Introduction to Optimization

Lectures 5&6: Benchmarking + Discrete Optimization

October 30 and November 15, 2019

TC2 - Optimisation

Université Paris-Saclay

Anne Auger and Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

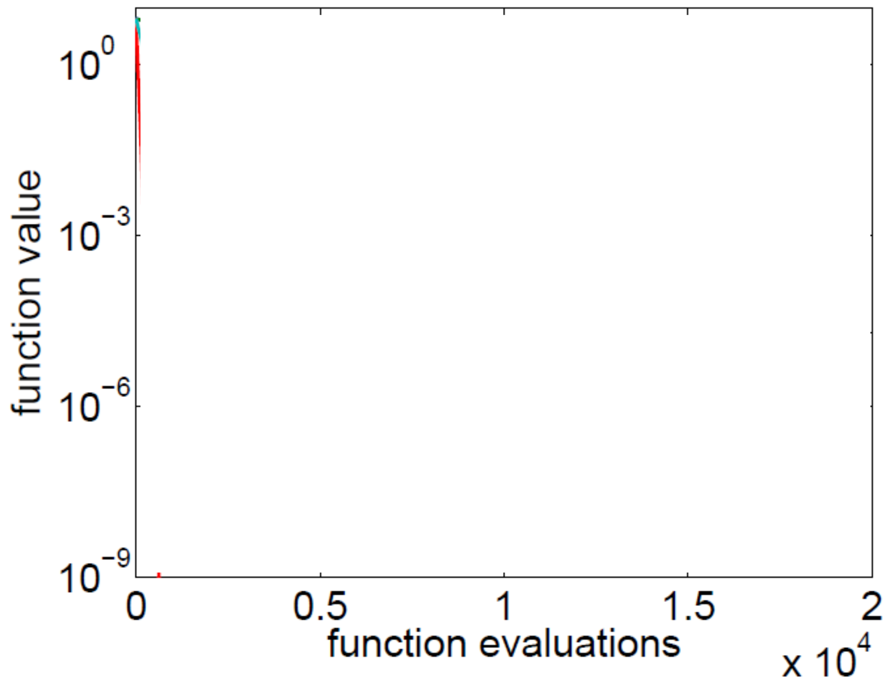
Date		Topic
Fri, 27.9.2019	DB	Introduction
Fri, 4.10.2019 (4hrs)	AA	Continuous Optimization I: differentiability, gradients, convexity, optimality conditions
Fri, 11.10.2019 (4hrs)	AA	Continuous Optimization II: constrained optimization, gradient-based algorithms, stochastic gradient
Fri, 18.10.2019 (4hrs)	DB	Continuous Optimization III: stochastic algorithms, derivative-free optimization, ~~critical performance assessment~~ [1st written test]
Wed, 30.10.2019	DB	Benchmarking + Discrete Optimization I: graph theory, greedy algorithms
Fri, 15.11.2019	DB	Discrete Optimization II: dynamic programming, heuristics [2nd written test]
Fri, 22.11.2018		final exam

Little Quiz (not graded)

CMA-ES as a stochastic search algorithm:

1) What's the underlying probability distribution?
2) How to update the mean?
3) When the progress is slower than expected, then …
4) When the progress is faster than expected, then …
5) With respect to which transformations is CMA-ES invariant?

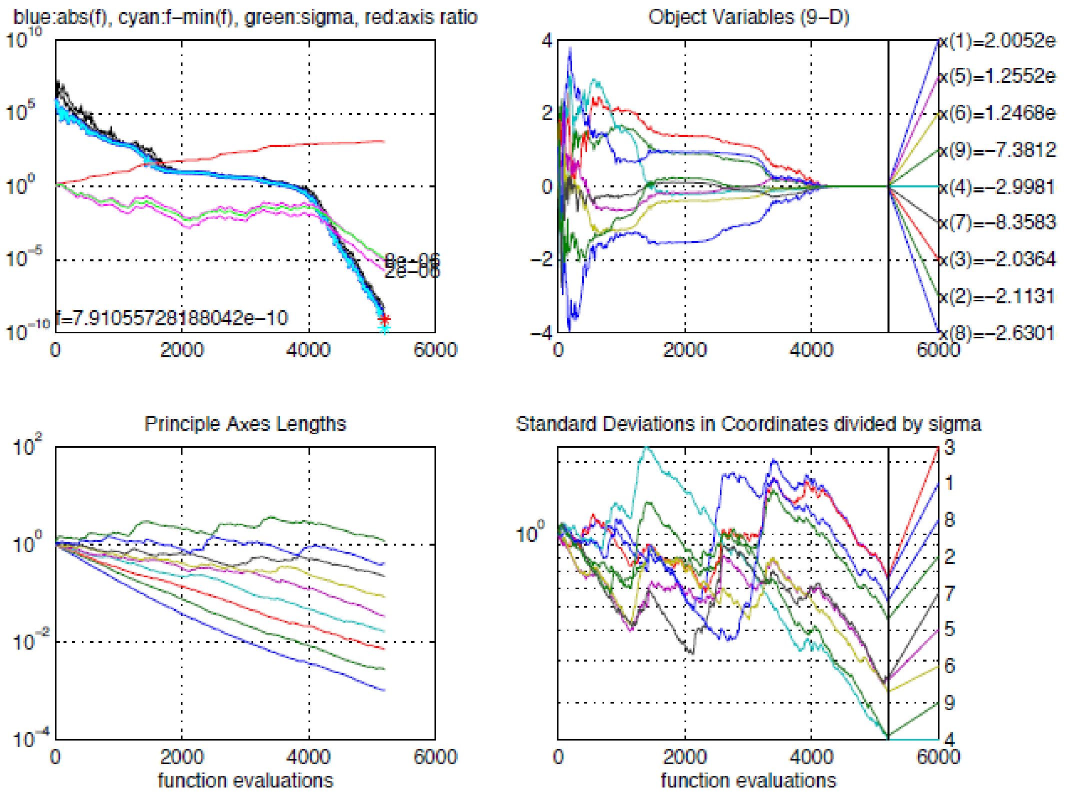
6) How does the constant stepsize (1+1)-ES looks like on this graph?



$$f(\boldsymbol{x}) = \sum_{i=1}^{n} x_i^2$$

in $[-0.2, 0.8]^n$
for $n = 10$

7) Is the function, optimized by CMA-ES here, separable?



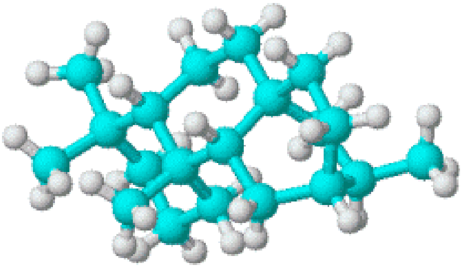
$$f(\boldsymbol{x}) = g\left(\boldsymbol{x}^{\mathrm{T}}\mathbf{H}\boldsymbol{x}\right),\ g : \mathbb{R} \to \mathbb{R} \text{ stricly increasing}$$

$\mathbf{C} \propto \boldsymbol{H}^{-1}$ for all $g, \mathbf{H}$

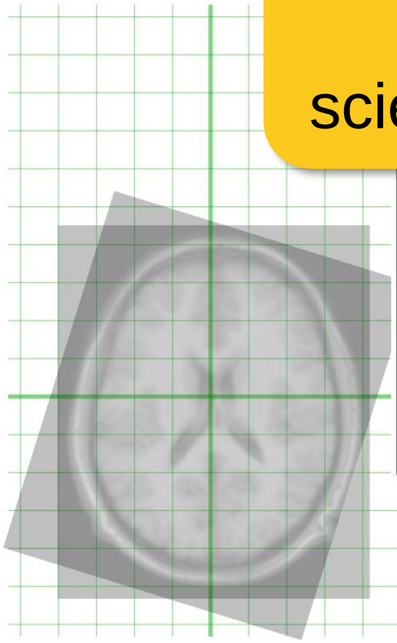
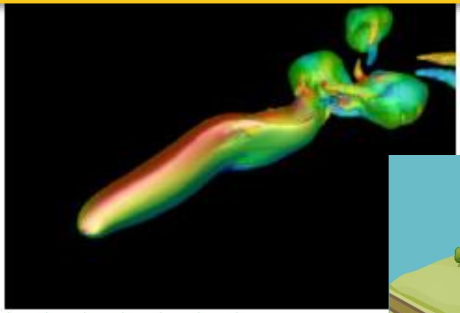
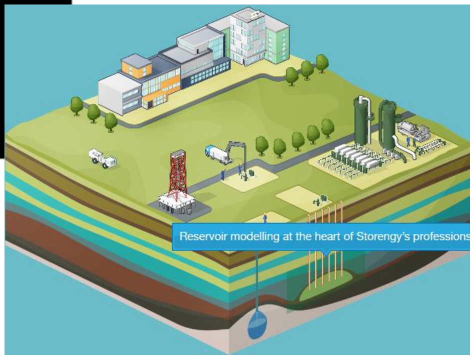
from [Hansen, p. 93]

Benchmarking Optimization Algorithms

or: critical performance assessment

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?

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

Do you remember the last Exercise?

How would you compare algorithms?

assumptions:

- continuous search space $\mathbb{R}^n$
- blackbox scenario w/o constraints
- two algorithms

a) Define a concrete experimental setup

- What to do if I want to compare algorithms A and B?
- Which experiment parameters you have to decide on?

b) What would you display to compare the performance?

c) Generalize

- arbitrary search space
- constraints
- any number of algorithms
- deterministic vs. stochastic algorithms

wouldn't

automatized benchmarking

be cool?

for this, we developed COCO

Comparing Continuous Optimizers Platform

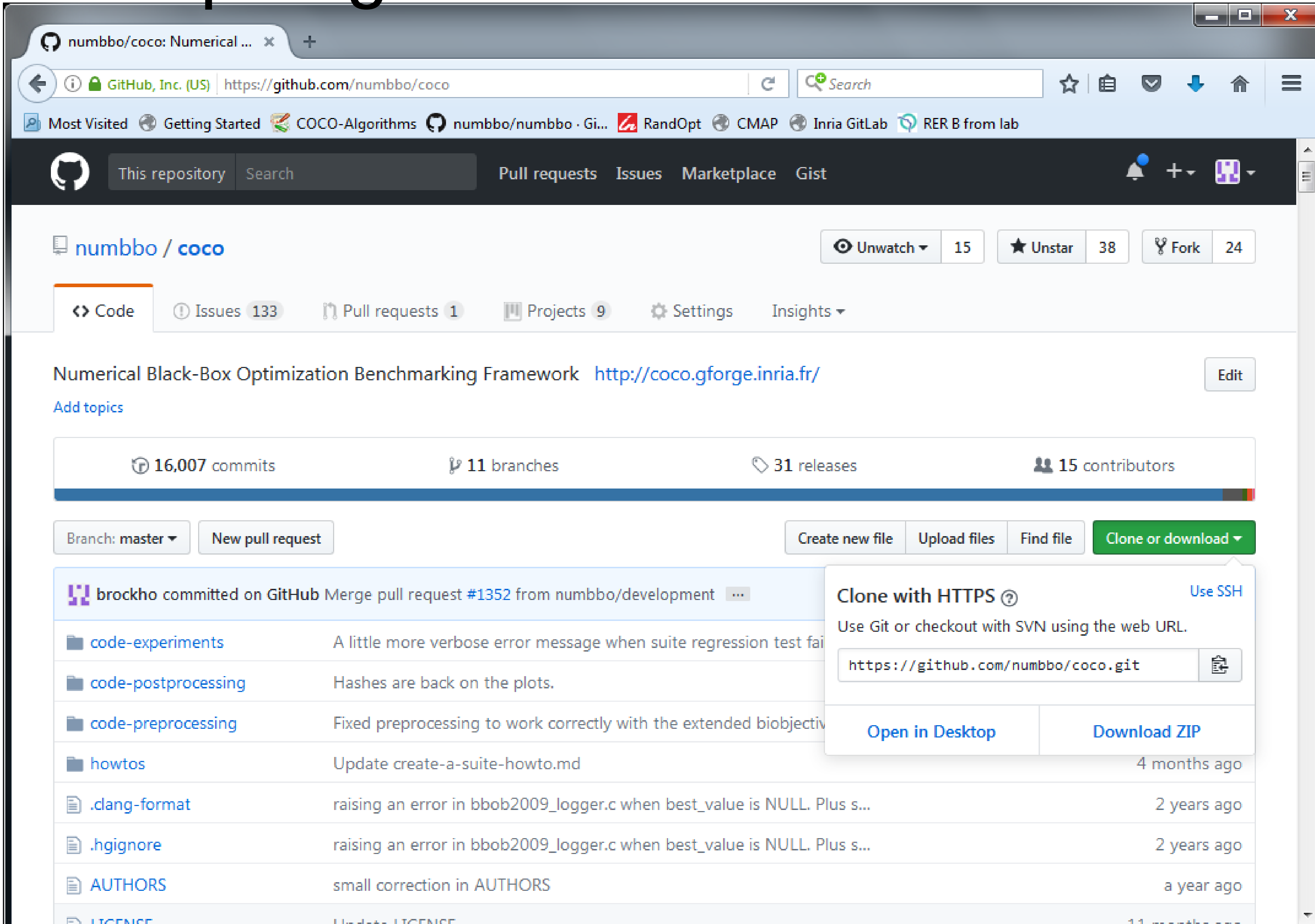
`https://github.com/numbbo/coco`

benchmarking is non-trivial

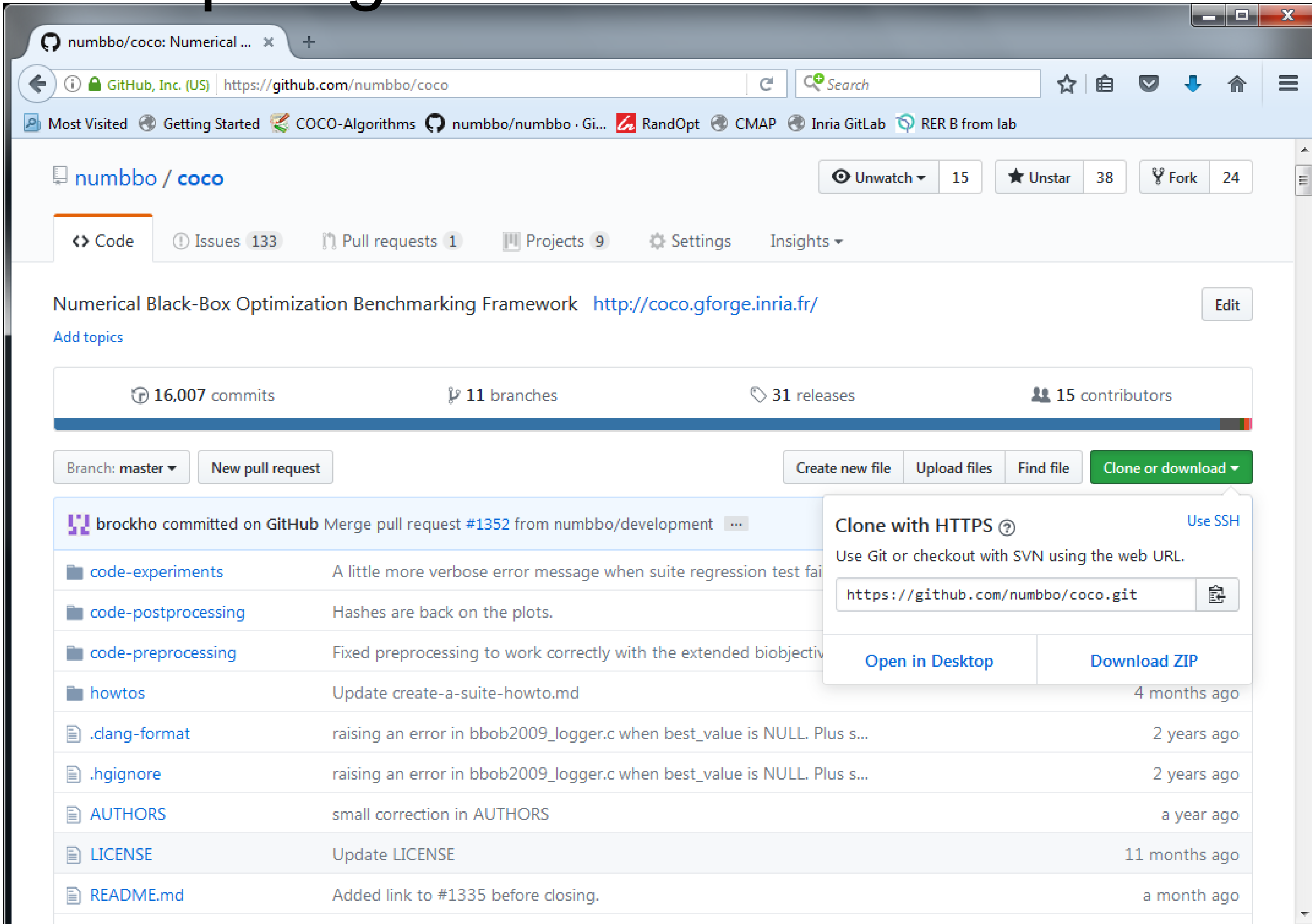
hence, COCO implements a
reasonable, well-founded, and
well-documented
pre-chosen methodology

How to benchmark algorithms with COCO?

