

For more information,

- read our benchmarking guidelines introduction
- read the COCO experimental setup description
- see the `bbob-biobj` and `bbob-biobj-ext` COCO multi-objective functions testbed documentation and the specificities of the performance assessment for the bi-objective testbeds.
- consult the BBOB workshops series,
- consider to register here for news,
- see the previous COCO home page here and
- see the links below to learn more about the ideas behind CoCO.

https://github.com/numbbo/coco

numbbo/coco: Numerical ...

GitHub, Inc. (US) https://github.com/numbbo/coco

Most Visited | Getting Started | COCO-Algorithms | numbbo/numbbo · Gi... | RandOpt | CMAP | Inria GitLab | RER B from lab

Getting Started

0. Check out the *Requirements* above.

1. **Download** the COCO framework code from github,

requirements & download

- either by clicking the Download ZIP button and unzip the `zip` file,
- or by typing `git clone https://github.com/numbbo/coco.git`. This way allows to remain up-to-date easily (but needs `git` to be installed). After cloning, `git pull` keeps the code up-to-date with the latest release.

The record of official releases can be found here. The latest release corresponds to the master branch as linked above.

2. In a system shell, `cd` **into** the `coco` or `coco-<version>` folder (framework root), where the file `do.py` can be found. Type, i.e. **execute**, one of the following commands once

```
python do.py run-c
python do.py run-java
python do.py run-matlab
python do.py run-octave
python do.py run-python
```

depending on which language shall be used to run the experiments. `run-*` will build the respective code and run the example experiment once. The build result and the example experiment code can be found under `code-experiments/build/<language>` (`<language>=matlab` for Octave). `python do.py` lists all available commands.

3. On the computer where experiment data shall be post-processed, run

```
python do.py install-postprocessing
```

https://github.com/numbbo/coco

Getting Started

0. Check out the *Requirements* above.

1. **Download** the COCO framework code from github,

- either by clicking the Download ZIP button and unzip the `zip` file,
- or by typing `git clone https://github.com/numbbo/coco.git`. This way allows to remain up-to-date easily (but needs `git` to be installed). After cloning, `git pull` keeps the code up-to-date with the latest release.

The record of official releases can be found here. The latest release corresponds to the master branch as linked above.

2. In a system shell, `cd` **into** the `coco` or `coco-<version>` folder (framework root), where the file `do.py` can be found. Type, i.e. **execute**, one of the following com

installation I: experiments

```
python do.py run-c
python do.py run-java
python do.py run-matlab
python do.py run-octave
python do.py run-python
```

depending on which language shall be used to run the experiments. `run-*` will build the respective code and run the example experiment once. The build result and the example experiment code can be found under `code-experiments/build/<language>` (`<language>=matlab` for Octave). `python do.py` lists all available commands.

3. On the computer where experiment data shall be post-processed, run

```
python do.py install-postprocessing
```

https://github.com/numbbo/coco

example experiment once. The build result and the example experiment code can be found under `code-experiments/build/<language>` (`<language>=matlab` for Octave). `python do.py` lists all available commands.

3. On the computer where experiment data shall b

```
python do.py install-postprocessing
```

installation II: postprocessing

to (user-locally) install the post-processing. From here on, `do.py` has done its job and is only needed again for updating the builds to a new release.

4. **Copy** the folder `code-experiments/build/YOUR-FAVORITE-LANGUAGE` and its content to another location. In Python it is sufficient to copy the file `example_experiment.py`. Run the example experiment (it already is compiled). As the details vary, see the respective read-me's and/or example experiment files:

- `C` read me and example experiment
- `Java` read me and example experiment
- `Matlab/Octave` read me and example experiment
- `Python` read me and example experiment

If the example experiment runs, **connect** your favorite algorithm to Coco: replace the call to the random search optimizer in the example experiment file by a call to your algorithm (see above). **Update** the output `result_folder`, the `algorithm_name` and `algorithm_info` of the observer options in the example experiment file.

Another entry point for your own experiments can be the `code-experiments/examples` folder.

5. Now you can **run** your favorite algorithm on the `bbob` suite (for single-objective algorithms) or on the `bbob-biobj` and `bbob-biobj-ext` suites (for multi-objective algorithms). Output is automatically generated in the specified data `result_folder`. By now, more suites might be available, see below.

https://github.com/numbbo/coco

example experiment once. The build result and the example experiment code can be found under `code-experiments/build/<language>` (`<language>=matlab` for Octave). `python do.py` lists all available commands.

3. On the computer where experiment data shall be post-processed, run

```
python do.py install-postprocessing
```

to (user-locally) install the post-processing. From here on, `do.py` has done its job and is only needed again for updating the builds to a new release.

4. **Copy** the folder `code-experiments/build/YOUR-FAVORITE-LANGUAGE` and its content to another location. In Python it is sufficient to copy the file `example_experiment.py`. Run the example experiment (it already is compiled). As the details vary, see the respective read-me's and/or example experiment files:

- `C` read me and example experiment
- `Java` read me and example experiment
- `Matlab/Octave` read me and example experiment
- `Python` read me and example experiment

coupling algo + COCO

If the example experiment runs, **connect** your favorite algorithm to Coco: replace the call to the random search optimizer in the example experiment file by a call to your algorithm (see above). **Update** the output `result_folder`, the `algorithm_name` and `algorithm_info` of the observer options in the example experiment file.

Another entry point for your own experiments can be the `code-experiments/examples` folder.

5. Now you can **run** your favorite algorithm on the `bbob` suite (for single-objective algorithms) or on the `bbob-biobj` and `bbob-biobj-ext` suites (for multi-objective algorithms). Output is automatically generated in the specified data `result_folder`. By now, more suites might be available, see below.

Simplified Example Experiment in Python

```python
import cocoex
import scipy.optimize

### input
suite_name = "bbob"
output_folder = "scipy-optimize-fmin"
fmin = scipy.optimize.fmin

### prepare
suite = cocoex.Suite(suite_name, "", "")
observer = cocoex.Observer(suite_name,
                           "result_folder: " + output_folder)

### go
for problem in suite:  # this loop will take several minutes
    problem.observe_with(observer)  # generates the data for
                                    # cocopp post-processing
    fmin(problem, problem.initial_solution)
```

Note: the actual example_experiment.py contains more advanced things like restarts, batch experiments, other algorithms (e.g. CMA-ES), etc.

https://github.com/numbbo/coco

numbbo/coco at develop...

GitHub, Inc. (US) https://github.com/numbbo/coco/tree/development

Most Visited · Getting Started · COCO-Algorithms · numbbo/numbbo · Gi... · RandOpt · CMAP · Inria GitLab · RER B from lab

Another entry point for your own experiments can be the `code-experiments/examples` folder.

5. Now you can **run** your favorite algorithm on the `bbob` suite (for single-objective algorithms) or on the `bbob-biobj` and `bbob-biobj-ext` suites (for multi-objective algorithms). Output is automatically generated in the specified data `result_folder`. By now, more suites might be available, see below.

6. **Postprocess** the data from the results folder by typing

```
python -m cocopp [-o OUTPUT_FOLDERNAME] YOURDAT
```

running the experiment

Any subfolder in the folder arguments will be searched fo... n different folders collected under a single "root" `YOURDATAFOLDER` folder. We can also compare more than one algorithm by specifying several data result folders generated by different algorithms.

A folder, p... file, usefu... the outpu...

A summa... template... template...

7. Once... indep...

tip:
start with small #funevals (until bugs fixed ☺)
then increase budget to get a feeling
how long a "long run" will take

8. The experiments can be **parallelized** with any re-distribution of single problem instances to batches (see `example_experiment.py` for an example). Each batch must write in a different target folder (this should happen automatically). Results of each batch must be kept under their separate folder as is. These folders then must be

https://github.com/numbbo/coco

numbbo/coco at develop...

GitHub, Inc. (US) https://github.com/numbbo/coco/tree/development

Most Visited · Getting Started · COCO-Algorithms · numbbo/numbbo · Gi... · RandOpt · CMAP · Inria GitLab · RER B from lab

Another entry point for your own experiments can be the `code-experiments/examples` folder.

5. Now you can **run** your favorite algorithm on the `bbob` suite (for single-objective algorithms) or on the `bbob-biobj` and `bbob-biobj-ext` suites (for multi-objective algorithms). Output is automatically generated in the specified data `result_folder`. By now, more suites might be available, see below.

6. **Postprocess** the data from the results folder by typing

```
python -m cocopp [-o OUTPUT_FOLDERNAME] YOURDATAFOLDER [MORE_DATAFOLDERS]
```

Any subfolder in the folder arguments will be searched for logged data. That is, experiments from different batches can be in different folders collected under a single "root" `YOURDATAFOLDER` ... specifying several data result folders generated by different algo...

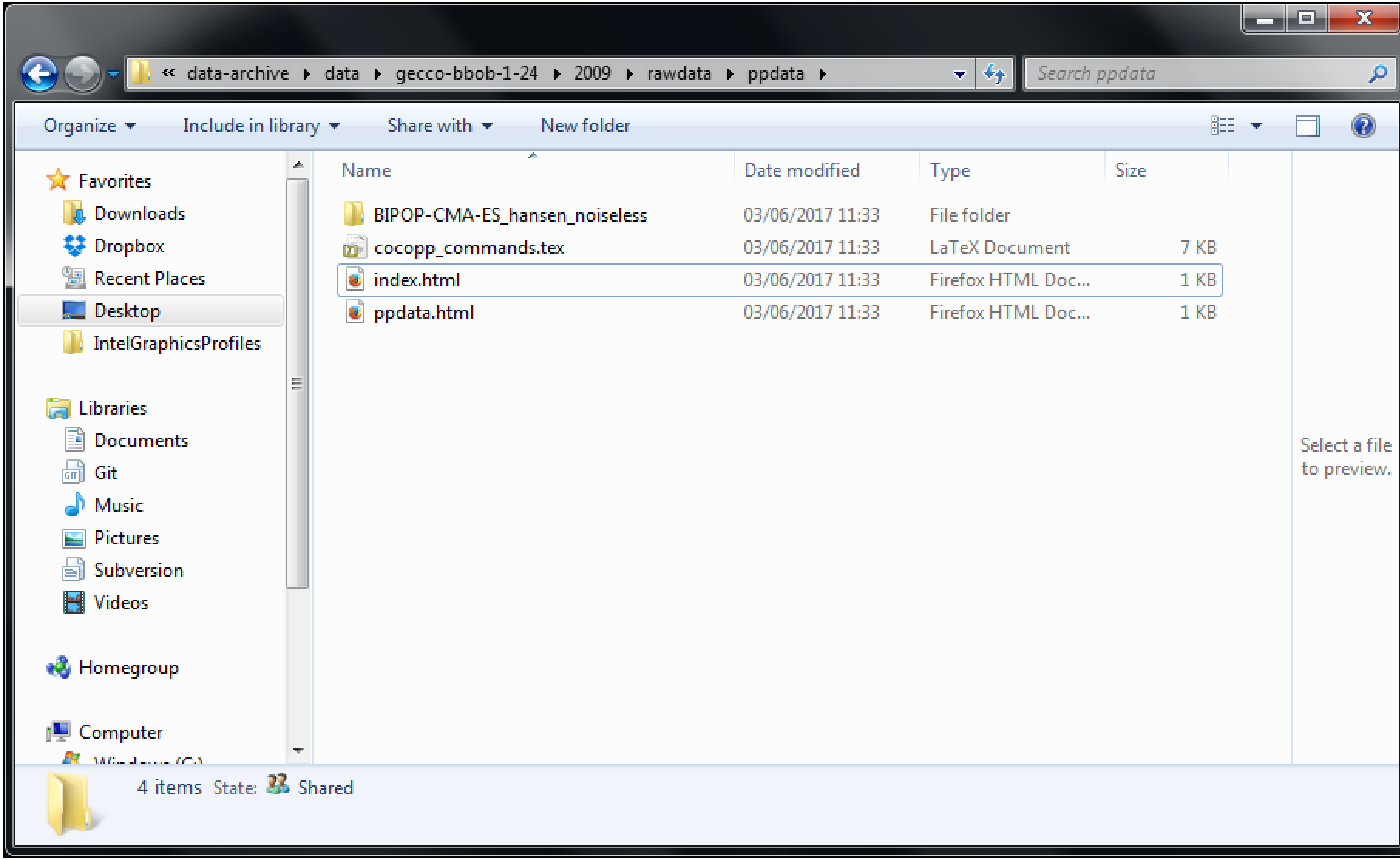
A folder, `ppdata` by default, will be generated, which contains a ... file, useful as main entry point to explore the result with a brows... the output folder name with the `-o OUTPUT_FOLDERNAME` option.

postprocessing

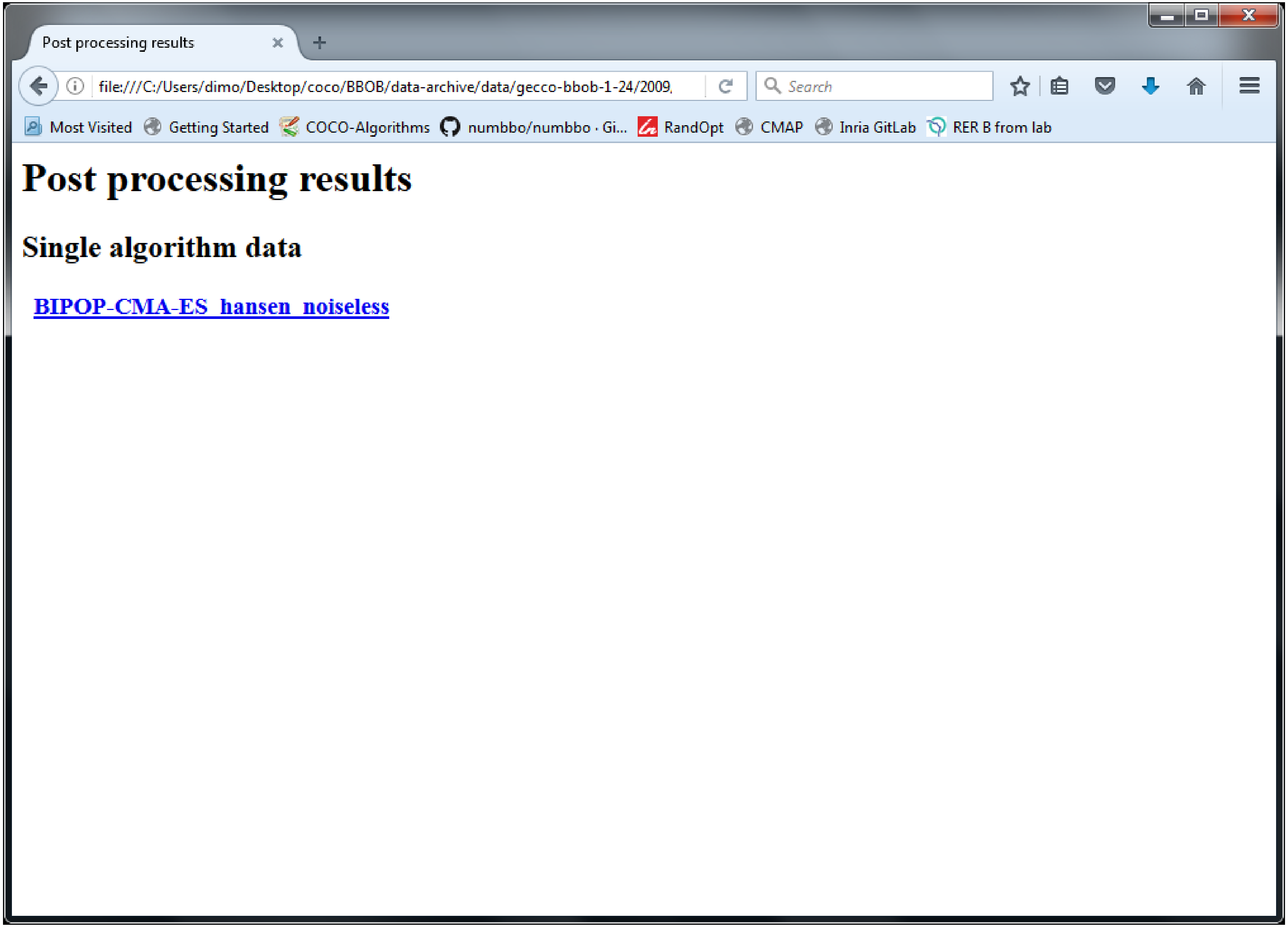
data from 200+ algorithms can be accessed directly through its name (see `http://coco.gforge.inria.fr/doku.php?id=algorithms`)

`example_experiment.py` for an example). Each batch must write in a different target folder (this should happen automatically). Results of each batch must be kept under their separate folder as is. These folders then must be

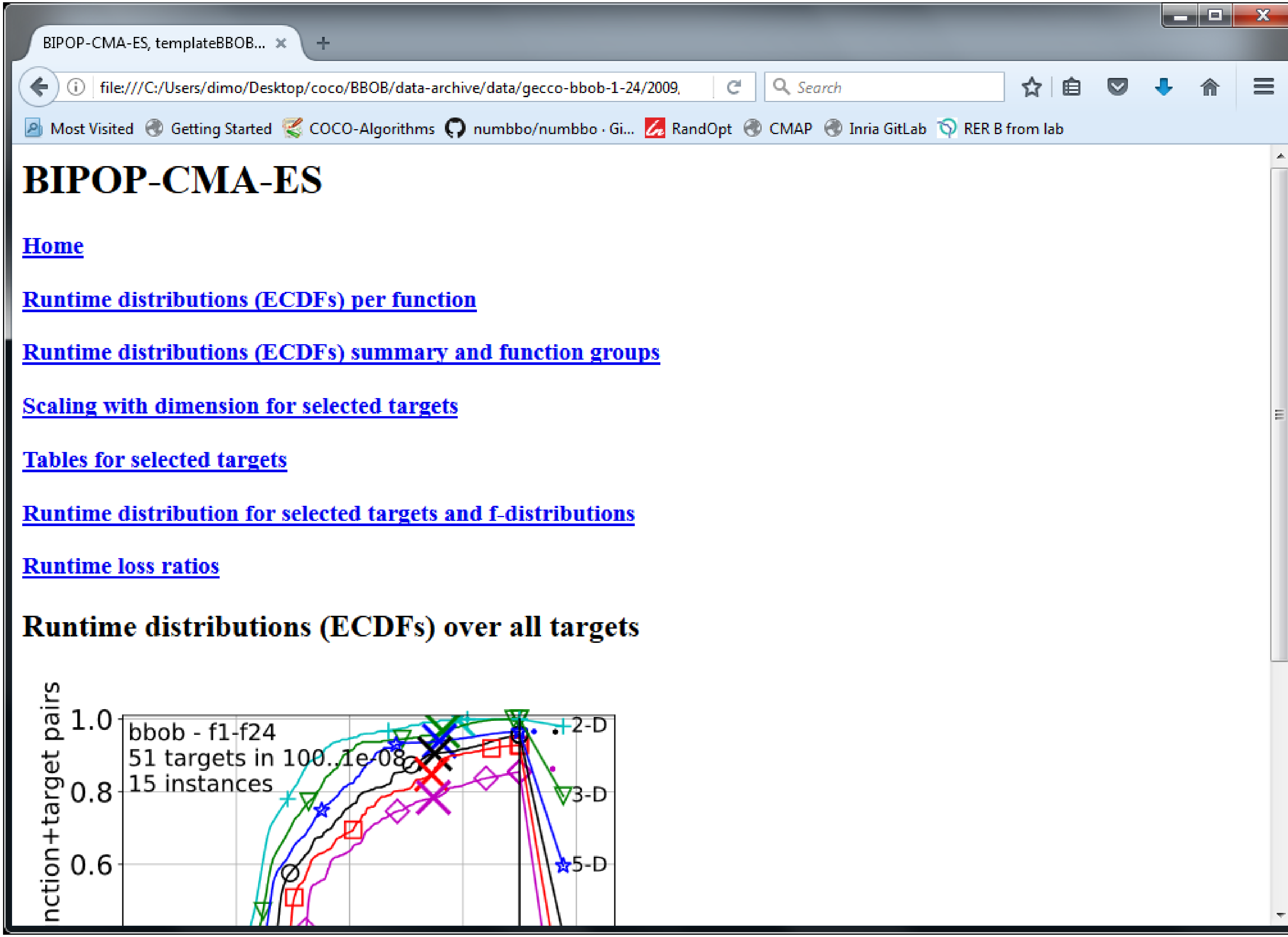
Result Folder



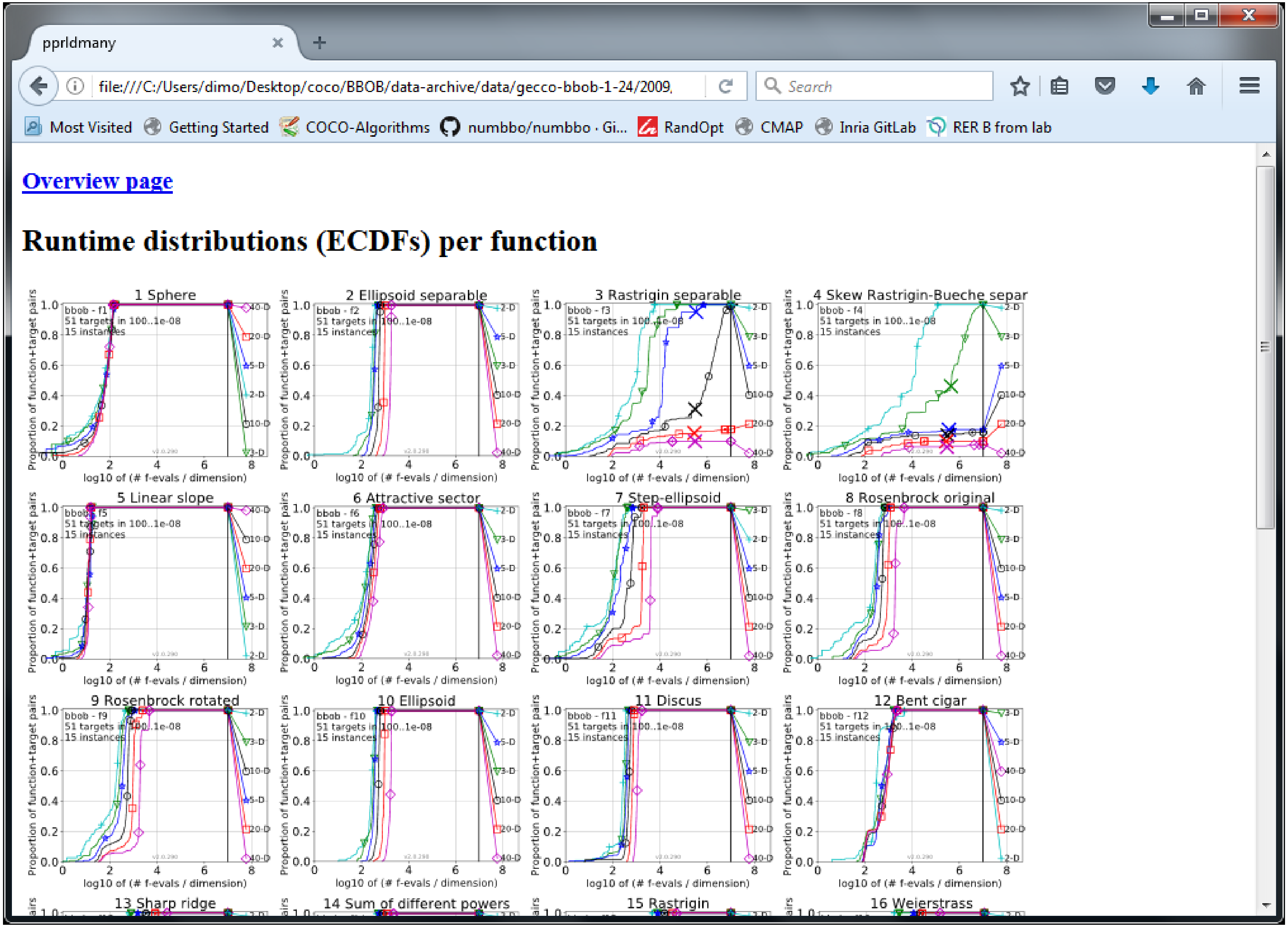
Automatically Generated Results



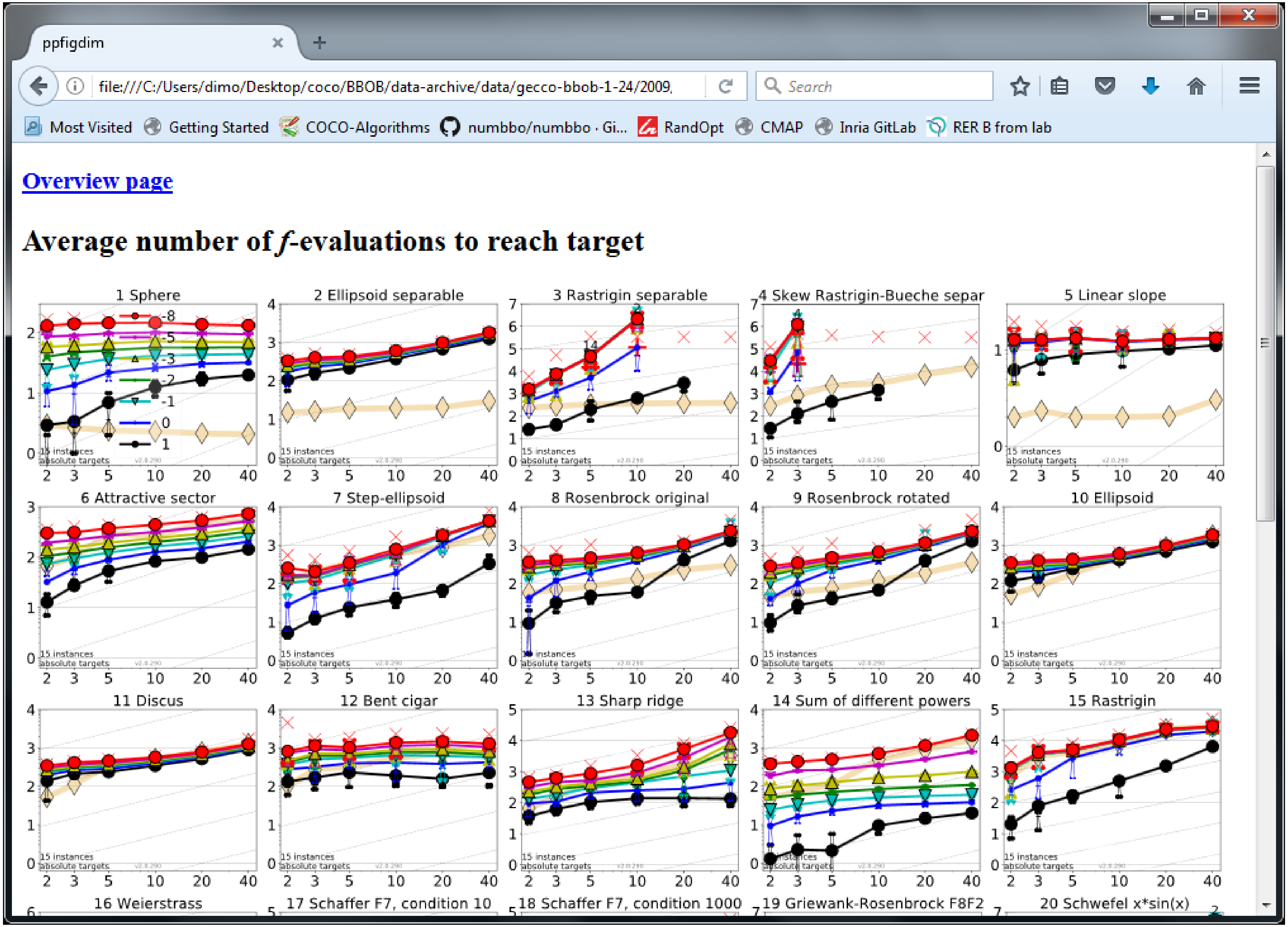
Automatically Generated Results



Automatically Generated Results



Automatically Generated Results



so far:

data for 200+ algorithm variants

(some of which on noisy or multiobjective test functions)

136 workshop papers

by 114 authors from 28 countries

used by another 77 students in the last two years

Measuring Performance

On

- real world problems
  - expensive
  - comparison typically limited to certain domains
  - experts have limited interest to publish
- "artificial" benchmark functions
  - cheap
  - controlled
  - data acquisition is comparatively easy
  - problem of representativeness

Test Functions

- define the "scientific question"

the relevance can hardly be overestimated

- should represent "reality"
- are often too simple?

remind separability

- a number of testbeds are around

- account for invariance properties

prediction of performance is based on “similarity”,
ideally equivalence classes of functions

Available Test Suites in COCO

bbob	24 noiseless fcts	140+ algo data sets
bbob-noisy	30 noisy fcts	40+ algo data sets
bbob-biobj	55 bi-objective fcts	16 algo data sets

soon to be released:

bbob-largescale

bbob-constrained

bbob-biobj-ext

How Do We Measure Performance?