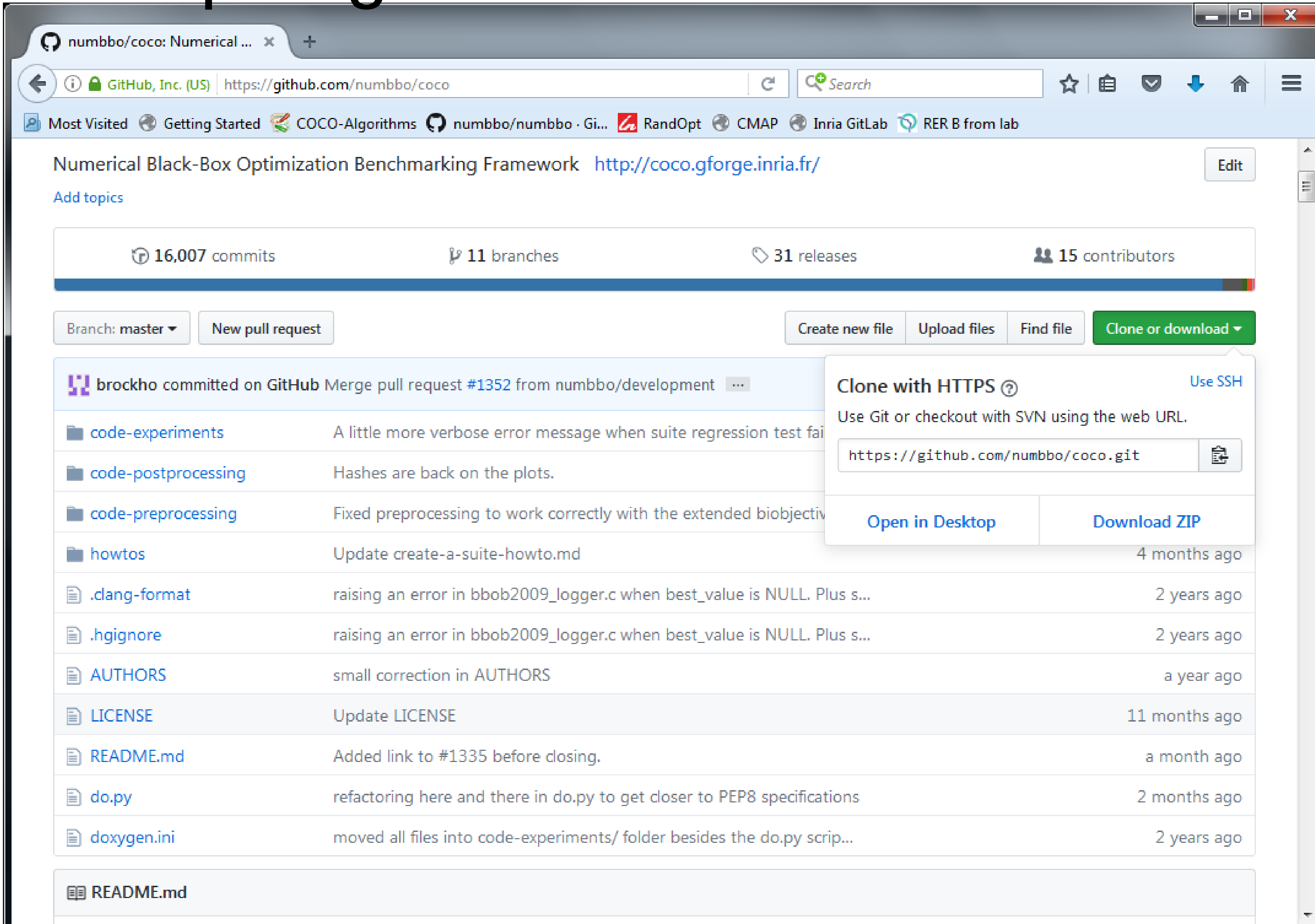
https://github.com/numbbo/coco



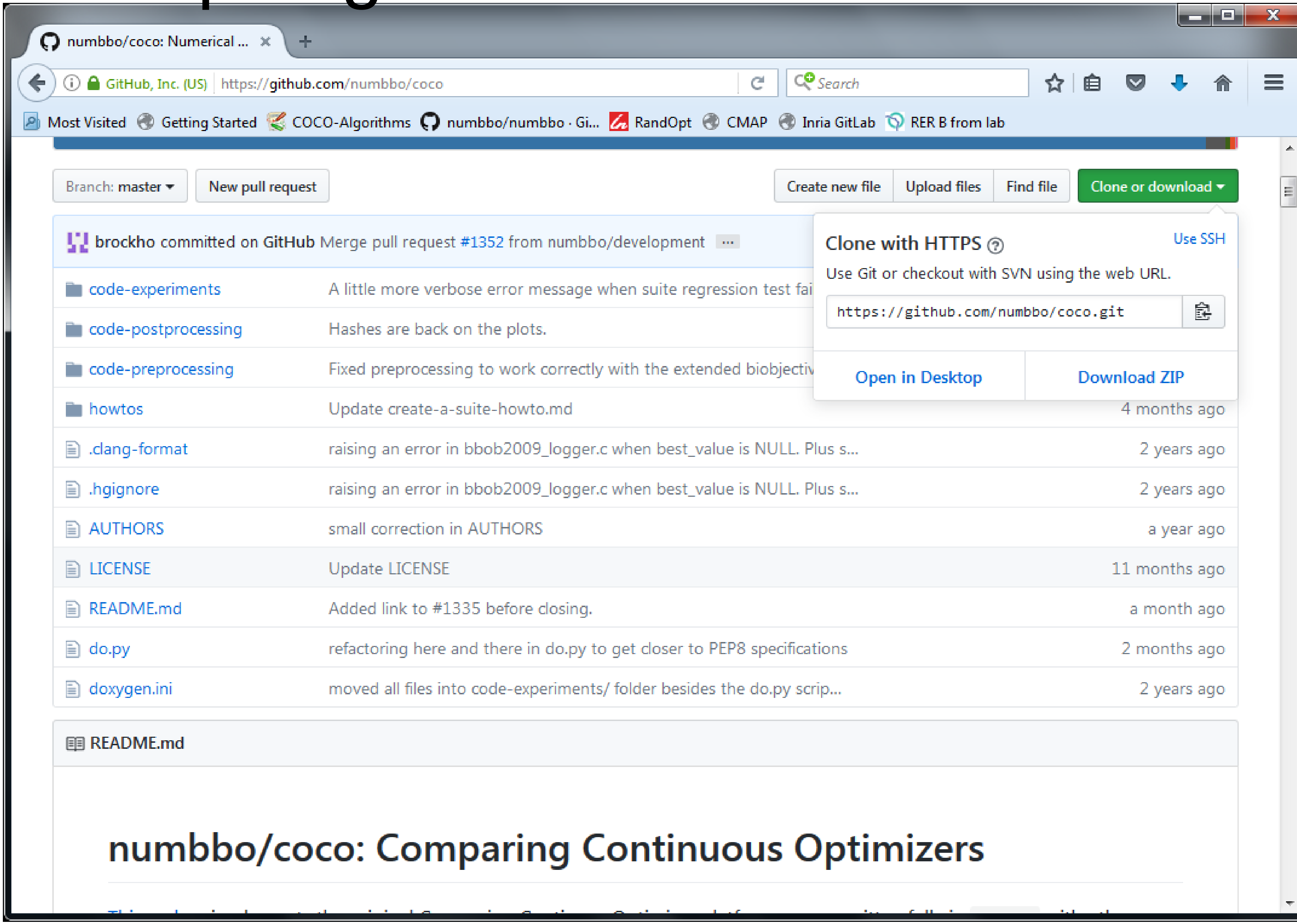
https://github.com/numbbo/coco



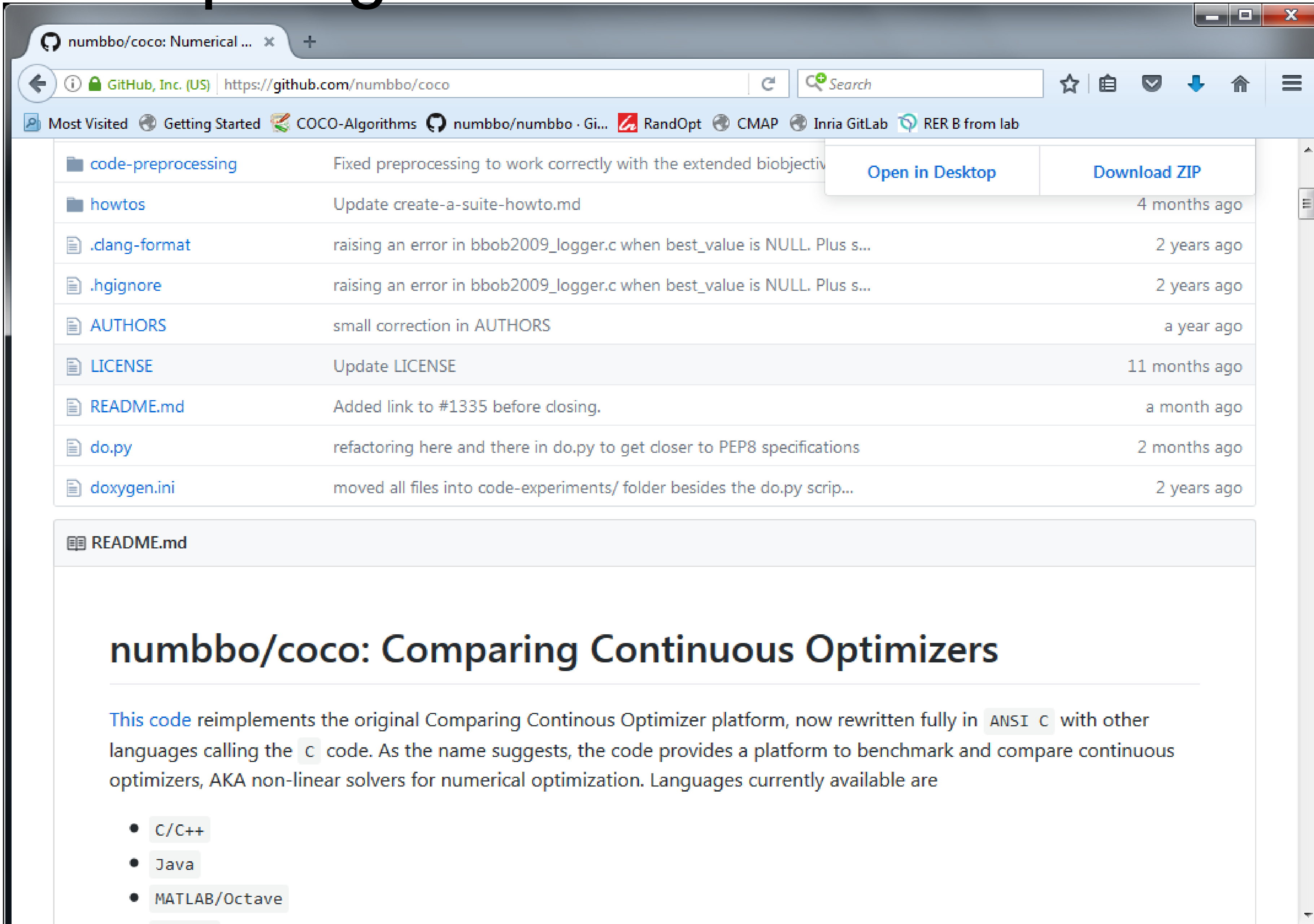
https://github.com/numbbo/coco



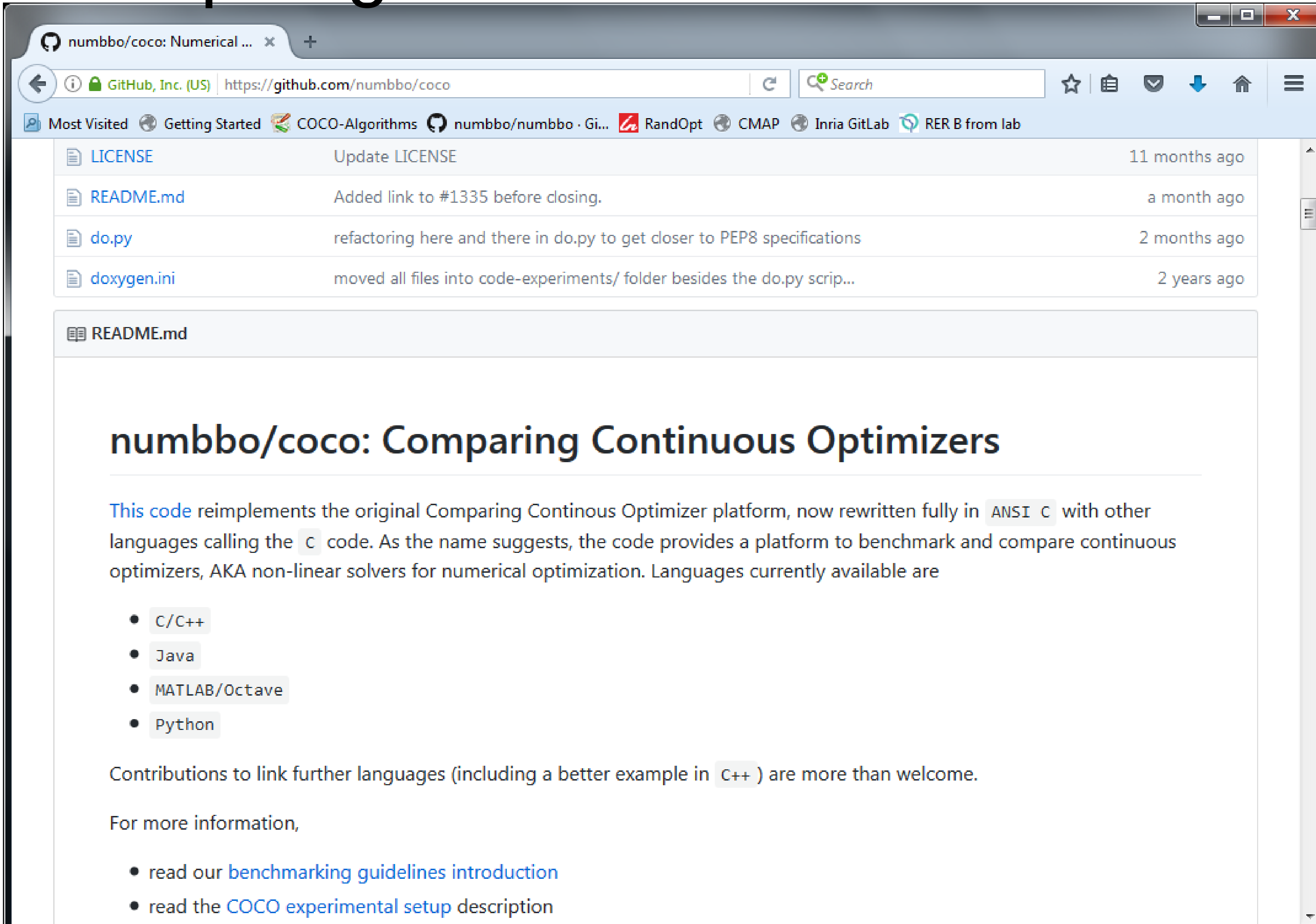
https://github.com/numbbo/coco



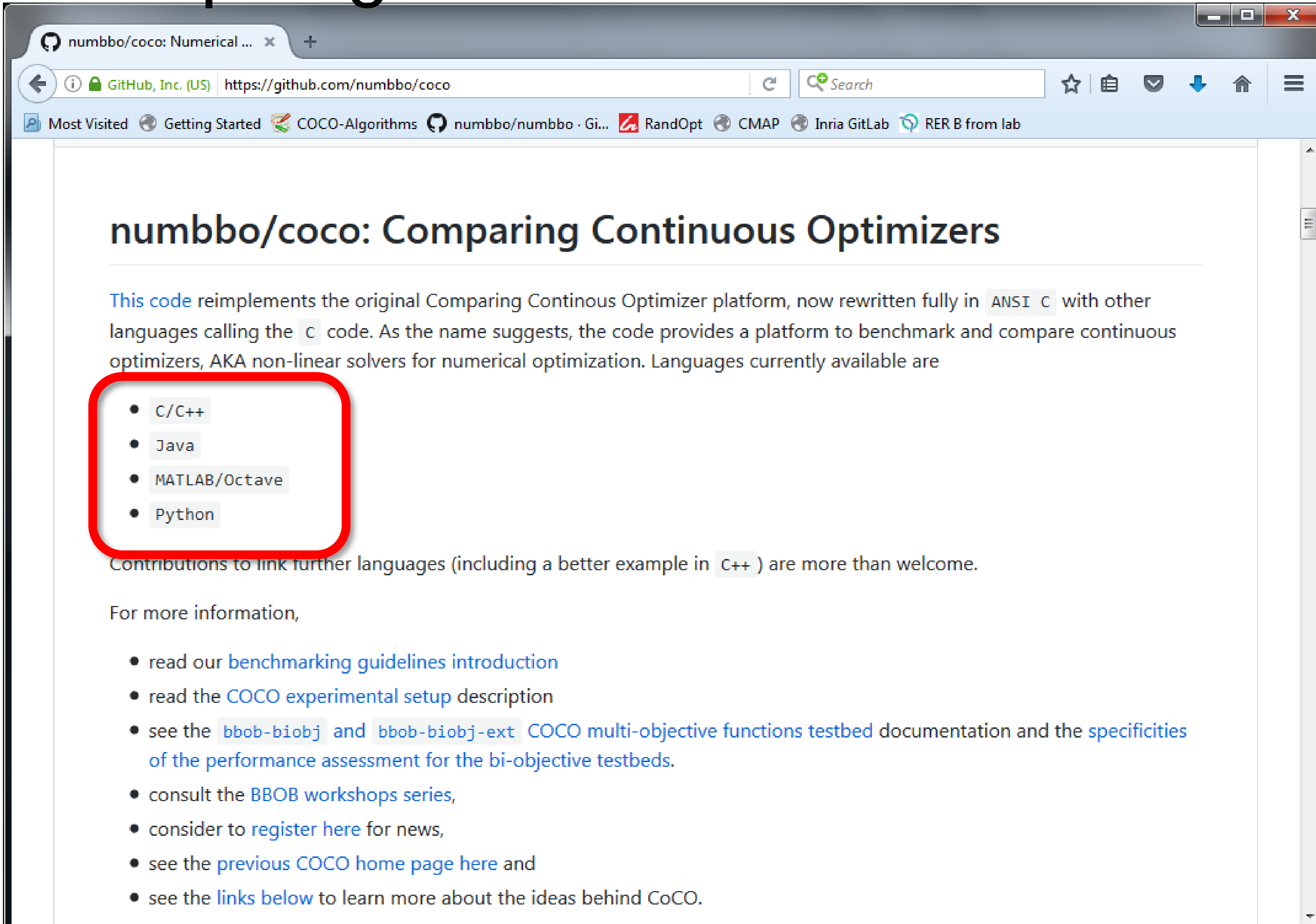
https://github.com/numbbo/coco



https://github.com/numbbo/coco



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

requirements & download

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 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 commands once

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

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

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

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

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

```
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... 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, ... file, usefu... the outpu...

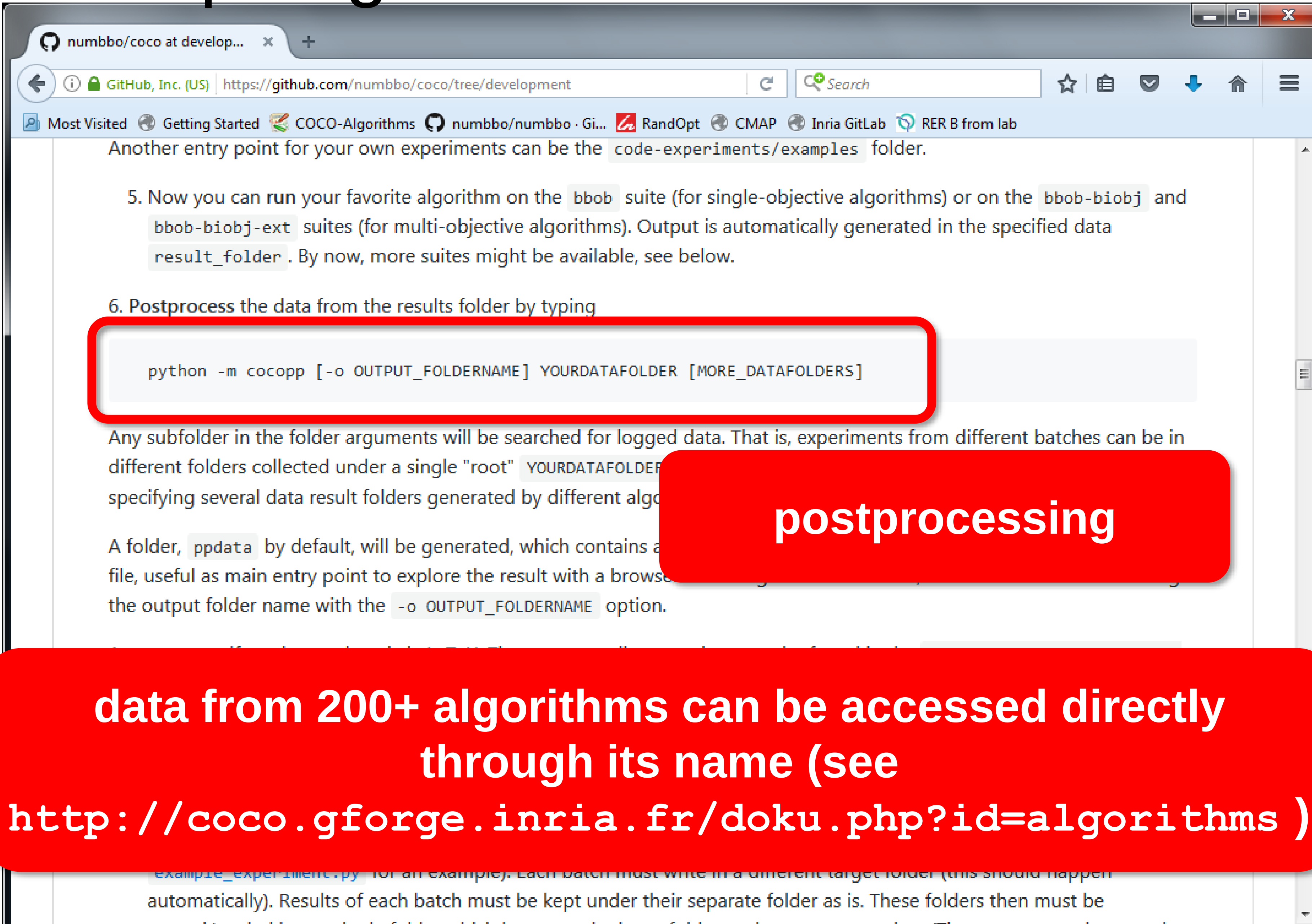
A summa... template... template...

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

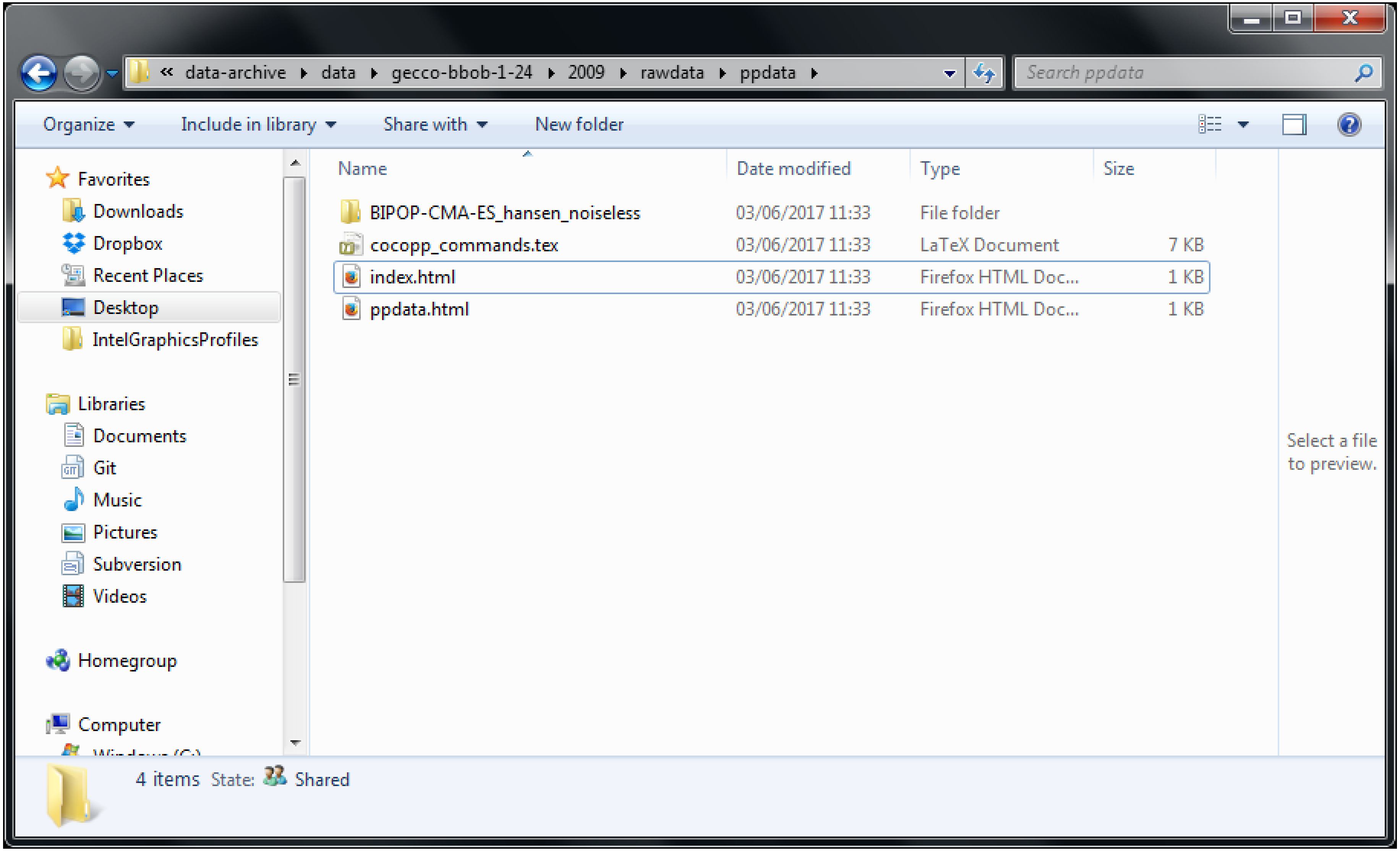
https://github.com/numbbo/coco



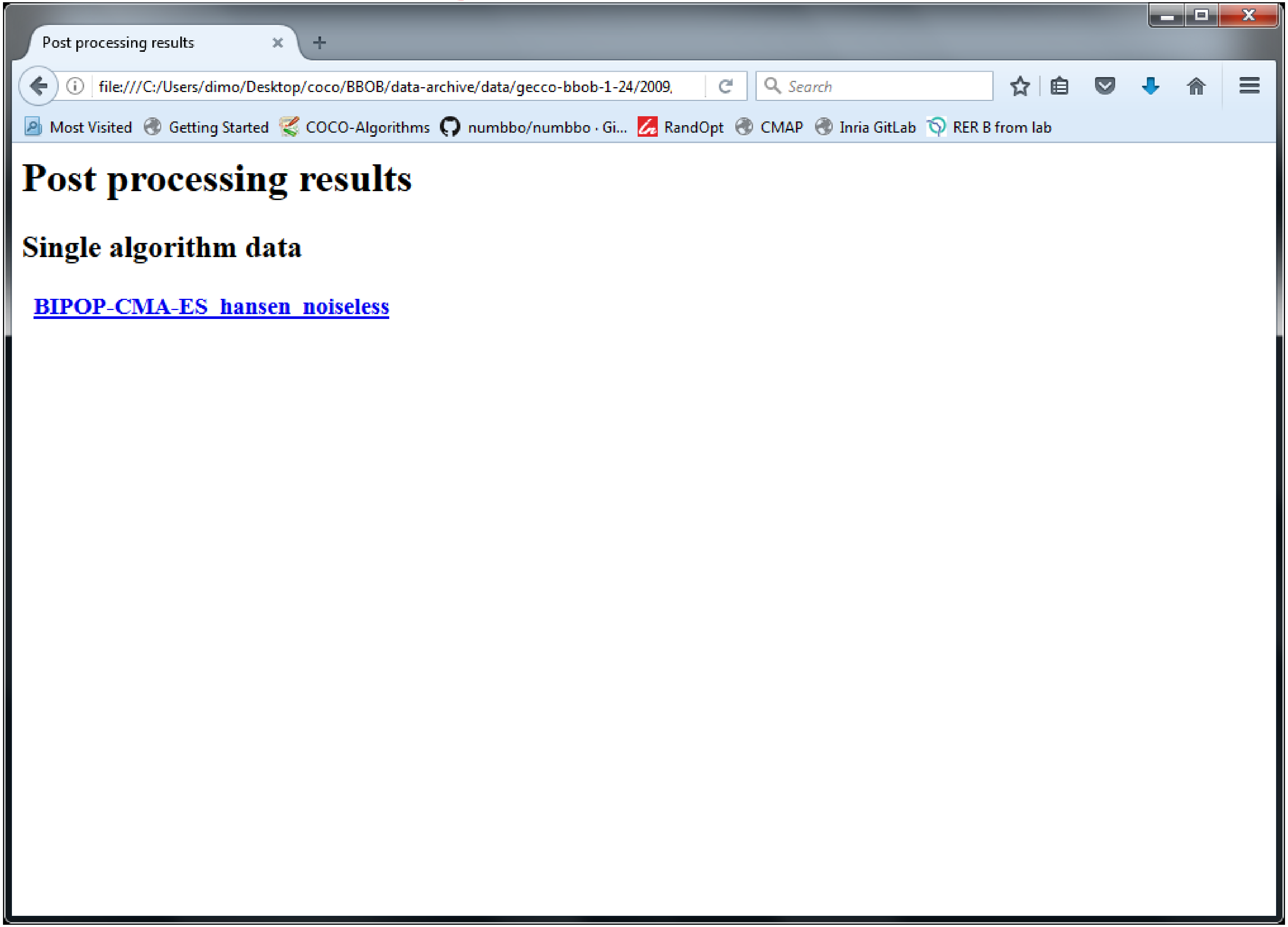
postprocessing

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

Result Folder



Automatically Generated Results



Automatically Generated Results

BIPOP-CMA-ES, templateBBOB...

file:///C:/Users/dimo/Desktop/coco/BBOB/data-archive/data/gecco-bbob-1-24/2009,

Search

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

BIPOP-CMA-ES

Home

Runtime distributions (ECDFs) per function

Runtime distributions (ECDFs) summary and function groups

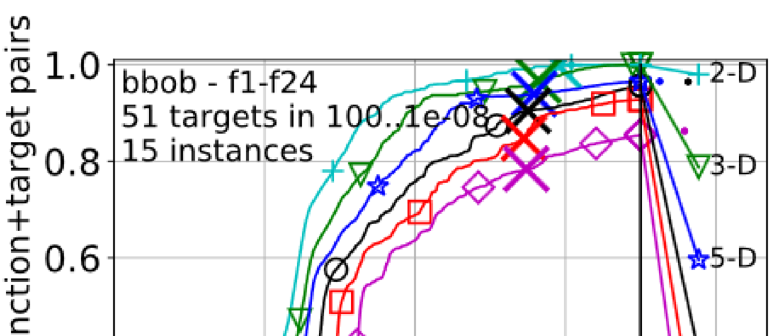
Scaling with dimension for selected targets

Tables for selected targets

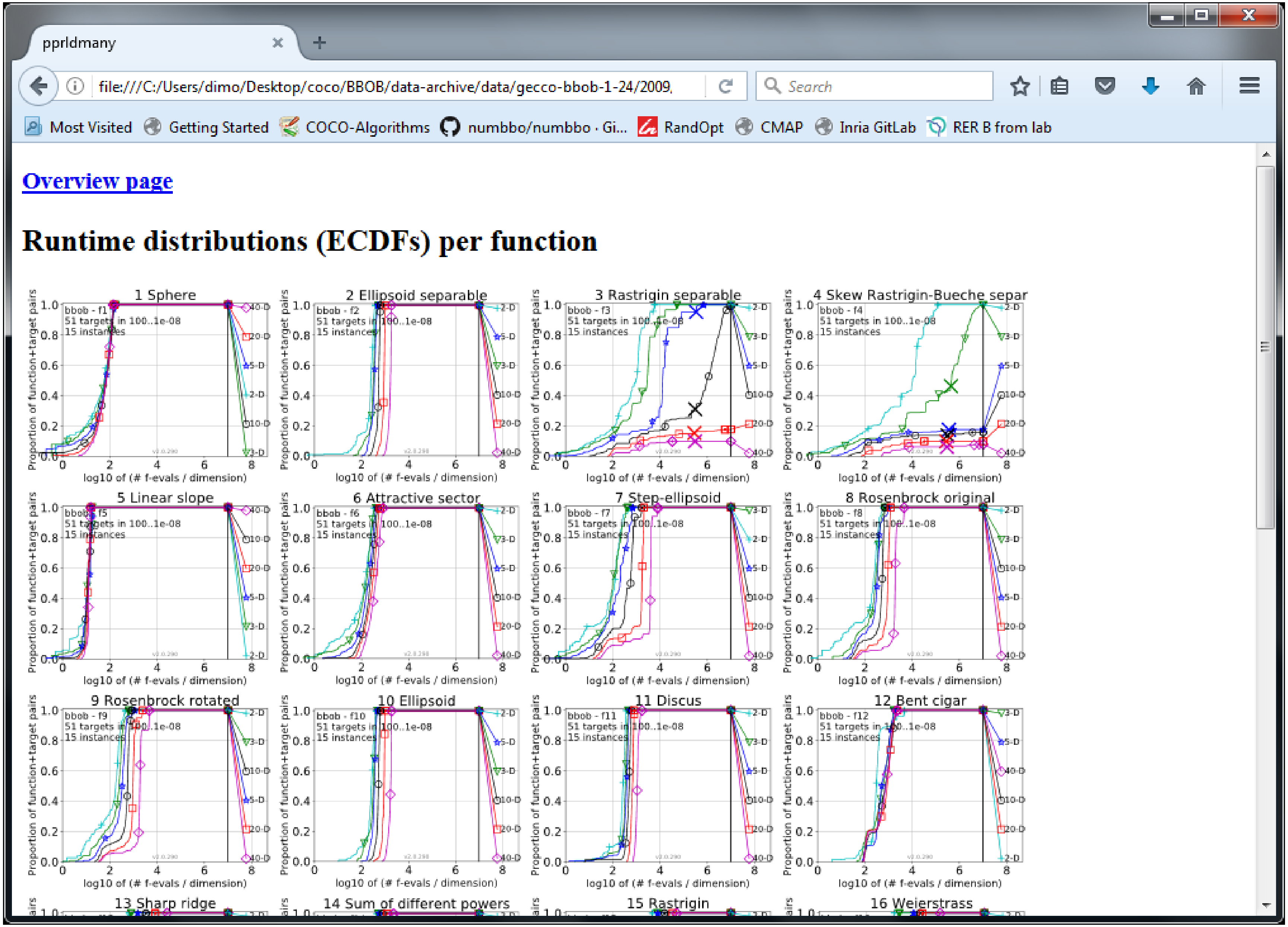
Runtime distribution for selected targets and f-distributions

Runtime loss ratios

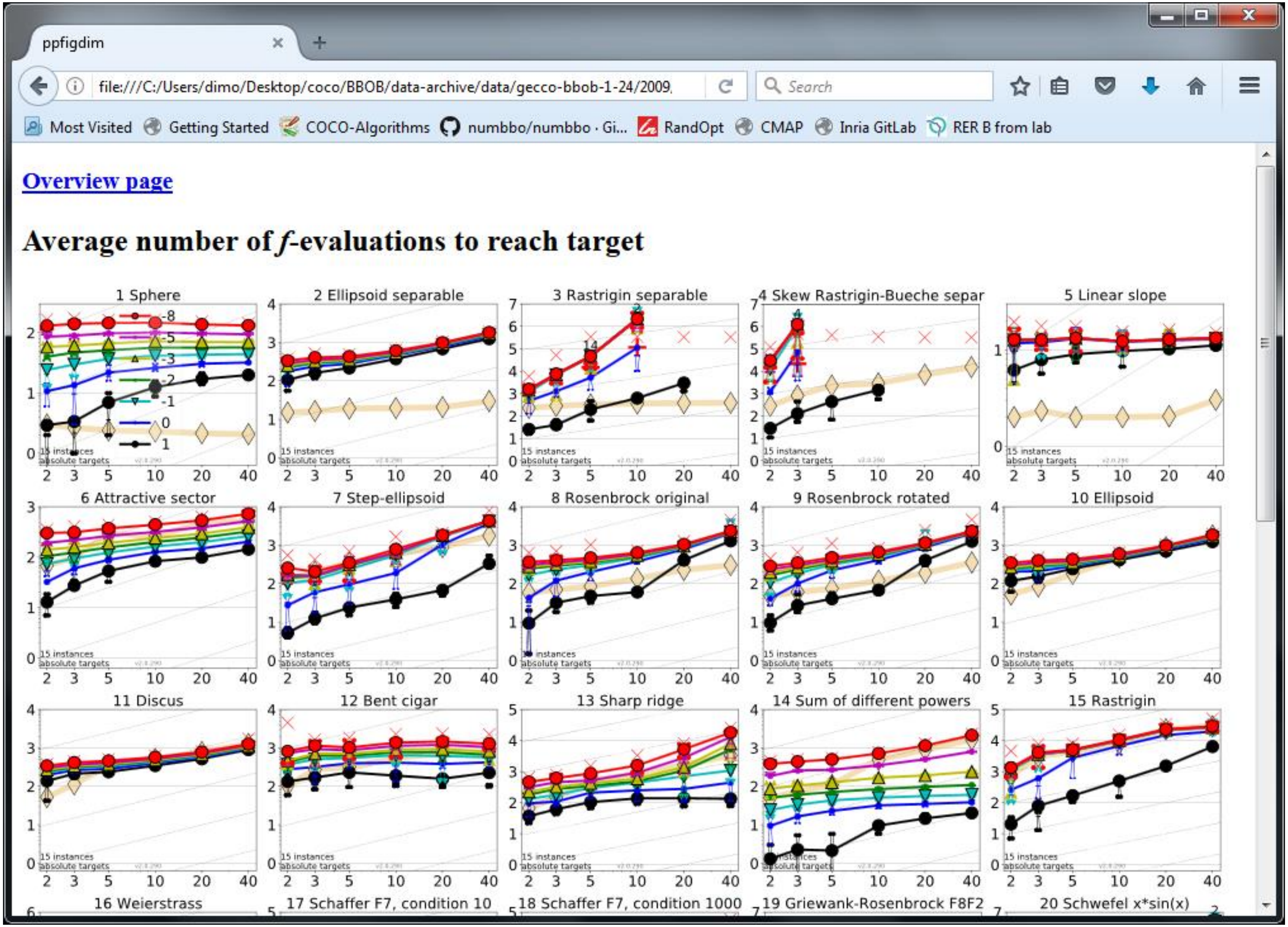
Runtime distributions (ECDFs) over all targets



Automatically Generated Results



Automatically Generated Results



so far:

data for 300+ algorithm variants

(some of which on noisy or multiobjective test functions)

143 workshop papers

by 109 authors from 28 countries

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	220+ algo data sets
bbob-noisy	30 noisy fcts	40+ algo data sets
bbob-biobj	55 bi-objective fcts	30+ algo data sets
bbob-largescale	24 noiseless fcts	11 algo data sets
bbob-mixint	24 mixed integer fcts	
bbob-biobj-mixint	92 mixed integer fcts	

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:

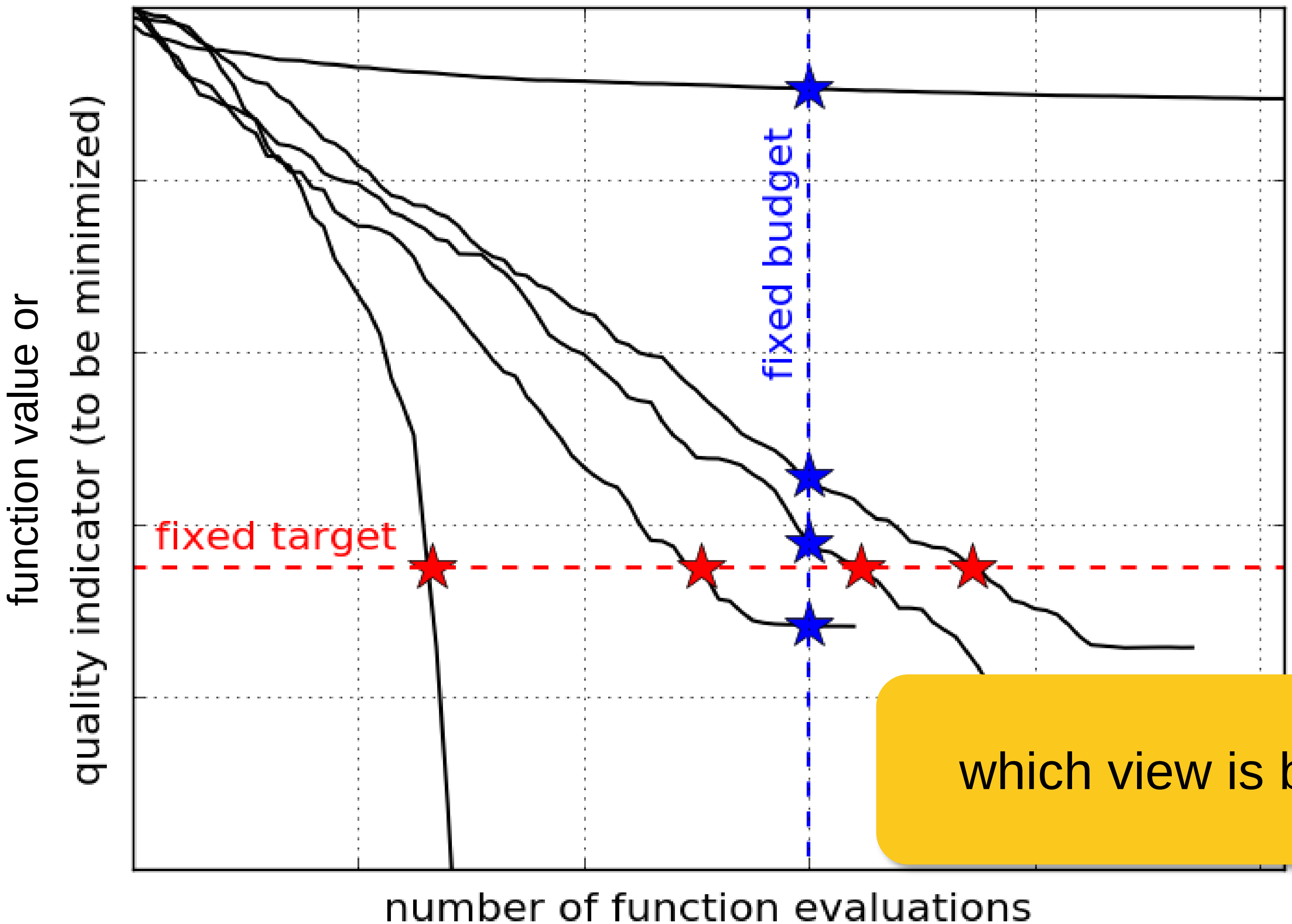
- Find solution with small(est possible) function/indicator value

- 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...



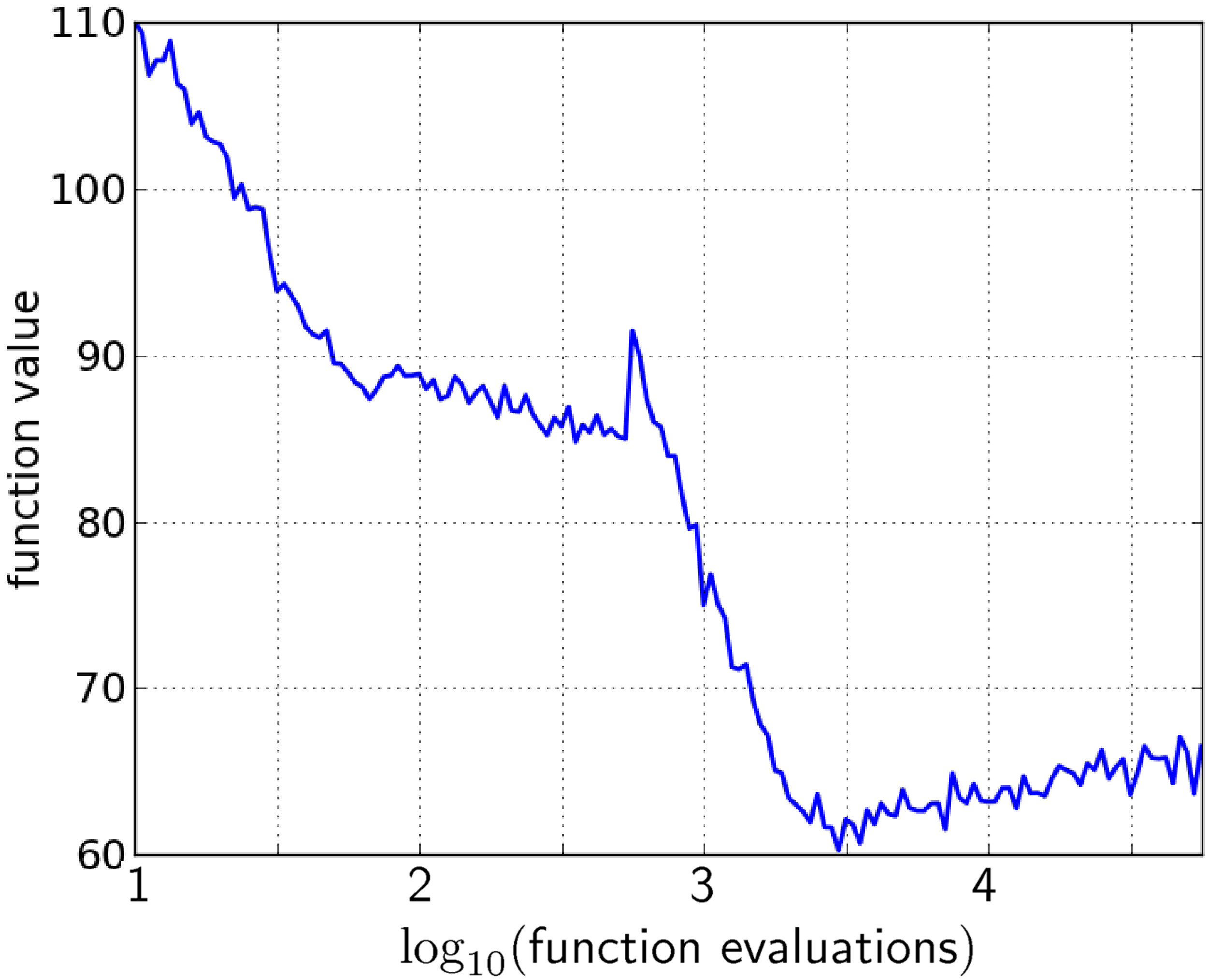
which view is better?

ECDF:

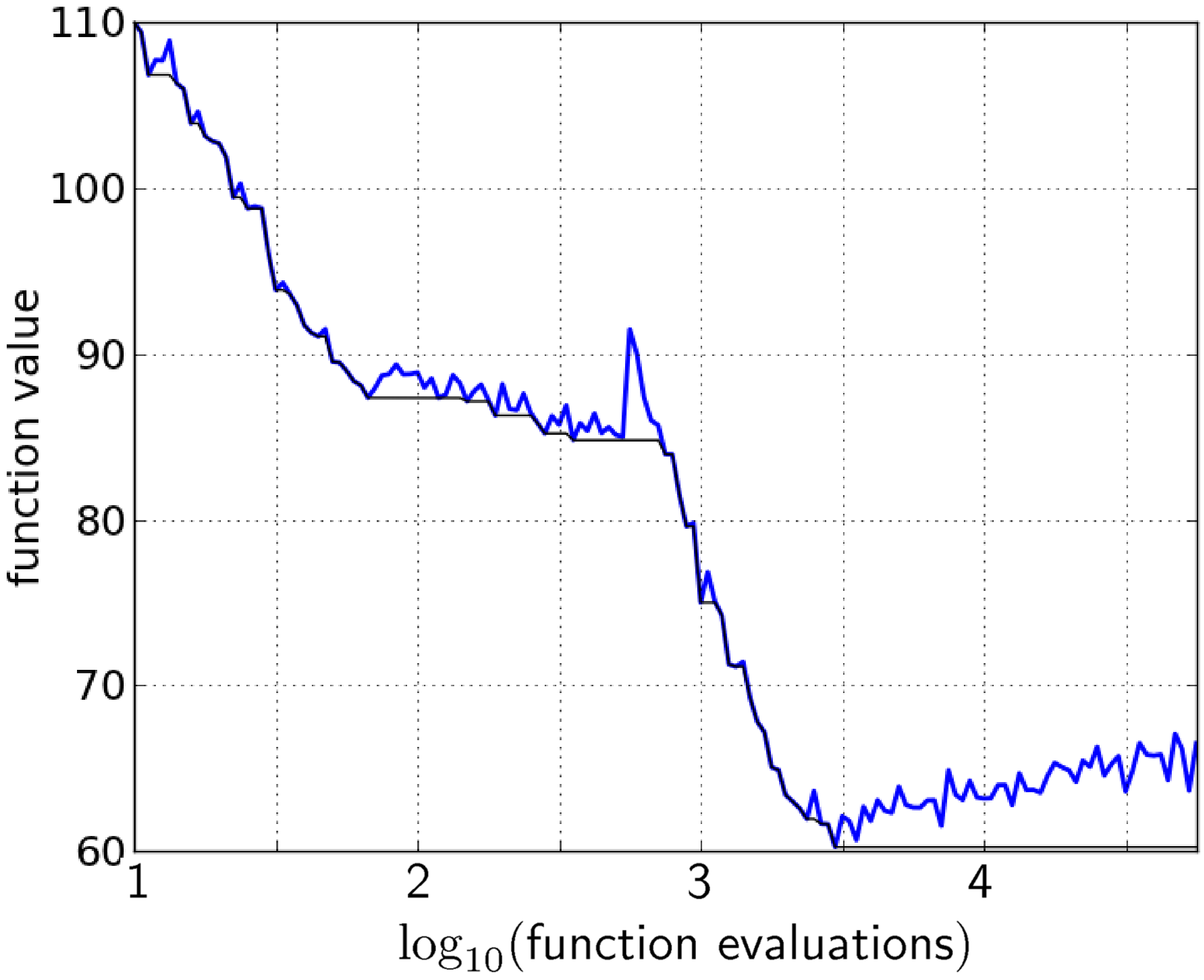
Empirical Cumulative Distribution Function of the Runtime

[aka data profile]

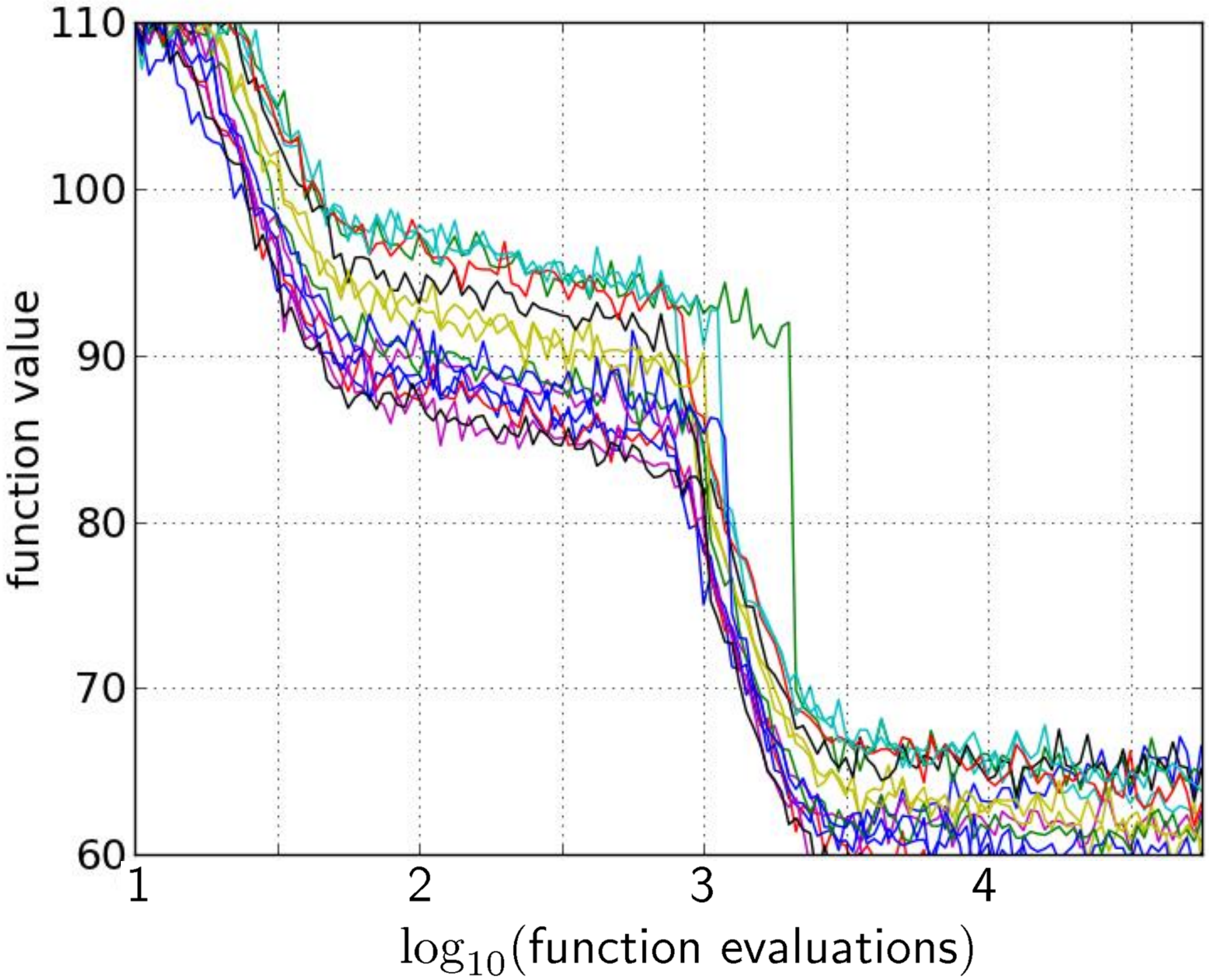
A Convergence Graph



First Hitting Time is Monotonous

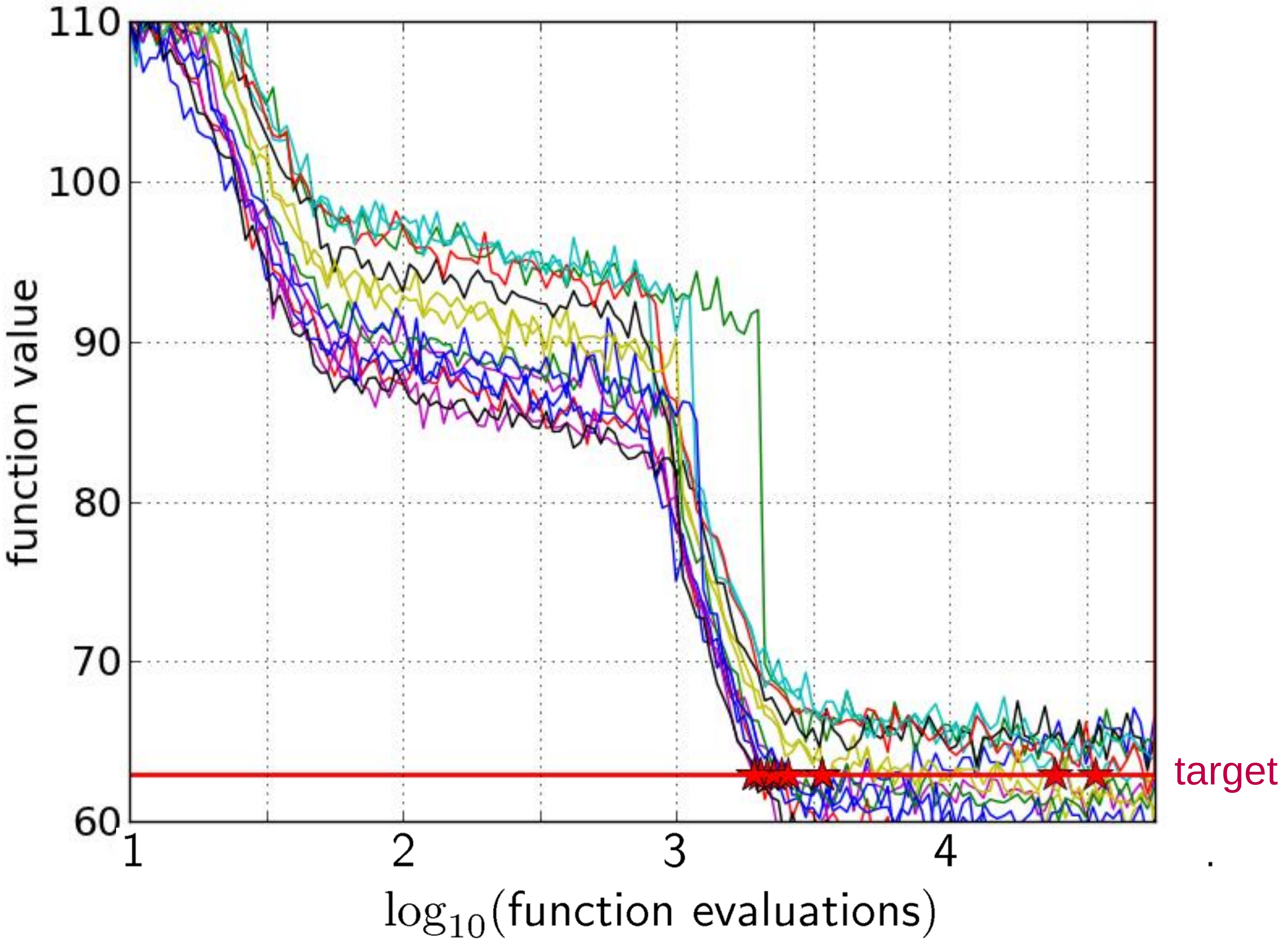


15 Runs

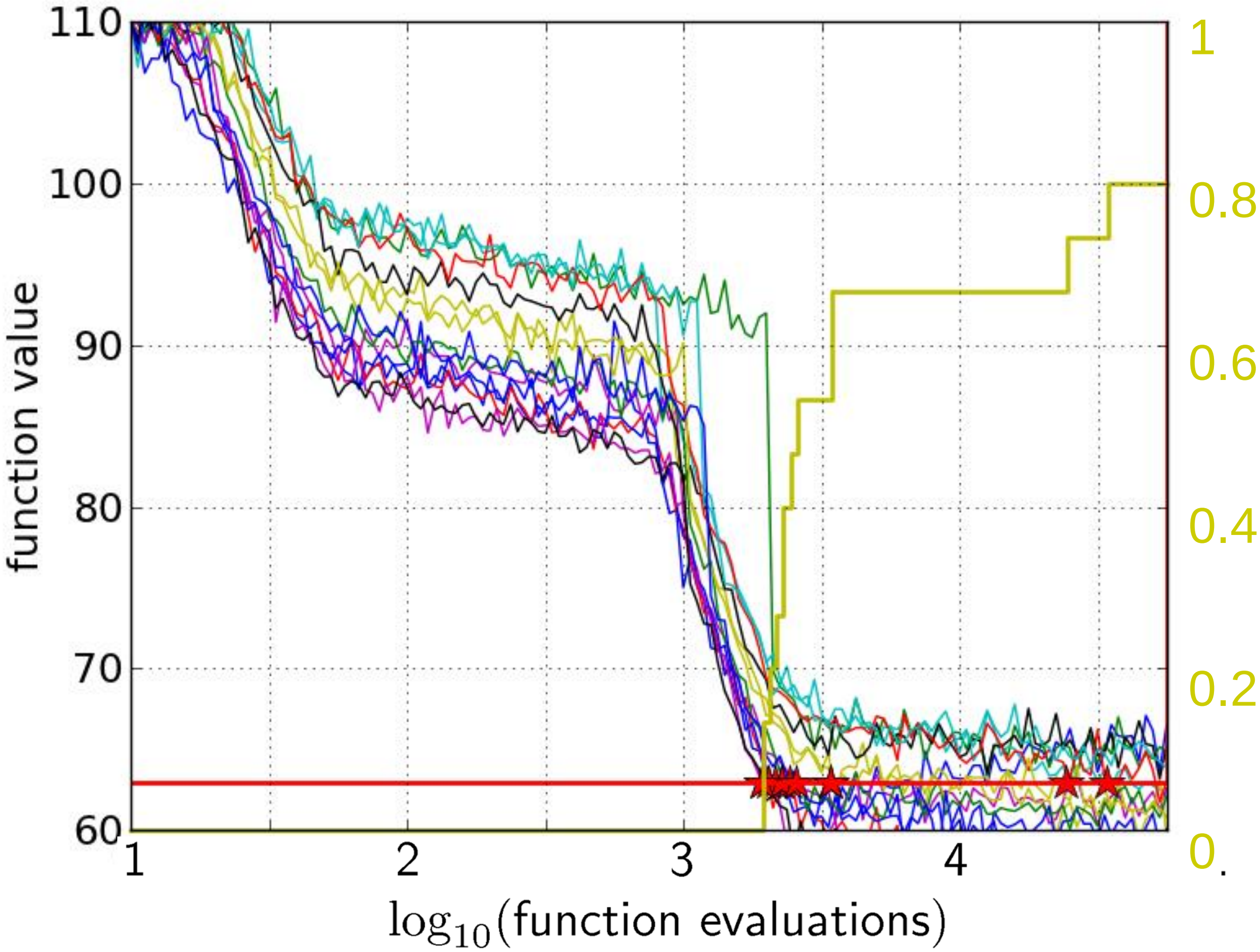


110
100
90
80
70
60
function value
1
2
3
4
$\log_{10}$(function evaluations)

15 Runs ≤ 15 Runtime Data Points

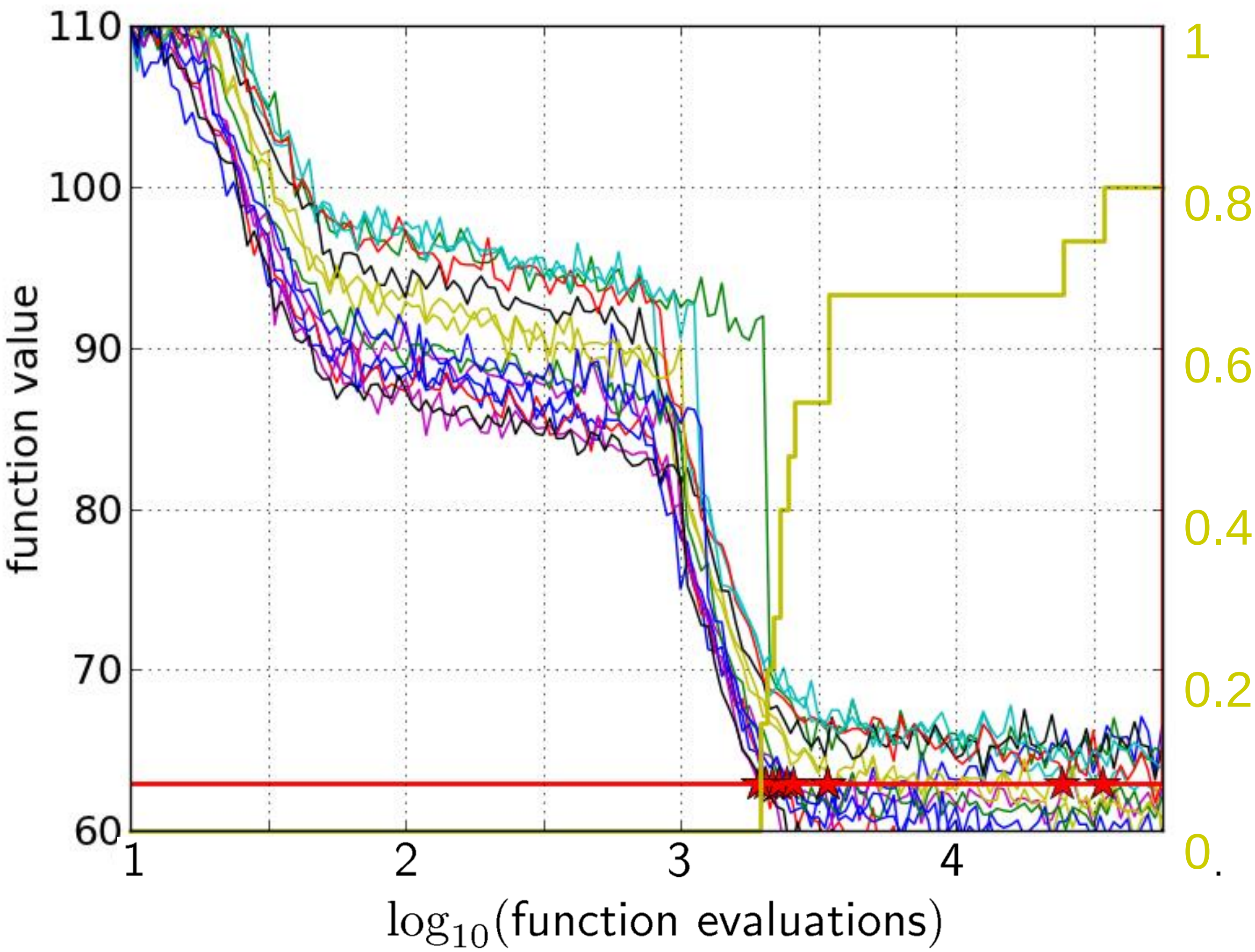


Empirical Cumulative Distribution



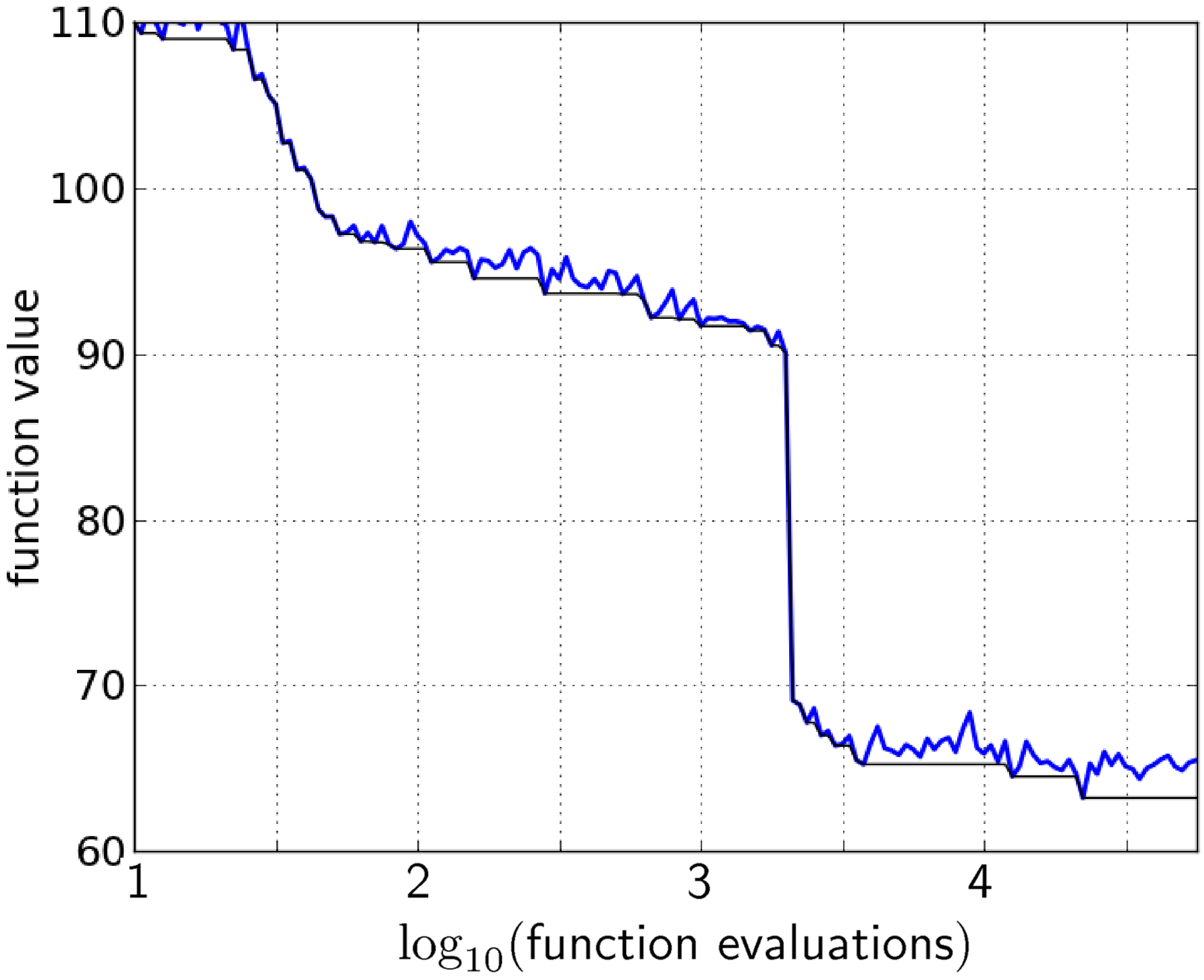
- the ECDF of run lengths to reach the target
- has for each data point a vertical step of constant size
- displays for each x-value (budget) the count of observations to the left (first hitting times)

Empirical Cumulative Distribution



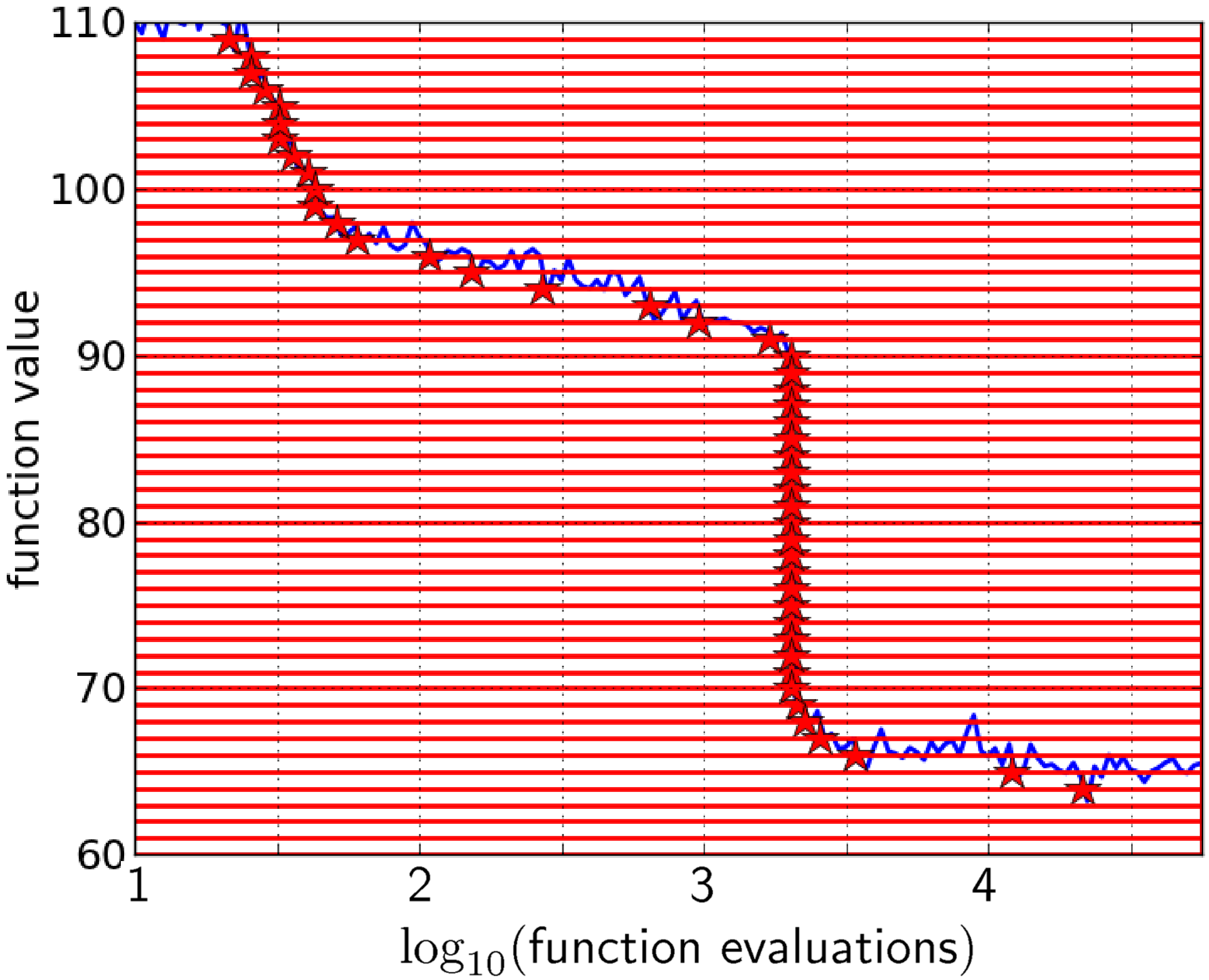
- interpretations possible:
- 80% of the runs reached the target
- e.g. 60% of the runs need between 2000 and 4000 evaluations

Reconstructing A Single Run



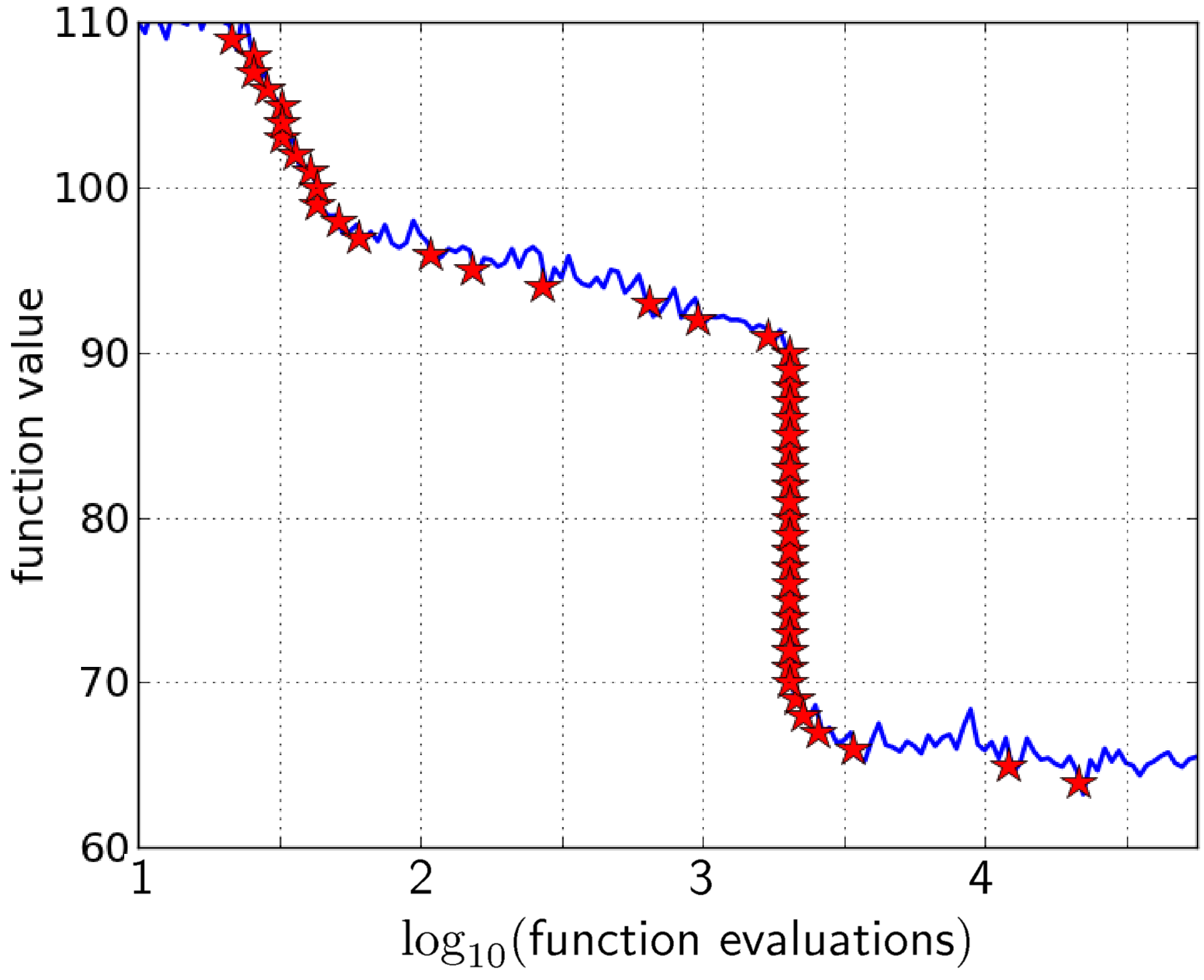
.

Reconstructing A Single Run

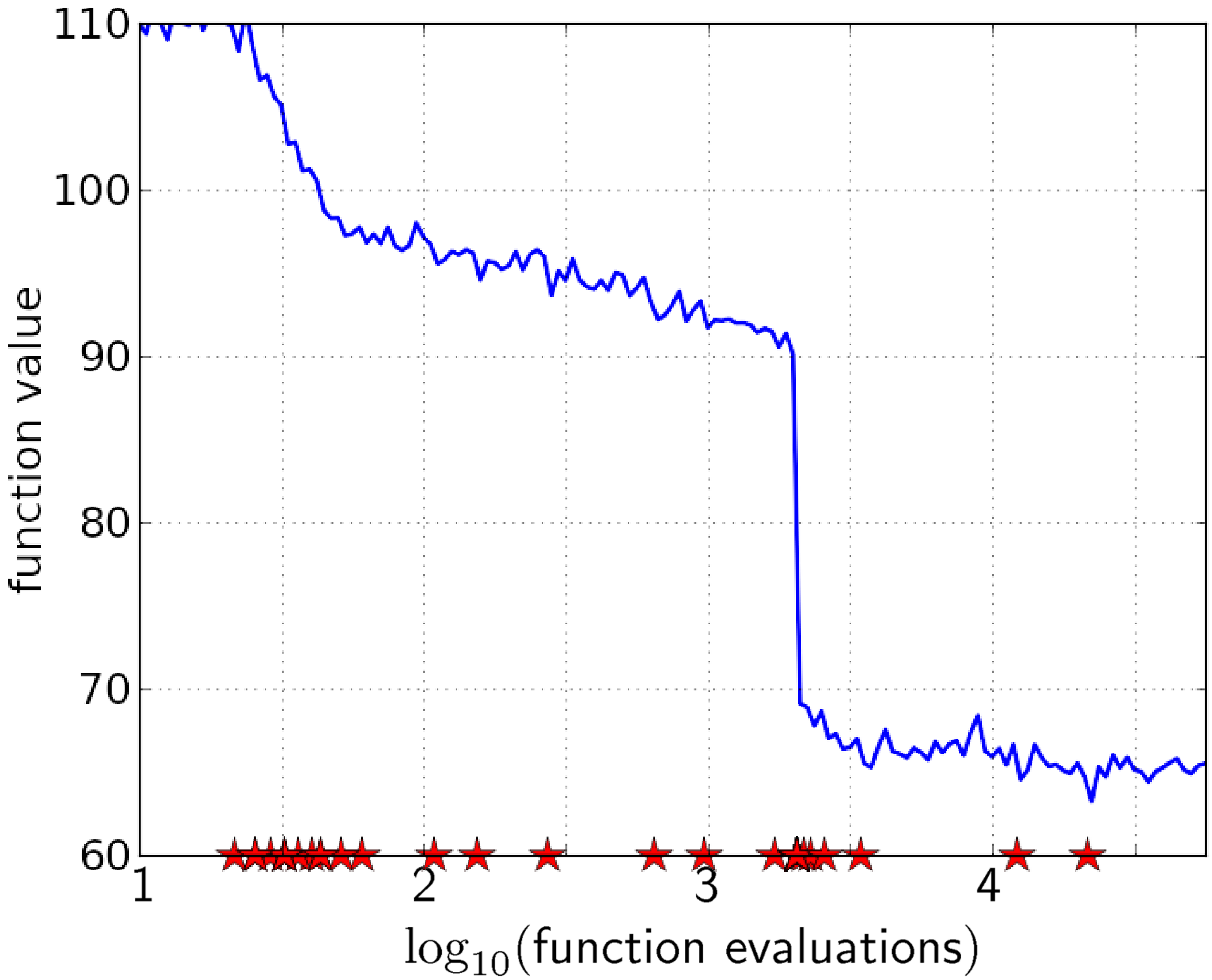


50 equally spaced targets

Reconstructing A Single Run

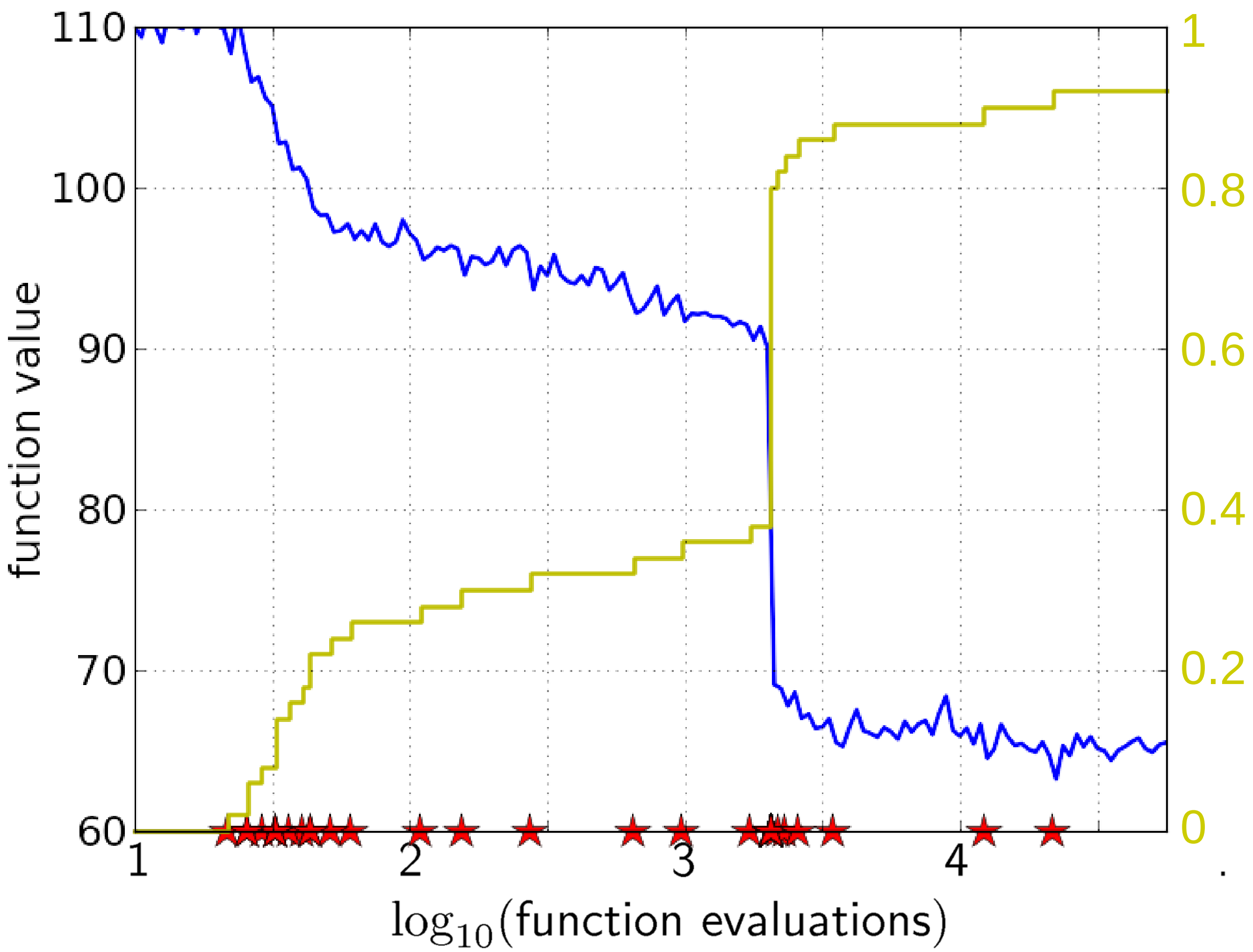


Reconstructing A Single Run



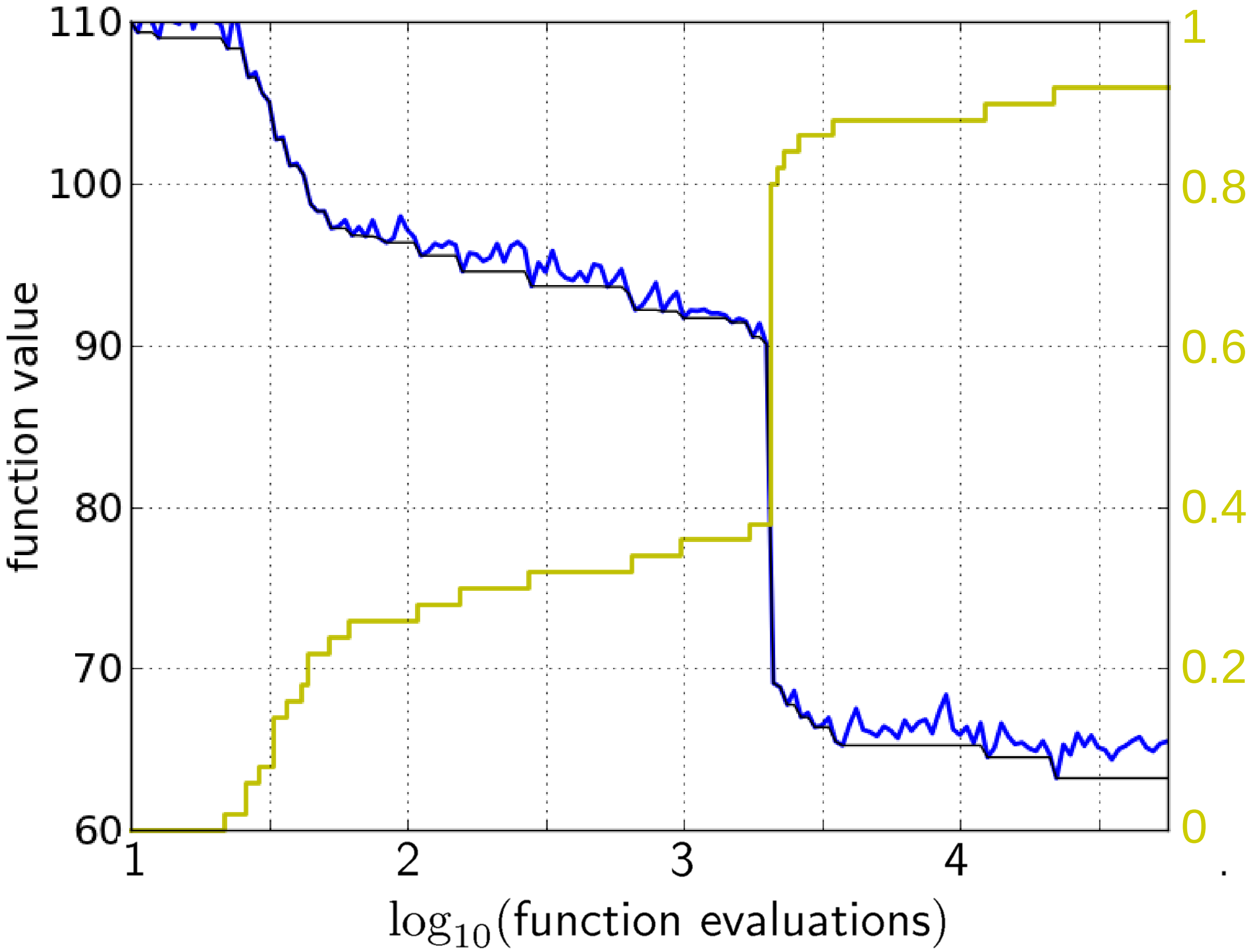
.

Reconstructing A Single Run



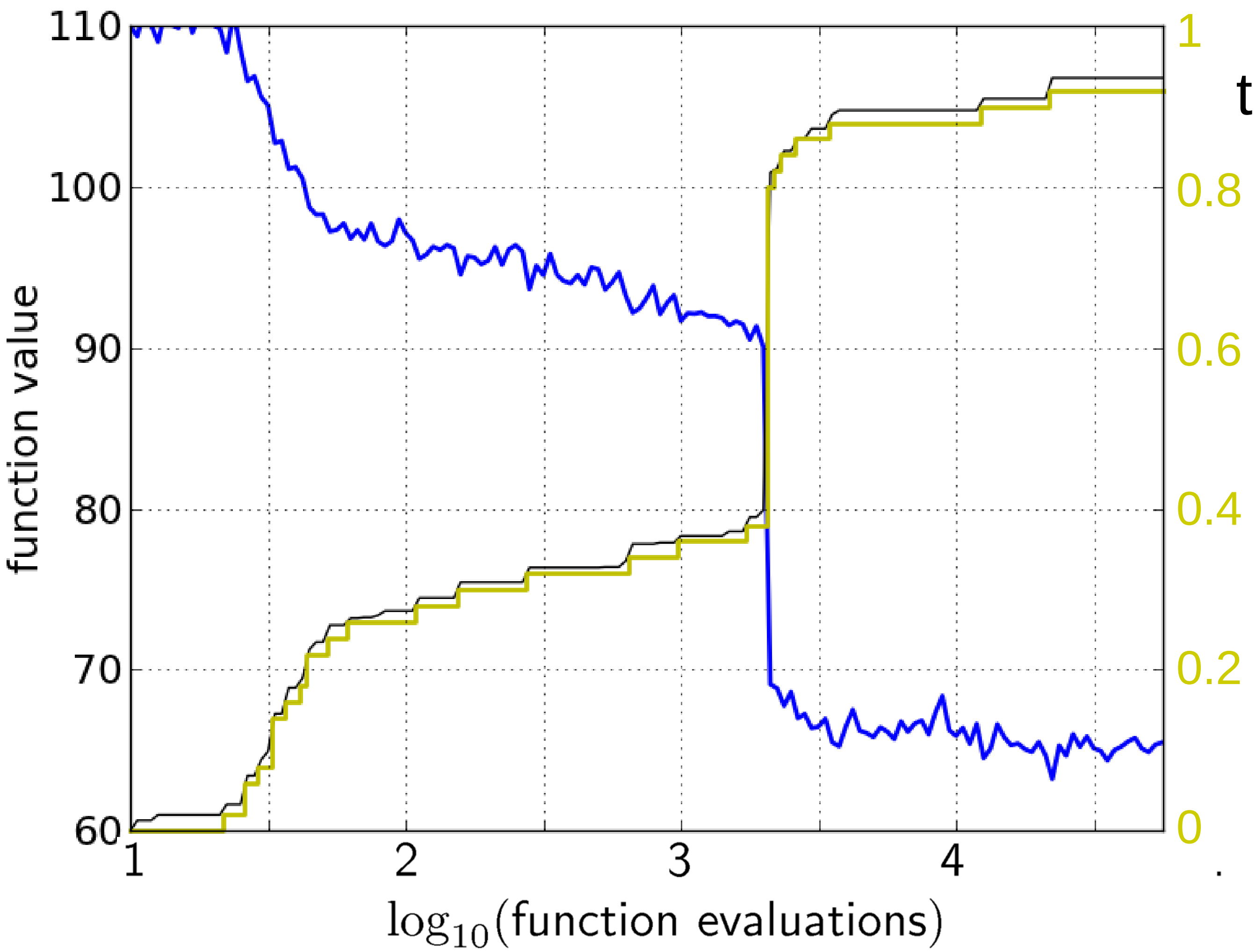
the empirical CDF makes a step for each star, is monotonous and displays for each budget the fraction of targets achieved within the budget

Reconstructing A Single Run



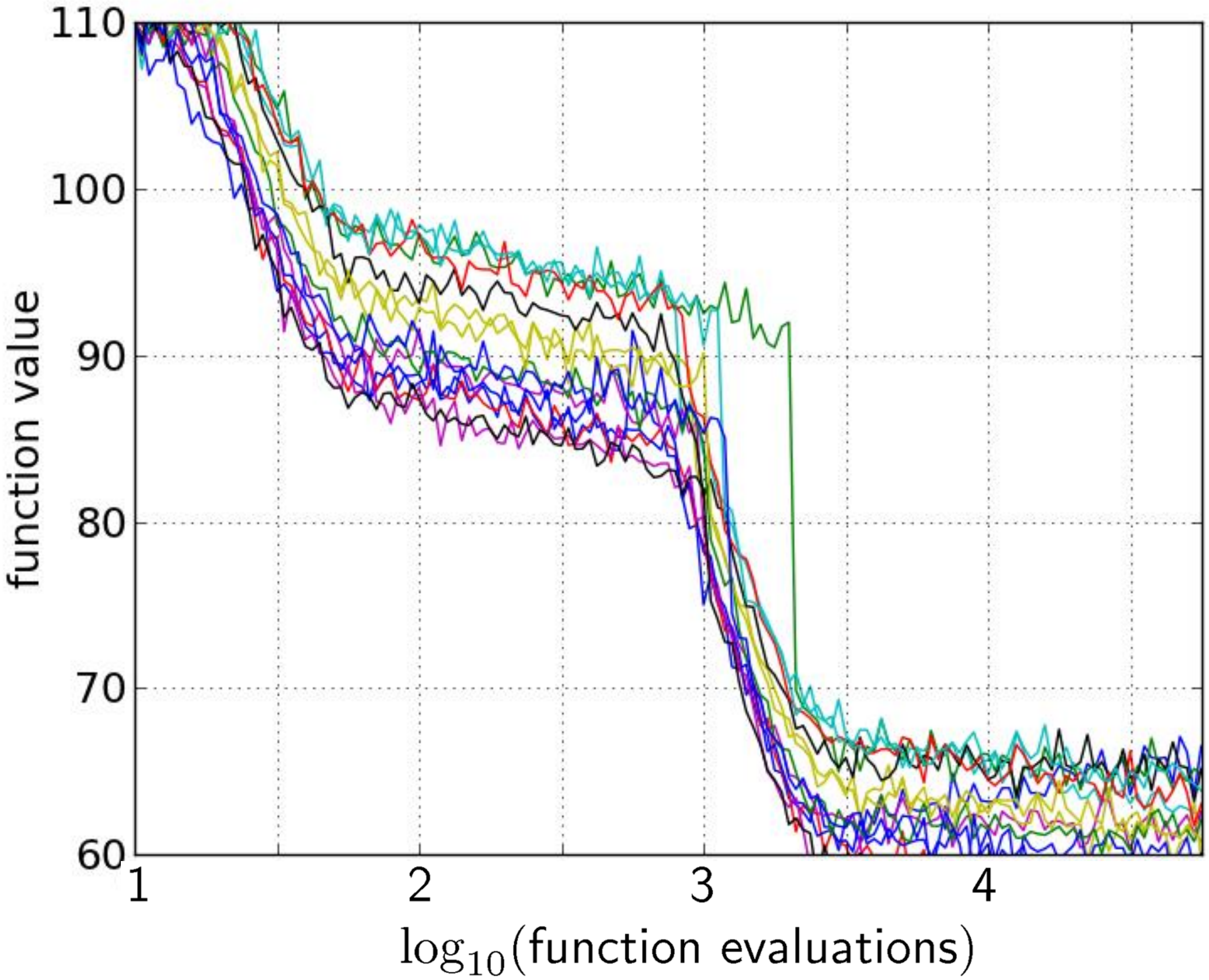
the ECDF recovers the monotonous graph, discretized and flipped

Reconstructing A Single Run



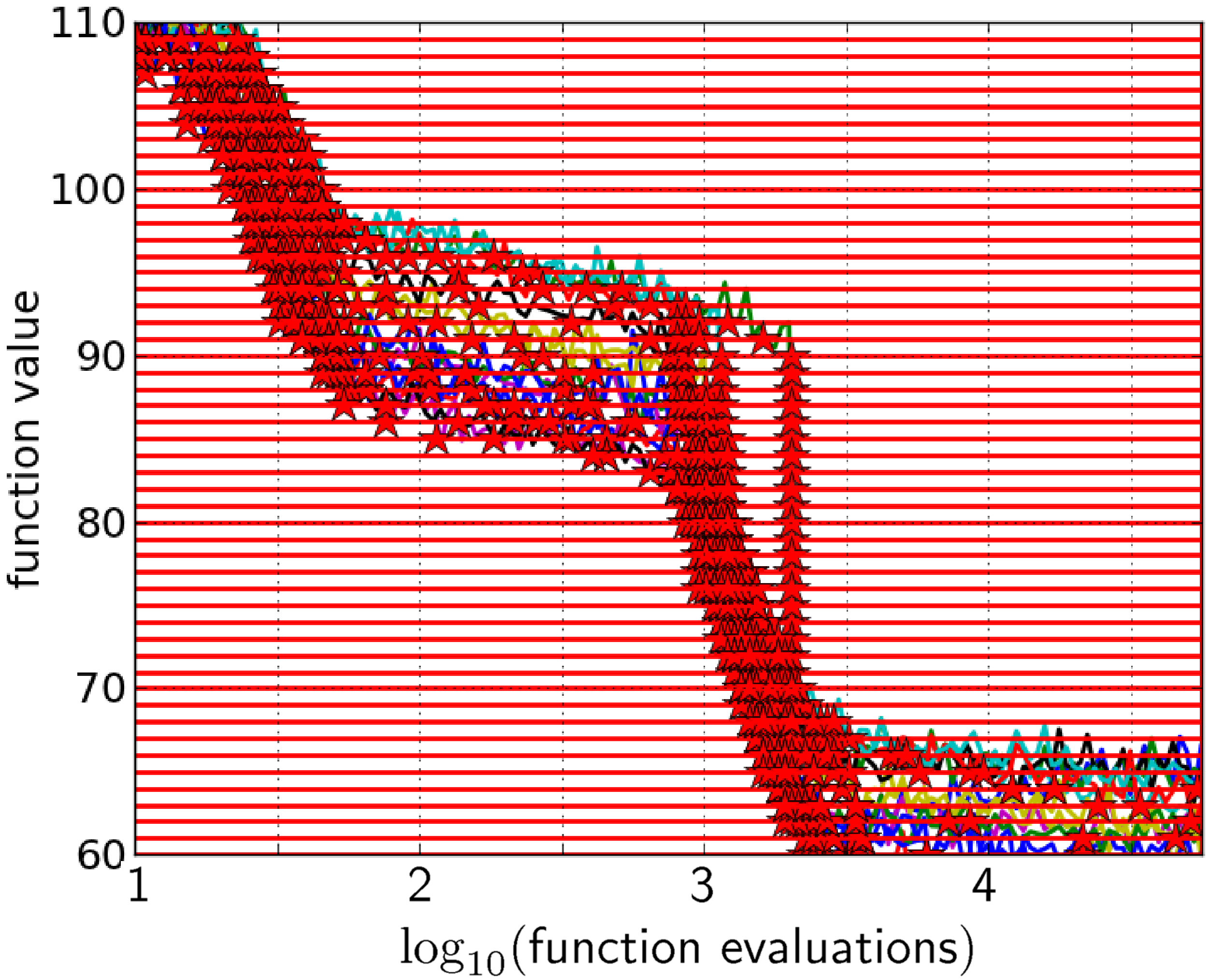
the ECDF recovers the monotonous graph, discretized and flipped

Aggregation



15 runs

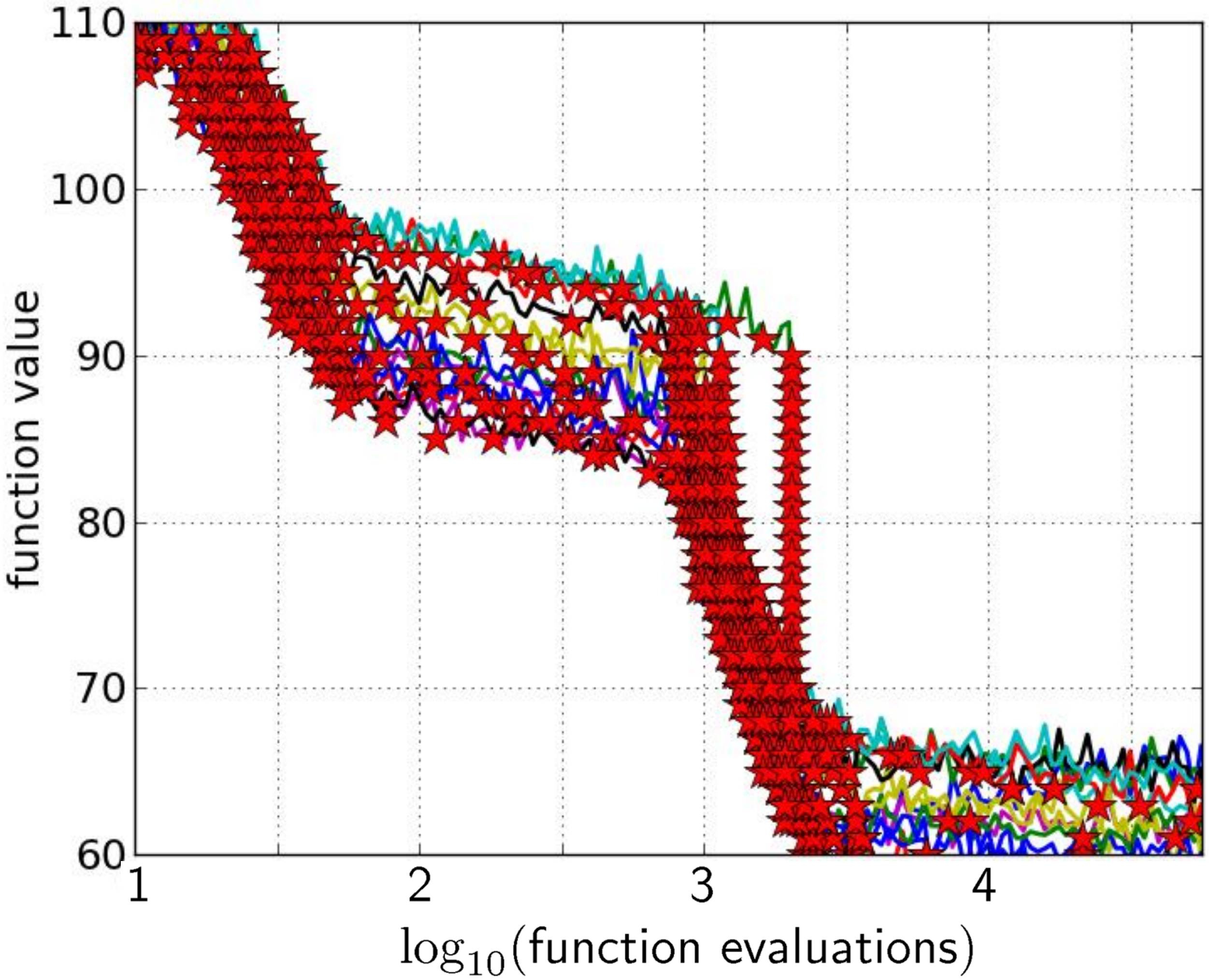
Aggregation



15 runs

50 targets

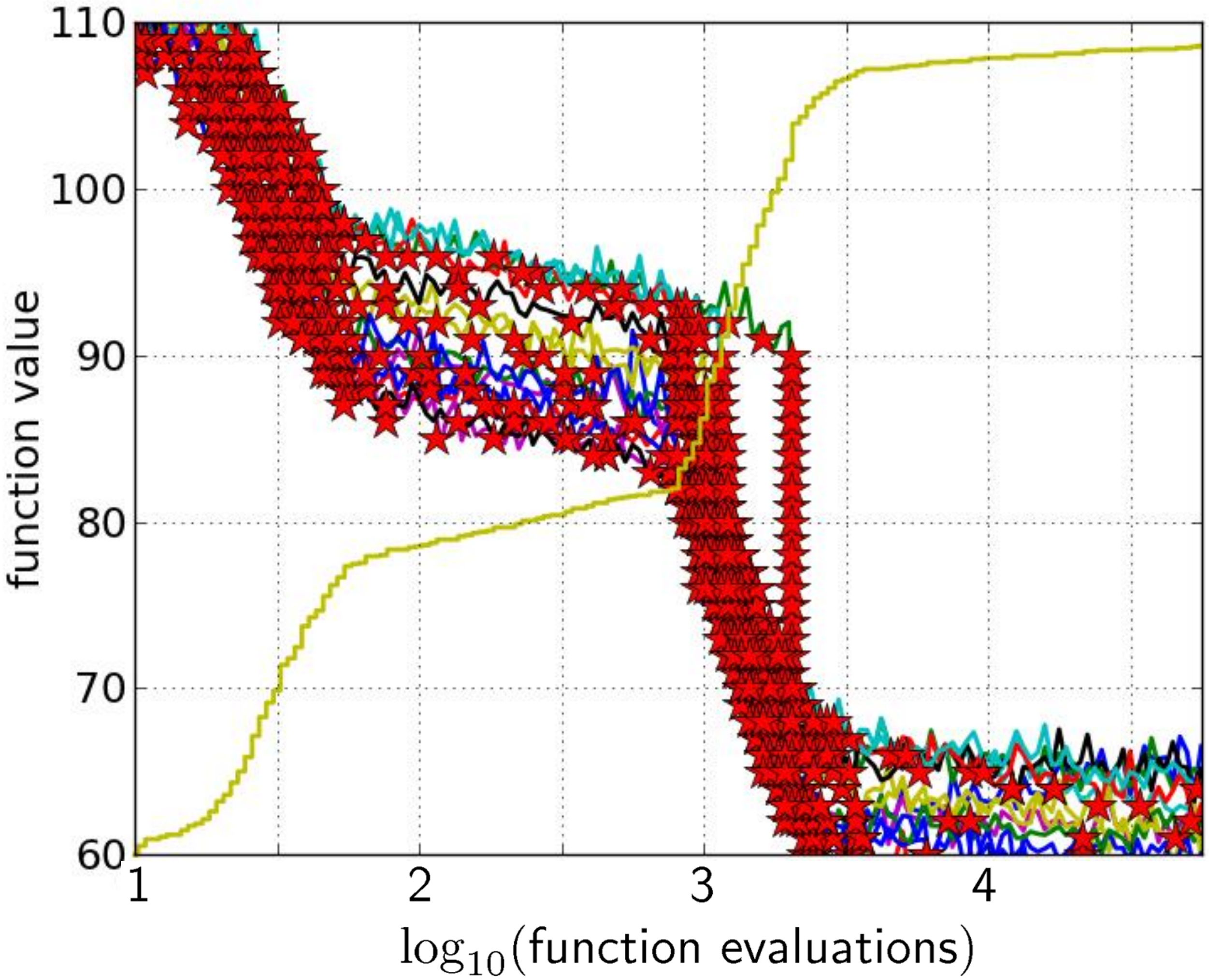
Aggregation



15 runs

50 targets

Aggregation

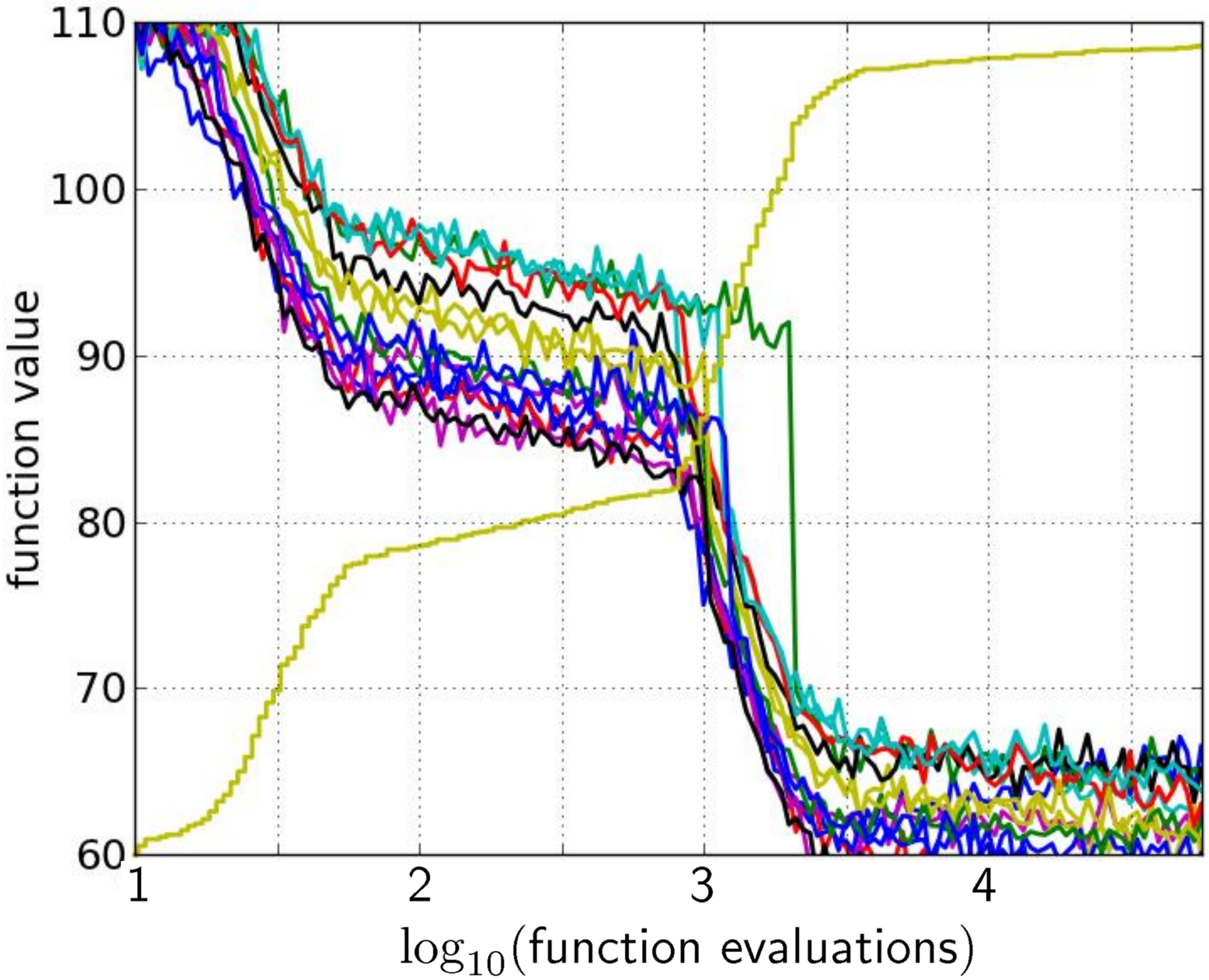


15 runs

50 targets

ECDF with 750 steps

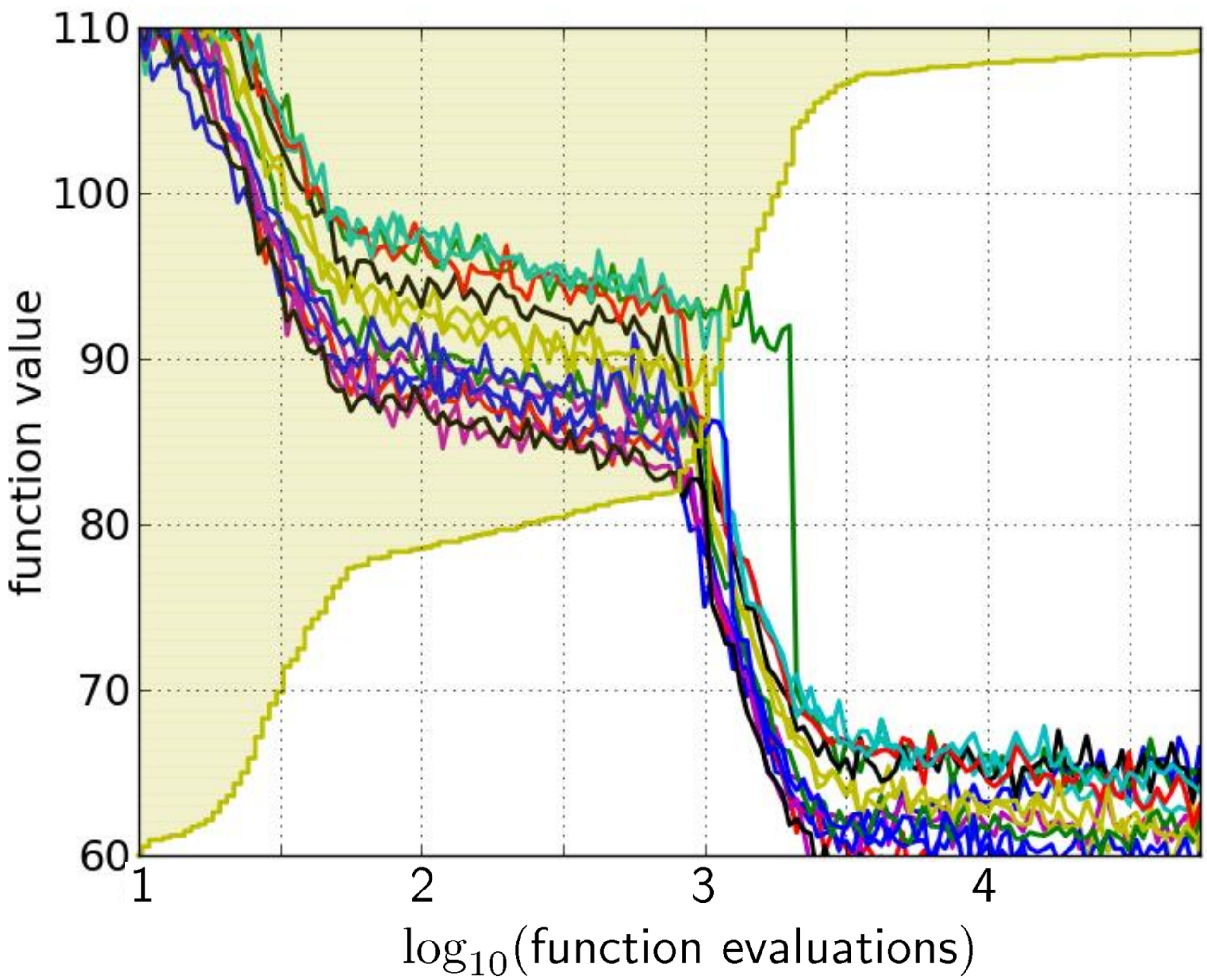
Aggregation



50 targets from 15 runs

...integrated in a single graph

Interpretation



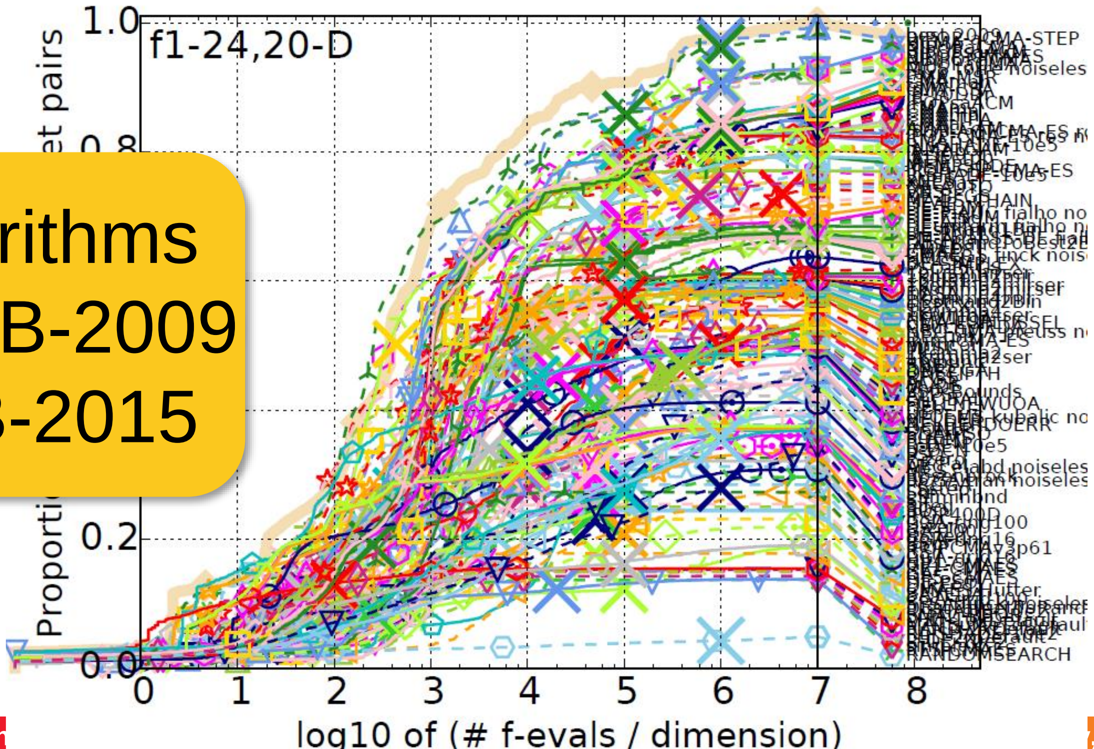
50 targets from
15 runs
integrated in a
single graph

area over the
ECDF curve
=
average log
runtime
(or geometric avg.
runtime) over all
targets (difficult and
easy) and all runs

Worth to Note: ECDFs in COCO

In COCO, ECDF graphs

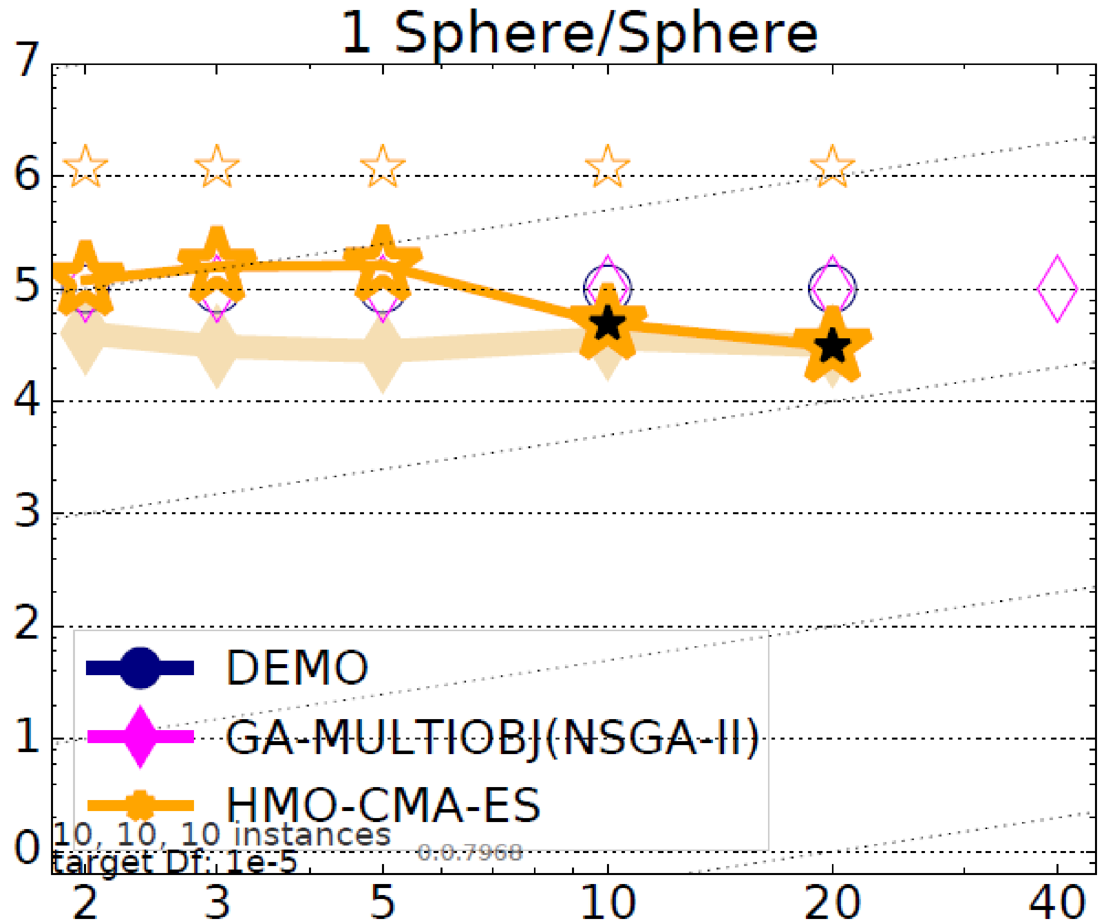
- never aggregate over dimension
  - but often over targets and functions
- can show data of more than 1 algorithm at a time



150 algorithms
from BBOB-2009
till BBOB-2015

Another Interesting Plot...

...compares average runtimes over several algorithms

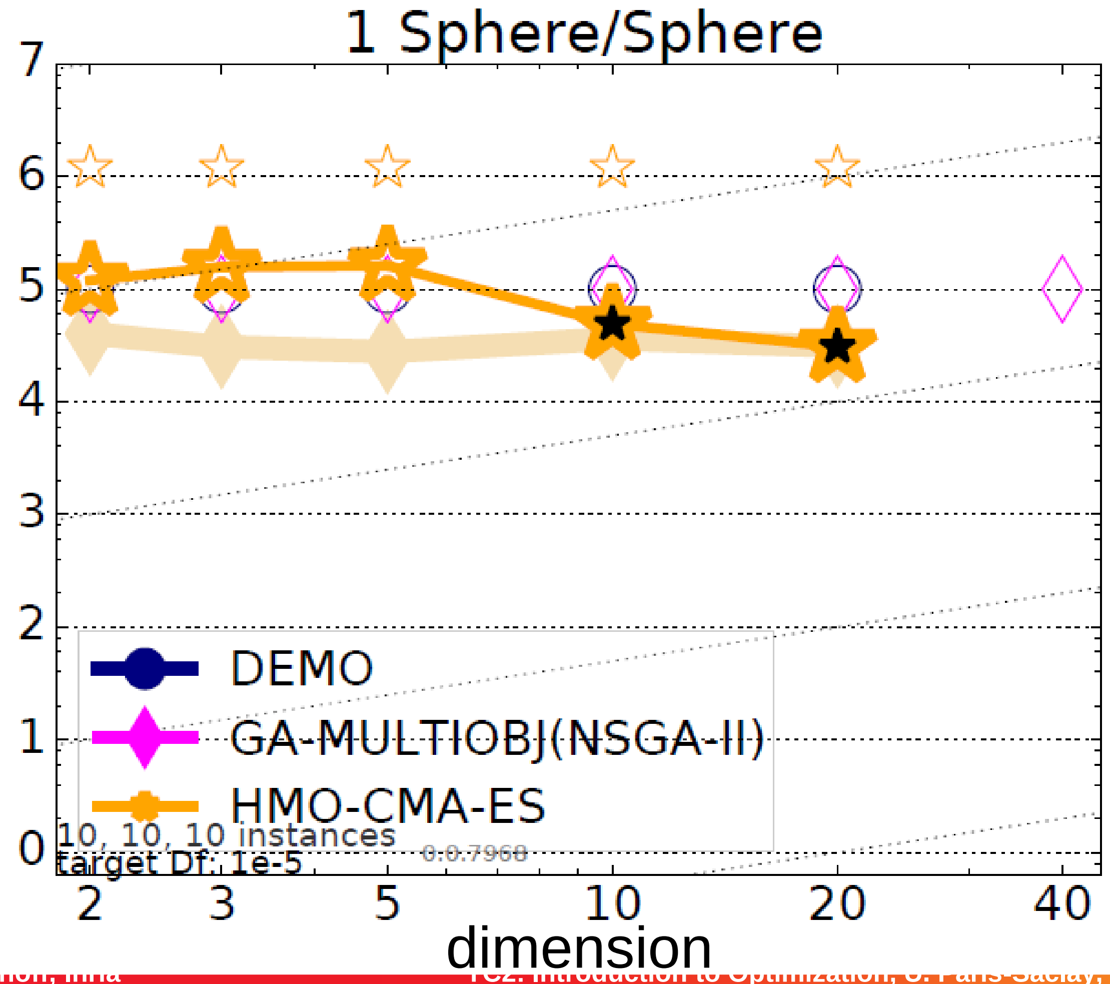


Another Interesting Plot...

...compares average runtimes over several algorithms

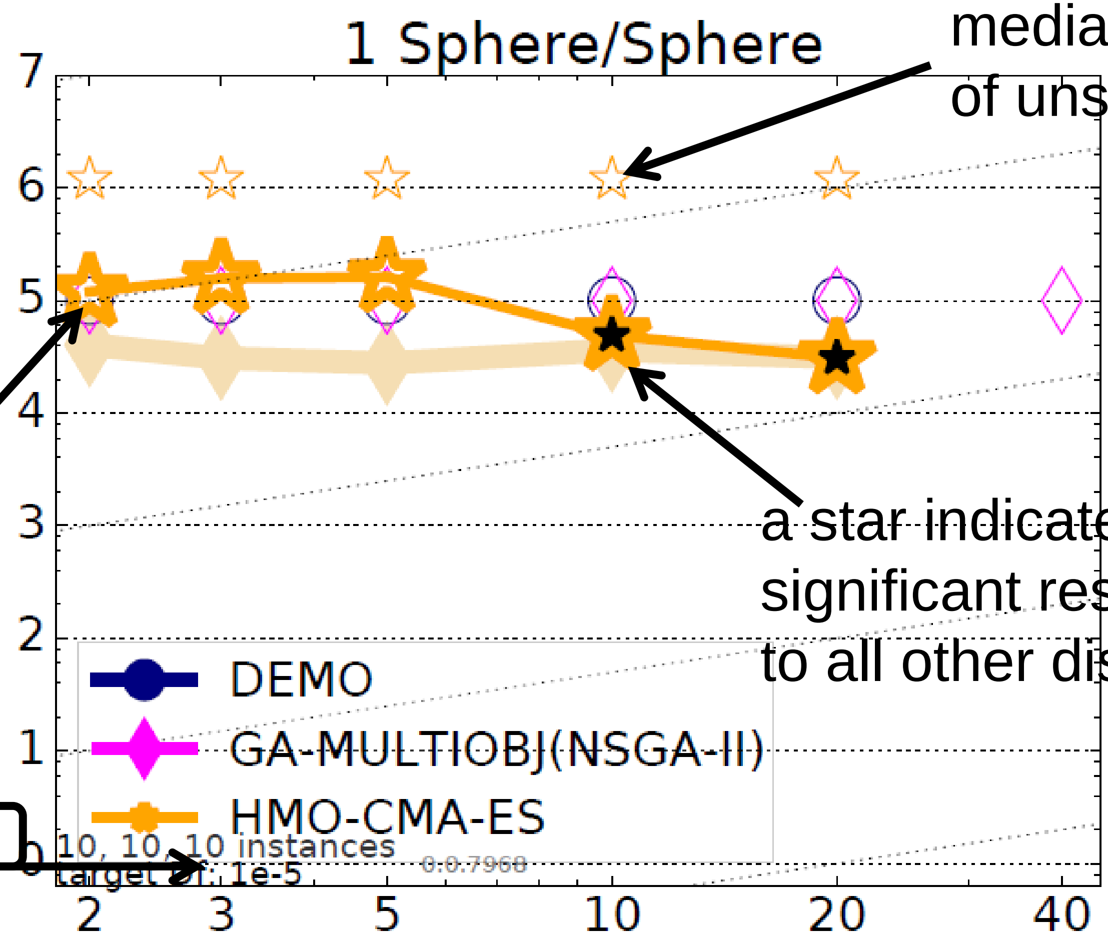
y axis shows runtime in log-scale:

5 "=" 1e5*DIM



Another Interesting Plot...

...compares average runtimes over several algorithms



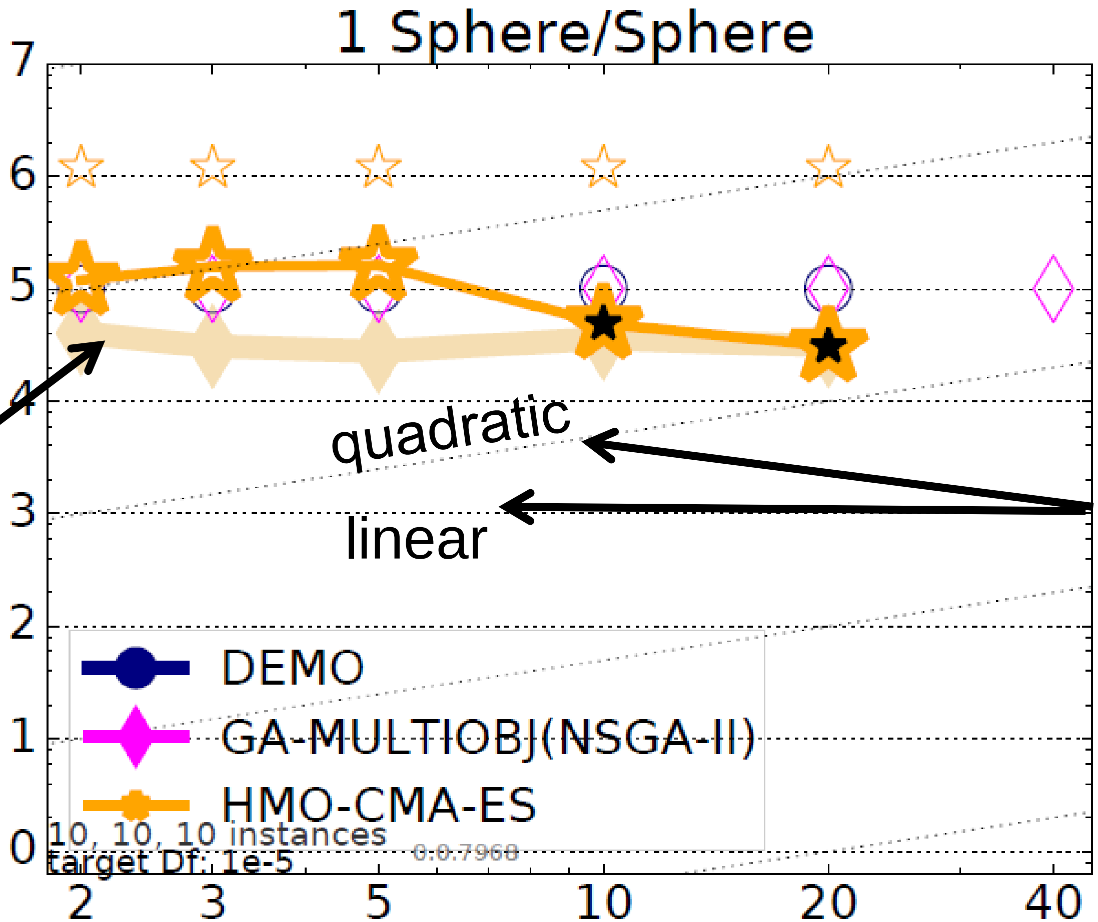
median runlength
of unsuccessful runs

a star indicates statistically
significant results compared
to all other displayed algos

aRT value
[if < ∞]
to reach
given target
precision

Another Interesting Plot...

...compares average runtimes over several algorithms



artificial best algorithm from BBOB-2016

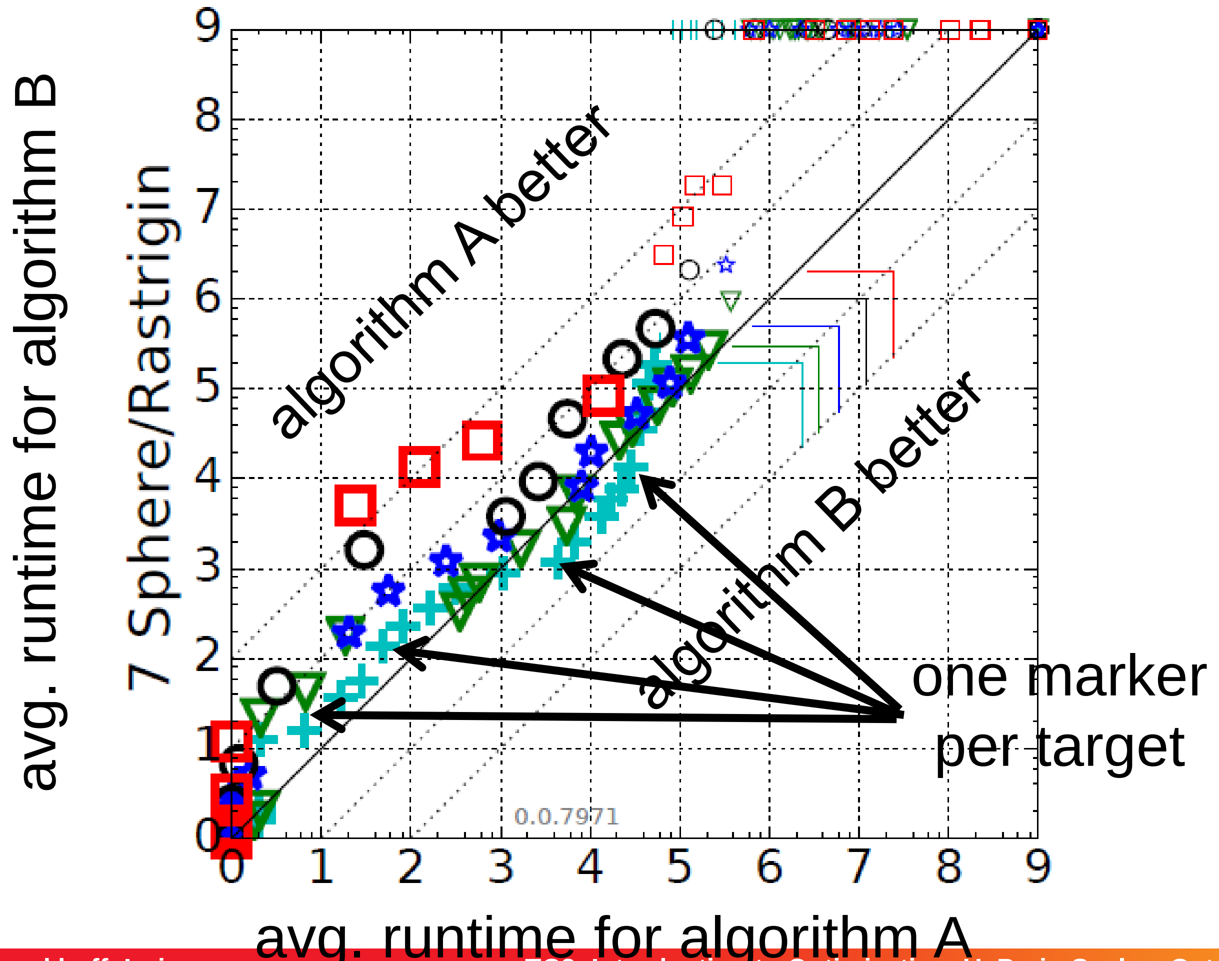
scaling with dimension

Interesting for 2 Algorithms...

...are scatter plots

dimensions:

2: +, 3: ▽, 5: ⋆, 10: ○, 20: □, 40: ◇.



Take Home Messages Benchmarking

I hope it became clear...

...that benchmarking is a non-trivial task

...details matter when comparing algorithms

...and that the COCO platform allows for an automated benchmarking and provides data from hundreds of benchmarking experiments

Discrete Optimization

Context discrete optimization:

- discrete variables
- or optimization over discrete structures (e.g. graphs)
- search space often finite, but typically too large for enumeration
- → need for smart algorithms

Algorithms for discrete problems:

- typically problem-specific
- but some general concepts are repeatedly used:
  - greedy algorithms
  - [branch and bound]
  - dynamic programming
  - randomized search heuristics

before 2 excursions:
the O-notation
& graph theory

Motivation for this Part:

  - get an idea of the most common algorithm design principles