Meaningful quantitative measure

- quantitative on the ratio scale (highest possible)

  "algo A is two *times* better than algo B" is a meaningful statement

- assume a wide range of values
- meaningful (interpretable) with regard to the real world

  possible to transfer from benchmarking to real world

runtime or first hitting time is the prime candidate
(we don't have many choices anyway)

How Do We Measure Performance?

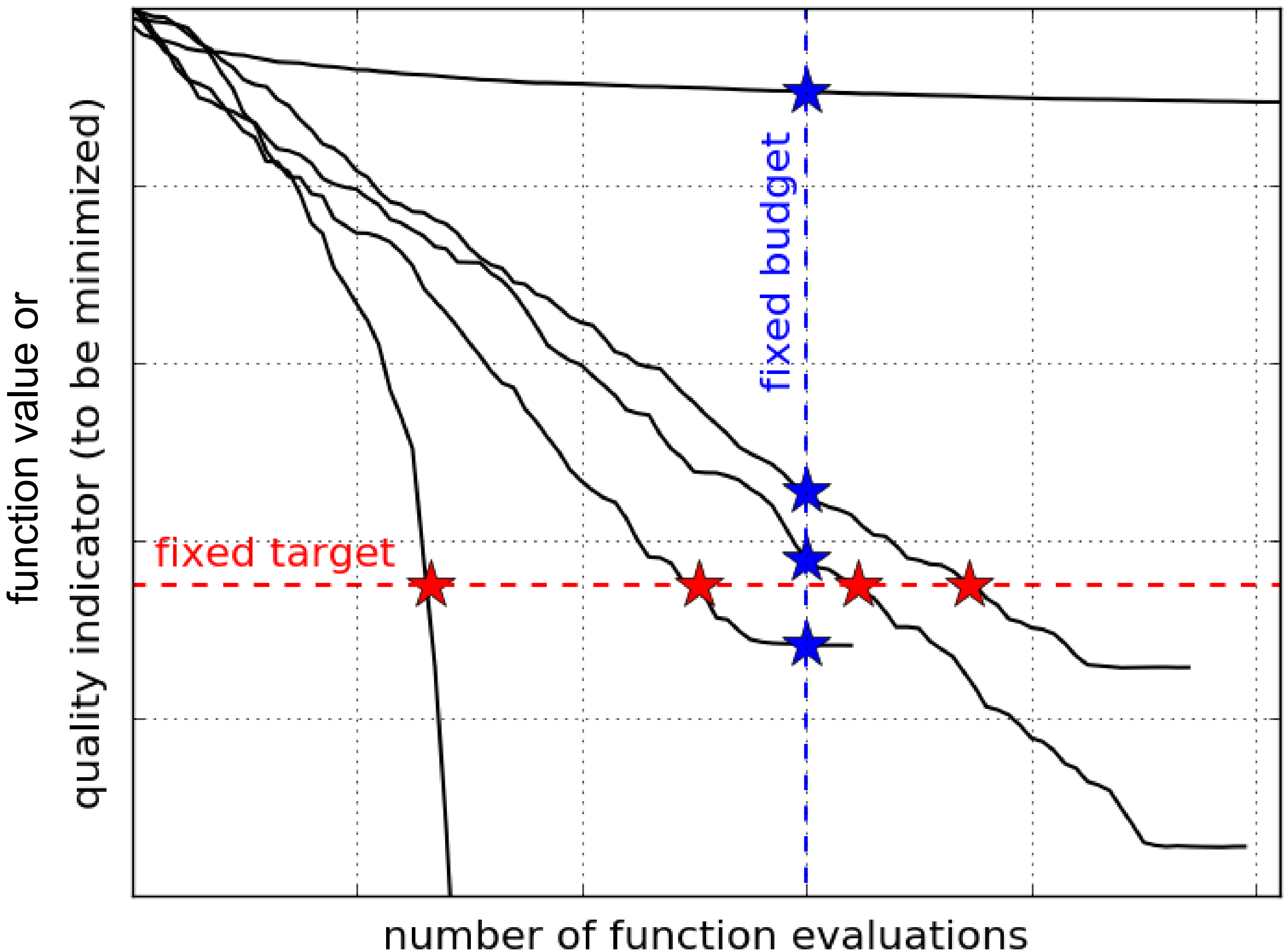
Two objectives:

- Find solution with small(est possible) function/indicator value
- With the least possible search costs (number of function evaluations)

For measuring performance: fix one and measure the other

Measuring Performance Empirically

convergence graphs is all we have to start with...

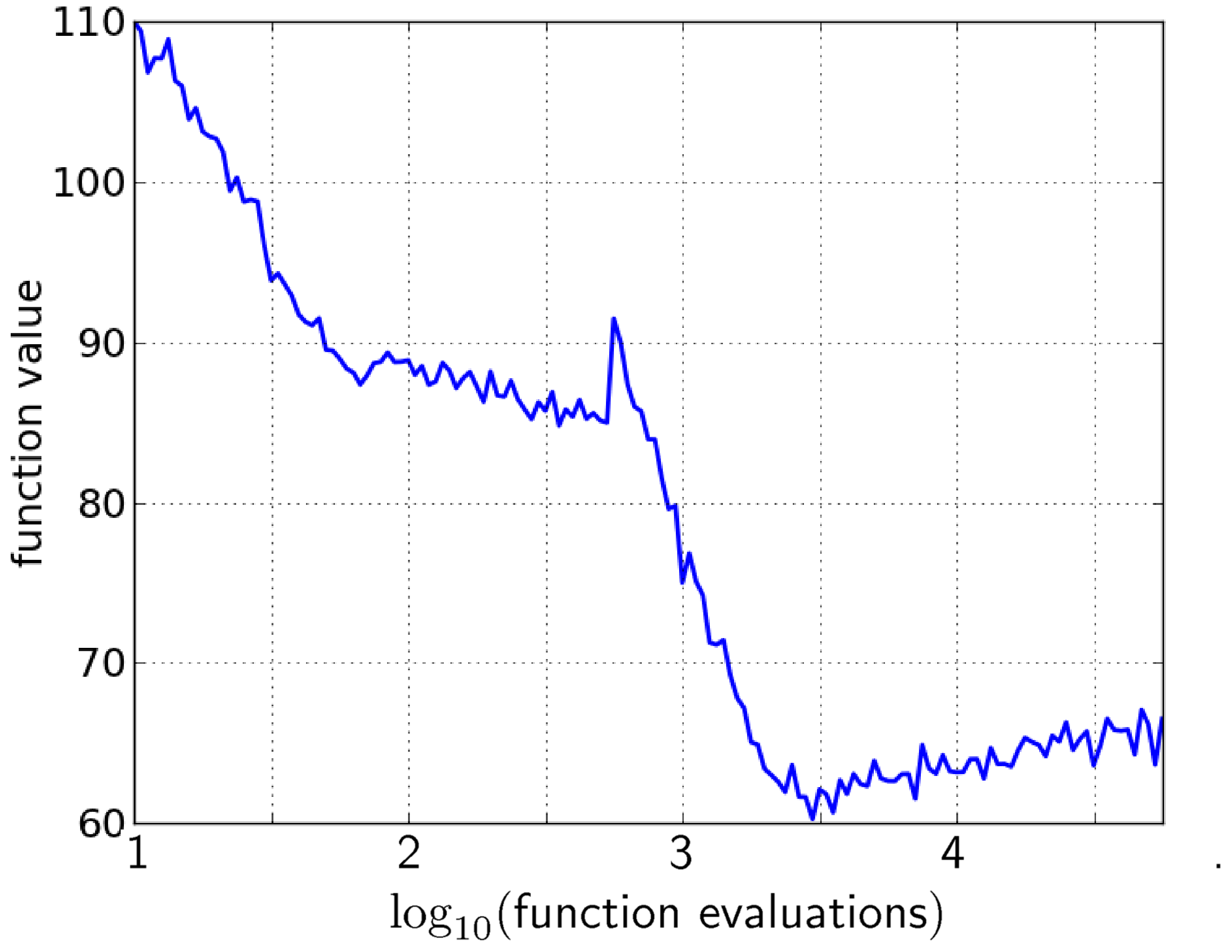


ECDF:

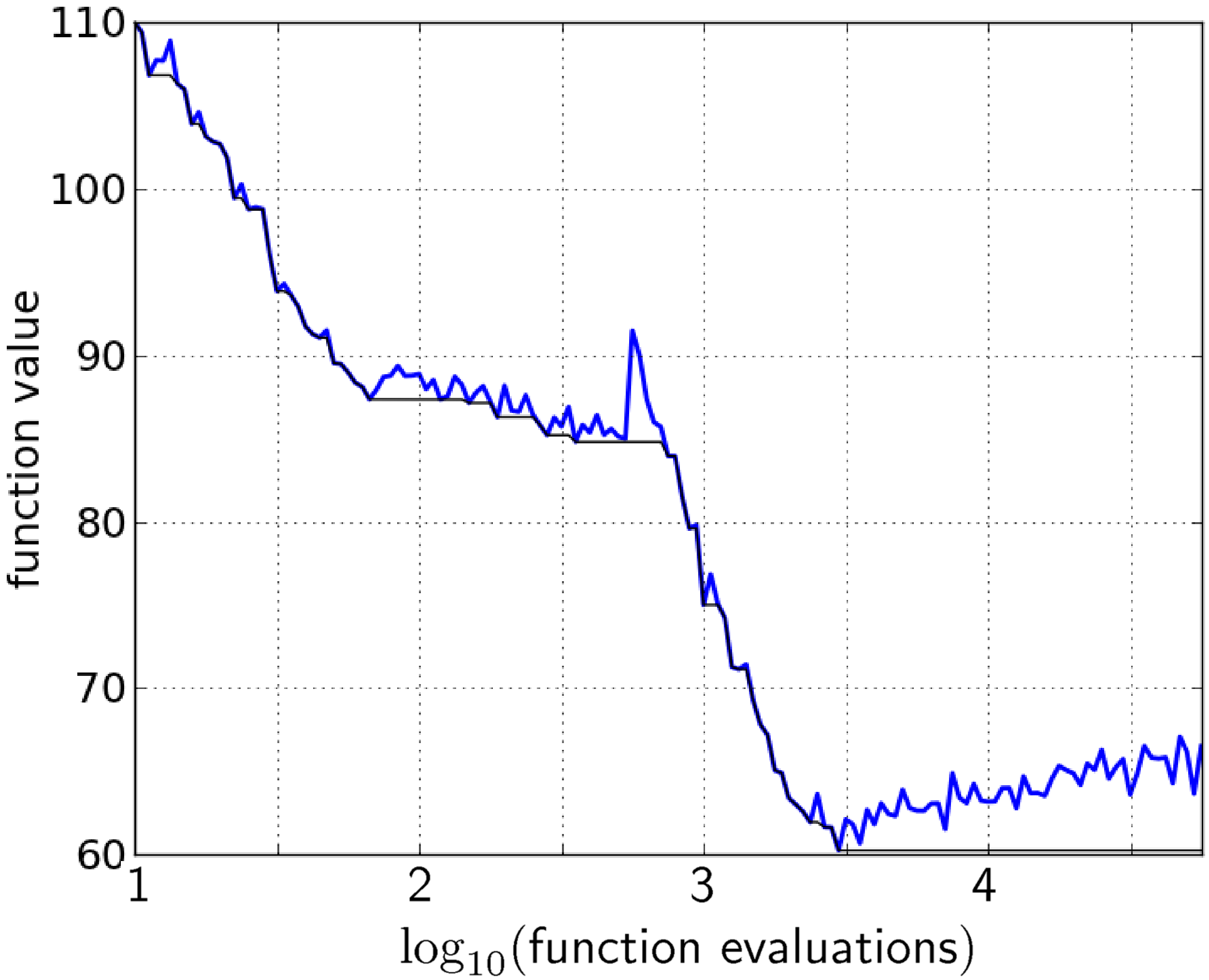
Empirical Cumulative Distribution Function of the Runtime

[aka data profile]

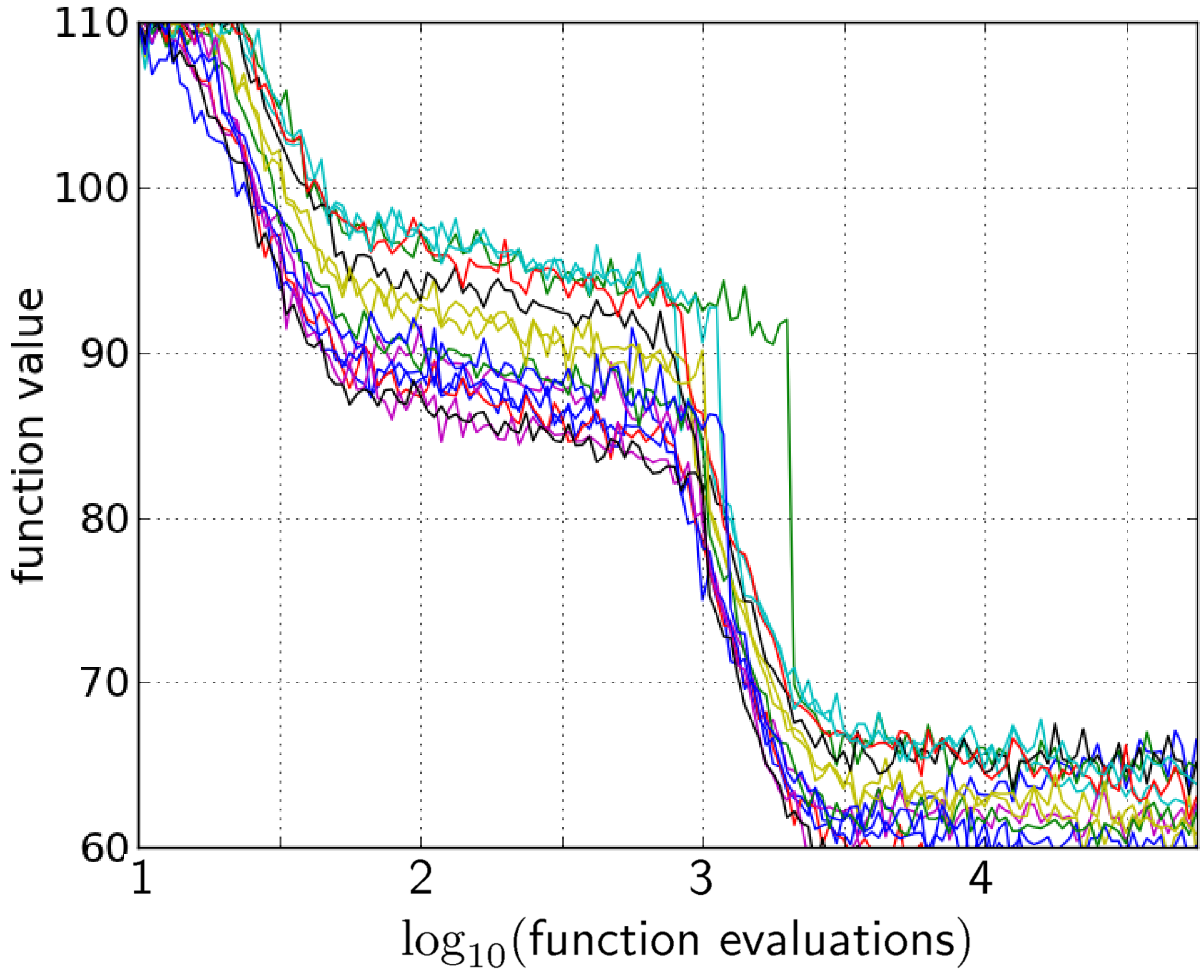
A Convergence Graph



First Hitting Time is Monotonous

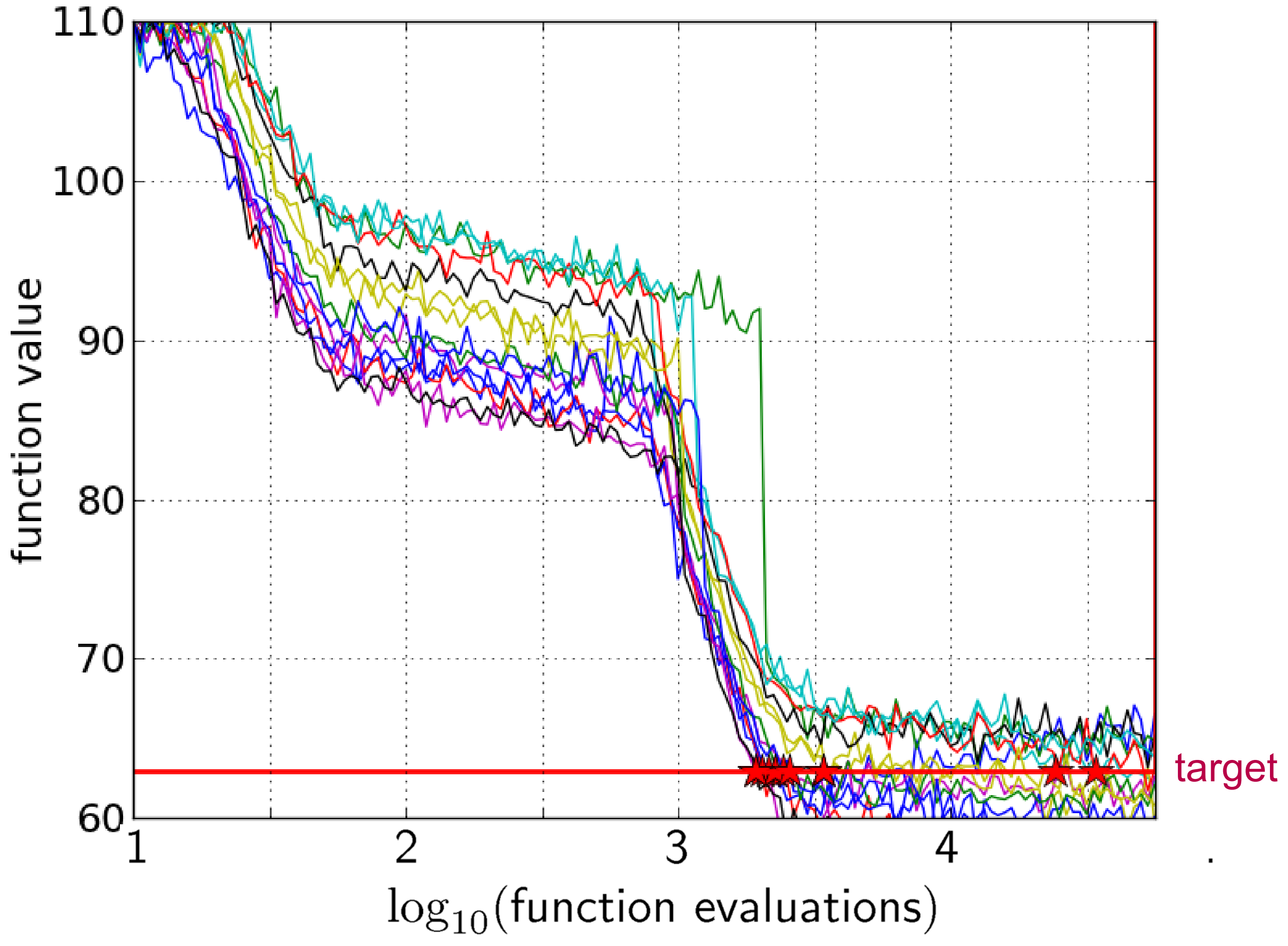


15 Runs

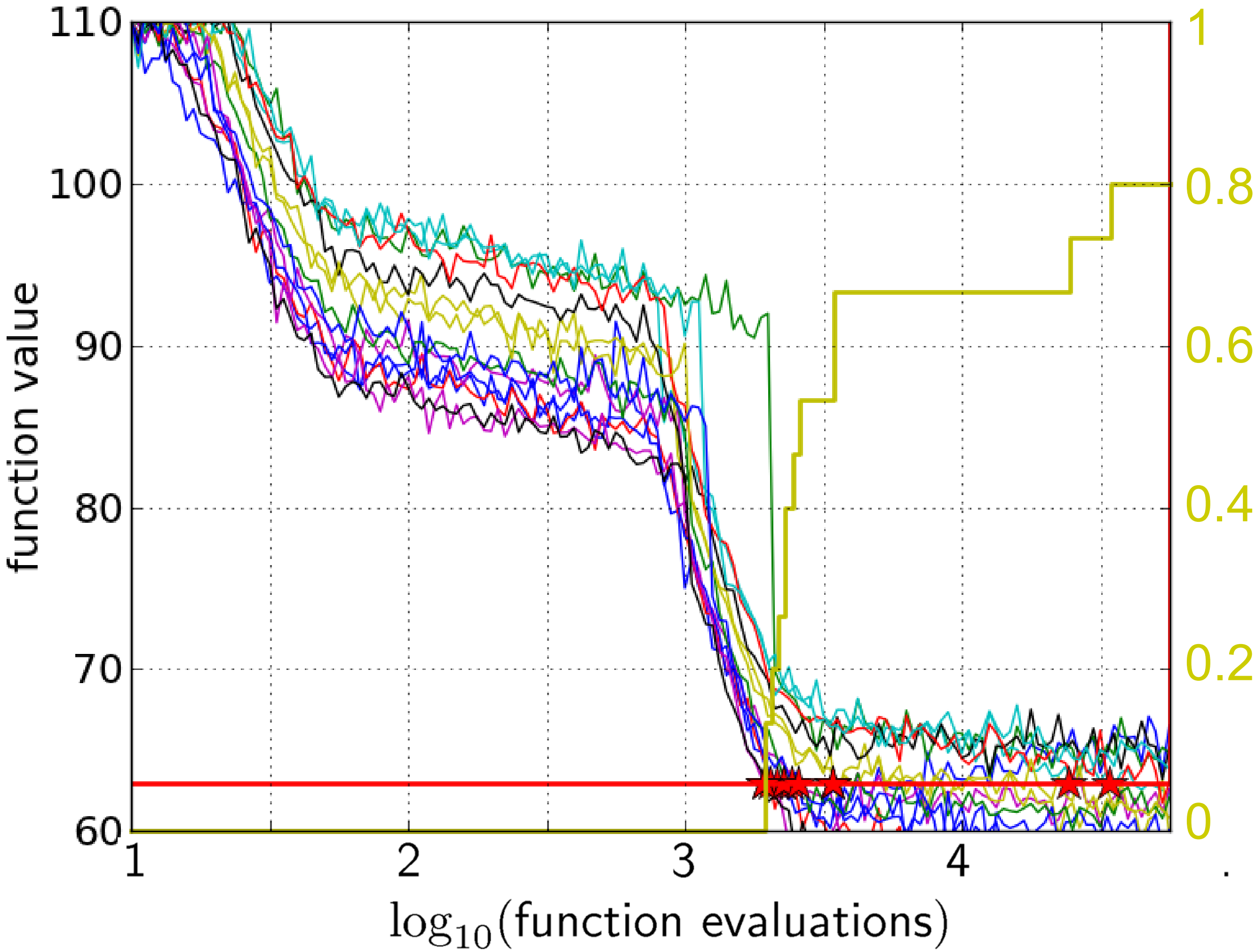


110
100
90
80
70
60
function value
1
2
3
4
$\log_{10}$(function evaluations)