Excursion: The O-Notation

Excursion: The O-Notation

Motivation:

- we often want to characterize how quickly a function f(x) grows asymptotically
- e.g. when we say an algorithm takes quadratically many steps (in the input size) to find the optimum of a problem with n (binary) variables, it is most likely not exactly n^2, but maybe n^2+1 or $(n+1)^2$

Big-O Notation

should be known, here mainly restating the definition:

Definition 1 We write $f(x) = O(g(x))$ iff there exists a constant $c > 0$ and an $x_0 > 0$ such that $|f(x)| \leq c \cdot g(x)$ holds for all $x > x_0$

we also view O(g(x)) as a set of functions growing at most as quick as g(x) and write $f(x) \in O(g(x))$

Big-O: Examples

- $f(x) + c = O(f(x))$ [if f(x) does not go to zero for x to infinity]
- $c \cdot f(x) = O(f(x))$
- $f(x) \cdot g(x) = O(f(x) \cdot g(x))$
- $3n^4 + n^2 - 7 = O(n^4)$

Intuition of the Big-O:

- if $f(x) = O(g(x))$ then $g(x)$ gives an upper bound (asymptotically) for f excluding constants and lower order terms
- With Big-O, you should have '≤' in mind
- An algorithm that solves a problem in polynomial time is "efficient"
- An algorithm that solves a problem in exponential time is not
- But be aware:
  In practice, often the line between efficient and non-efficient lies around $n \log n$ or even n (or even $\log n$ in the big data context) and the constants **do** matter!!!

Excursion: The O-Notation

Further definitions to generalize from '≤' to '≥' and '=':

- $f(x) = \Omega(g(x))$ if $g(x) = O(f(x))$
- $f(x) = \Theta(g(x))$ if $f(x) = O(g(x))$ and $g(x) = O(f(x))$

Note: extensions to '<' and '>' exist as well, but are not needed here.

Example:

- Algo A solves problem P in time $O(n)$
- Algo B solves problem P in time $O(n^2)$
- which one is faster?

only proving upper bounds to compare algorithms is not sufficient!

Excursion: The O-Notation

Further definitions to generalize from ‘≤’ to ‘≥‘ and ‘=‘:

- $f(x) = \Omega(g(x))$ if $g(x) = O(f(x))$
- $f(x) = \Theta(g(x))$ if $f(x) = O(g(x))$ and $g(x) = O(f(x))$

Note: extensions to ‘<‘ and ‘>’ exist as well, but are not needed here.

Example:

- Algo A solves problem P in time $O(n)$
- Algo B solves problem P in time $\cancel{O(n^2)}$ $\Omega(n^2)$
- which one is faster?

only proving upper bounds to compare algorithms is not sufficient!

Exercise O-Notation

❶ Please order the following functions in terms of their asymptotic behavior (from smallest to largest):

- $\exp(n^2)$
- log n
- ln n / ln ln n
- n
- n log n
- exp(n)
- ln n!

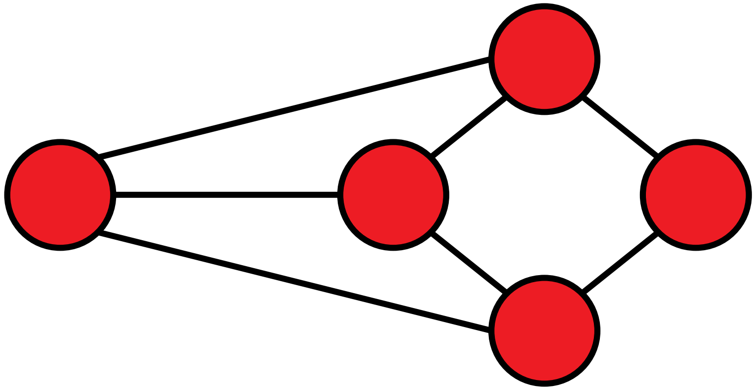
❷ Pick one pair of runtimes and give a formal proof for the relation.

Excursion:
Basic Concepts of Graph Theory

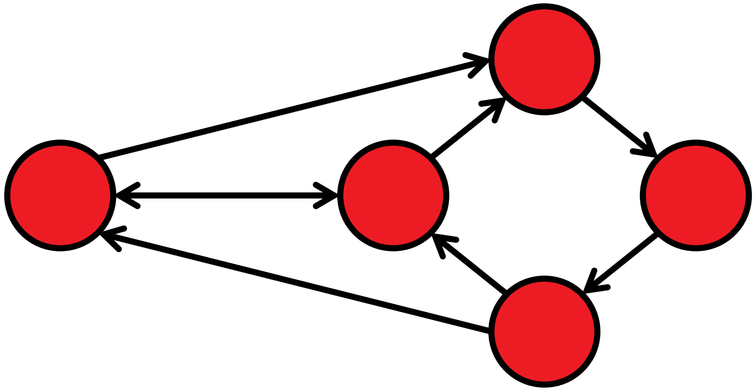
[following for example http://math.tut.fi/~ruohonen/GT_English.pdf]

Graphs

Definition 1 *An undirected graph G is a tupel $G = (V, E)$ of edges $e = \{u, v\} \in E$ over the vertex set V (i.e., $u, v \in V$).*

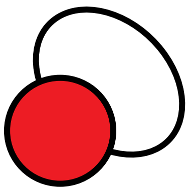


- vertices = nodes
- edges = lines
- Note: edges cover two *unordered* vertices (*undirected* graph)
  - if they are *ordered*, we call G a *directed* graph



Graphs: Basic Definitions

- G is called *empty* if E empty
- u and v are *end vertices* of an edge {u,v}
- Edges are *adjacent* if they share an end vertex
- Vertices u and v are *adjacent* if {u,v} is in E

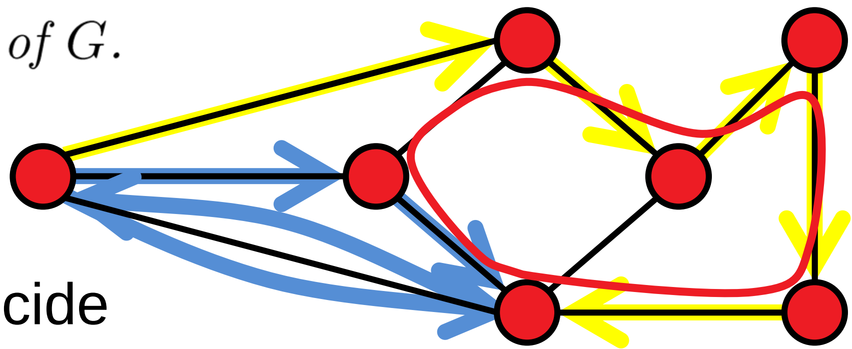


a loop

Definition 1 *A* walk *in a graph* $G = (V, E)$ *is a sequence*

$$v_{i_0}, e_{i_1} = (v_{i_0}, v_{i_1}), v_{i_1}, e_{i_2} = (v_{i_1}, v_{i_2}), \ldots, e_{i_k}, v_{i_k},$$

alternating vertices and adjacent edges of G.



A walk is

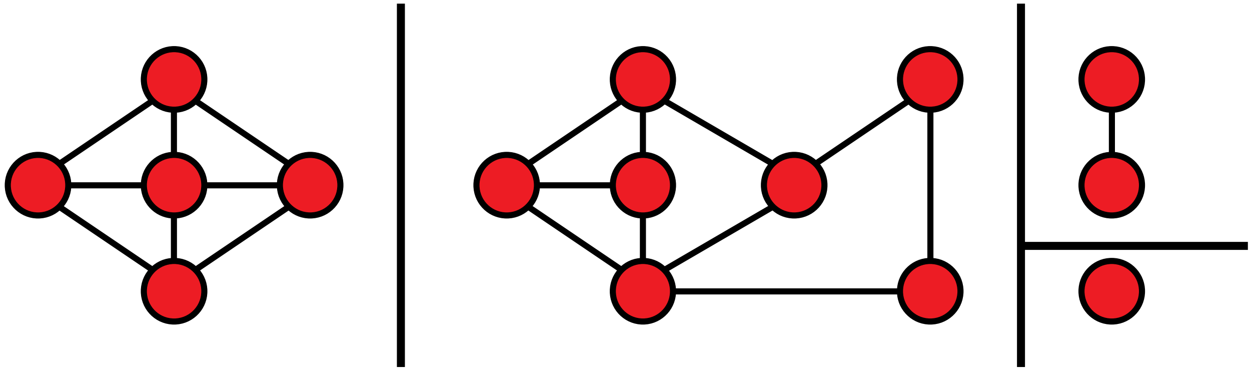
- *closed* if first and last node coincide
- a *trail* if each edge traversed at most once
- a *path* if each vertex is visited at most once

a closed path is called a *circuit* or *cycle*

Graphs: Connectedness

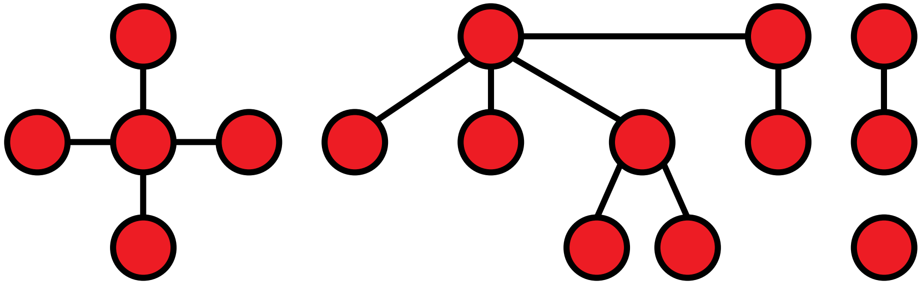
- Two vertices are called *connected* if there is a walk between them in G
- If all vertex pairs in G are connected, G is called connected

- The *connected components* of G are the (maximal) subgraphs which are connected.

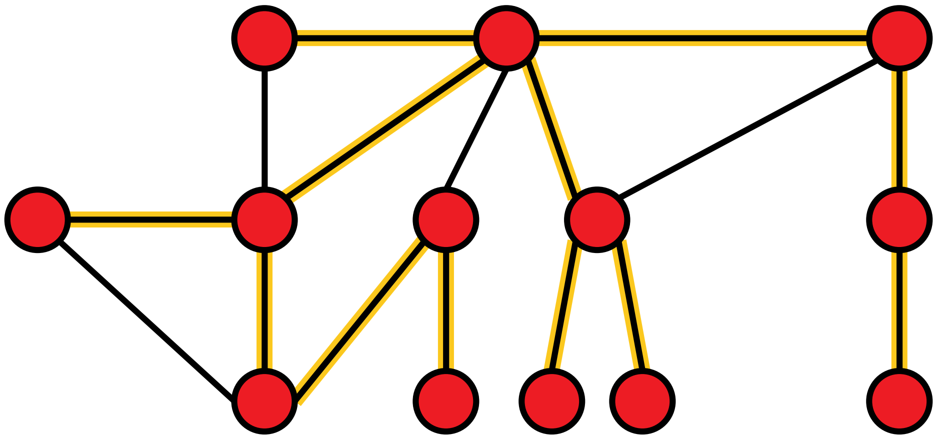


Trees and Forests

- A *forest* is a cycle-free graph
- A *tree* is a connected forest



A *spanning tree* of a connected graph G is a tree in G which contains all vertices of G



Greedy Algorithms

Greedy Algorithms

From Wikipedia:

"A *greedy algorithm* is an algorithm that follows the problem solving *heuristic* of making the locally optimal choice at each stage with the hope of finding a global optimum."

- Note: typically greedy algorithms do not find the global optimum

Lecture Outline Greedy Algorithms

What we will see:

➊ Example 1: Money Change problem

➋ Example 2: Minimal Spanning Trees (MST) and the algorithm of Kruskal

Example 1: Money Change

Change-making problem

- Given n coins of distinct values $w_1=1, w_2, ..., w_n$ and a total change W (where $w_1, ..., w_n$, and W are integers).
- Minimize the total amount of coins Σx_i such that $\Sigma w_i x_i = W$ and where x_i is the number of times, coin i is given back as change.

Greedy Algorithm

Unless total change not reached:

add the largest coin which is not larger than the remaining amount to the change

Note: only optimal for standard coin sets, not for arbitrary ones!

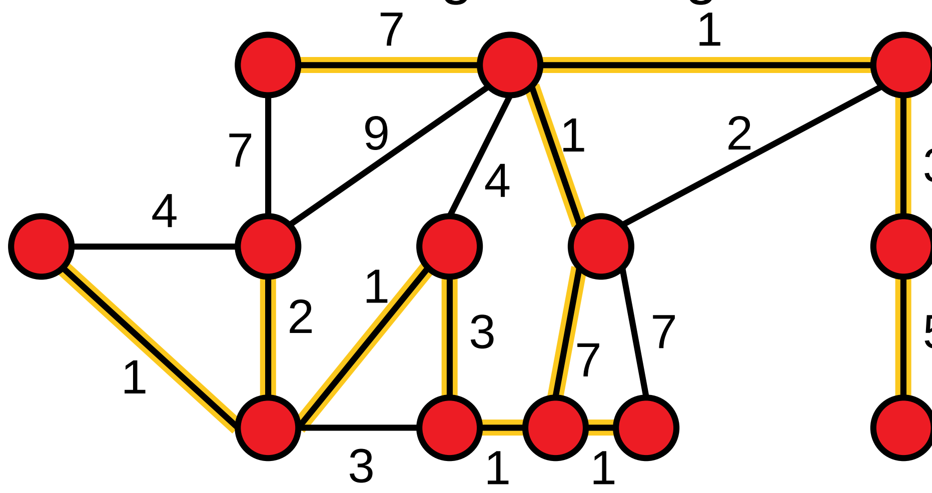
Related Problem:

finishing darts (from 501 to 0 with 9 darts)

Example 2: Minimum Spanning Trees (MST)

Minimum Spanning Tree problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i. Find the spanning tree with the smallest weight among all spanning trees.



weight of a spanning tree:

$$w(T) = \sum_{e_i \text{ in } T} w_i$$

w(T) = 33

Applications

Setting up a new wired telecommunication/water supply/electricity network

Constructing minimal delay trees for broadcasting in networks

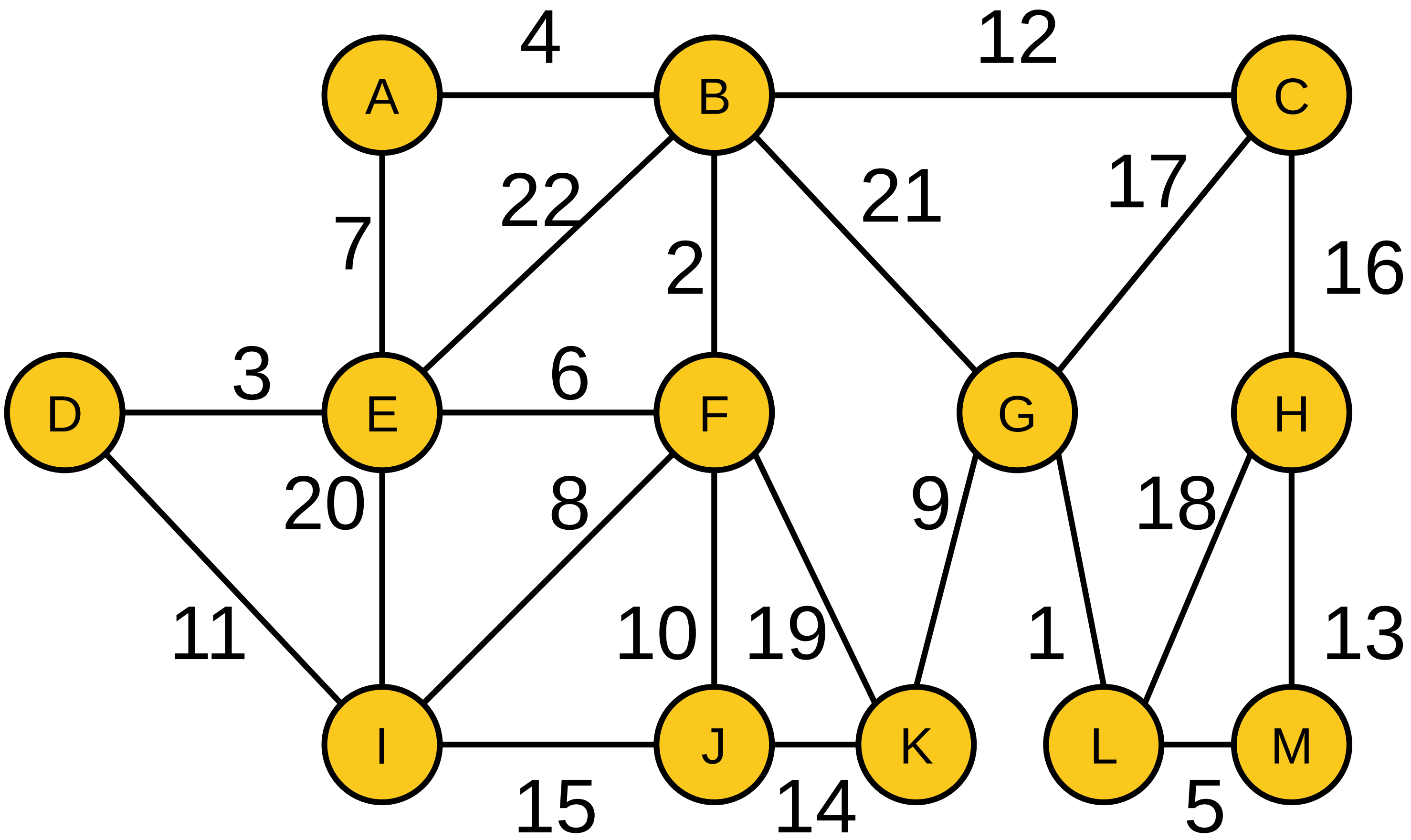
Kruskal's Algorithm: Idea

Algorithm, see [1]

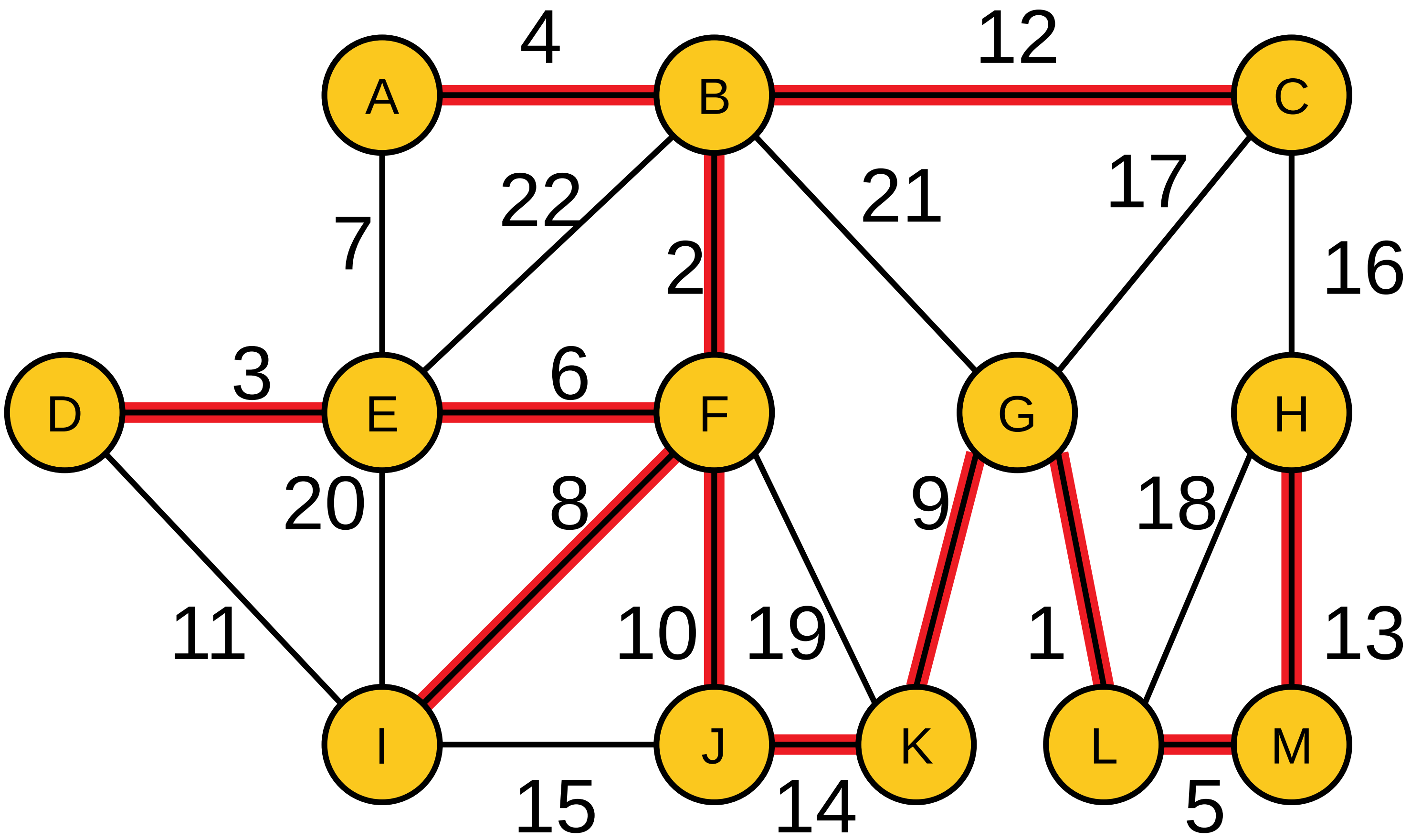
- Create forest F = (V,{}) with n components and no edge
- Put sorted edges (such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$) into set S
- While S non-empty and F not spanning:
  - delete cheapest edge from S
  - add it to F if no cycle is introduced

[1] Kruskal, J. B. (1956). "On the shortest spanning subtree of a graph and the traveling salesman problem". *Proceedings of the American Mathematical Society* **7**: 48–50. doi:10.1090/S0002-9939-1956-0078686-7

Kruskal's Algorithm: Example



Kruskal's Algorithm: Example



Kruskal's Algorithm: Runtime Considerations

First question: how to implement the algorithm?

- sorting of edges needs O(|E| log |E|)

Algorithm

Create forest F = (V,{}) with n components and no edge

Put sorted edges (such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$) into set S

While S non-empty and F not spanning:

delete cheapest edge from S

add it to F if no cycle is introduced

simple

?

forest implementation:

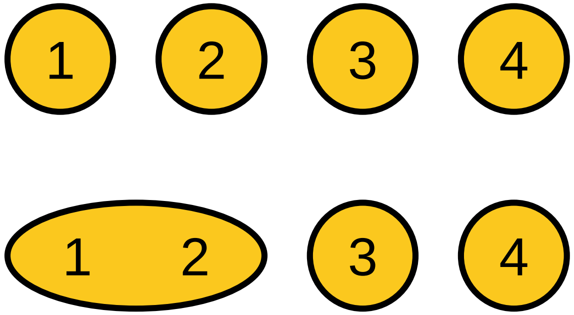
Disjoint-set data structure

Disjoint-set Data Structure (“Union&Find”)

Data structure: ground set 1...N grouped to disjoint sets

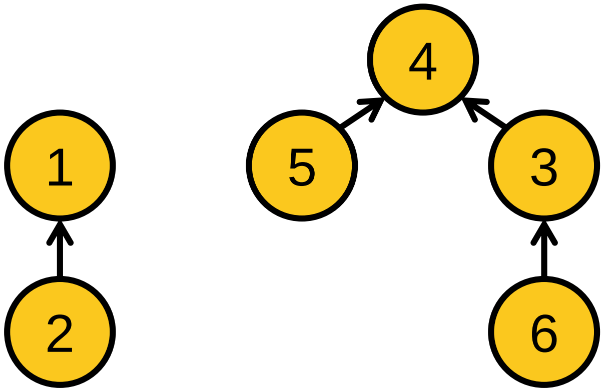
Operations:

- FIND(i): to which set does i belong?
- UNION(i,j): union the sets of i and j!



Implemented as trees:

- UNION(T1, T2): hang root node of smaller tree under root node of larger tree (constant time), thus
- FIND(u): traverse tree from u to root (to return a representative of u’s set) takes logarithmic time in total number of nodes



Implementation of Kruskal's Algorithm

Algorithm, rewritten with UNION-FIND:

- Create initial disjoint-set data structure, i.e. for each vertex v_i, store v_i as representative of its set
- Create empty forest F = {}
- Sort edges such that w.l.o.g. $w_1 < w_2 < ... < w_{|E|}$
- for each edge e_i={u,v} starting from i=1:
  - if FIND(u) ≠ FIND(v): # no cycle introduced?
    - F = F ∪ {{u,v}}
    - UNION(u,v)
- return F

Back to Runtime Considerations

- Sorting of edges needs O(|E| log |E|)
- forest: **Disjoint-set data structure**
  - initialization: O(|V|)
  - log |V| to find out whether the minimum-cost edge {u,v} connects two sets (no cycle induced) or is within a set (cycle would be induced)
  - 2x FIND + potential UNION needs to be done O(|E|) times
  - total O(|E| log |V|)
- Overall: O(|E| log |E|)

Kruskal's Algorithm: Proof of Correctness

Two parts needed:

❶ Algo always produces a spanning tree

final F contains no cycle and is connected by definition ✓

❷ Algo always produces a *minimum* spanning tree

- argument by induction
- P: If *F* is forest at a given stage of the algorithm, then there is some minimum spanning tree that contains *F*.
- clearly true for F = (V, {})
- assume that P holds when new edge e is added to F and be T a MST that contains F
  - if e in T, fine
  - if e not in T: T + e has cycle C with edge f in C but not in F (otherwise e would have introduced a cycle in F)
    - now T – f + e is a tree with same weight as T (since T is a MST and f was not chosen to F)
    - hence T – f + e is MST including F + e (i.e. P holds) ✓

Conclusion Greedy Algorithms I

What we have seen so far:

- two problems where a greedy algorithm was optimal
  - money change
  - minimum spanning tree (Kruskal's algorithm)
- but also: greedy not always optimal
  - for some sets of coins for example

Obvious Question: when is greedy good?

Answer: if the problem is a matroid (no further details here)

From Wikipedia: [...] a matroid is a structure that captures and generalizes the notion of linear independence in vector spaces. There are many equivalent ways to define a matroid, the most significant being in terms of independent sets, bases, circuits, closed sets or flats, closure operators, and rank functions.

Conclusions Greedy Algorithms II

I hope it became clear...

...what a greedy algorithm is

...that it not always results in the optimal solution

...but that it does if and only if the problem is a matroid

Dynamic Programming

Dynamic Programming

Wikipedia:

"[...] **dynamic programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems."

But that's not all:

- dynamic programming also makes sure that the subproblems are not solved too often but only once by keeping the solutions of simpler subproblems in memory ("trading space vs. time")
- it is an exact method, i.e. in comparison to the greedy approach, it always solves a problem to optimality

Note:

the reason why the approach is called "dynamic programming" is historical: at the time of invention by Richard Bellman, no computer "program" existed

Two Properties Needed

Optimal Substructure

A solution can be constructed efficiently from optimal solutions of sub-problems

Overlapping Subproblems

Wikipedia: “[...] a problem is said to have **overlapping subproblems** if the problem can be broken down into subproblems which are reused several times or a recursive algorithm for the problem solves the same subproblem over and over rather than always generating new subproblems.”

Note: in case of optimal substructure but independent subproblems, often greedy algorithms are a good choice; in this case, dynamic programming is often called “divide and conquer” instead

Main Idea Behind Dynamic Programming

Main idea: solve larger subproblems by breaking them down to smaller, easier subproblems in a recursive manner

Typical Algorithm Design:

❶ decompose the problem into subproblems and think about how to solve a larger problem with the solutions of its subproblems

❷ specify how you compute the value of a larger problem recursively with the help of the optimal values of its subproblems ("Bellman equation")

❸ bottom-up solving of the subproblems (i.e. computing their optimal value), starting from the smallest by using the Bellman equality and a table structure to store the optimal values (top-down approach also possible, but less common)

❹ eventually construct the final solution (can be omitted if only the value of an optimal solution is sought)

Lecture Outline Dynamic Programming (DP)

What we will see:

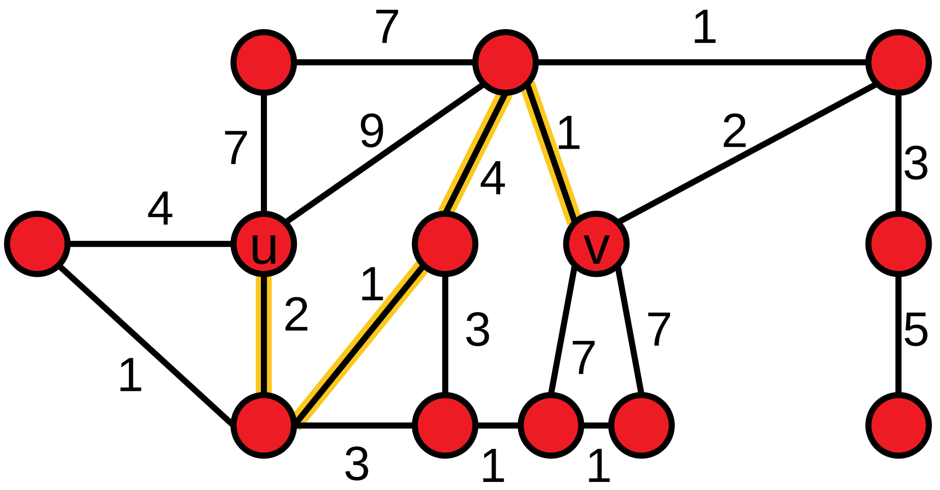
➊ Example 1: The All-Pairs Shortest Path Problem

➋ Example 2: The knapsack problem

Example 1: The Shortest Path Problem

Shortest Path problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i. Find the shortest path from a vertex v to a vertex u, i.e., the path (v, $e_1=\{v, v_1\}$, v_1, ..., v_k, $e_k=\{v_k,u\}$, u) such that $w_1 + ... + w_k$ is minimized.



Obvious Applications

Google maps

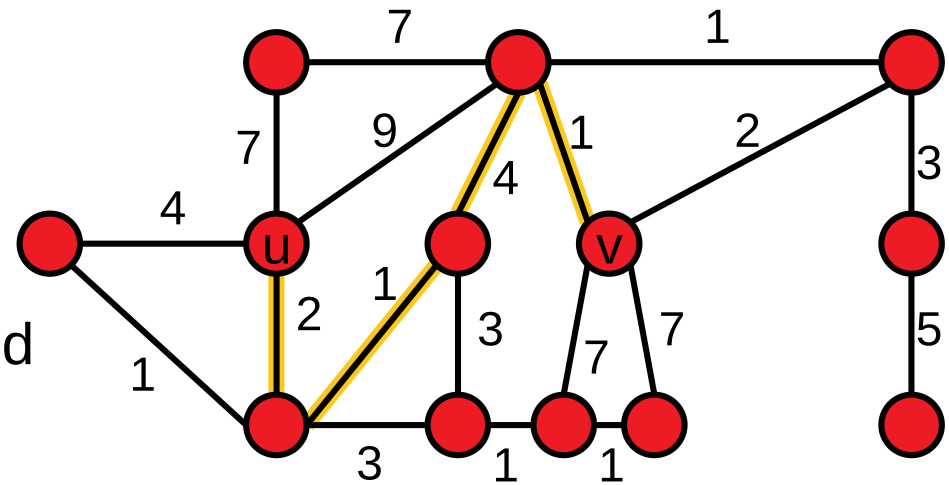
Autonomous cars

Finding routes for packages in a computer network

...

Shortest Path problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i. Find the shortest path from a vertex v to a vertex u, i.e., the path (v, e_1={v, v_1}, v_1, ..., v_k, e_k={v_k,u}, u) such that w_1 + ... + w_k is minimized.



Note:

We can often assume that the edge weights are stored in a distance matrix D of dimension |E|x|E| where an entry $D_{i,j}$ gives the weight between nodes i and j and "non-edges" are assigned a value of ∞

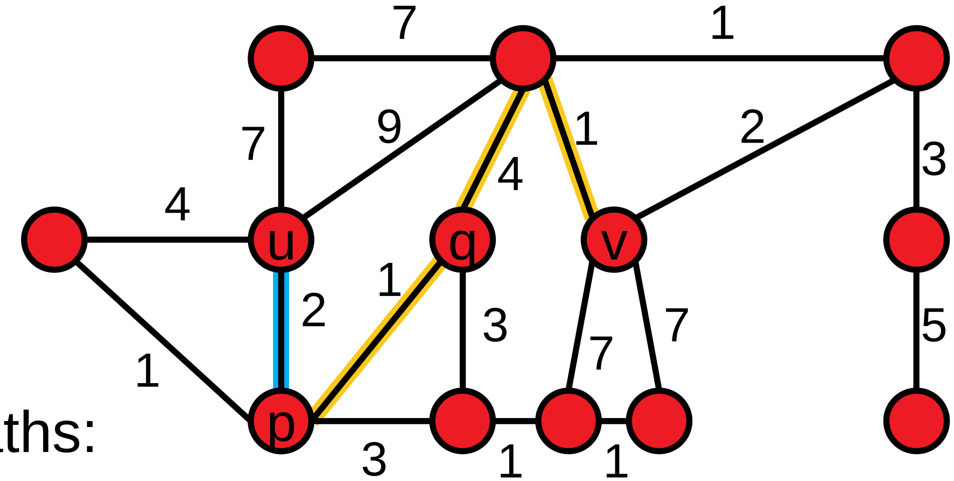
Why important? ⇨ determines input size

Optimal Substructure

The optimal path from u to v, if it contains another vertex p can be constructed by simply joining the optimal path from u to p with the optimal path from p to v.

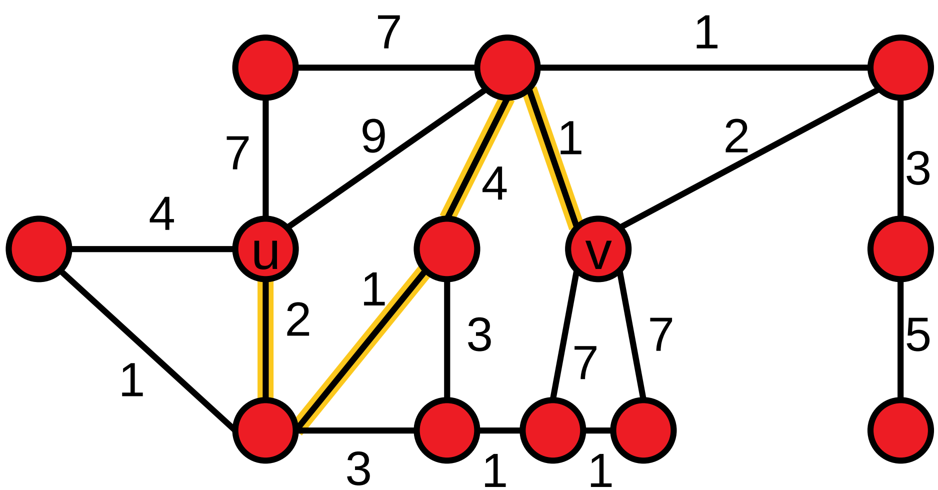
Overlapping Subproblems