15 Runs ≤ 15 Runtime Data Points

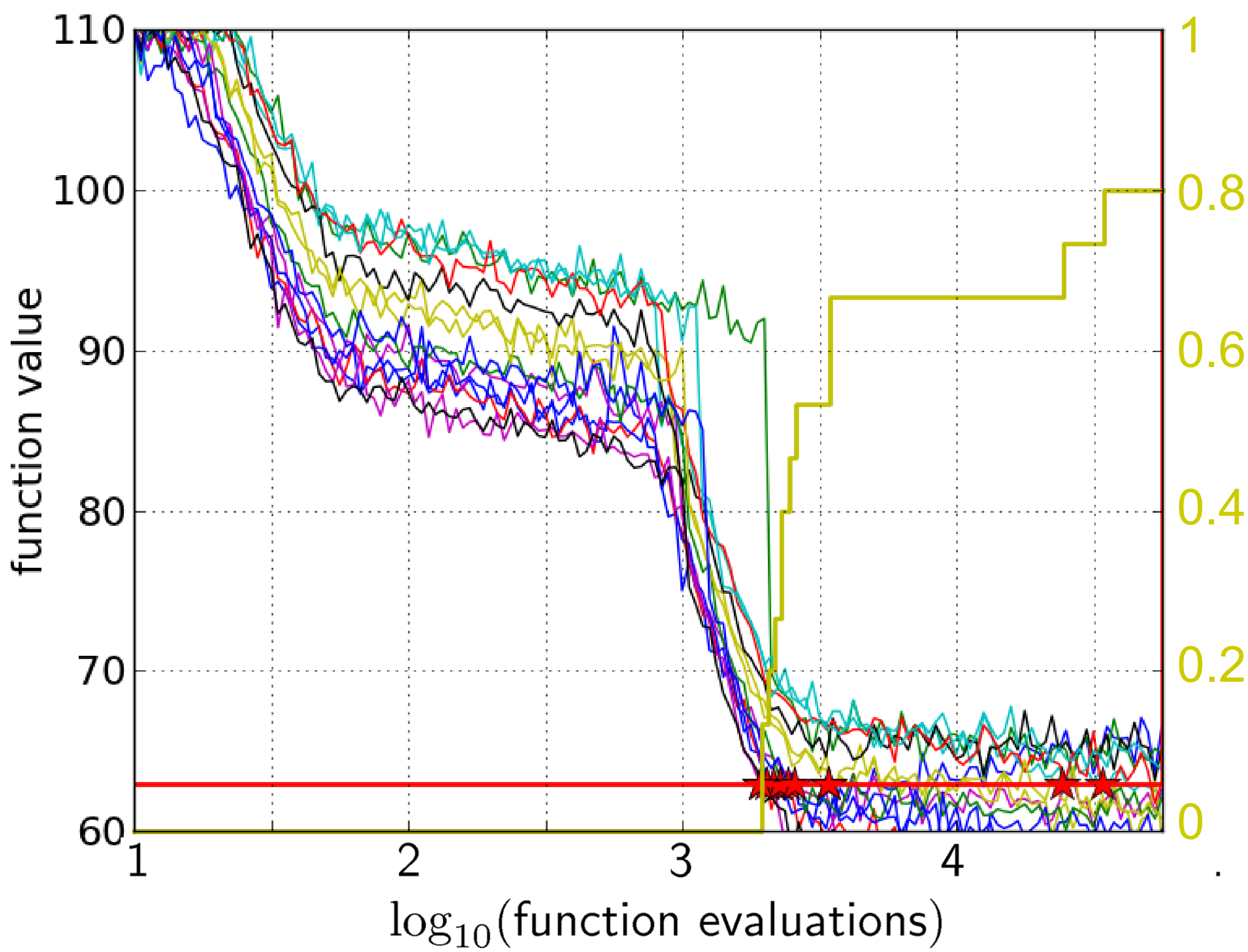


Empirical Cumulative Distribution



- the ECDF of run lengths to reach the target
  - has for each data point a vertical step of constant size
  - displays for each x-value (budget) the count of observations to the left (first hitting times)

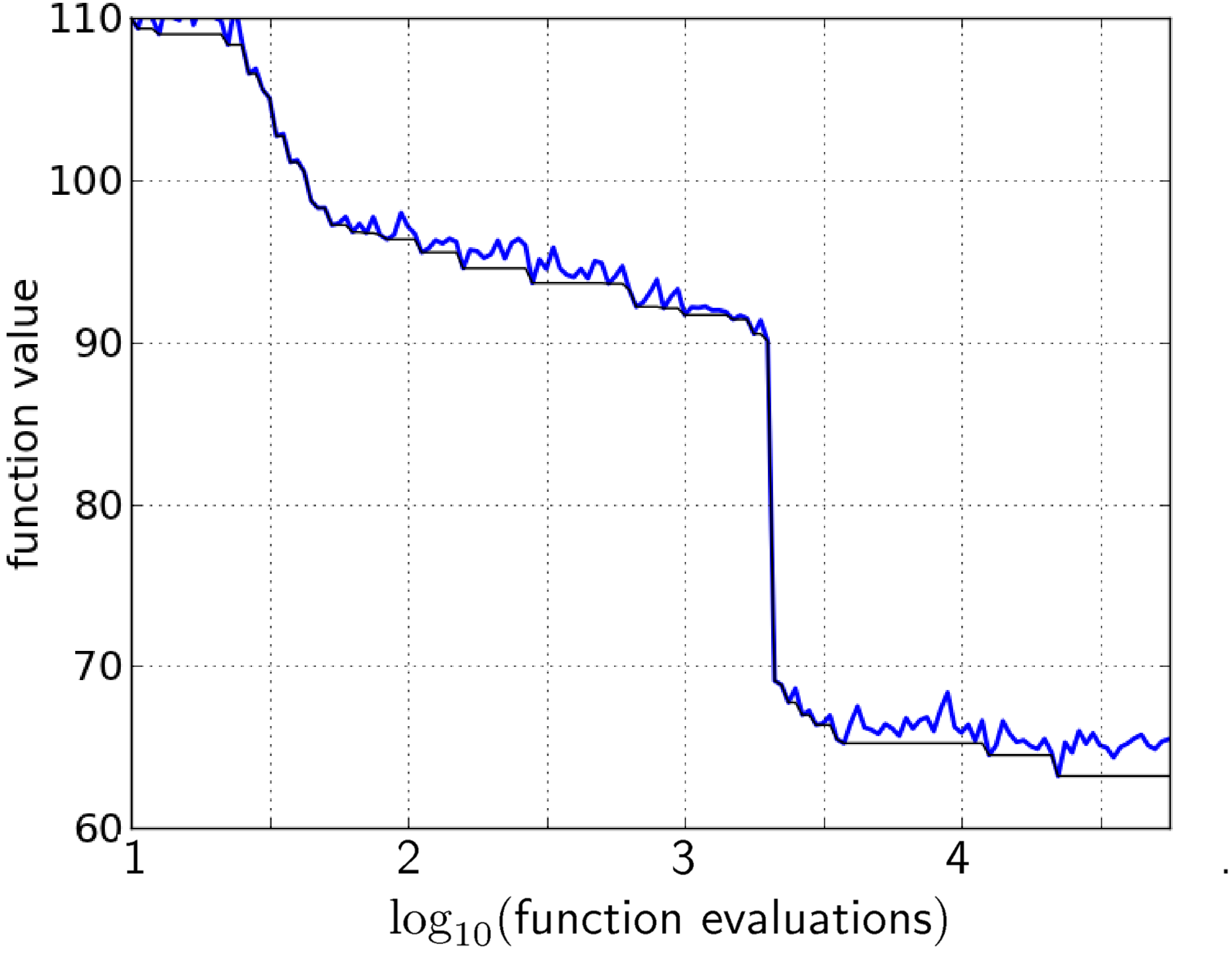
Empirical Cumulative Distribution



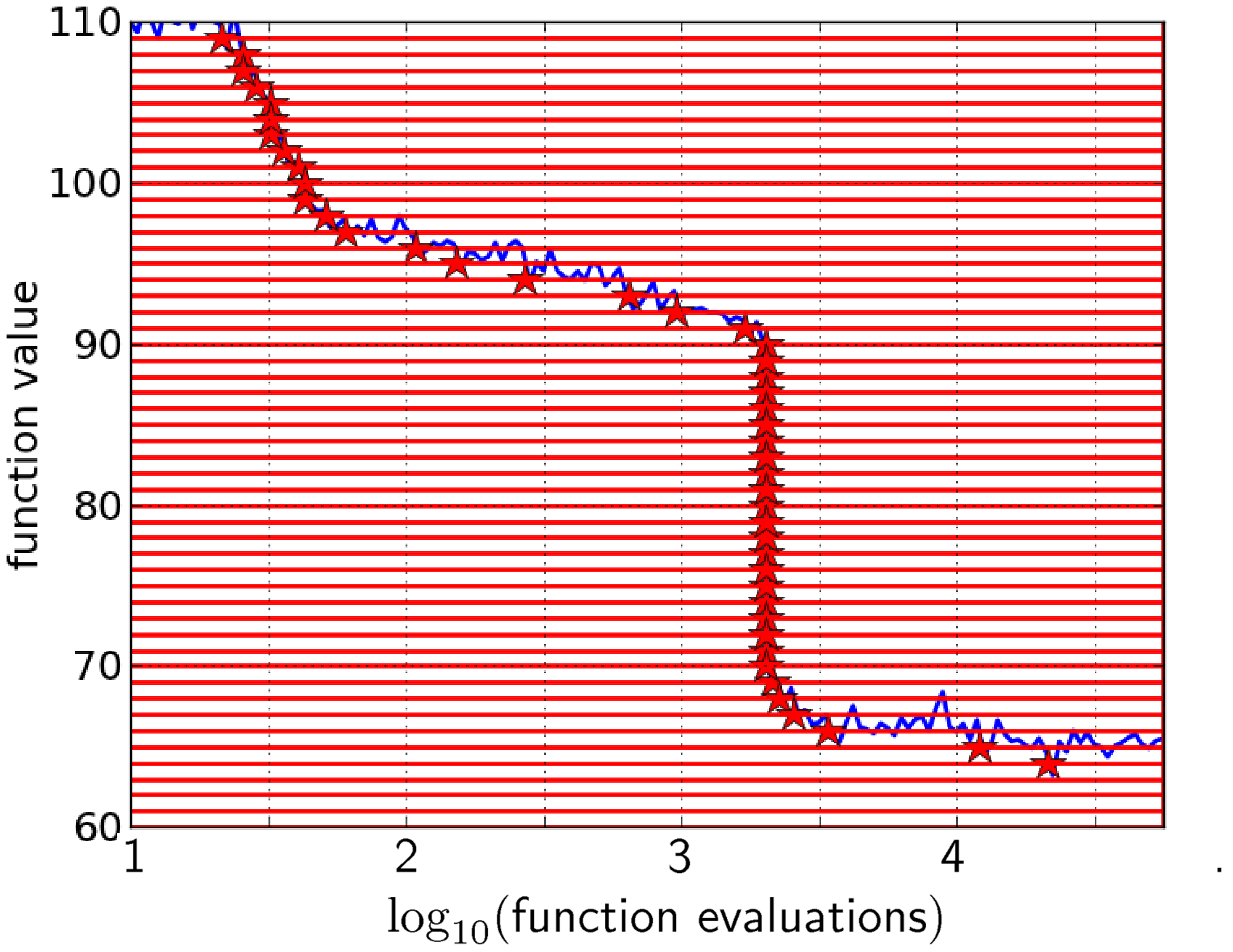
interpretations possible:

- 80% of the runs reached the target
- e.g. 60% of the runs need between 2000 and 4000 evaluations

Reconstructing A Single Run

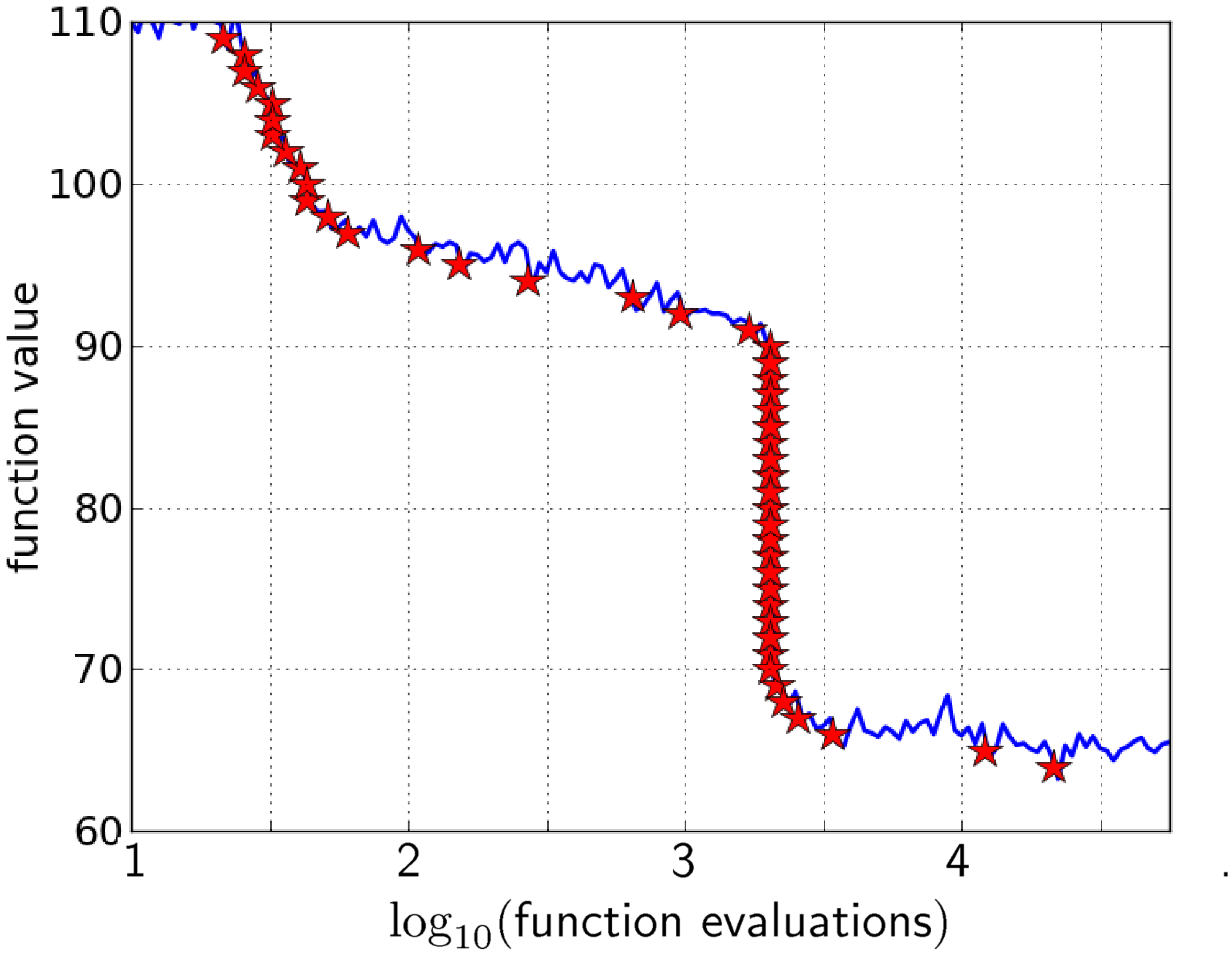


Reconstructing A Single Run

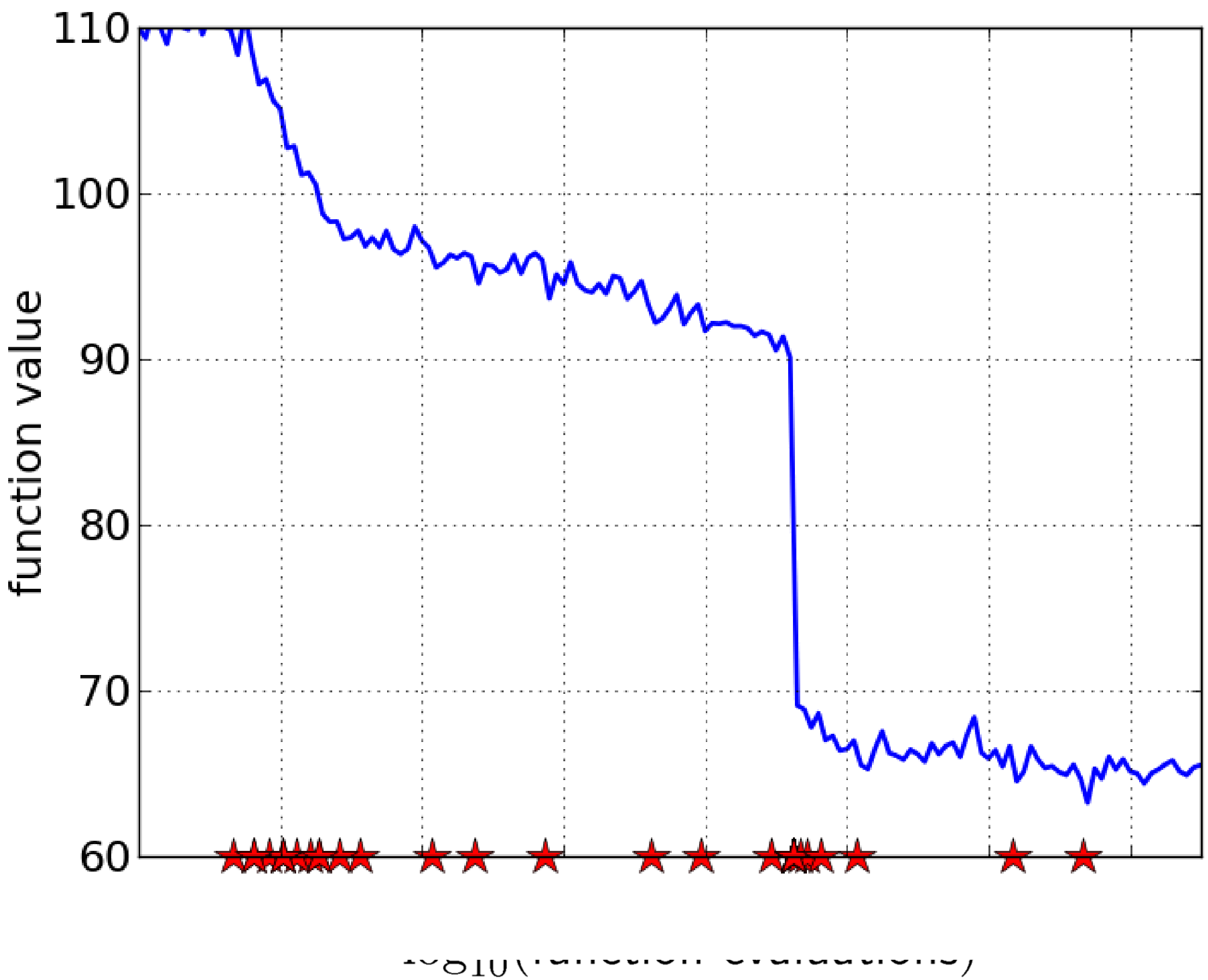


50 equally spaced targets

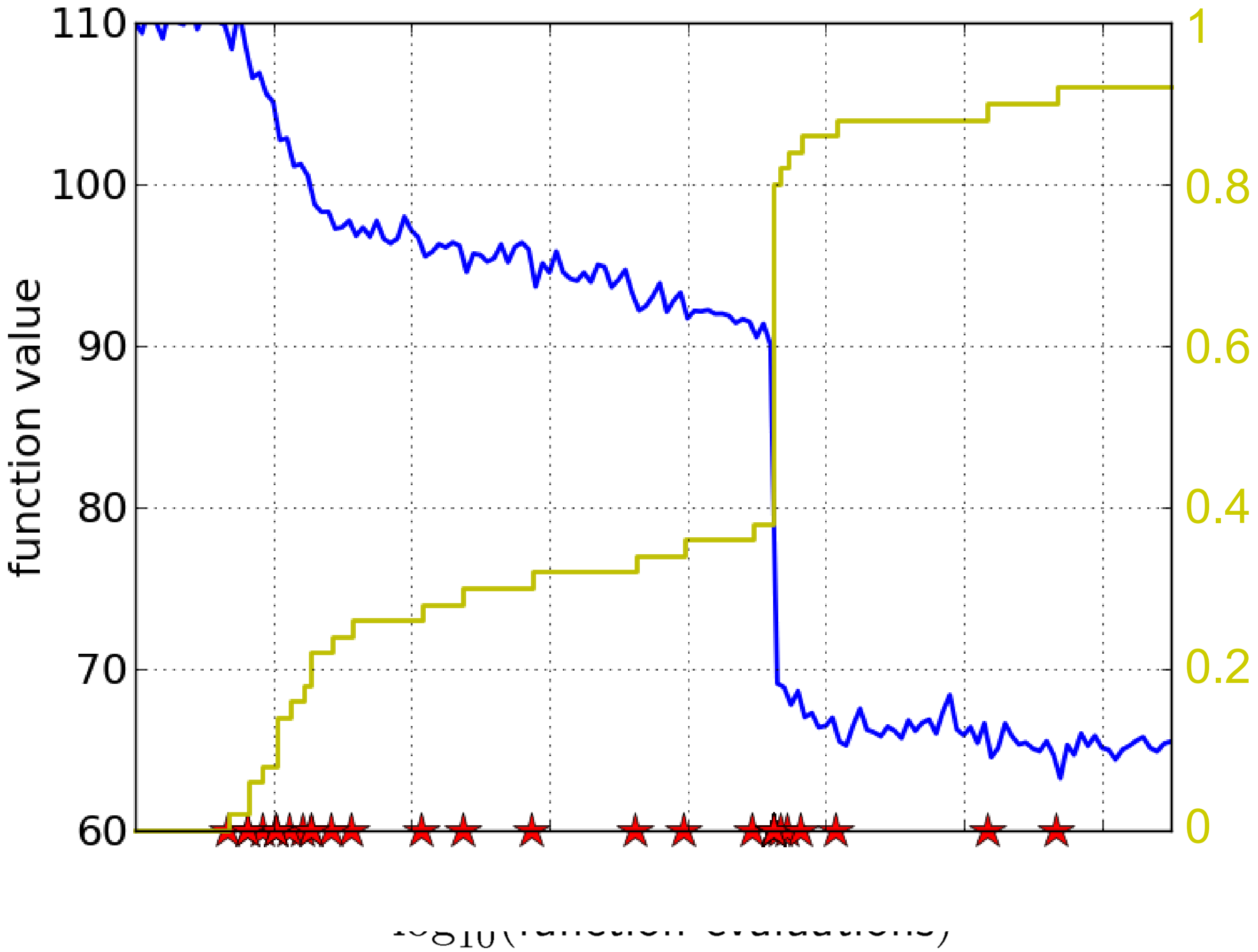
Reconstructing A Single Run



Reconstructing A Single Run

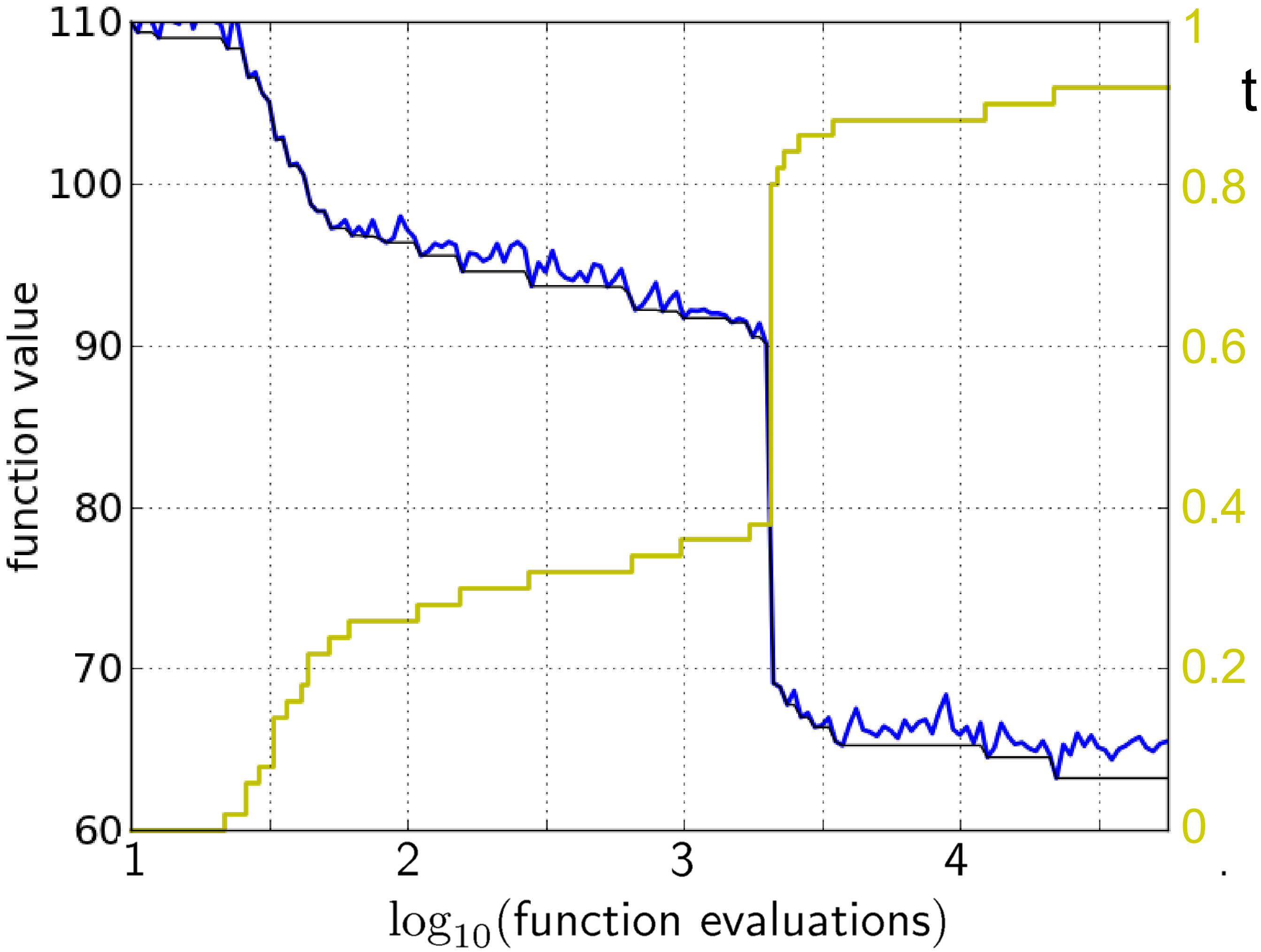


Reconstructing A Single Run



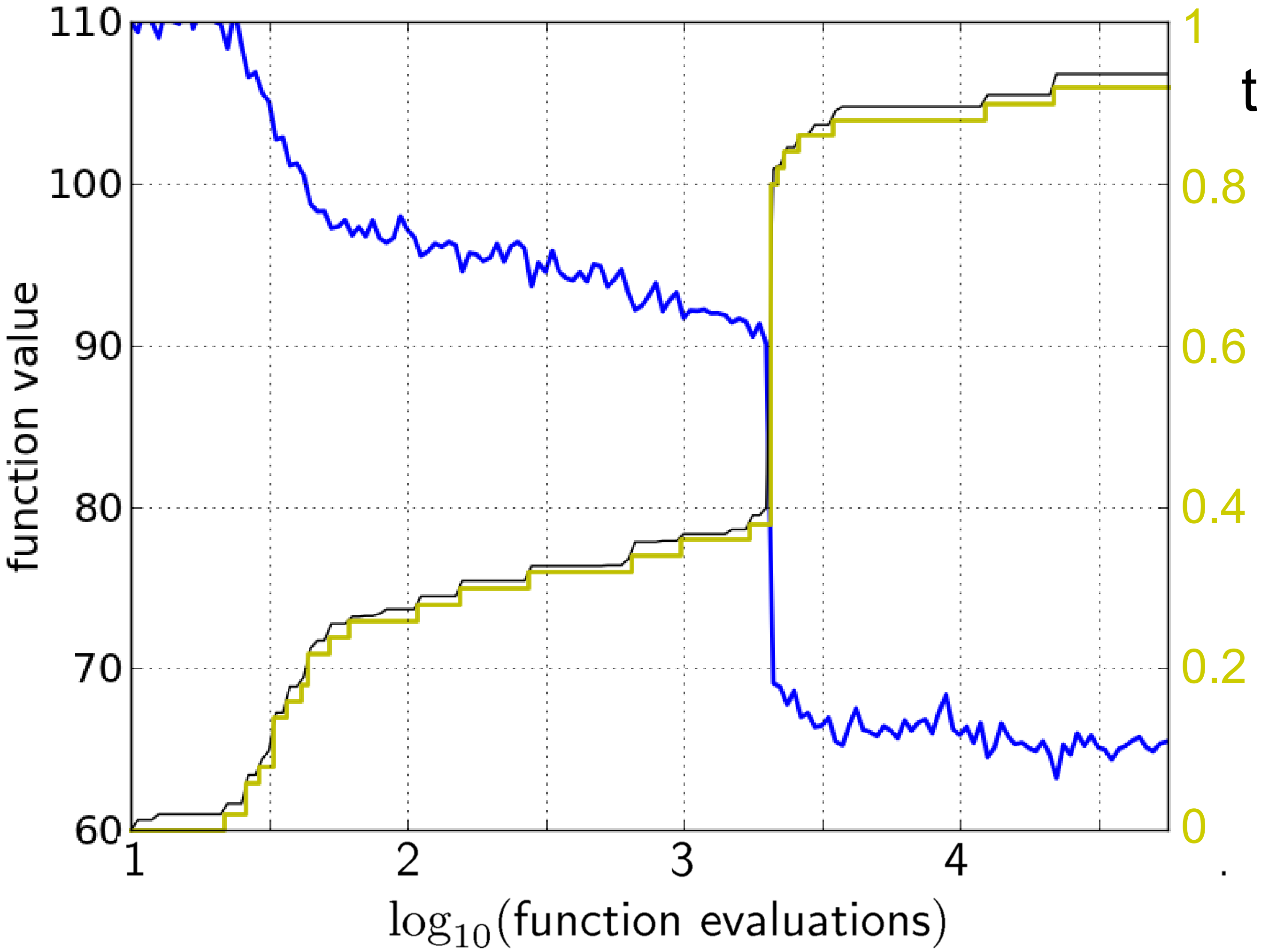
e empirical CDF makes a step for each star, is monotonous and displays for each budget the fraction of targets achieved within the budget

Reconstructing A Single Run



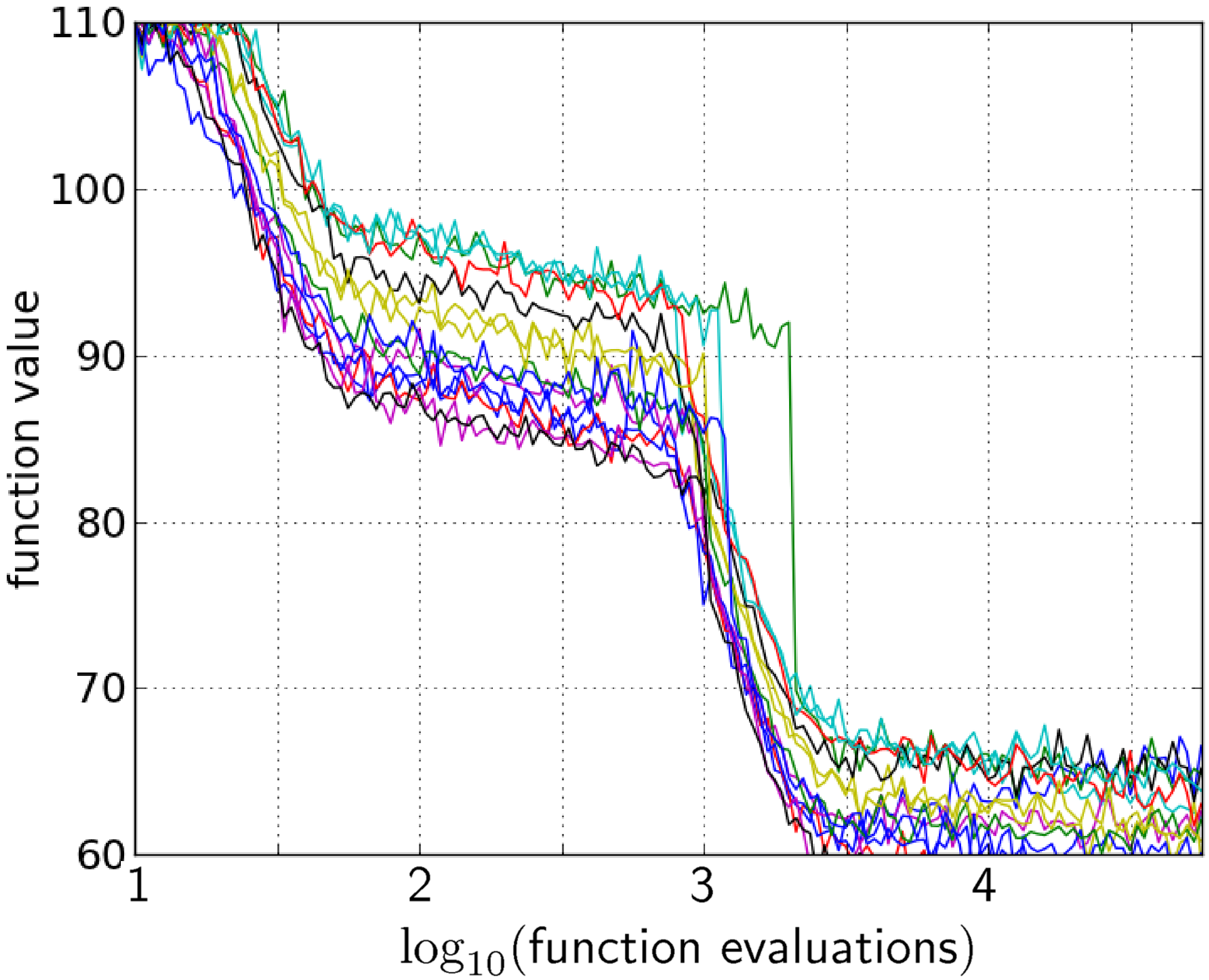
the ECDF recovers the monotonous graph, discretized and flipped

Reconstructing A Single Run



the ECDF recovers the monotonous graph, discretized and flipped

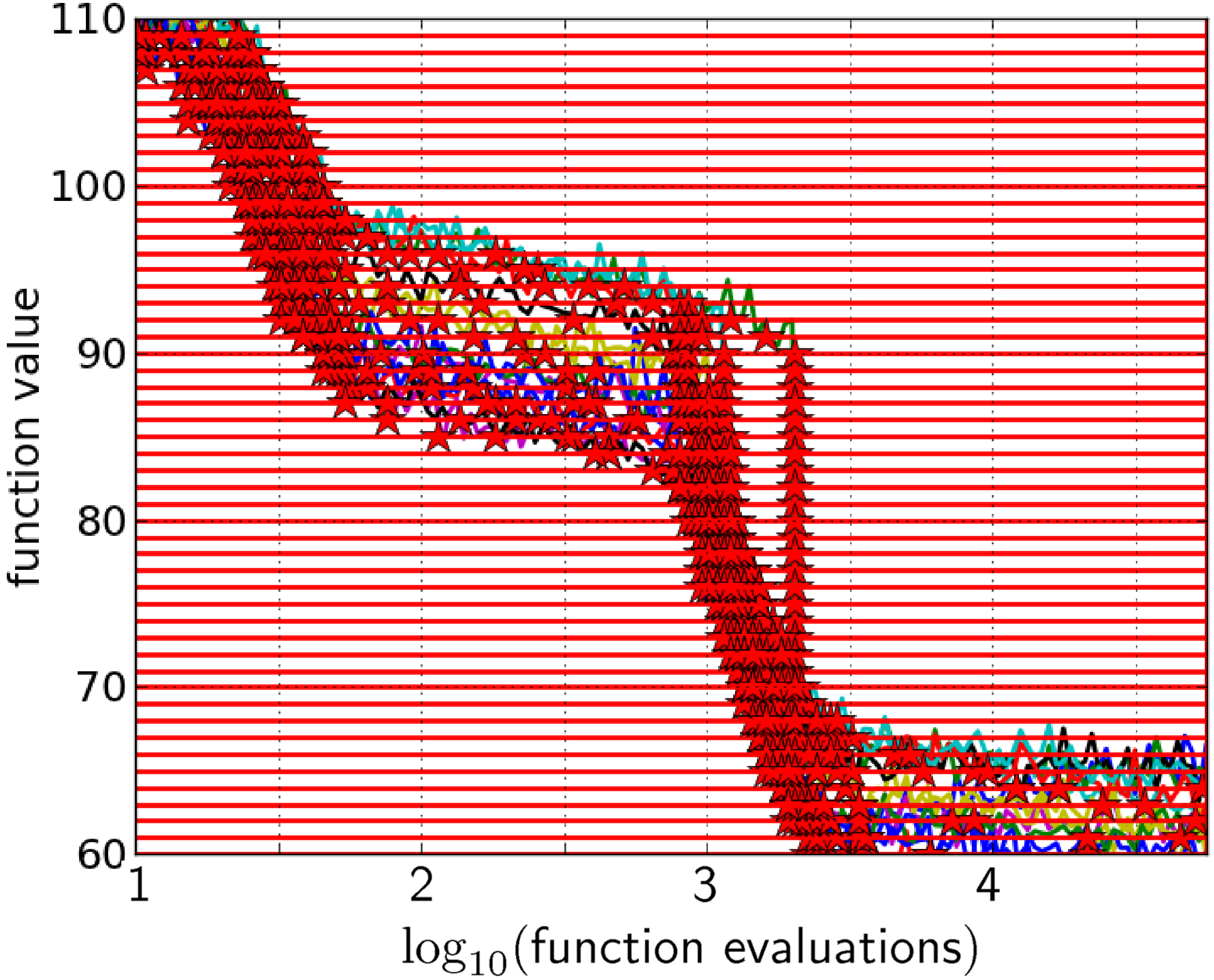
Aggregation



15 runs

.