Optimal shortest
sub-paths can be reused
when computing longer paths:
e.g. the optimal path from u to p
is contained in the optimal path from
u to q and in the optimal path from u to v.



All Pairs Shortest Path problem:

Given a graph G=(V,E) with edge weights w_i for each edge e_i.
Find the shortest path from each source vertex v to each other target vertex u, i.e., the paths (v, e_1={v, v_1}, v_1, ..., v_k, e_k={v_k,u}, u) such that $w_1 + ... + w_k$ is minimized for all pairs (u,v) in V^2.



The Algorithm of Robert Floyd (1962)

Idea:

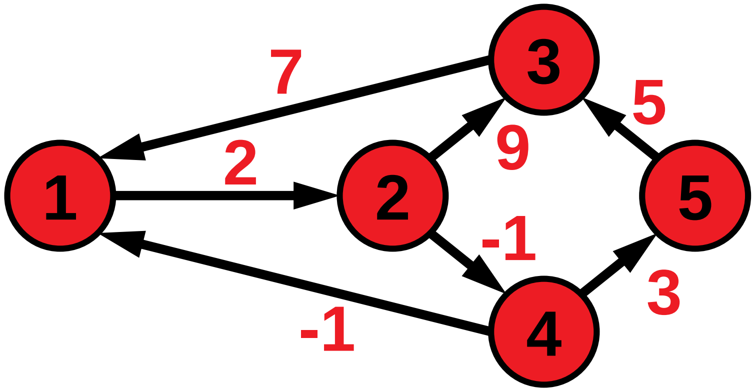
- if we knew that the shortest path between source and target goes through node v, we would be able to construct the optimal path from the shorter paths “source→v” and “v→target”
- subproblem P(k): compute all shortest paths where the intermediate nodes can be chosen from $v_1, \ldots, v_k$

AllPairsShortestPathFloyd(G, D)

- Init: for all $1 \leq i,j \leq |V|$: $dist(i,j) = D_{i,j}$
- For $k = 1$ to $|V|$ # solve subproblems P(k)
  - for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
    - $dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$

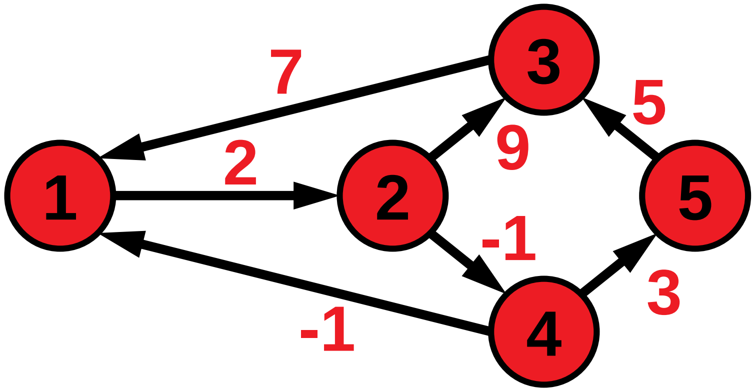
Note: Bernard Roy in 1959 and Stephen Warshall in 1962 essentially proposed the same algorithm independently.

Example



k=0	1	2	3	4	5
1					
2					
3					
4					
5					

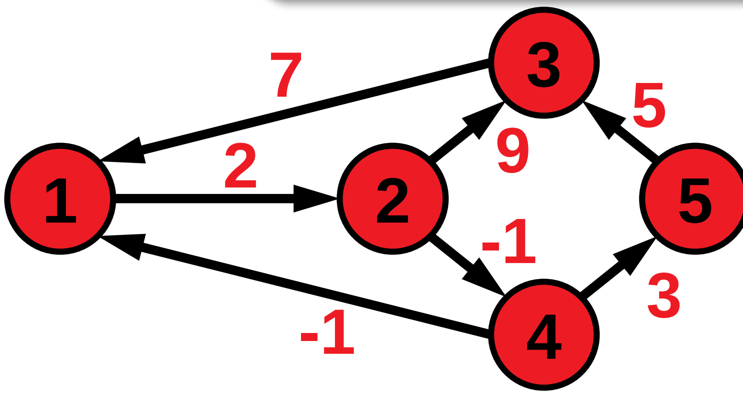
Example



k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

Example

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



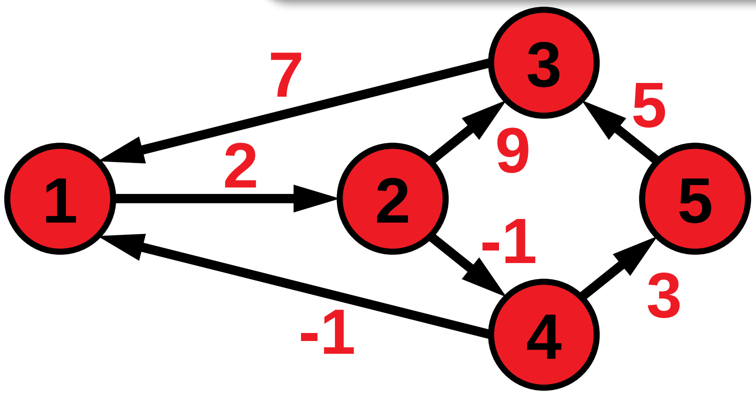
allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



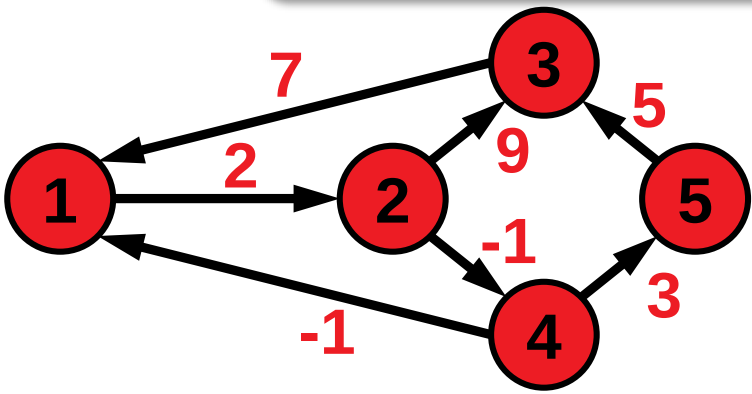
allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



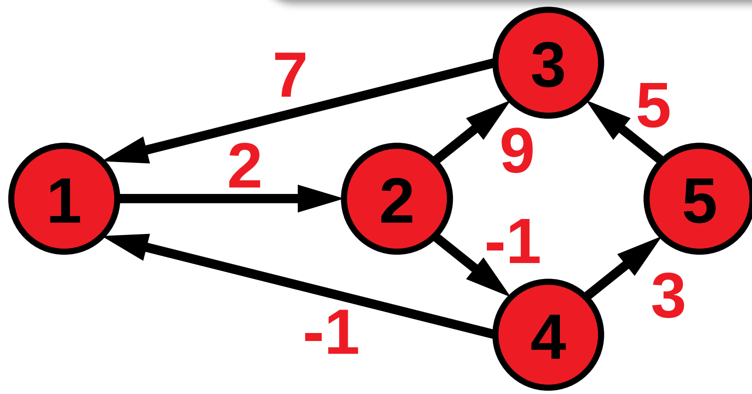
allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3					
4					
5					

Example

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



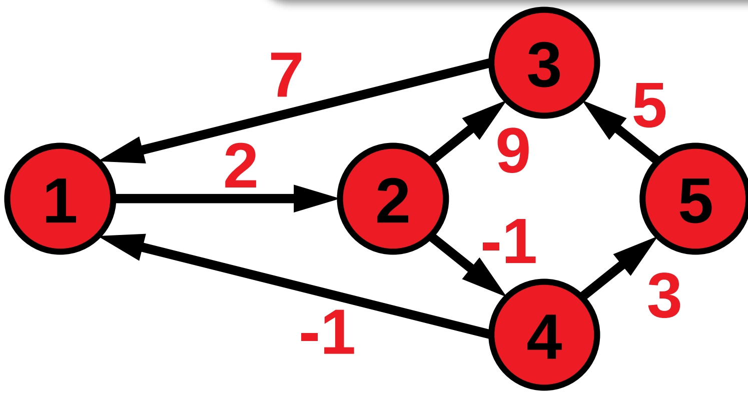
allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1					
2					
3		9			
4		1			
5					

Example

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$



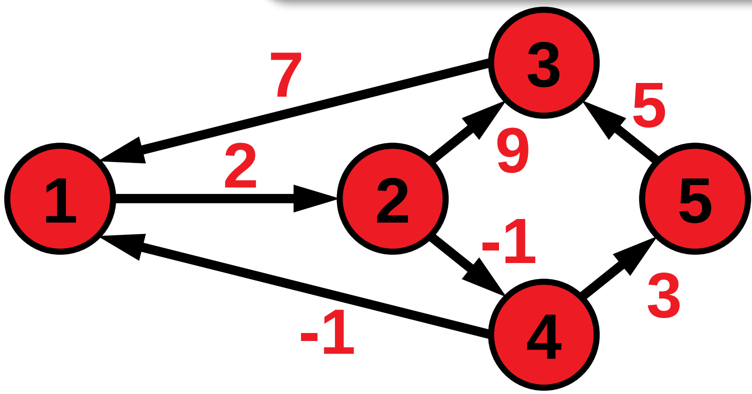
allow 1 as intermediate node

k=0	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	∞	∞	∞	∞
4	-1	∞	∞	∞	3
5	∞	∞	5	∞	∞

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

Example

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



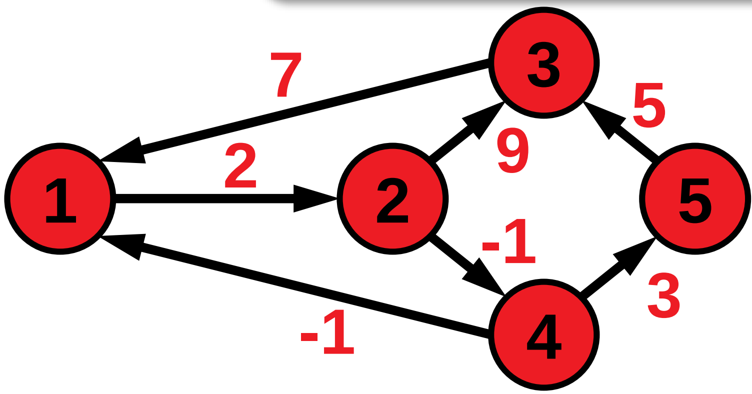
allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

k=2	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

Example

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



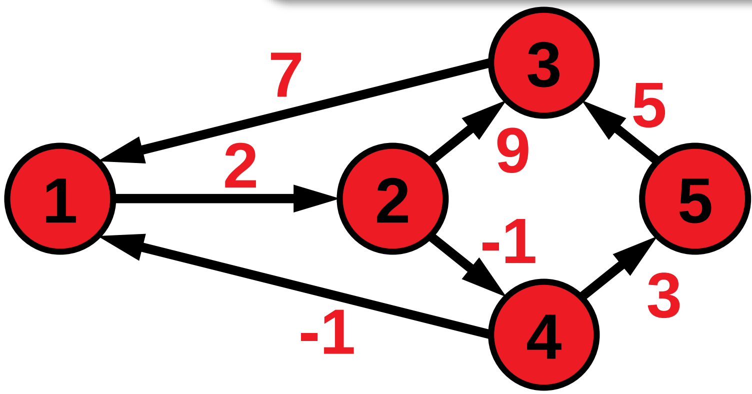
allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

k=2	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

Example

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



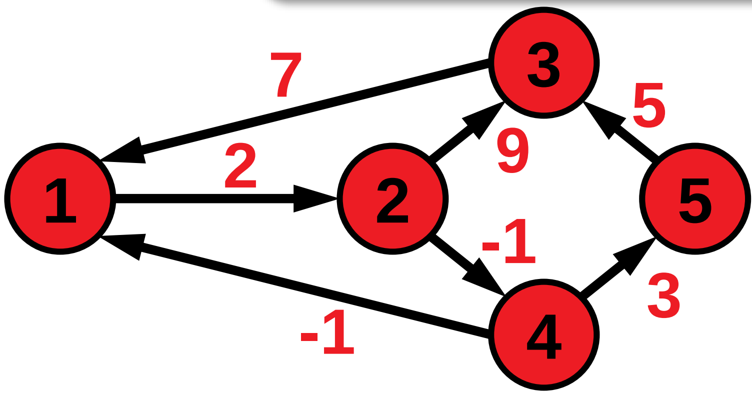
allow 1 & 2 as intermediate nodes

k=1	1	2	3	4	5
1	∞	2	∞	∞	∞
2	∞	∞	9	-1	∞
3	7	9	∞	∞	∞
4	-1	1	∞	∞	3
5	∞	∞	5	∞	∞

k=2	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

Example

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



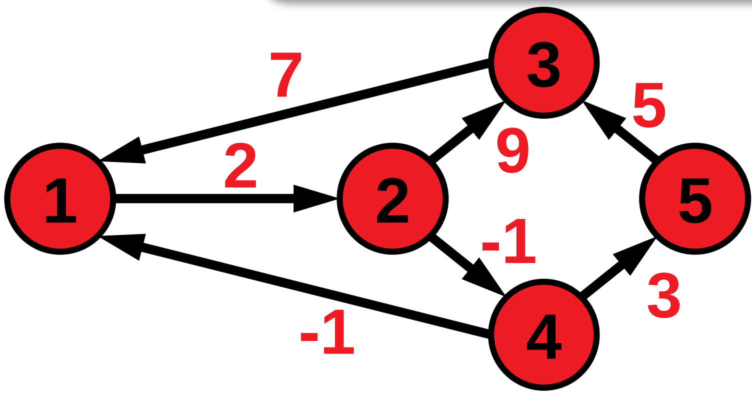
allow {1,2,3} as intermediate nodes

k=2	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

k=3	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

Example

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



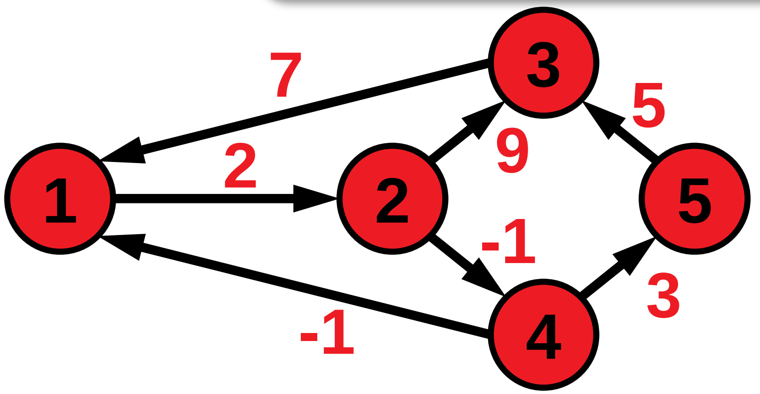
allow {1,2,3} as intermediate nodes

k=2	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

k=3	1	2	3	4	5
1			11		∞
2			9		∞
3	7	9	18	8	∞
4			10		3
5			5		∞

Example

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$



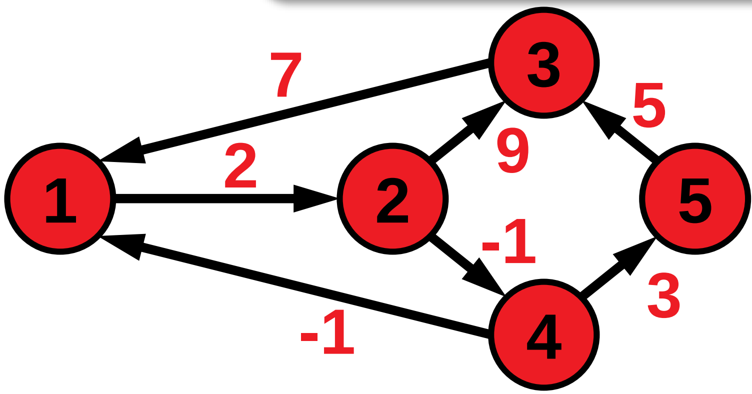
allow {1,2,3} as intermediate nodes

k=2	1	2	3	4	5
1	∞	2	11	1	∞
2	∞	∞	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	∞	∞	5	∞	∞

k=3	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

Example

for all pairs of nodes (i.e. 1 ≤ i,j ≤ |V|):
dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }



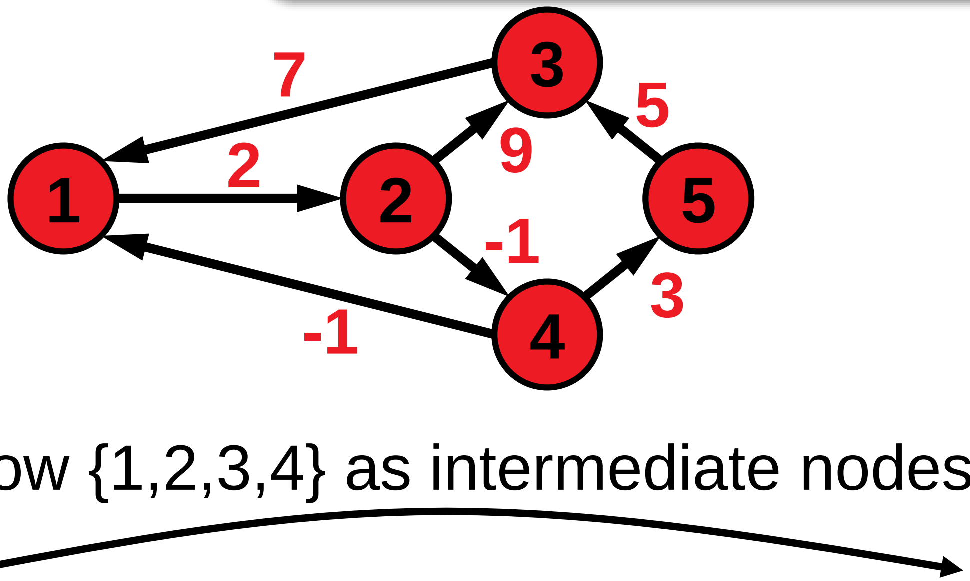
allow {1,2,3,4} as intermediate nodes

k=3	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

k=4	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

Example

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$

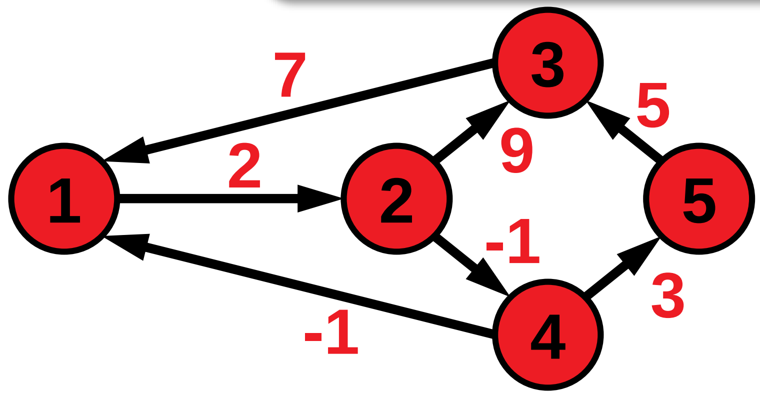


k=3	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

k=4	1	2	3	4	5
1				1	
2				-1	
3				8	
4	-1	1	10	0	3
5				13	

Example

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$\text{dist}(i,j) = \min \{ \text{dist}(i,j), \text{dist}(i,k) + \text{dist}(k,j) \}$



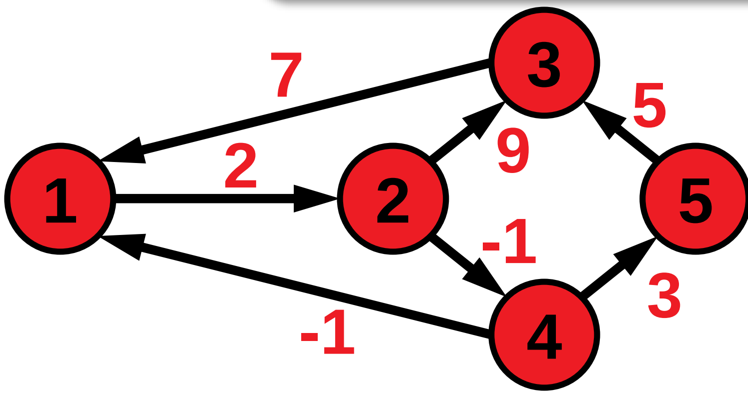
allow {1,2,3,4} as intermediate nodes

k=3	1	2	3	4	5
1	18	2	11	1	∞
2	16	18	9	-1	∞
3	7	9	18	8	∞
4	-1	1	10	0	3
5	12	14	5	13	∞

k=4	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

Example

for all pairs of nodes (i.e. $1 \leq i,j \leq |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



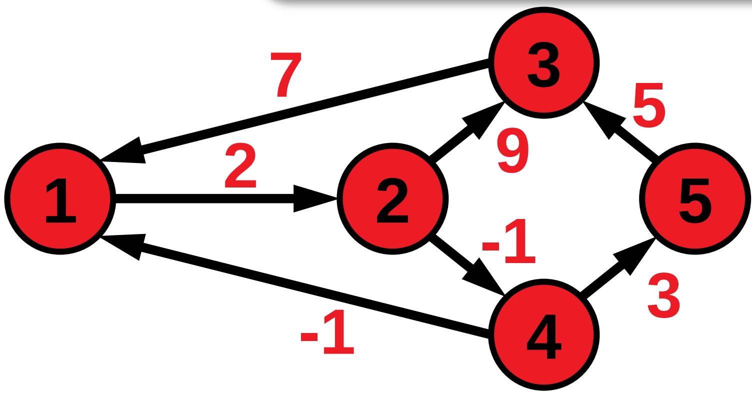
allow all nodes as intermediate nodes

k=4	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

k=5	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

Example

for all pairs of nodes (i.e. $1 \le i,j \le |V|$):
$dist(i,j) = \min \{ dist(i,j), dist(i,k) + dist(k,j) \}$



allow all nodes as intermediate nodes

k=4	1	2	3	4	5
1	0	2	11	1	4
2	-2	0	9	-1	2
3	7	9	18	8	11
4	-1	1	10	0	3
5	12	14	5	13	16

k=5	1	2	3	4	5
1	0	2	9	1	4
2	-2	0	7	-1	2
3	7	9	16	8	11
4	-1	1	8	0	3
5	12	14	5	13	16

Runtime Considerations and Correctness

$O(|V|^3)$ easy to show

- $O(|V|^2)$ many distances need to be updated $O(|V|)$ times

Correctness

- given by the Bellman equation

  dist(i,j) = min { dist(i,j), dist(i,k) + dist(k,j) }
- only correct if cycles do not have negative total weight (can be checked in final distance matrix if diagonal elements are negative)

But How Can We Actually Construct the Paths?

- Construct matrix of predecessors P alongside distance matrix
- $P_{i,j}(k)$ = predecessor of node j on path from i to j (at algo. step k)
- no extra costs (asymptotically)

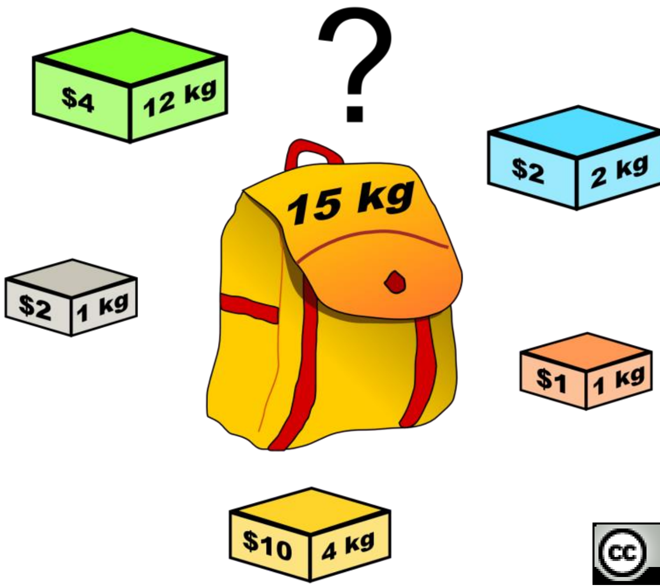
$$P_{i,j}(0) = \begin{cases} 0 & \text{if } i = j \text{ or } d_{i,j} = \infty \\ i & \text{in all other cases} \end{cases}$$

$$P_{i,j}(k) = \begin{cases} P_{i,j}(k-1) & \text{if } \mathrm{dist}(i,j) \leq \mathrm{dist}(i,k) + \mathrm{dist}(k,j) \\ P_{k,j}(k-1) & \text{if } \mathrm{dist}(i,j) > \mathrm{dist}(i,k) + \mathrm{dist}(k,j) \end{cases}$$

Knapsack Problem

$$\text{max.} \sum_{j=1}^{n} p_j x_j \text{ with } x_j \in \{0,1\}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_j \leq W$$



Opt. Substructure and Overlapping Subproblems

Consider the following subproblem:

$P(i,j)$: optimal profit when packing the first i items into a knapsack of size j