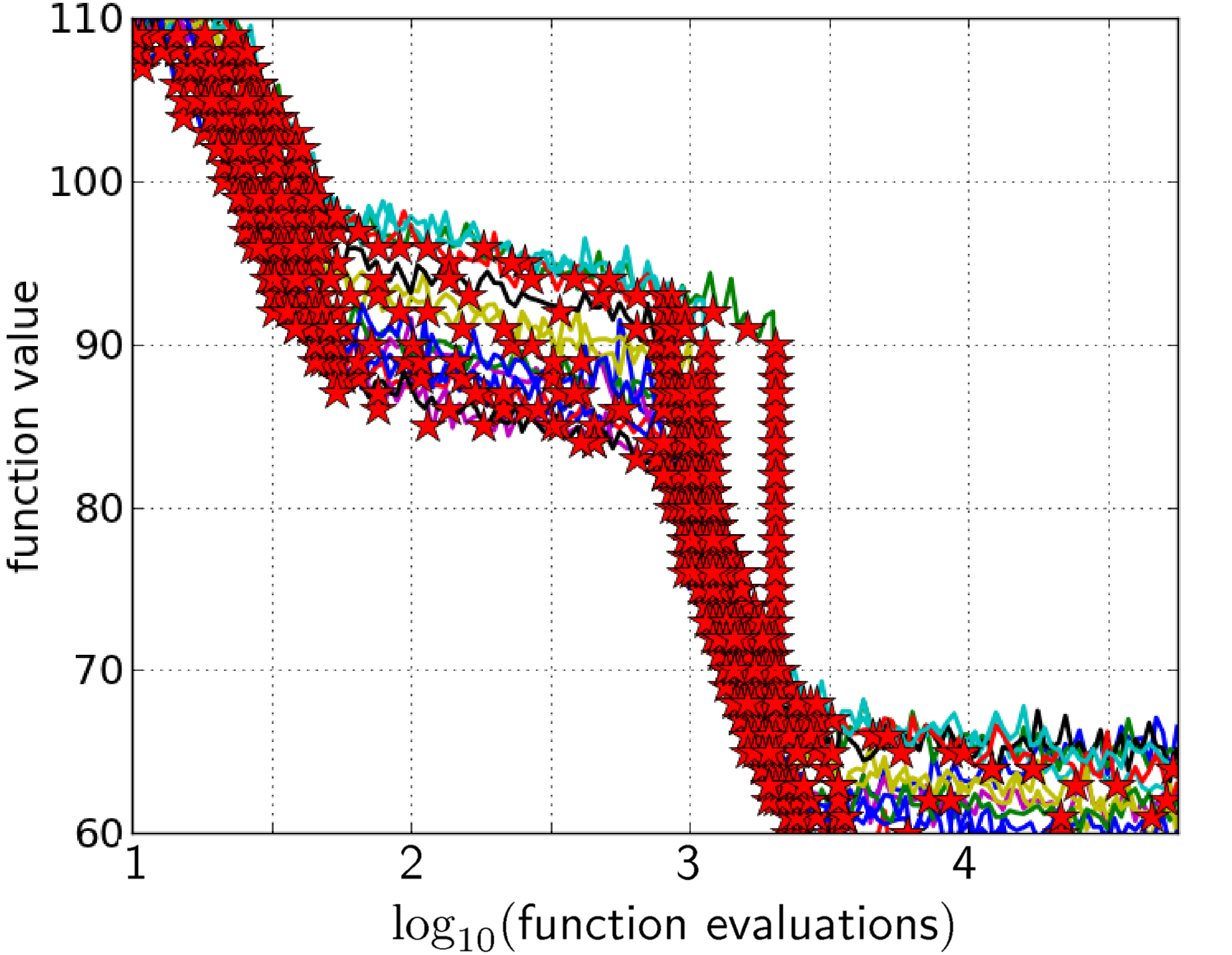
Aggregation



15 runs

50 targets

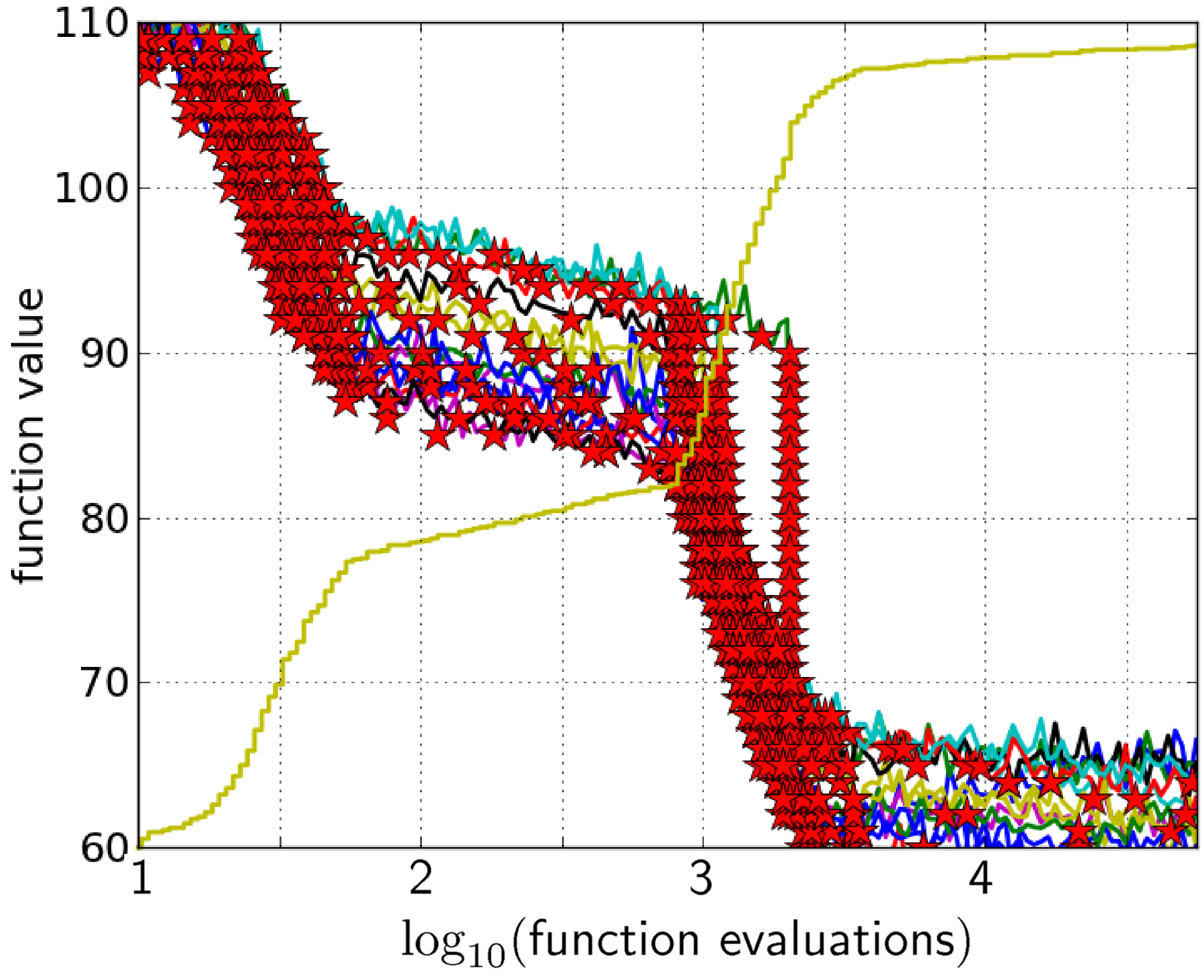
Aggregation



15 runs

50 targets

Aggregation

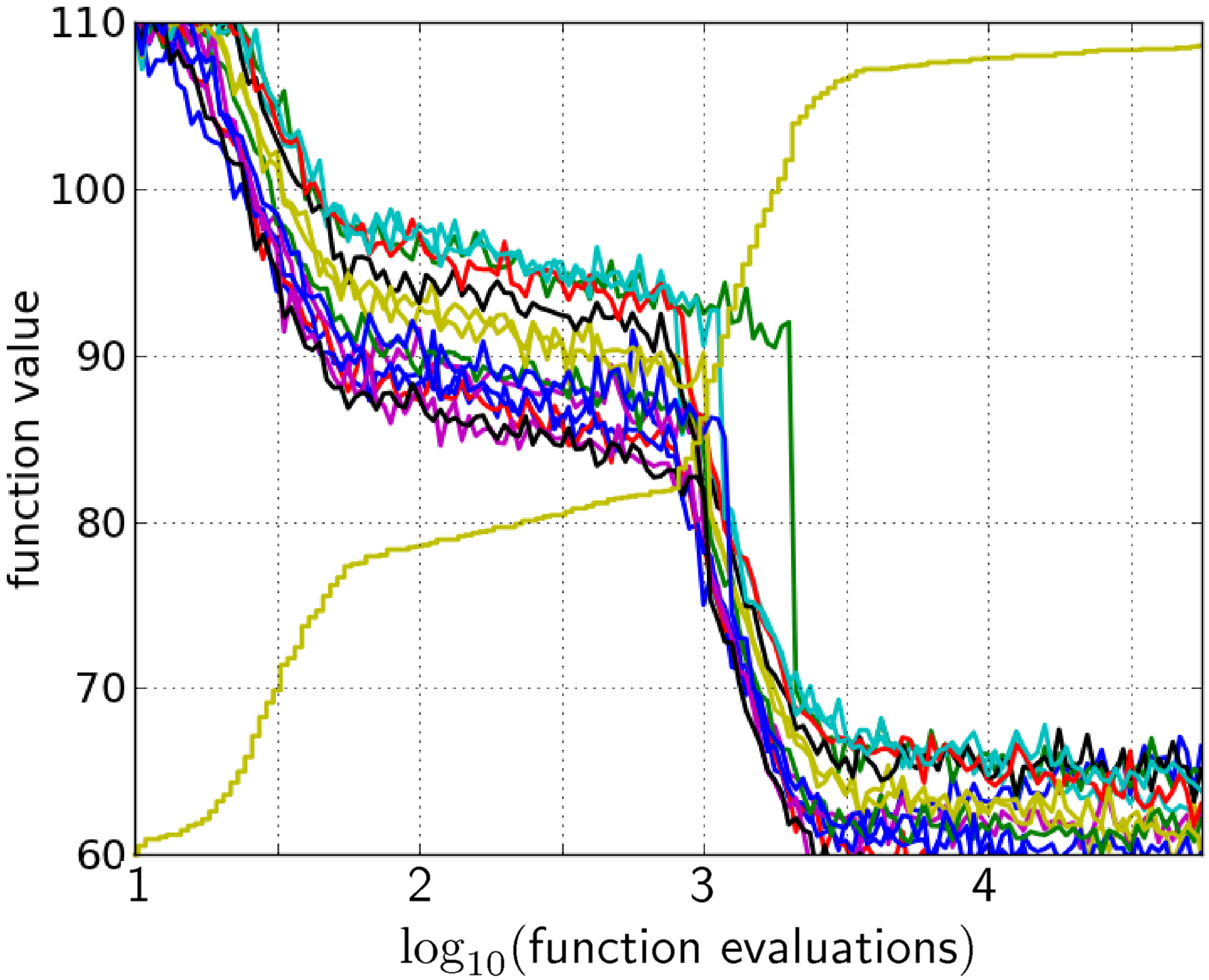


15 runs

50 targets

ECDF with 750 steps

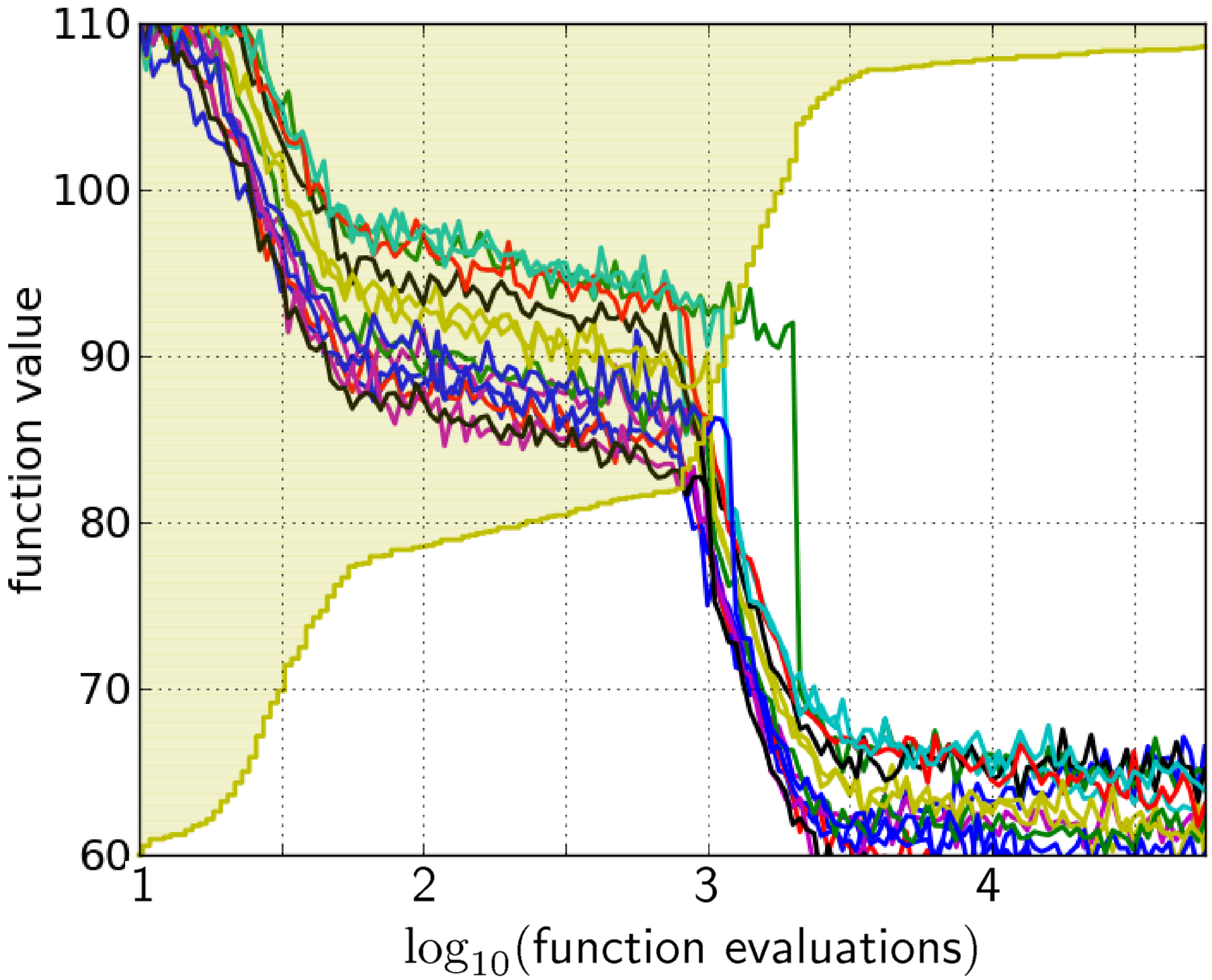
Aggregation



50 targets from 15 runs

...integrated in a single graph

Interpretation



50 targets from 15 runs integrated in a single graph

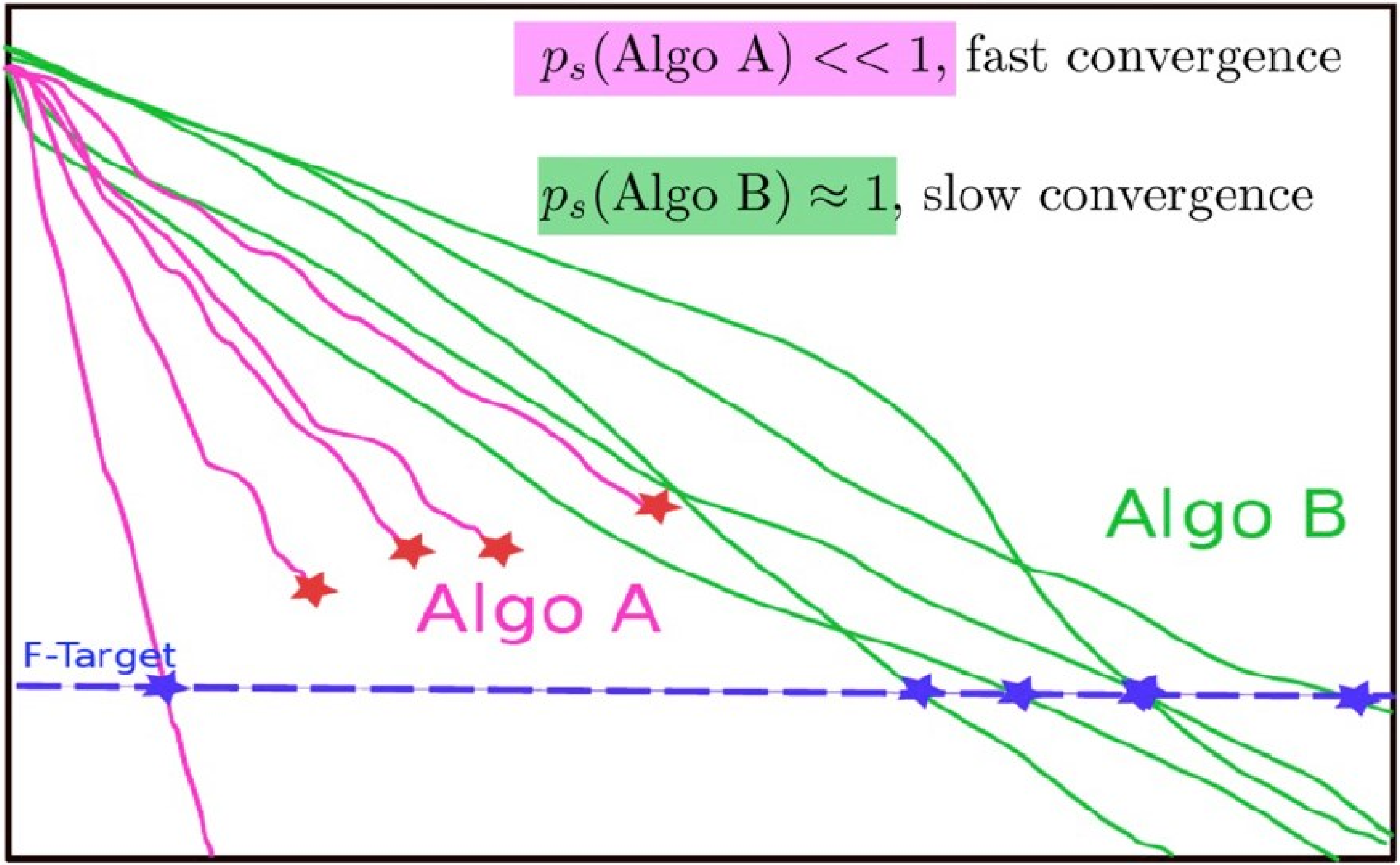
area over the ECDF curve

=

average log runtime

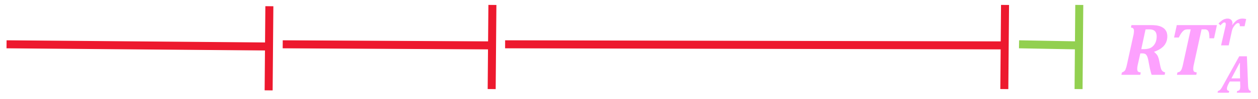
(or geometric avg. runtime) over all targets (difficult and easy) and all runs

Fixed-target: Measuring Runtime



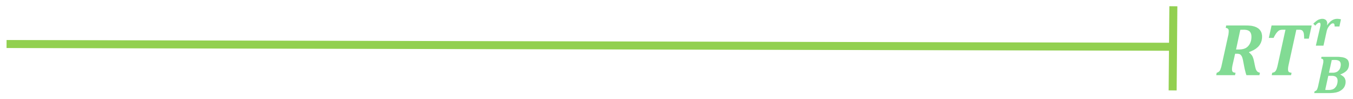
Fixed-target: Measuring Runtime

- Algo Restart A:



p_s(Algo Restart A) = 1

- Algo Restart B:



p_s(Algo Restart B) = 1

Fixed-target: Measuring Runtime

- Expected running time of the restarted algorithm:

$$E[RT^r] = \frac{1-p_s}{p_s} E\left[RT_{unsuccessful}\right] + E[RT_{successful}]$$

- Estimator average running time (aRT):

$$\widehat{p_s} = \frac{\text{#successes}}{\text{#runs}}$$

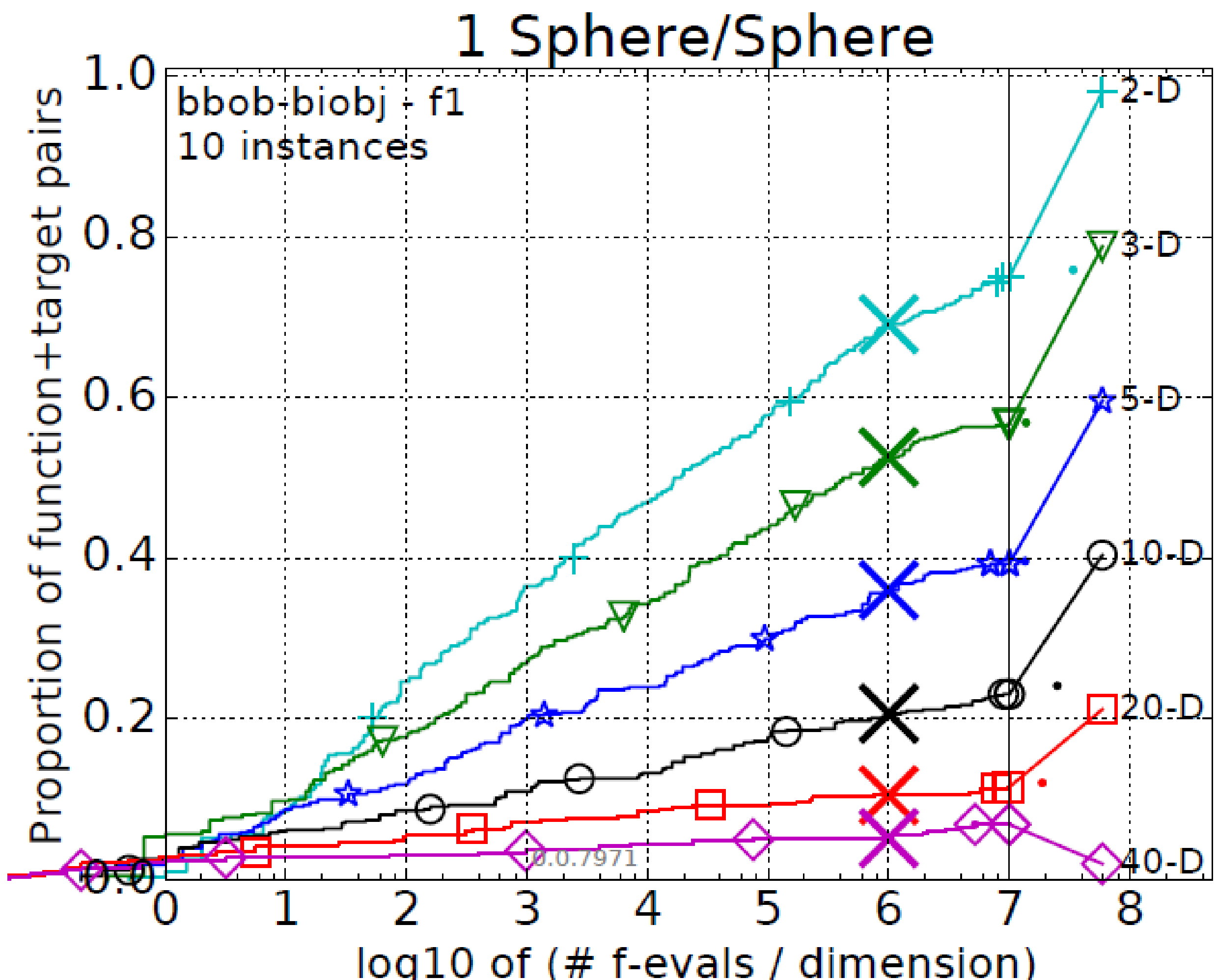
$$\widehat{RT_{unsucc}} = \text{Average evals of unsuccessful runs}$$

$$\widehat{RT_{succ}} = \text{Average evals of successful runs}$$

$$aRT = \frac{\text{total #evals}}{\text{#successes}}$$

ECDFs with Simulated Restarts

What we typically plot are ECDFs of the simulated restarted algorithms:

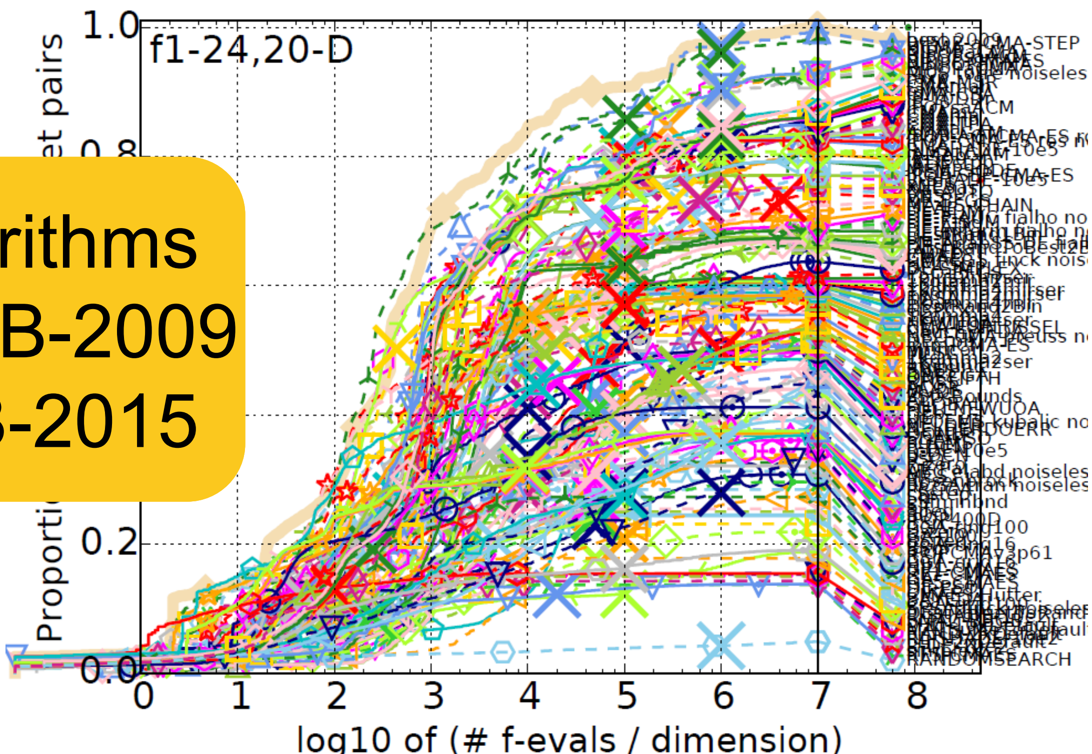


Worth to Note: ECDFs in COCO

In COCO, ECDF graphs

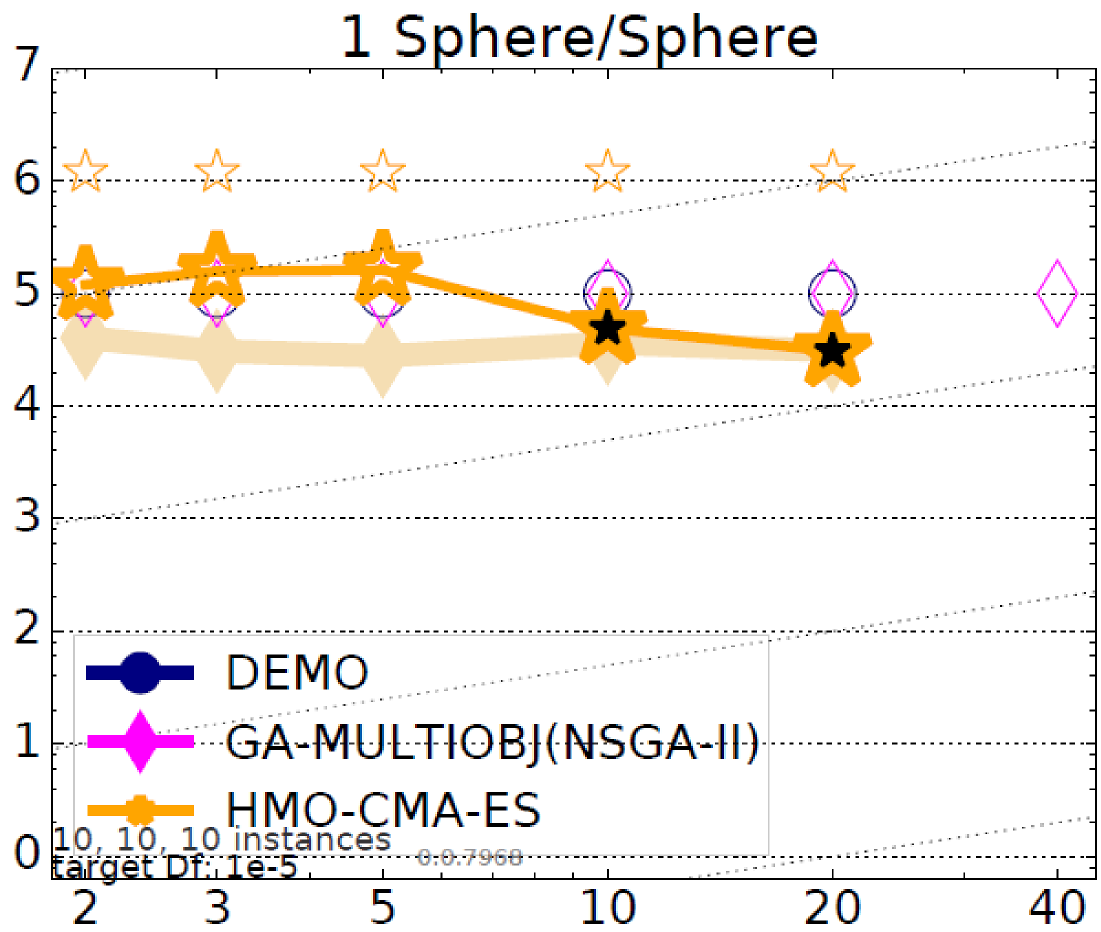
- never aggregate over dimension
  - but often over targets and functions
- can show data of more than 1 algorithm at a time

150 algorithms from BBOB-2009 till BBOB-2015



Another Interesting Plot...

...comparing aRT values over several algorithms

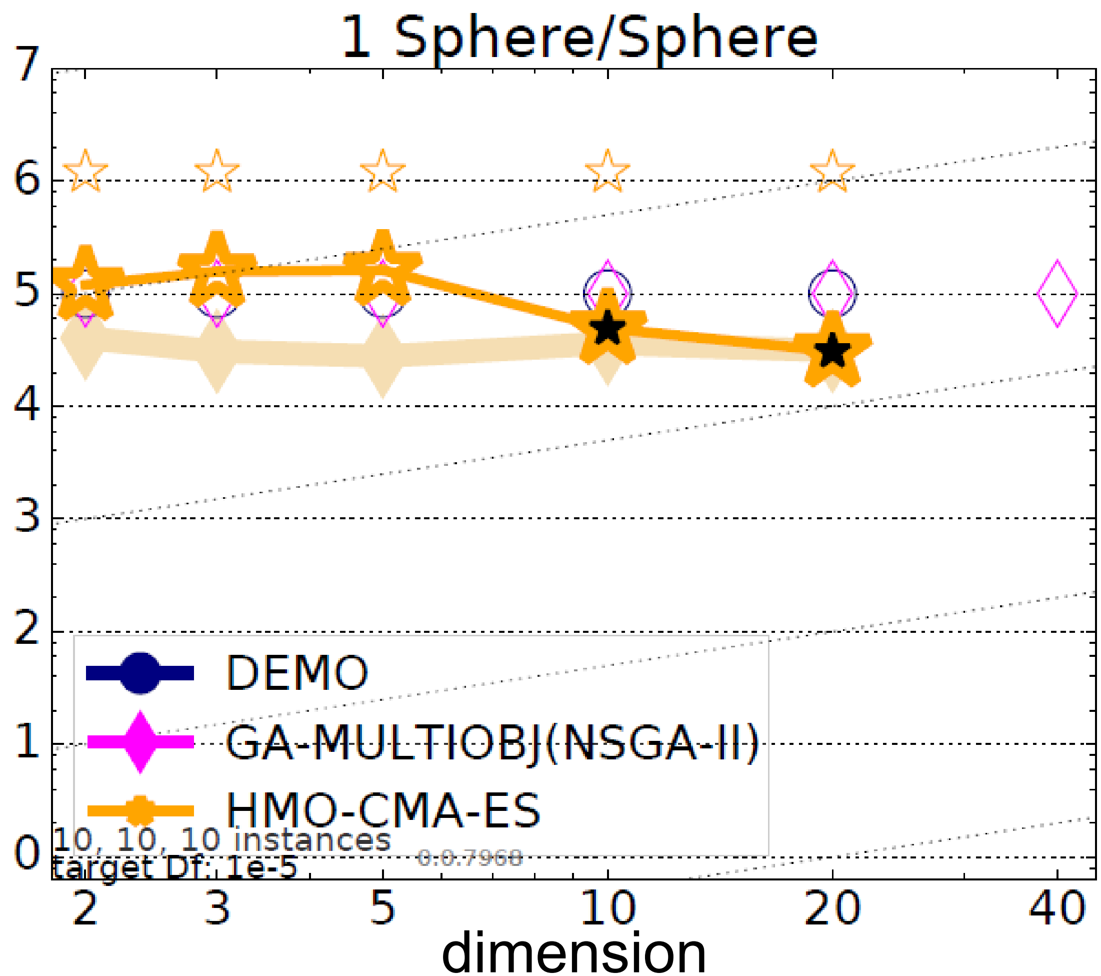


Another Interesting Plot...

...comparing aRT values over several algorithms

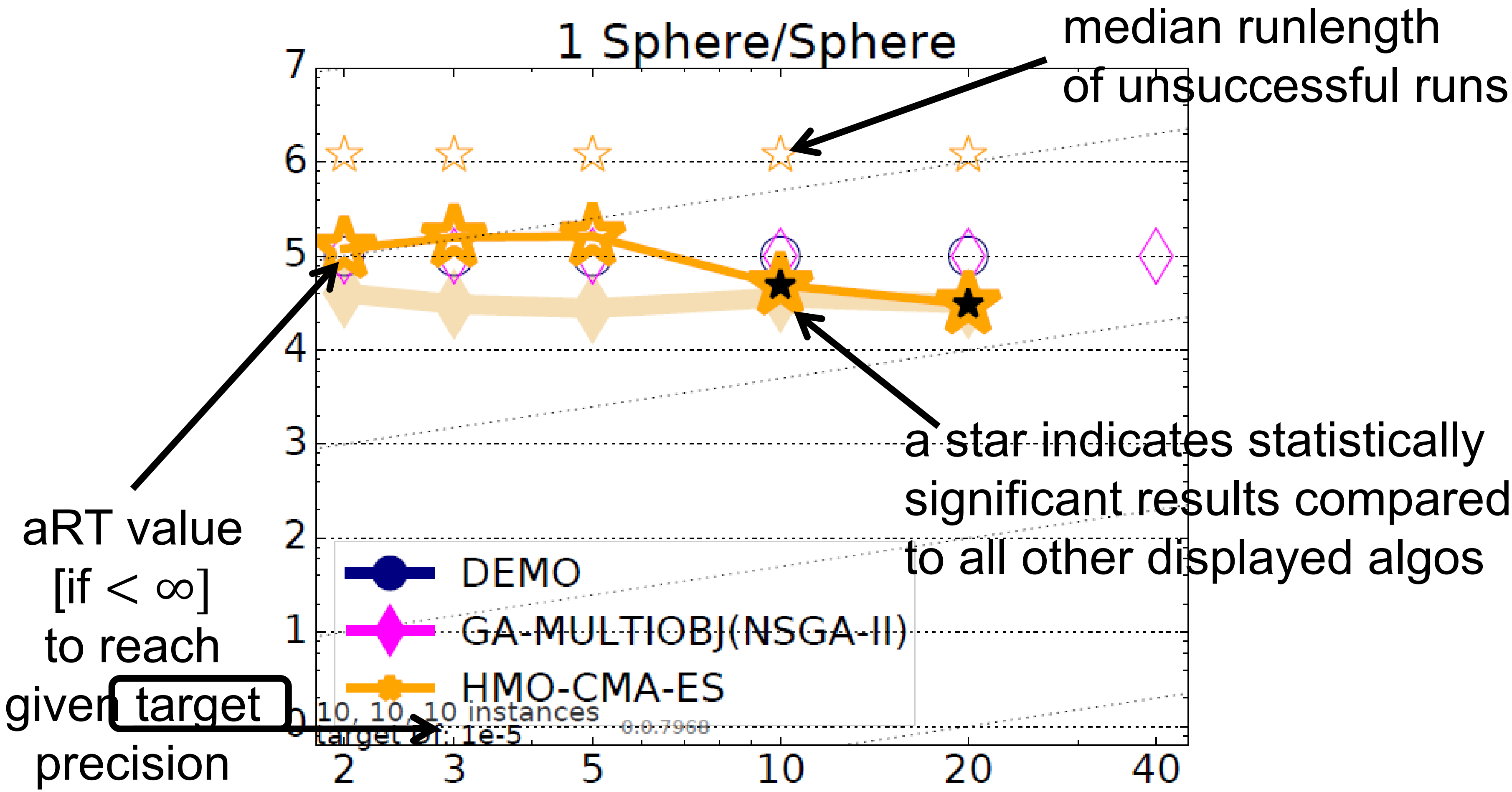
y axis shows runtime in log-scale:

5 "=" 1e5*DIM



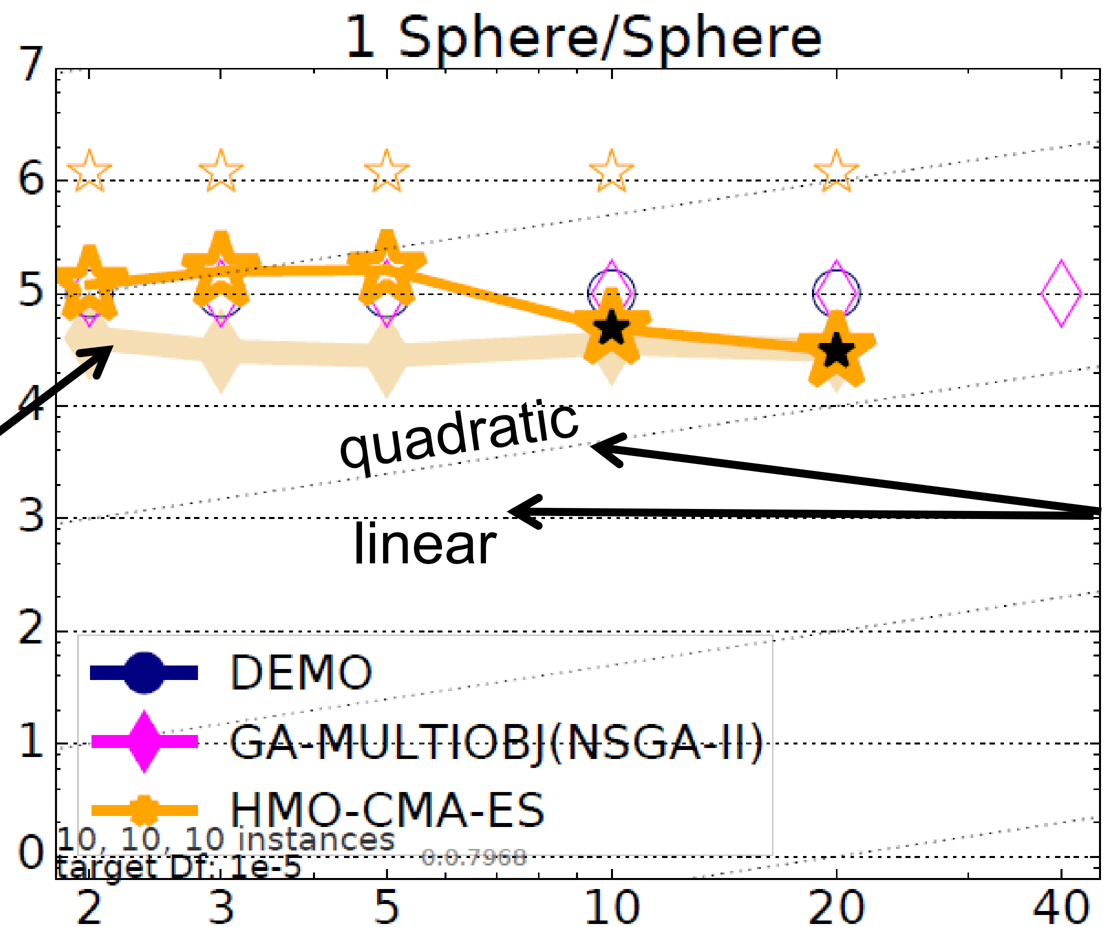
Another Interesting Plot...

...comparing aRT values over several algorithms



Another Interesting Plot...

...comparing aRT values over several algorithms



artificial best algorithm from BBOB-2016

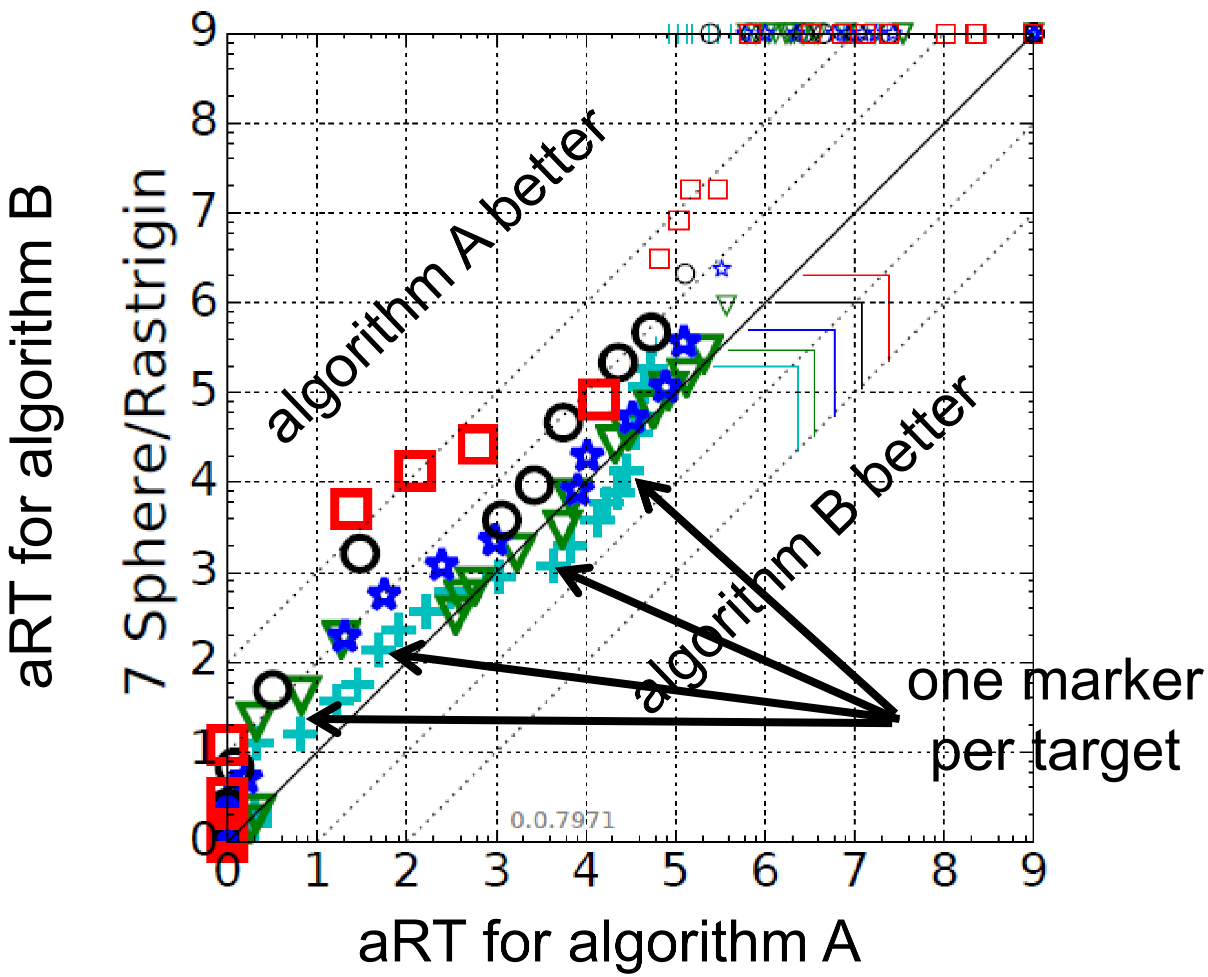
scaling with dimension

Interesting for 2 Algorithms...

...are scatter plots

dimensions:

2:+, 3:▽, 5:⋆, 10:○, 20:□, 40:◇.



There are more Plots...

...but they are probably less interesting for us here

The single-objective BBOB functions

bbob Testbed

- 24 functions in 5 groups:

1 Separable Functions	
f1	Sphere Function
f2	Ellipsoidal Function
f3	Rastrigin Function
f4	Büche-Rastrigin Function
f5	Linear Slope
2 Functions with low or moderate conditioning	
f6	Attractive Sector Function
f7	Step Ellipsoidal Function
f8	Rosenbrock Function, original
f9	Rosenbrock Function, rotated
3 Functions with high conditioning and unimodal	
f10	Ellipsoidal Function
f11	Discus Function
f12	Bent Cigar Function
f13	Sharp Ridge Function
f14	Different Powers Function

4 Multi-modal functions with adequate global structure	
f15	Rastrigin Function
f16	Weierstrass Function
f17	Schaffers F7 Function
f18	Schaffers F7 Functions, moderately ill-conditioned
f19	Composite Griewank-Rosenbrock Function F8F2
5 Multi-modal functions with weak global structure	
f20	Schwefel Function
f21	Gallagher's Gaussian 101-me Peaks Function
f22	Gallagher's Gaussian 21-hi Peaks Function
f23	Katsuura Function
f24	Lunacek bi-Rastrigin Function

- 6 dimensions: 2, 3, 5, 10, 20, (40 optional)

Notion of Instances

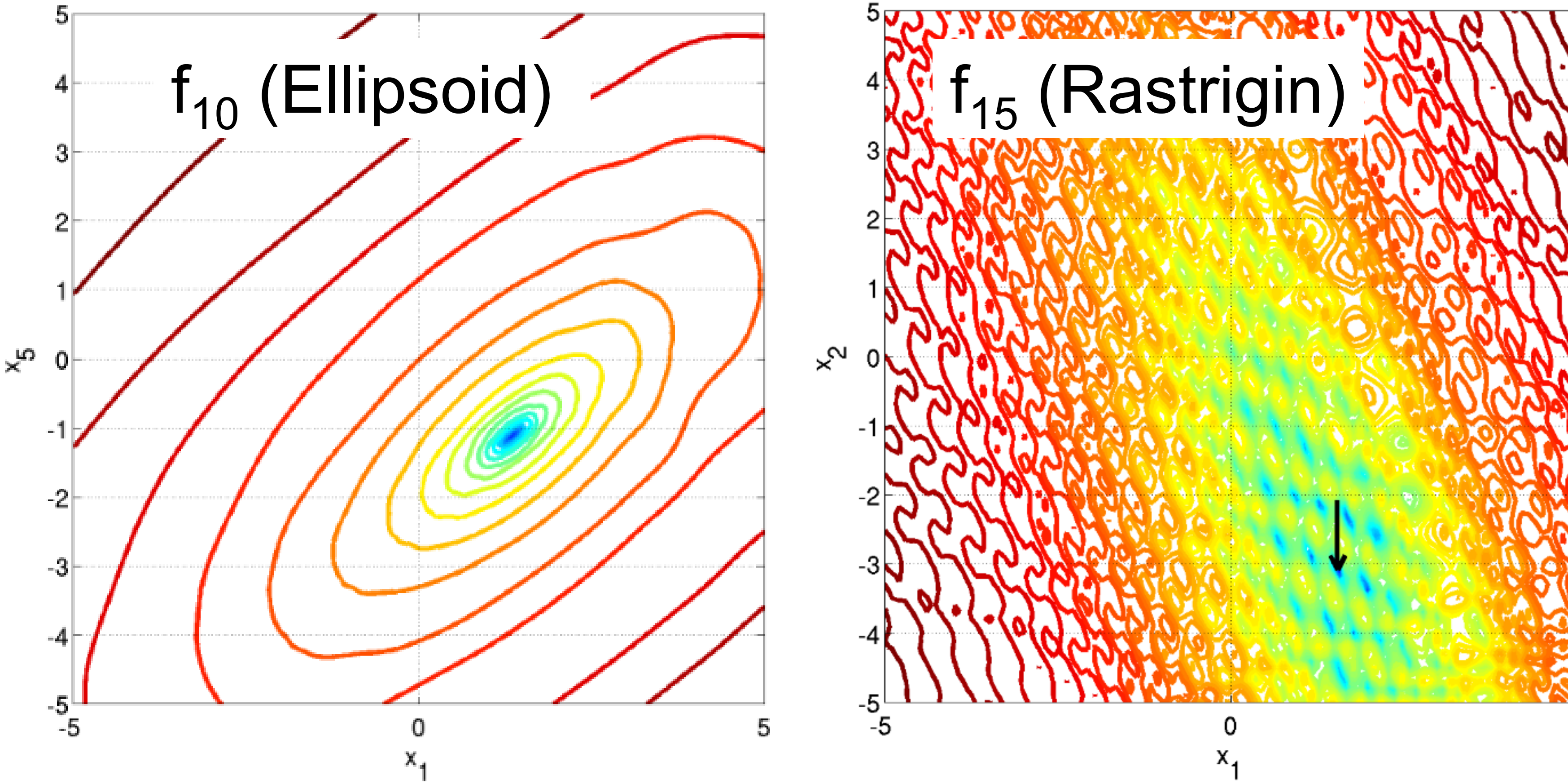
- All COCO problems come in form of instances
  - e.g. as translated/rotated versions of the same function
- Prescribed instances typically change from year to year
  - avoid overfitting
  - 5 instances are always kept the same

Plus:

  - the bbob functions are locally perturbed by non-linear transformations

Notion of Instances

- All COCO problems come in form of instances

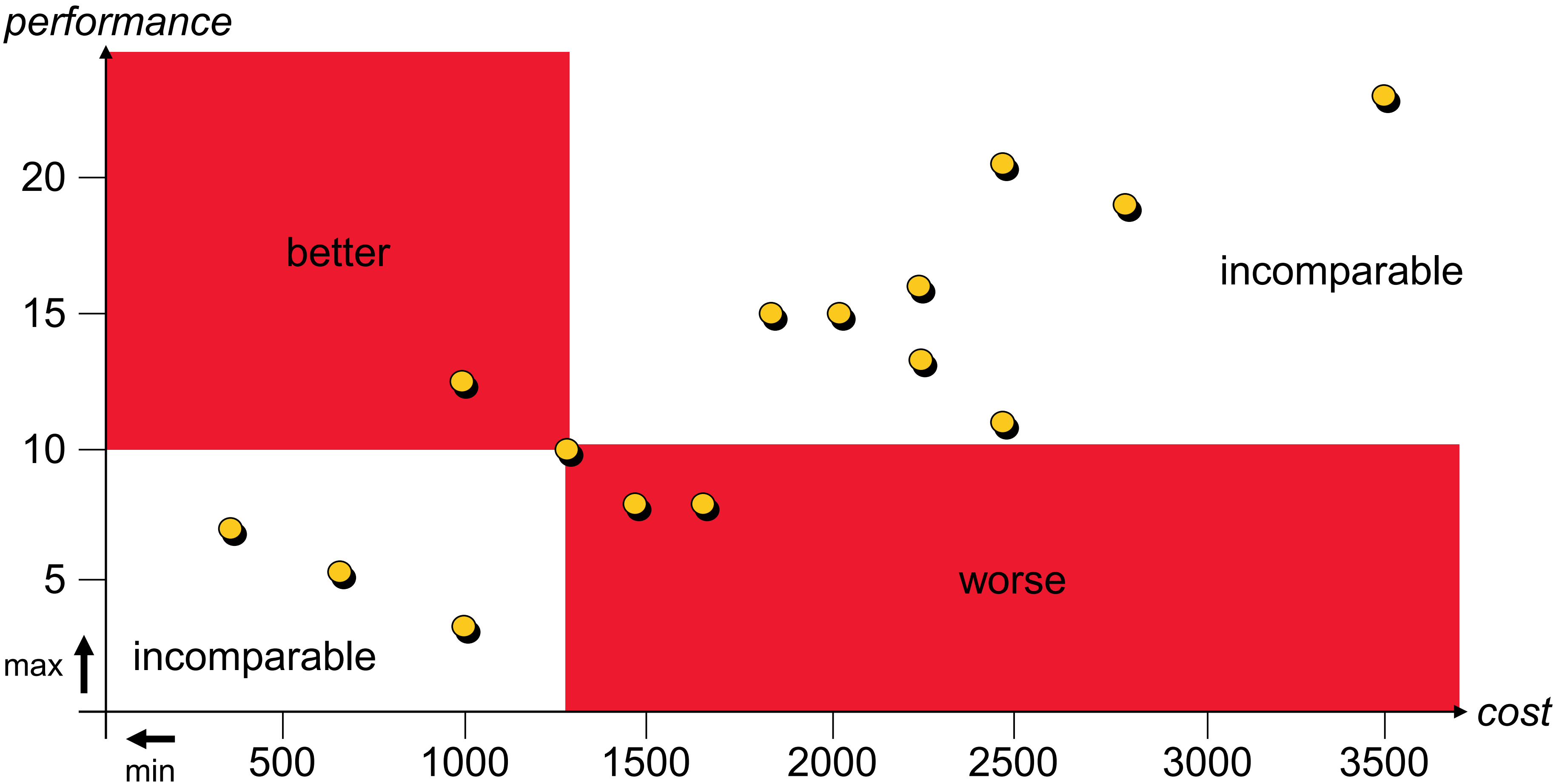


linear transformations

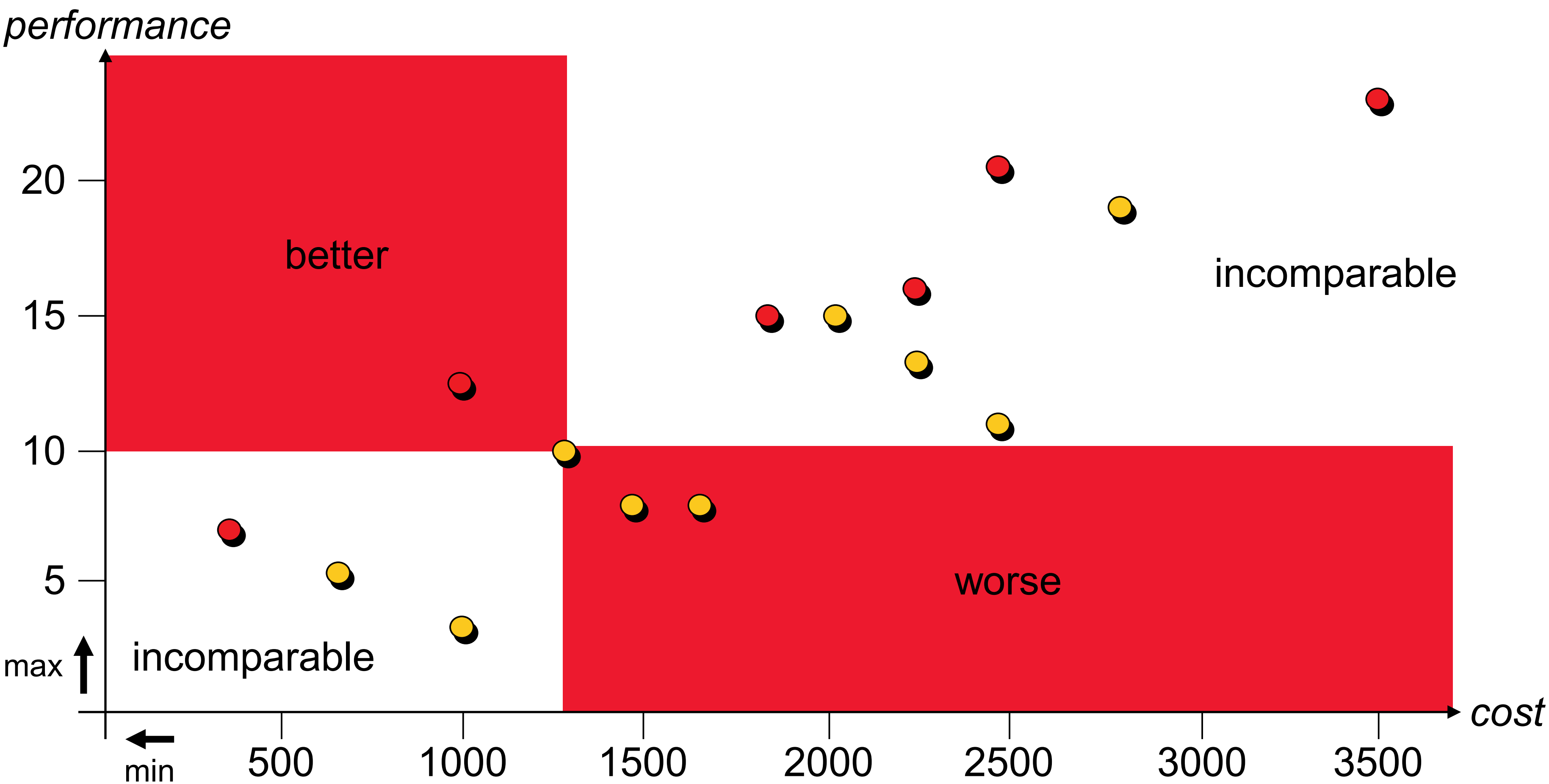
the recent extension to multi-objective optimization

Multiobjective Optimization (MOO)

Multiple objectives that have to be optimized simultaneously

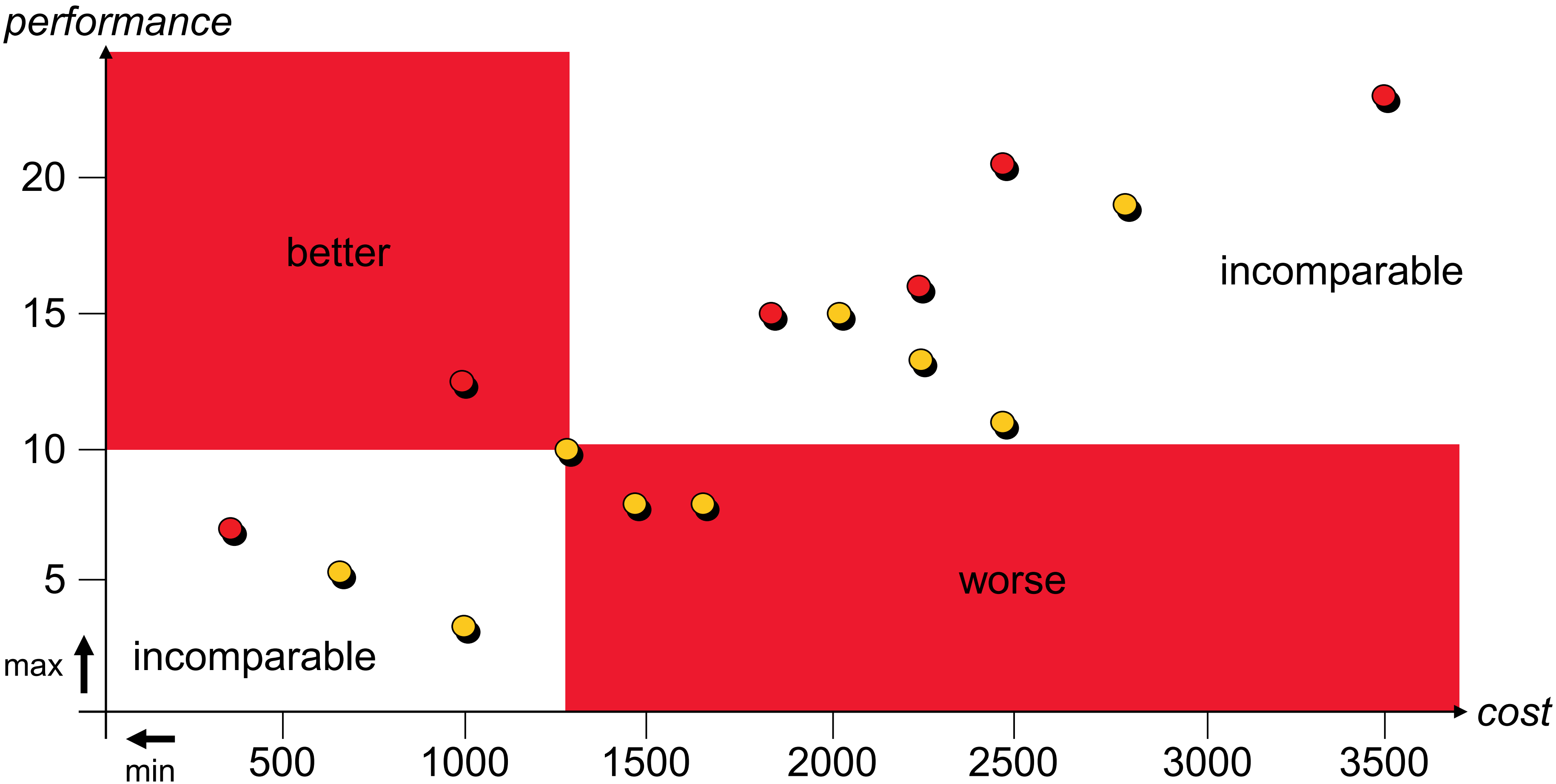


Observations: ❶ there is no single optimal solution, but
❷ some solutions ( ) are better than others ( )