Optimal Substructure

The optimal choice of whether taking item i or not can be made easily for a knapsack of weight j if we know the optimal choice for items $1 \ldots i-1$:

$$P(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Overlapping Subproblems

a recursive implementation of the Bellman equation is simple, but the $P(i,j)$ might need to be computed more than once!

Dynamic Programming Approach to the KP

To circumvent computing the subproblems more than once, we can store their results (in a matrix for example)...

knapsack weight →

items ↓

P(i,j)	0	1	2	3	...			W-1	W
0									
1					P(i,j)				
2									
...									
n-1									
n									

best achievable profit with items 1...i and a knapsack of size j

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is W=11.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0												
1												
2												
3												
4												
5												

initialization:
$P(i,j) = 0$ if $i = 0$ or $j = 0$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits (5,4), (7,10), (2,3), (4,5), and (3,3). Weight restriction is W=11.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											

initialization:
$P(i,j) = 0$ if $i = 0$ or $j = 0$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0											
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j),\, p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0										
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0									
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0							
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4						
2	0											
3	0											
4	0											
5	0											

$+p_1(= 4)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4					
2	0											
3	0											
4	0											
5	0											

$+p_1 (= 4)$

for $i = 1$ to n:

  for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j),\, p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0											
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4					
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1, j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight $\longrightarrow$

items $\downarrow$

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10				
3	0											
4	0											
5	0											

$+p_2(= 10)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0											
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3							
4	0											
5	0											

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4						
4	0											
5	0											

$+p_3(= 3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \le j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4					
4	0											
5	0											

$+p_3(= 3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1, j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5{,}4)$, $(7{,}10)$, $(2{,}3)$, $(4{,}5)$, and $(3{,}3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	etc.			
4	0											
5	0											

$+p_3(=3)$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j),\, p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1, j-w_i)\} & \text{if } w_i \le j \end{cases}$$

Dynamic Programming Approach to the KP

Example instance with 5 items with weights and profits $(5,4)$, $(7,10)$, $(2,3)$, $(4,5)$, and $(3,3)$. Weight restriction is $W = 11$.

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	**15**

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Dynamic Programming Approach to the KP

Question: How to obtain the actual packing?

Answer: we just need to remember where the $\max$ came from!

knapsack weight →

items ↓

P(i,j)	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	4	4	4	4	4	4	4
2	0	0	0	0	0	4	4	10	10	10	10	10
3	0	0	3	3	3	4	4	10	10	13	13	13
4	0	0	3	3	5	5	8	10	10	13	13	15
5	0	0	3	3	5	6	8	10	10	13	13	15

$x_1 = 0$ $x_2 = 1$ $x_3 = 0$ $x_4 = 1$ $x_5 = 0$

for $i = 1$ to n:

for $j = 1$ to W:

$$P(i,j) = \begin{cases} P(i-1,j) & \text{if } w_i > j \\ \max\{P(i-1,j), p_i + P(i-1,j-w_i)\} & \text{if } w_i \leq j \end{cases}$$

Conclusions

I hope it became clear...

...what the algorithm design ideas of dynamic programming are

...and for which problem types it is supposed to be suitable

(Randomized) Search Heuristics

Motivation General Search Heuristics

- often, problem complicated and not much time available to develop a problem-specific algorithm
- search heuristics are a good choice:
  - relatively easy to implement
  - easy to adapt/change/improve
    - e.g. when the problem formulation changes in an early product design phase
    - or when slightly different problems need to be solved over time
- randomized/stochastic algorithms are a good choice because they are robust to noise

Lecture Outline Randomized Search Heuristics

Which algorithms will we touch?

❶ Randomized Local Search (RLS)

❷ Variable Neighborhood Search (VNS)

❸ Tabu Search (TS)

❹ Evolutionary Algorithms (EAs)

Neighborhoods

For most (stochastic) search heuristics, we need to define a *neighborhood structure*

- which search points are close to each other?

Example: k-bit flip / Hamming distance k neighborhood

- search space: bitstrings of length n ($\Omega=\{0,1\}^n$)
- two search points are neighbors if their Hamming distance is k
- in other words: x and y are neighbors if we can flip exactly k bits in x to obtain y
- 0001001101 is neighbor of
  0001000101 for k=1
  0101000101 for k=2
  1101000101 for k=3

Neighborhoods II

Example: possible neighborhoods for the knapsack problem

- search space again bitstrings of length n ($\Omega=\{0,1\}^n$)
- Hamming distance 1 neighborhood:
  - add an item or remove it from the packing
- replacing 2 items neighborhood:
  - replace one chosen item with an unchosen one
  - makes only sense in combination with other neighborhoods because the number of items stays constant
- Hamming distance 2 neighborhood on the contrary:
  - allows to change 2 arbitrary items, e.g.
    - add 2 new items
    - remove 2 chosen items
    - or replace one chosen item with an unchosen one

Randomized Local Search (RLS)

Idea behind (Randomized) Local Search:

- explore the local neighborhood of the current solution (randomly)

Pure Random Search:

- go to randomly chosen neighbor

First Improvement Local Search:

- go to first (randomly) chosen neighbor which is better

Best Improvement strategy:

- always go to the best neighbor
- not random anymore
- computationally expensive if neighborhood large

Variable Neighborhood Search

Main Idea: [Mladenovic and P. Hansen, 1997]

- change the neighborhood from time to time
  - local optima are not the same for different neighborhood operators
  - but often close to each other