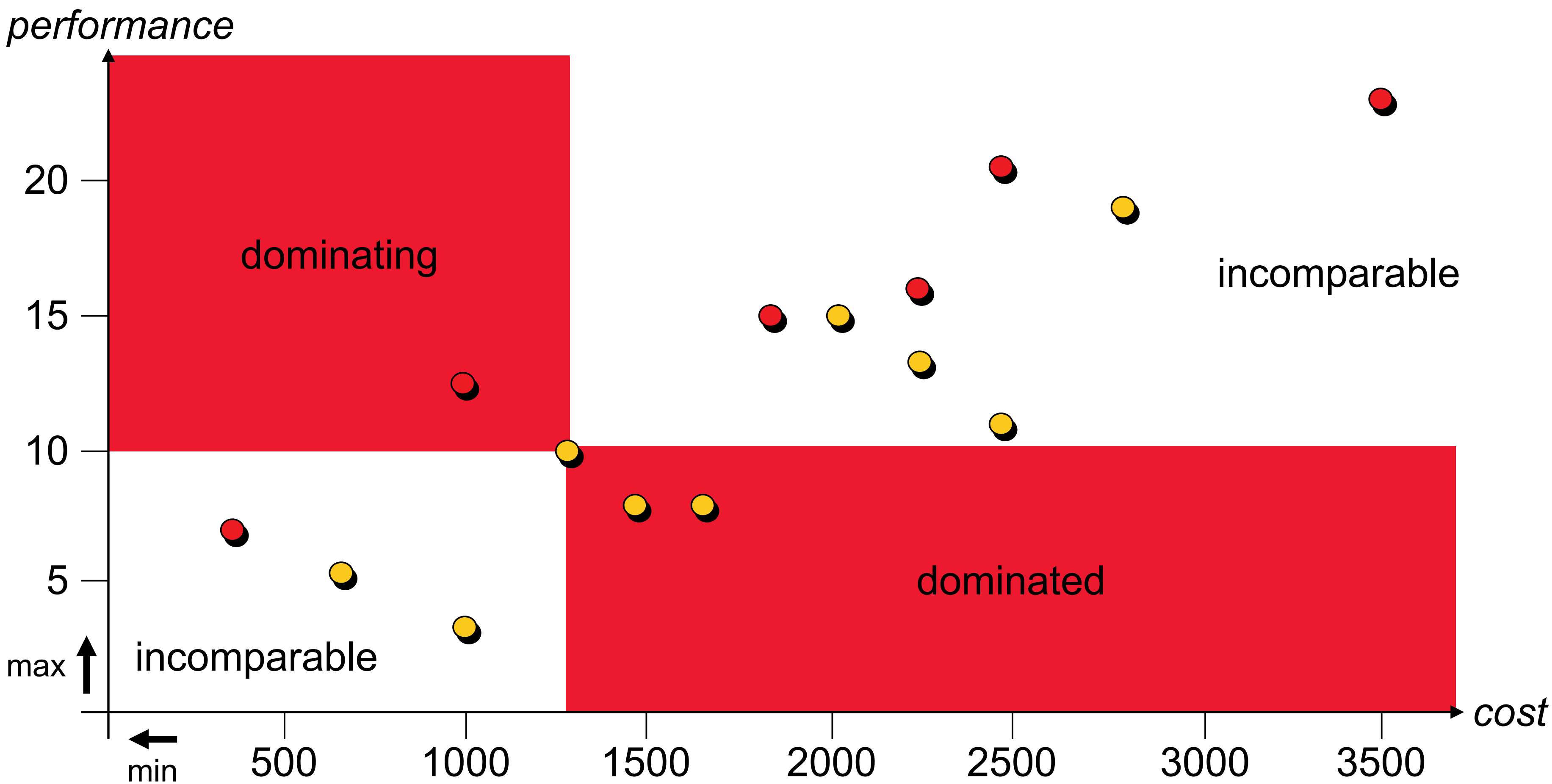
$$u \text{ weakly Pareto dominates } v \ (u \preccurlyeq_{par} v): \quad \forall 1 \leq i \leq k : f_i(u) \leq f_i(v)$$

$$u \text{ Pareto dominates } v \ (u \prec_{par} v): \quad u \preccurlyeq_{par} v \ \wedge \ v \not\preccurlyeq_{par} u$$

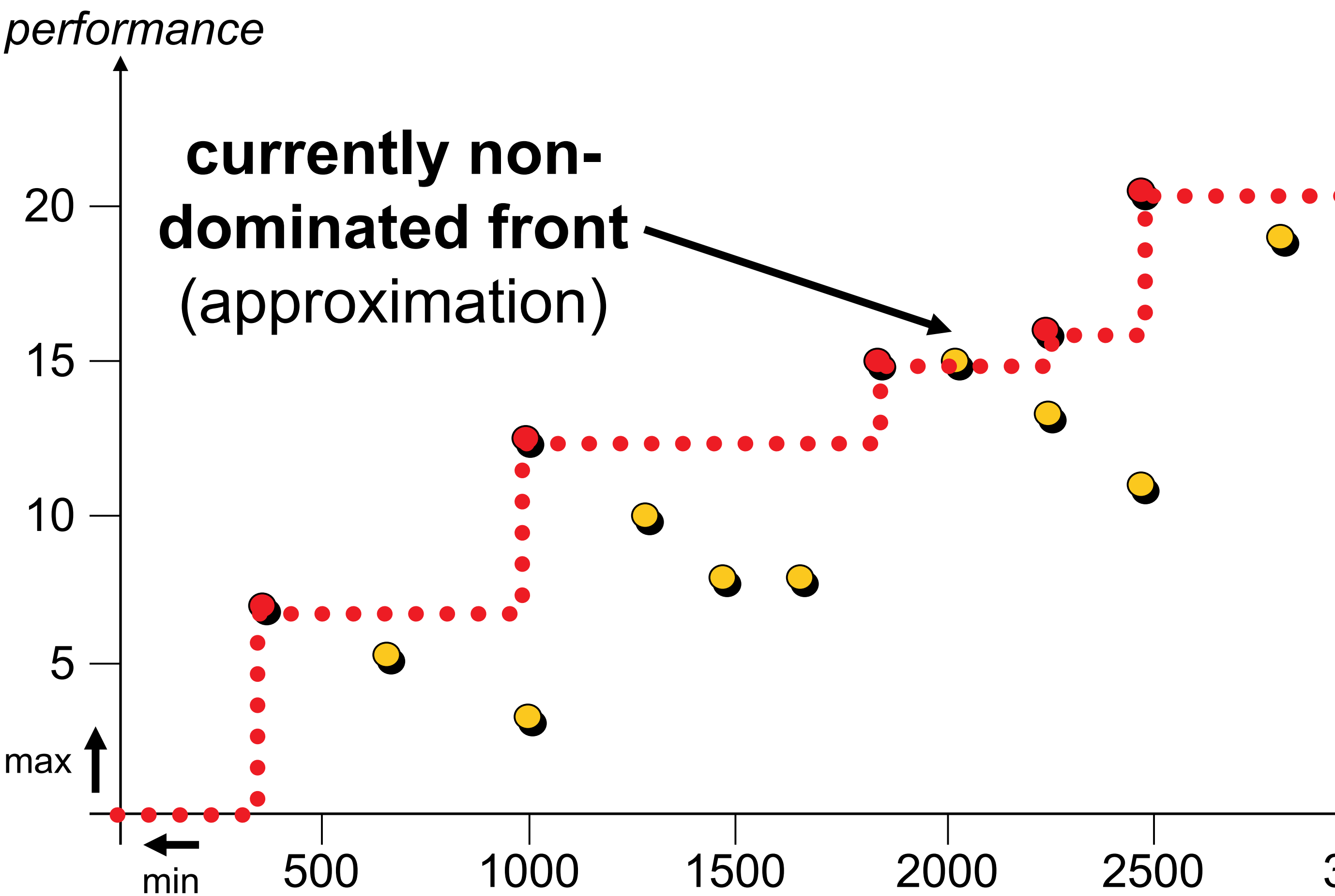


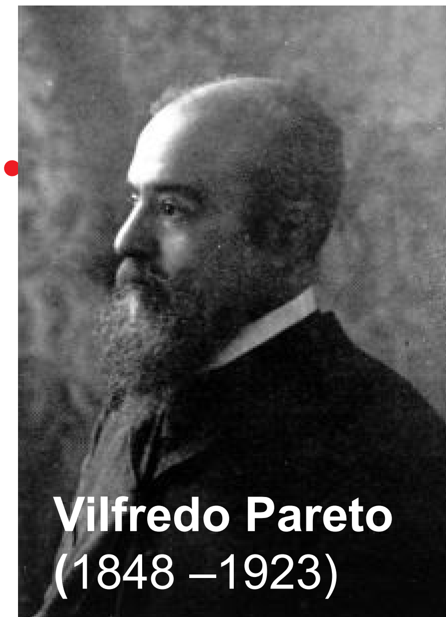
$$u \text{ weakly Pareto dominates } v \ (u \preccurlyeq_{par} v): \quad \forall 1 \leq i \leq k: f_i(u) \leq f_i(v)$$

$$u \text{ Pareto dominates } v \ (u \prec_{par} v): \quad u \preccurlyeq_{par} v \ \wedge \ v \not\preccurlyeq_{par} u$$



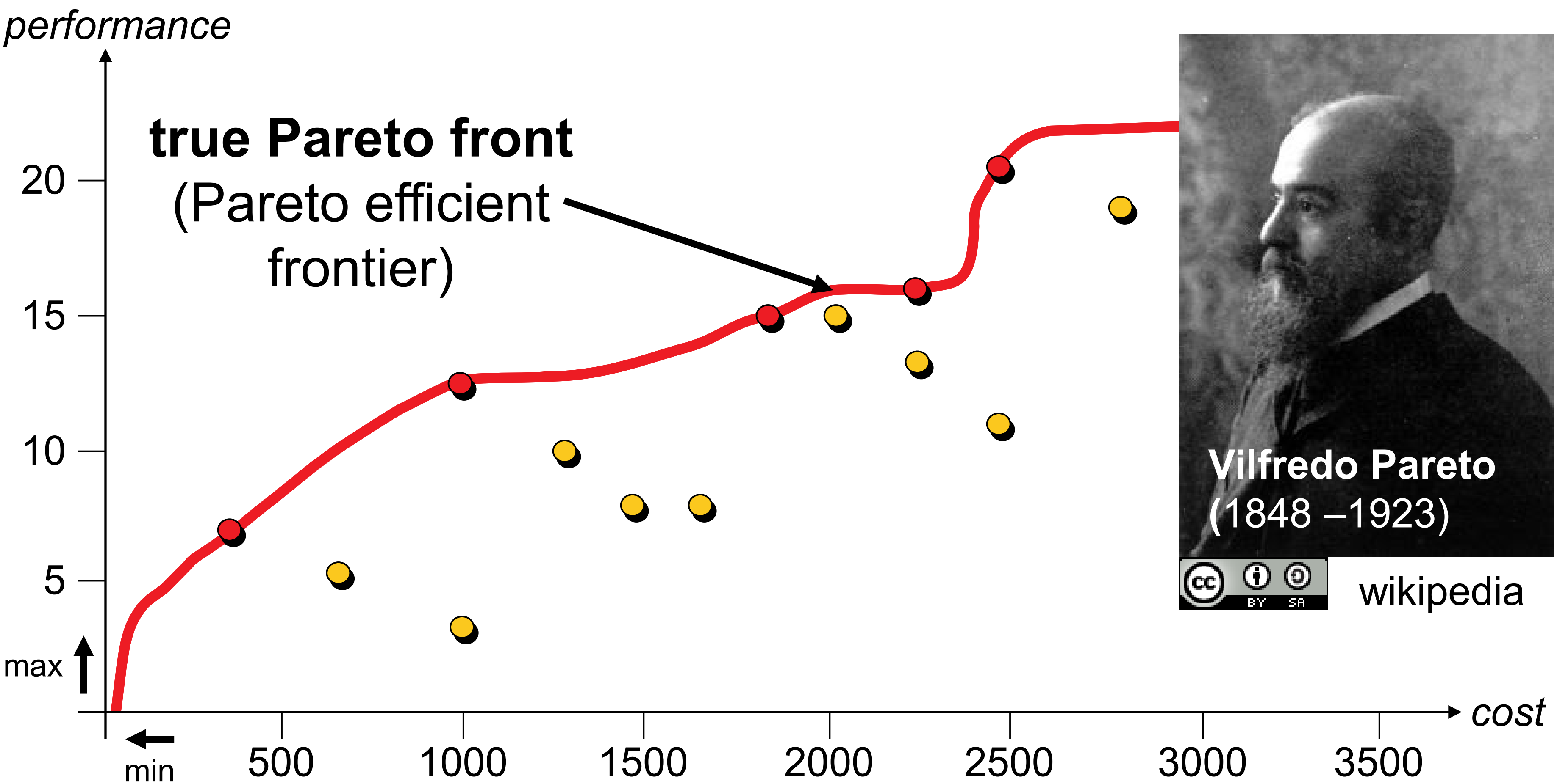
Pareto set: set of all non-dominated solutions (decision space)
Pareto front: its image in the objective space

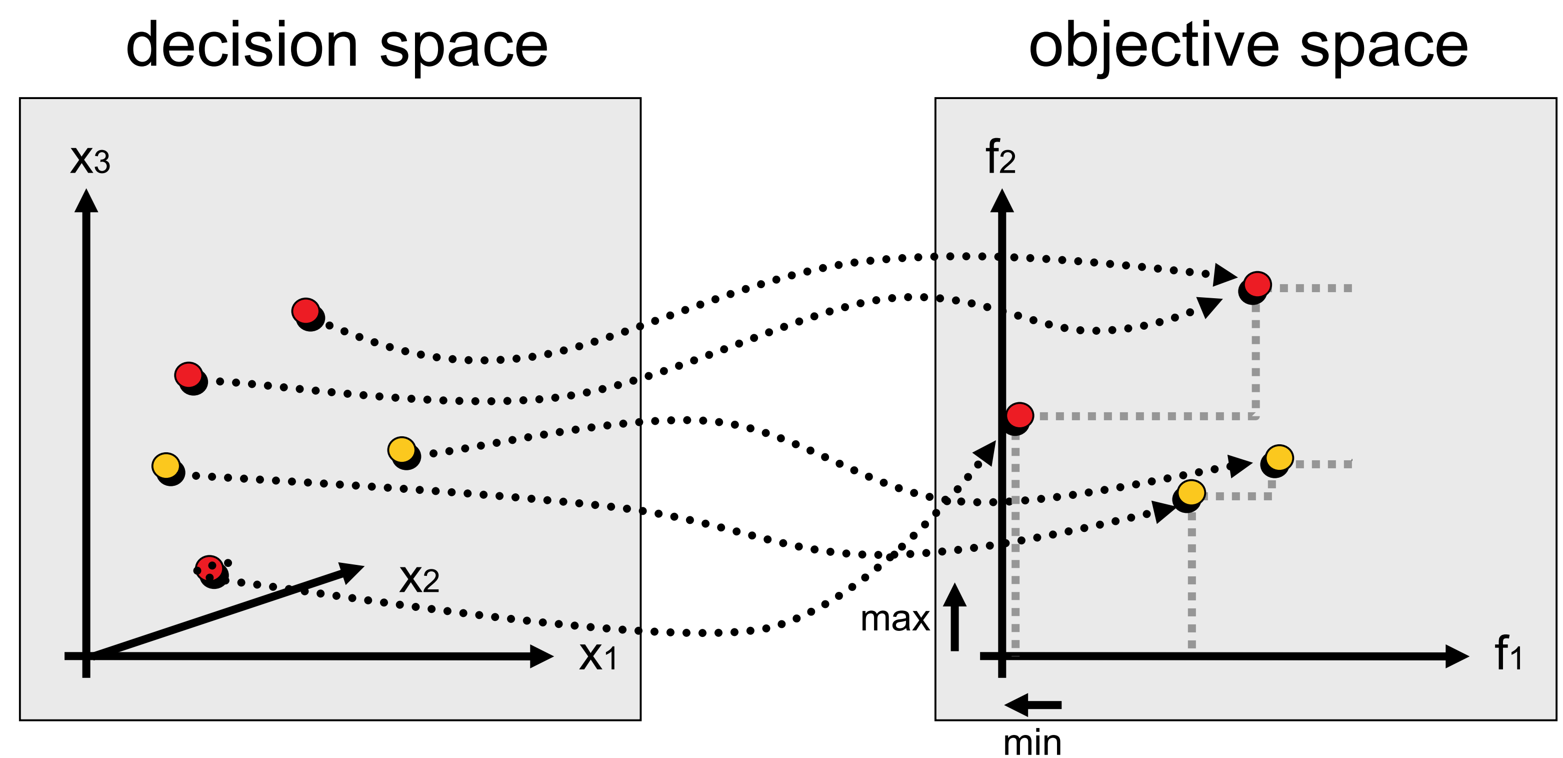




Pareto set: set of all non-dominated solutions (decision space)

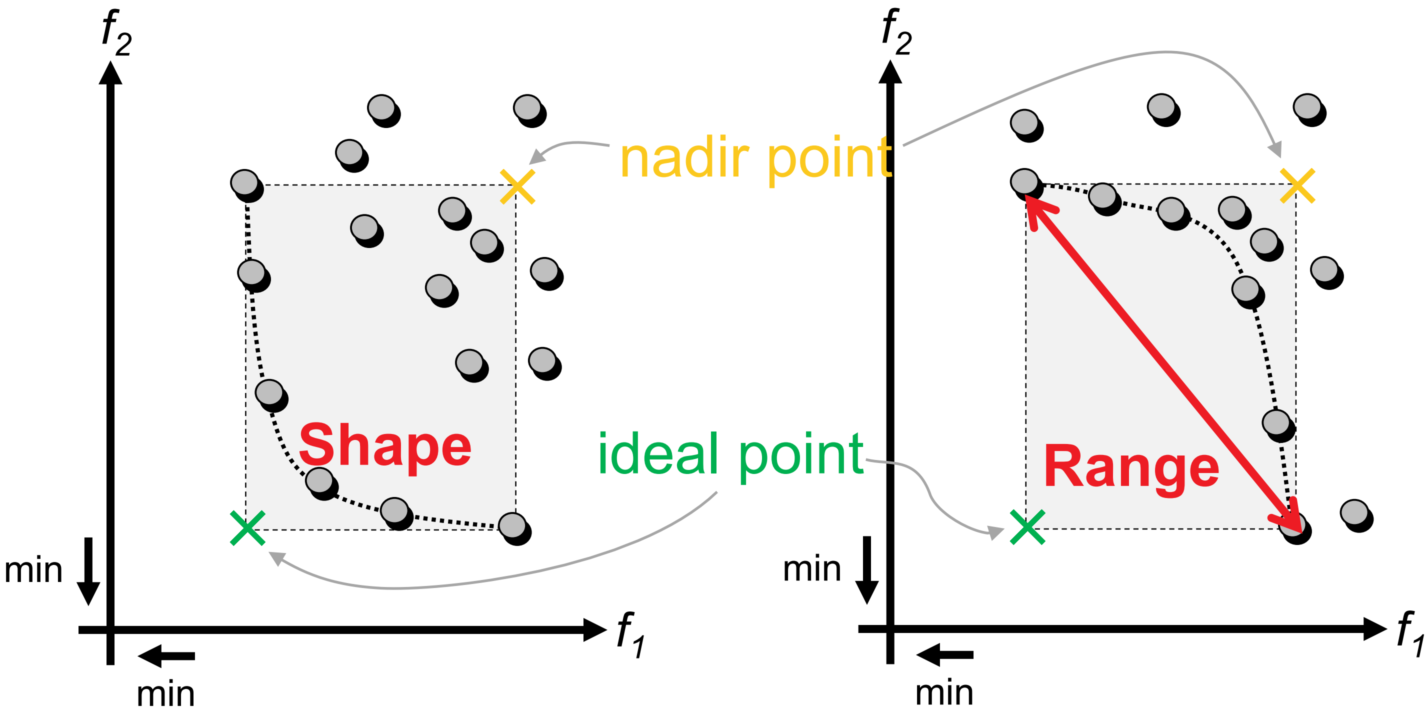
Pareto front: its image in the objective space





solution of Pareto-optimal set ● vector of Pareto-optimal front

non-optimal decision vector ● non-optimal objective vector



ideal point: best values
nadir point: worst values

} obtained for *Pareto-optimal* points

Quality Indicator Approach to MOO

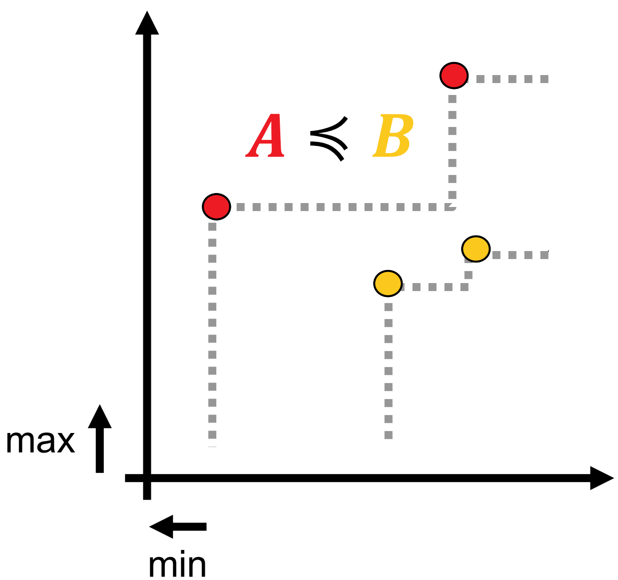
Idea:

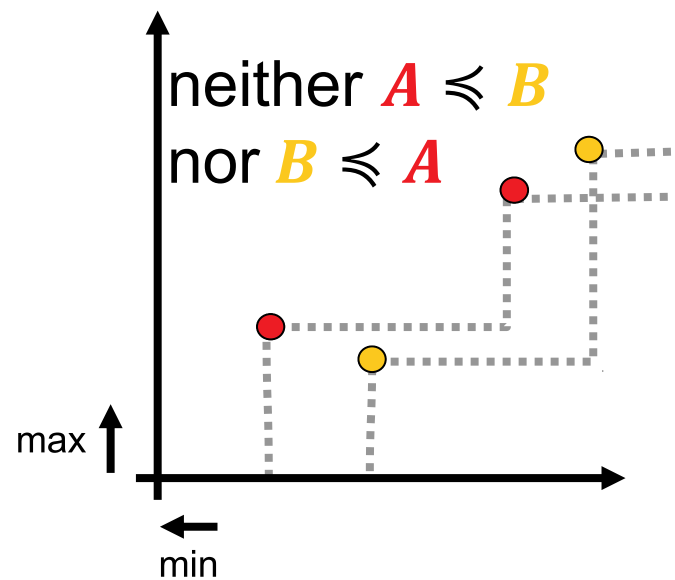
- transfer multiobjective problem into a set problem
- define an objective function (“quality indicator”) on sets

Important:

$\Rightarrow$ Underlying dominance relation (on sets) should be reflected by the resulting set comparisons!

$$A \preceq B :\Leftrightarrow \forall_{y \in B} \exists_{x \in A} x \leq_{par} y$$





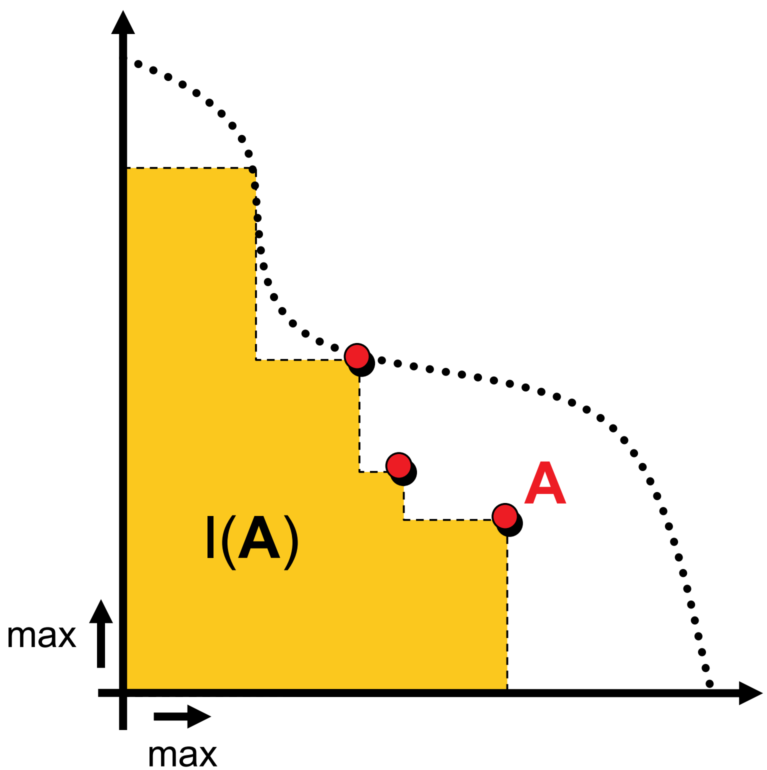
Examples of Quality Indicators

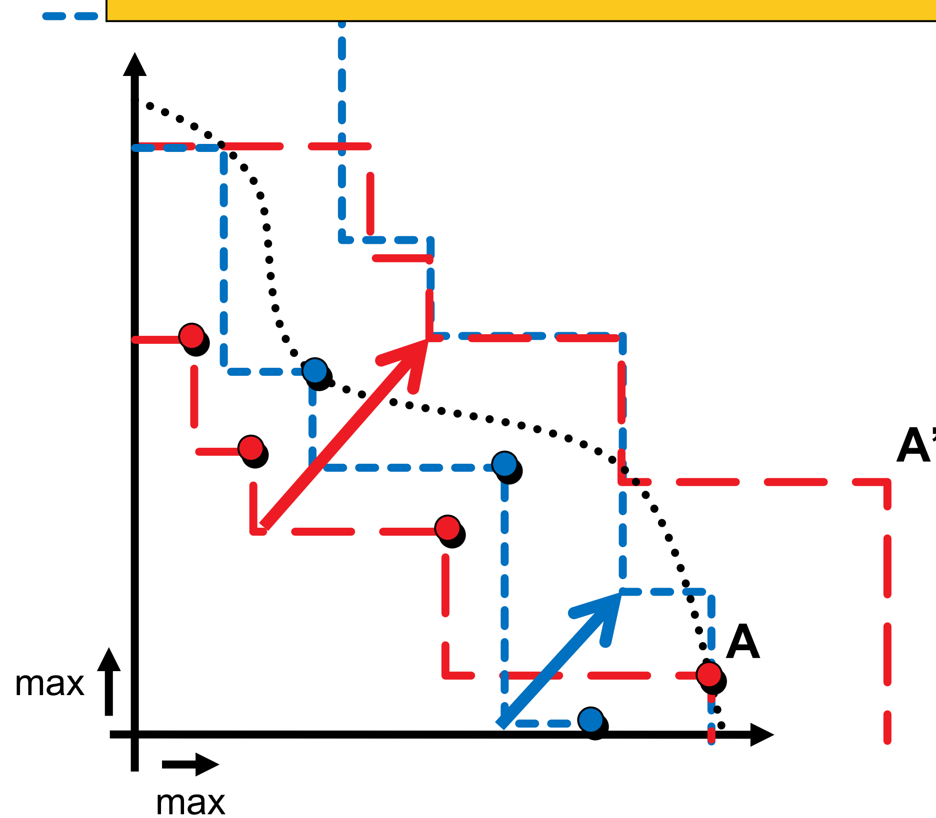
$$A \overset{\text{ref}}{\preccurlyeq} B :\Leftrightarrow I(A) \geq I(B)$$

$$A \overset{\text{ref}}{\preccurlyeq} B :\Leftrightarrow I(A,B) \leq I(B,A)$$

I(A) = volume of the weakly dominated area in objective space

I(A,B) = how much needs A to be moved to weakly dominate B





unary hypervolume indicator

binary epsilon indicator

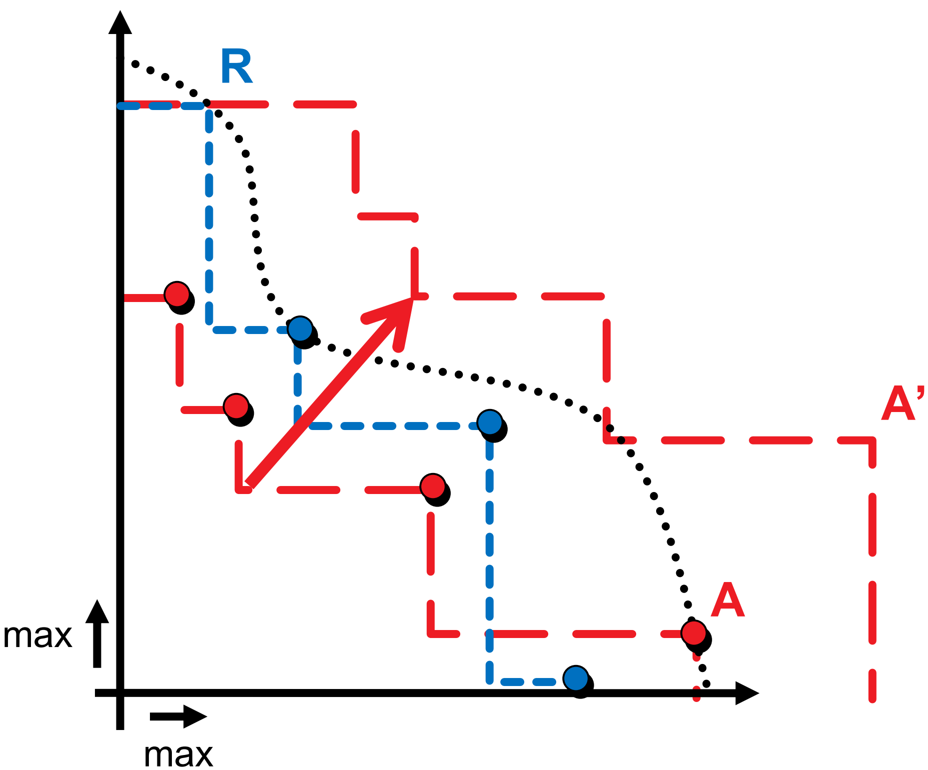
Examples of Quality Indicators II

$$A \overset{\text{ref}}{\preccurlyeq} B :\Leftrightarrow I(A,R) \leq I(B,R)$$

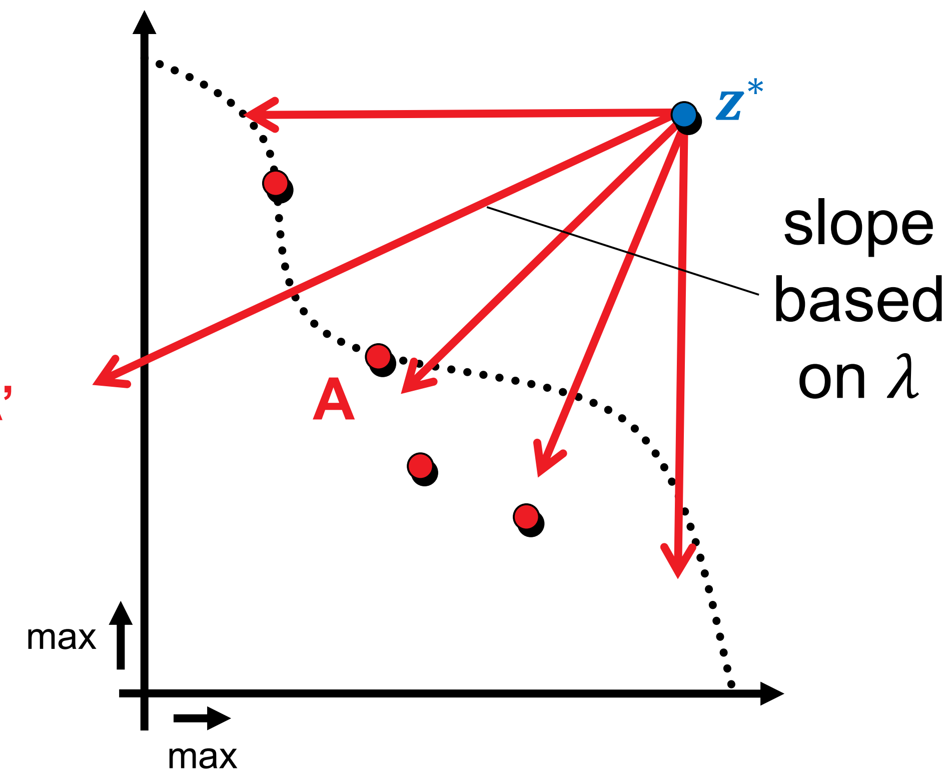
$$A \overset{\text{ref}}{\preccurlyeq} B :\Leftrightarrow I(A) \leq I(B)$$

I(A,R) = how much needs A to be moved to weakly dominate R

I(A) =
$$\frac{1}{|\Lambda|} \sum_{\lambda \in \Lambda} \min_{a \in A} \left(\max_{j=1..m} \lambda_j |z_j^* - a_j| \right)$$



unary epsilon indicator

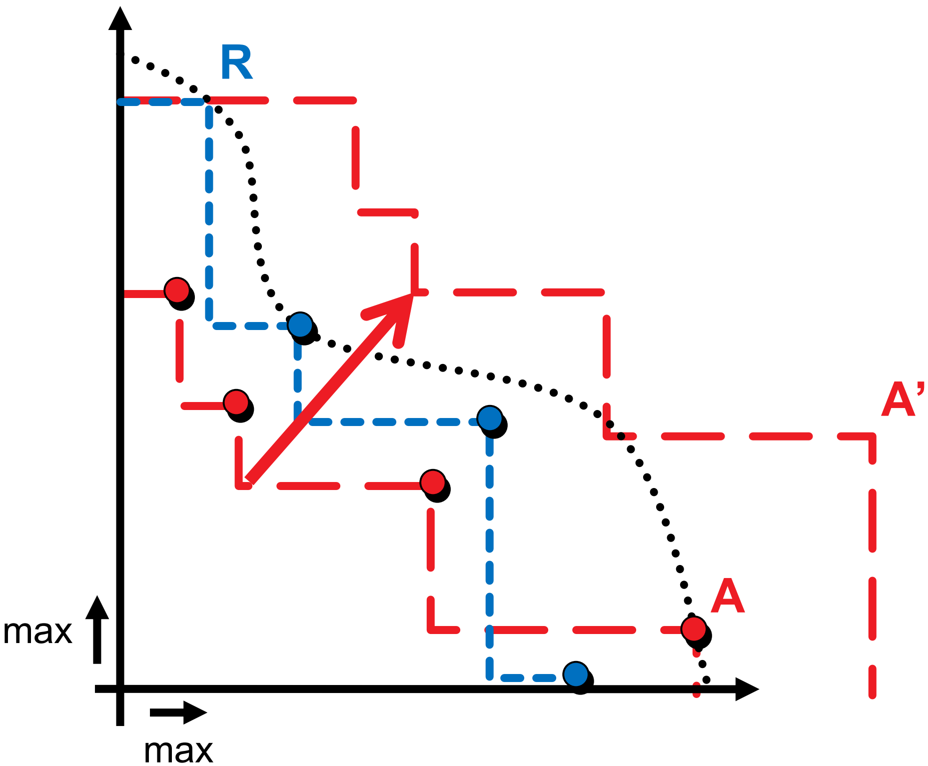


unary R2 indicator

Examples of Quality Indicators II

$$A \overset{\text{ref}}{\preccurlyeq} B :\Leftrightarrow I(A,R) \leq I(B,R)$$

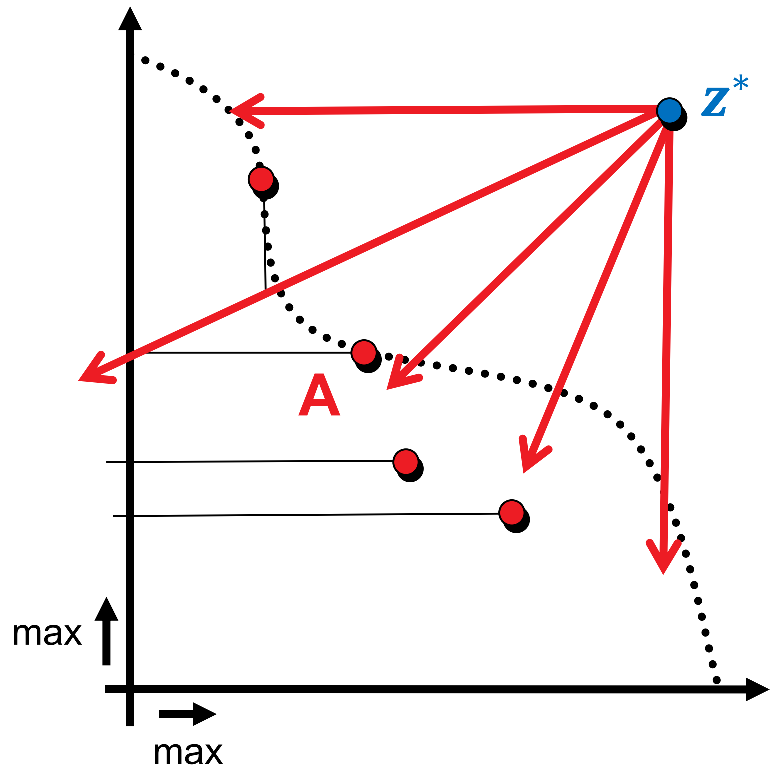
I(A,R) = how much needs A to be moved to weakly dominate R



unary epsilon indicator

$$A \overset{\text{ref}}{\preccurlyeq} B :\Leftrightarrow I(A) \leq I(B)$$

I(A) =
$$\frac{1}{|\Lambda|}\sum_{\lambda\in\Lambda} \min_{a\in A}\left(\max_{j=1..m} \lambda_j |z_j^* - a_j|\right)$$



unary R2 indicator

bbob-biobj Testbed

- 55 functions by combining 2 **bbob** functions

1 Separable Functions	
f1	Sphere Function ✓
f2	Ellipsoidal Function ✓
f3	Rastrigin Function
f4	Büche-Rastrigin Function
f5	Linear Slope
2 Functions with low or moderate conditioning	
f6	Attractive Sector Function ✓
f7	Step Ellipsoidal Function
f8	Rosenbrock Function, original ✓
f9	Rosenbrock Function, rotated
3 Functions with high conditioning and unimodal	
f10	Ellipsoidal Function
f11	Discus Function
f12	Bent Cigar Function
f13	Sharp Ridge Function ✓
f14	Different Powers Function ✓

4 Multi-modal functions with adequate global structure	
f15	Rastrigin Function ✓
f16	Weierstrass Function
f17	Schaffers F7 Function ✓
f18	Schaffers F7 Functions, moderately ill-conditioned
f19	Composite Griewank-Rosenbrock Function F8F2
5 Multi-modal functions with weak global structure	
f20	Schwefel Function ✓
f21	Gallagher's Gaussian 101-me Peaks Function ✓
f22	Gallagher's Gaussian 21-hi Peaks Function
f23	Katsuura Function
f24	Lunacek bi-Rastrigin Function

bbob-biobj Testbed

- 55 functions by combining 2 **bbob** functions

1 Separable Functions	
f1	Sphere Function ✓
f2	Ellipsoidal Function ✓
f3	Rastrigin Function
f4	Büche-Rastrigin Function
f5	Linear Slope
2 Functions with low or moderate conditionir	
f6	Attractive Sector Function ✓
f7	Step Ellipsoidal Function
f8	Rosenbrock Function, original ✓
f9	Rosenbrock Function, rotated
3 Functions with high conditioning and unimo	
f10	Ellipsoidal Function
f11	Discus Function
f12	Bent Cigar Function
f13	Sharp Ridge Function ✓
f14	Different Powers Function ✓

4 Multi-modal functions with adequate global structure	
f15	Rastrigin Function ✓
f16	Weierstrass Function
f17	Schaffers F7 Function ✓

	f_1	f_2	f_6	f_8	f_{13}	f_{14}	f_{15}	f_{17}	f_{20}	f_{21}
f_1	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10
f_2		f11	f12	f13	f14	f15	f16	f17	f18	f19
f_6			f20	f21	f22	f23	f24	f25	f26	f27
f_8				f28	f29	f30	f31	f32	f33	f34
f_{13}					f35	f36	f37	f38	f39	f40
f_{14}						f41	f42	f43	f44	f45
f_{15}							f46	f47	f48	f49
f_{17}								f50	f51	f52
f_{20}									f53	f54
f_{21}										f55

`bbob-biobj` Testbed

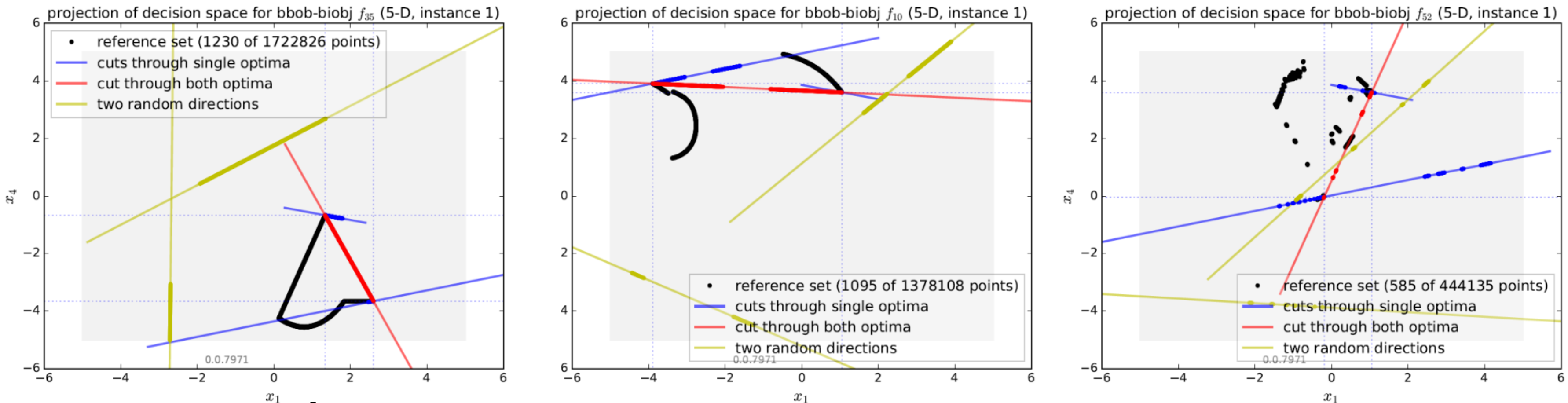
- 55 functions by combining 2 `bbob` functions
- 15 function groups with 3-4 functions each
  - separable – separable, separable – moderate, separable - ill-conditioned, ...
- 6 dimensions: 2, 3, 5, 10, 20, (40 optional)
- instances derived from `bbob` instances:
- no normalization (algo has to cope with different orders of magnitude)
- for performance assessment: ideal/nadir points known

`bbob-biobj` Testbed (cont'd)

- Pareto set and Pareto front unknown
  - but we have a good idea of where they are by running quite some algorithms and keeping track of all non-dominated points found so far
- Various types of shapes

bbob-biobj Testbed (cont'd)

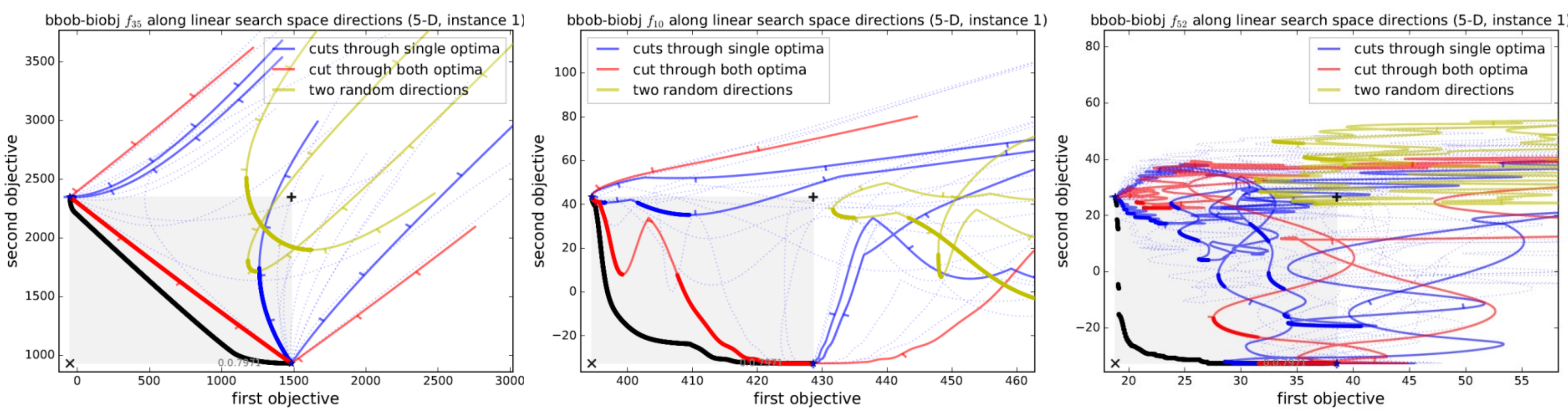
search space



connected
uni-modal

disconnected
multi-modal

objective space



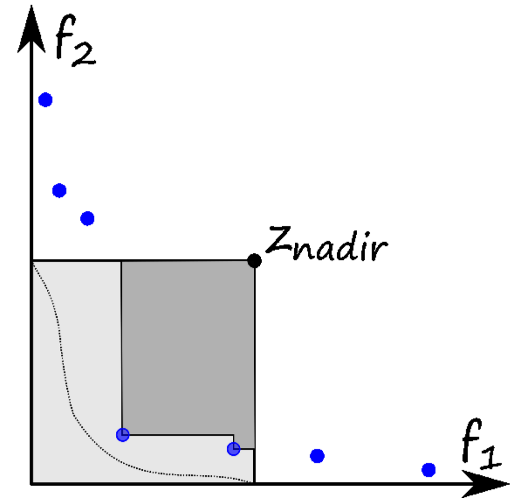
Bi-objective Performance Assessment

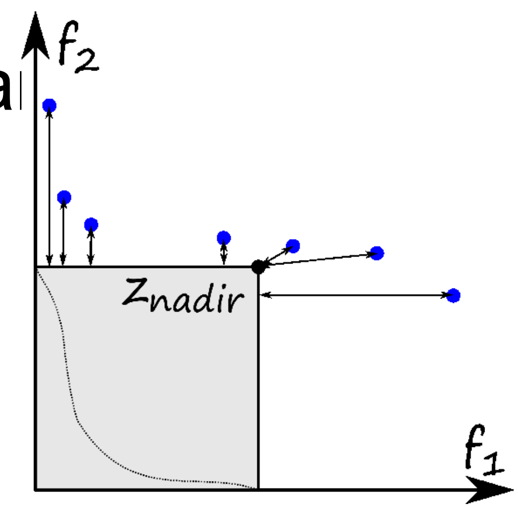
algorithm quality =

normalized* hypervolume (HV)
of all non-dominated solutions
if a point dominates nadir

closest normalized* negative dista
to region of interest $[0,1]^2$
if no point dominates nadir

* such that ideal=[0,0] and nadir=[1,1]





Bi-objective Performance Assessment

We measure runtimes to reach (HV indicator) targets:

- relative to a reference set, given as the best Pareto front approximation known (since exact Pareto set not known)
- actual absolute hypervolume targets used are

HV(refset) – targetprecision

with 58 fixed targetprecisions between +1 and -10^{-4} (same for all functions, dimensions, and instances) in the displays

Course Overview

1	Mon, 17.9.2018 Thu, 20.9.2018	Monday's lecture: introduction, example problems, problem types groups defined via wiki everybody went (actively!) through the Getting Started part of github.com/numbbo/coco ❷ remaining part difficulties in cont. opt.
2	Fri, 21.9.2018	❸ today's lecture "Benchmarking", ❶ final adjustments of groups everybody can run and postprocess the example experiment (❹ ~1h for final questions/help during the lecture)
3	Fri, 28.9.2018	lecture "Introduction to Continuous Optimization"
4	Fri, 5.10.2018	lecture "Gradient-Based Algorithms"
5	Fri, 12.10.2018	lecture "Stochastic Algorithms and DFO"
6	Fri, 19.10.2018	lecture "Discrete Optimization I: graphs, greedy algos, dyn. progr." deadline for submitting data sets
7	Wed, 24.10.2018 Fri, 26.10.2018	deadline for paper submission final lecture "Discrete Optimization II: dyn. progr., B&B, heuristics"
	29.10.-2.11.2018	vacation aka learning for the exams
	Thu, 8.11.2018 / Fri, 9.11.2018	oral presentations (individual time slots)
	Fri, 16.11.2018	written exam

All deadlines:
23:59pm Paris time

Conclusions Benchmarking Continuous Optimizers

I hope it became clear...

...what are important problem difficulties in continuous optimization
...what are the important issues in algorithm benchmarking
...which functionality is behind the COCO platform
...and how to measure performance in particular
...what are the basics of multiobjective optimization
...and what are the next important steps to do:

- read the assigned paper and implement the algorithm
- document everything on the wiki
- run COCO experiment with your algorithm and share your data until Friday 19th of October, 2018

And now...

...time for your questions and problems around COCO