  - global optimum is local optimum for all neighborhoods
- rather a framework than a concrete algorithm
  - e.g. deterministic and stochastic neighborhood changes
- typically combined with (i) first improvement, (ii) a random order in which the neighbors are visited and (iii) restarts

N. Mladenovic and P. Hansen (1997). "Variable neighborhood search". Computers and Operations Research 24 (11): 1097–1100.

Tabu Search

Disadvantages of local searches (with or without varying neighborhoods)

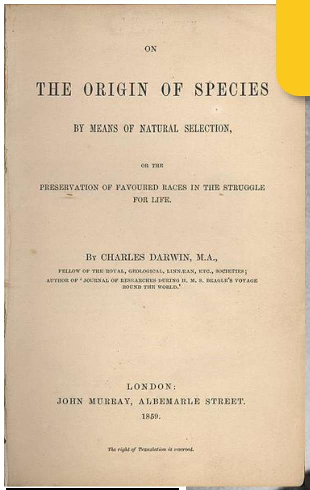
- they get stuck in local optima
- have problems to traverse large plateaus of equal objective function value (“random walk”)

Tabu search addresses these by

- allowing worsening moves if all neighbors are explored
- introducing a tabu list of temporarily not allowed moves
- those restricted moves are
  - problem-specific and
  - can be specific solutions or not permitted “search directions” such as “don’t include this edge anymore” or “do not flip this specific bit”
- the tabu list is typically restricted in size and after a while, restricted moves are permitted again

One class of (bio-inspired) stochastic optimization algorithms: Evolutionary Algorithms (EAs)

1859


ON

THE ORIGIN OF SPECIES

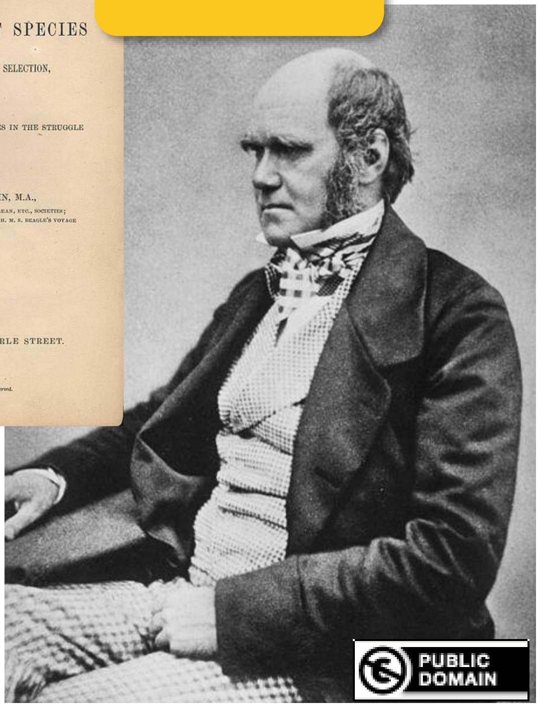
BY MEANS OF NATURAL SELECTION,

OR THE

PRESERVATION OF FAVOURED RACES IN THE STRUGGLE FOR LIFE.

By CHARLES DARWIN, M.A.,

LONDON:
JOHN MURRAY, ALBEMARLE STREET.
1859.

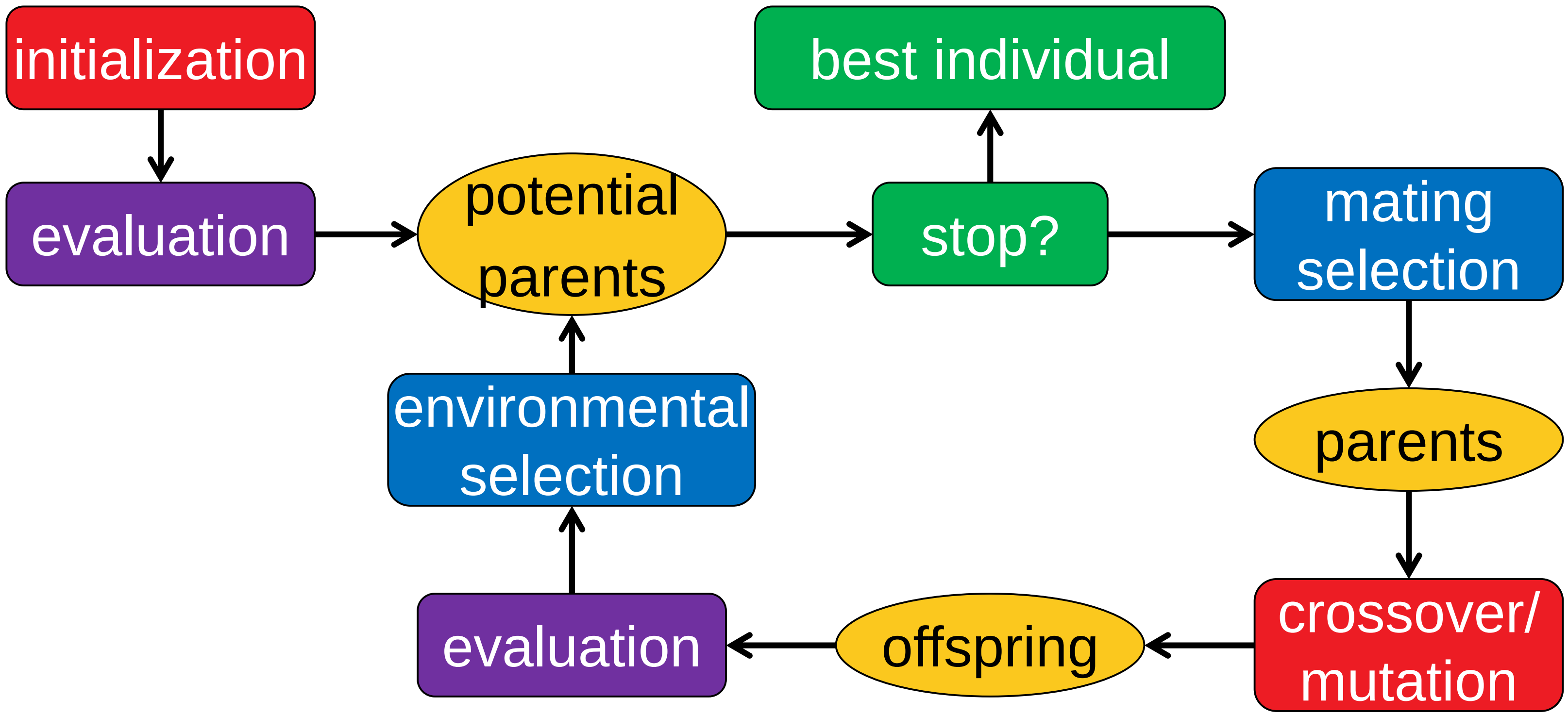


- Class of optimization algorithms originally inspired by the idea of biological evolution
- selection, mutation, recombination

Metaphors

Classical Optimization	Evolutionary Computation
variables or parameters	variables or chromosomes
candidate solution vector of decision variables / design variables / object variables	individual, offspring, parent
set of candidate solutions	population
objective function loss function cost function error function	fitness function
iteration	generation

Generic Framework of an EA



stochastic operators

“Darwinism”

stopping criteria

Important:
representation (search space)

The Historic Roots of EAs

Genetic Algorithms (GA)

J. Holland 1975 and D. Goldberg (USA)

$$\Omega = \{0,1\}^n$$

Evolution Strategies (ES)

I. Rechenberg and H.P. Schwefel, 1965 (Berlin)

$$\Omega = \mathbb{R}^n$$

Evolutionary Programming (EP)

L.J. Fogel 1966 (USA)

Genetic Programming (GP)

J. Koza 1990 (USA)

$$\Omega = \text{space of all programs}$$

nowadays one umbrella term: evolutionary algorithms

Genotype – Phenotype mapping

The genotype – phenotype mapping

- related to the question: how to come up with a fitness ("quality") of each individual from the representation?
- related to DNA vs. actual animal (which then has a fitness)

fitness of an individual not always = f(x)

- include constraints
- include diversity
- others
- but needed: always a total order on the solutions

Handling Constraints

Several possible ways to handle constraints, e.g.:

- resampling until a new feasible point is found ("often bad idea")
- penalty function approach: add constraint violation term (potentially scaled)
- repair approach: after generation of a new point, repair it (e.g. with a heuristic) to become feasible again if infeasible
  - continue to use repaired solution in the population or
  - use repaired solution only for the evaluation?
- multiobjective approach: keep objective function and constraint functions separate and try to optimize all of them in parallel
- ...

Examples for some EA parts

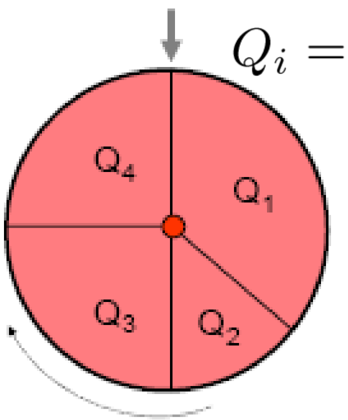
Selection is the major determinant for specifying the trade-off between exploitation and exploration

Selection is either

stochastic or deterministic

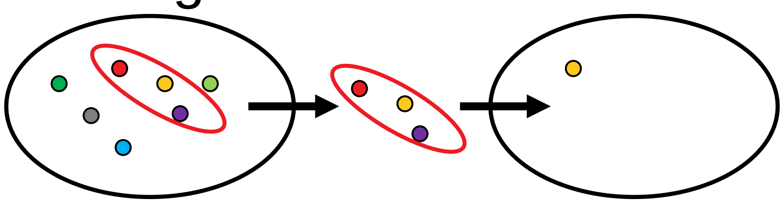
e.g. fitness proportional

$$Q_i = \frac{f(x_i)}{\sum_{j=1}^{\mu} f(x_j)}$$



Disadvantage: depends on scaling of f

e.g. via a tournament



e.g. (μ+λ), (μ,λ)

best μ from offspring *and* parents

best μ from offspring only

Mating selection (selection for variation): usually stochastic

Environmental selection (selection for survival): often deterministic

Variation Operators

Variation aims at generating new individuals on the basis of those individuals selected for mating

Variation = Mutation and Recombination/Crossover

mutation: *mut:* $\Omega \to \Omega$
recombination: *recomb:* $\Omega^r \to \Omega^s$ where $r \geq 2$ and $s \geq 1$

- choice always depends on the problem and the chosen representation
- however, there are some operators that are applicable to a wide range of problems and tailored to standard representations such as vectors, permutations, trees, etc.

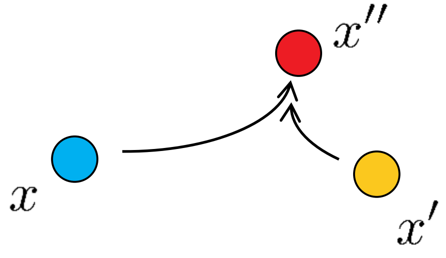
Variation Operators: Guidelines

Two desirable properties for mutation operators:

- every solution can be generation from every other with a probability greater than 0 (“exhaustiveness”)
- $d(x,x') < d(x,x'') => Prob(\mathrm{mut}(x) = x') > Prob(\mathrm{mut}(x) = x'')$ (“locality”)

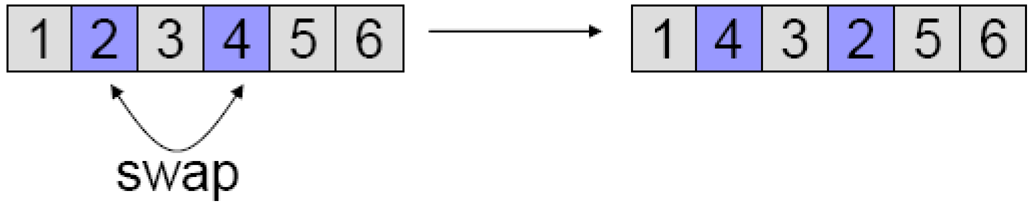
Desirable property of recombination operators (“in-between-ness”):

$$x'' = \ \mathrm{recomb}(x,x') \Rightarrow d(x'',x) \leq d(x,x') \wedge d(x'',x') \leq d(x,x')$$

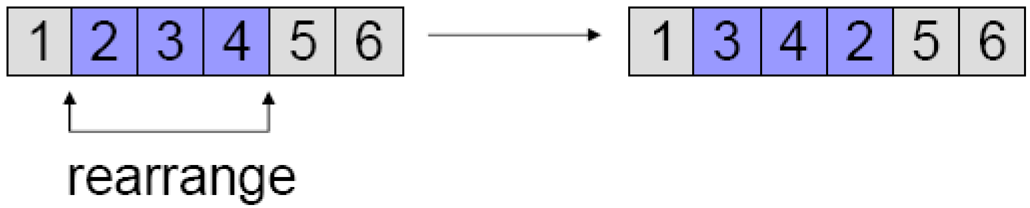


Examples of Mutation Operators on Permutations

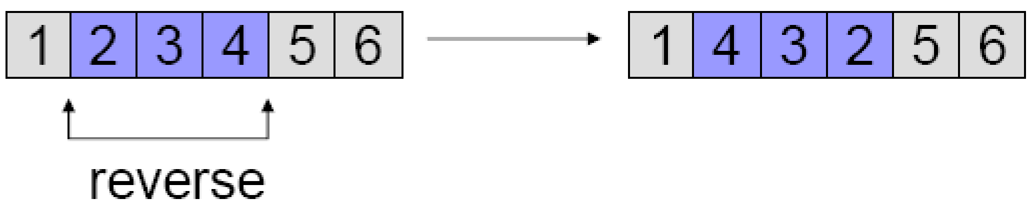
Swap:



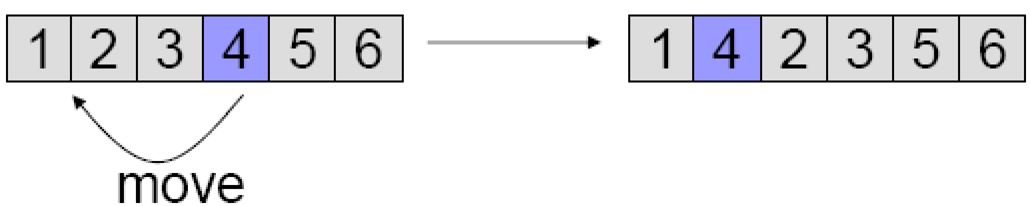
Scramble:



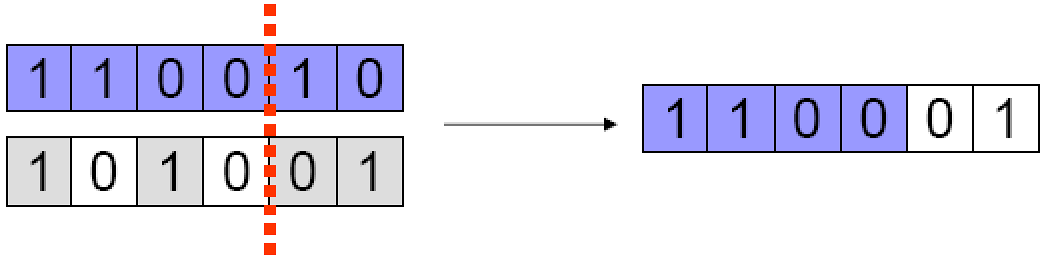
Invert:



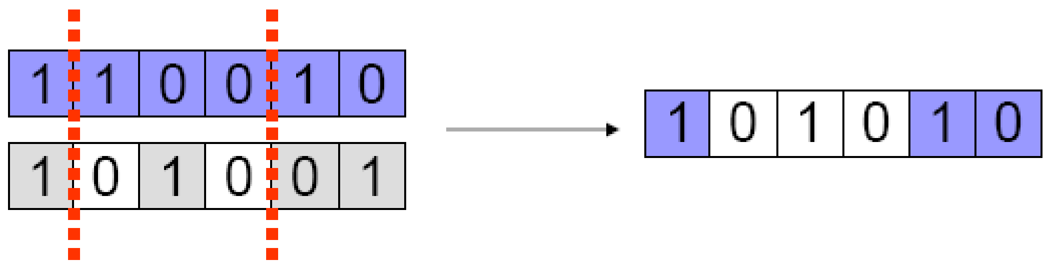
Insert:



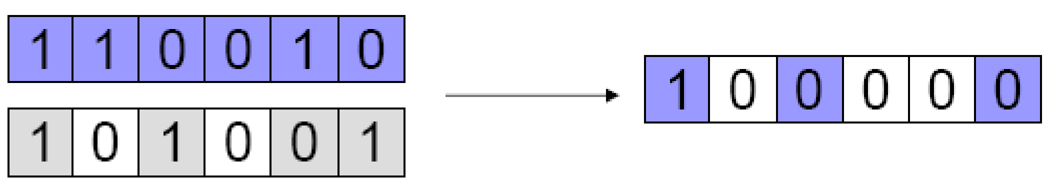
1-point crossover



n-point crossover



uniform crossover



choose each bit independently from one parent or another

A Canonical Genetic Algorithm

- binary search space, maximization
- uniform initialization
- generational cycle: of the population
  - evaluation of solutions
  - mating selection (e.g. roulette wheel)
  - crossover (e.g. 1-point)
  - environmental selection (e.g. plus-selection)

Conclusions

- EAs are generic algorithms (randomized search heuristics, meta-heuristics, ...) for black box optimization

  no or almost no assumptions on the objective function

- They are typically less efficient than problem-specific (exact) algorithms (in terms of #funevals)

  less differences in the continuous case (as we have seen)

- Allow for an easy and rapid implementation and therefore to find good solutions fast

  easy to incorporate (and recommended!) to incorporate problem-specific knowledge to improve the algorithm

Conclusions

I hope it became clear...

...that heuristics is what we typically can afford in practice (no guarantees and no proofs)

...what are the main ideas behind evolutionary algorithms

...and that evolutionary algorithms and genetic algorithms are no synonyms