AutoML: Beyond AutoML

Overview: Algorithm Configuration

Bernd Bischl   Frank Hutter   Lars Kotthoff
Marius Lindauer   Joaquin Vanschoren

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
    1. the algorithm at hand has parameters that influence its performance

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm
- a limitation of HPO is that we assume that we care only about a single task (i.e., dataset or input to the algorithm)

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm
- a limitation of HPO is that we assume that we care only about a single task (i.e., dataset or input to the algorithm)

$\rightsquigarrow$ Can we find an algorithm's configuration that performs well and robustly across a set of tasks?

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm
- a limitation of HPO is that we assume that we care only about a single task (i.e., dataset or input to the algorithm)

$\rightsquigarrow$ Can we find an algorithm's configuration that performs well and robustly across a set of tasks?

  - A hyperparameter configuration for a set of datasets

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm
- a limitation of HPO is that we assume that we care only about a single task (i.e., dataset or input to the algorithm)

$\rightsquigarrow$ Can we find an algorithm's configuration that performs well and robustly across a set of tasks?

  - A hyperparameter configuration for a set of datasets
  - A parameter configuration of a SAT solver for a set of SAT instances

Generalization of HPO

- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm
- a limitation of HPO is that we assume that we care only about a single task (i.e., dataset or input to the algorithm)

$\rightsquigarrow$ Can we find an algorithm's configuration that performs well and robustly across a set of tasks?

- A hyperparameter configuration for a set of datasets
- A parameter configuration of a SAT solver for a set of SAT instances
- A parameter configuration of an AI planning solver for a set of planning problems
- ...

Generalization of HPO

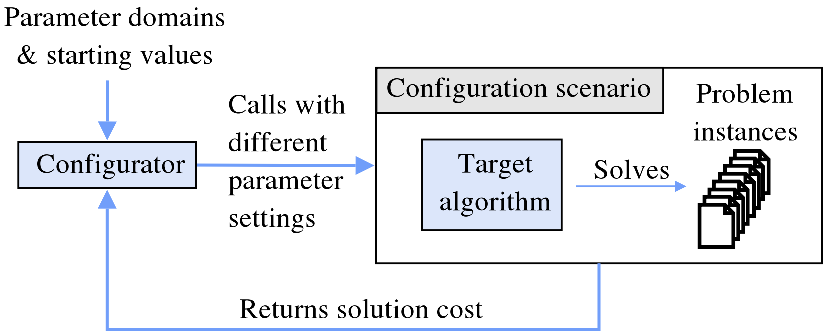
- hyperparameter optimization (HPO) is not limited to ML
- in fact, you can optimize the performance of any algorithm by means of HPO if
  1. the algorithm at hand has parameters that influence its performance
  2. you care about the empirical performance of an algorithm
- a limitation of HPO is that we assume that we care only about a single task (i.e., dataset or input to the algorithm)

$\rightsquigarrow$ Can we find an algorithm's configuration that performs well and robustly across a set of tasks?

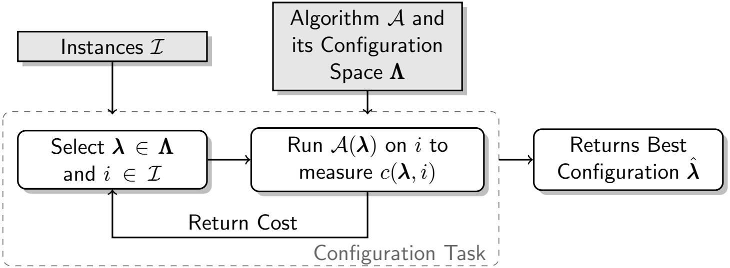
- A hyperparameter configuration for a set of datasets
- A parameter configuration of a SAT solver for a set of SAT instances
- A parameter configuration of an AI planning solver for a set of planning problems
- ...

$\rightsquigarrow$ Algorithm configuration

Algorithm Configuration Visualized



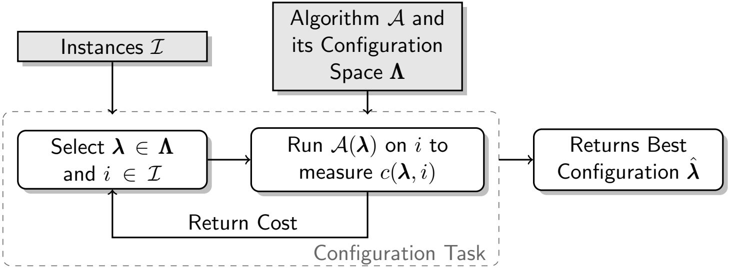
Algorithm Configuration – in More Detail



Definition

Given a parameterized algorithm $\mathcal{A}$ with possible (hyper-)parameter settings $\boldsymbol{\Lambda}$,

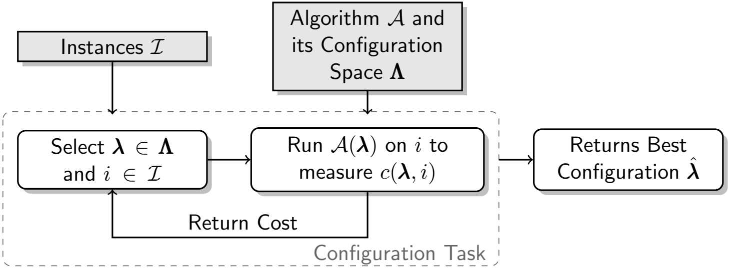
Algorithm Configuration – in More Detail



Definition

Given a parameterized algorithm $\mathcal{A}$ with possible (hyper-)parameter settings $\boldsymbol{\Lambda}$, a set of training problem instances $\mathcal{I}$,

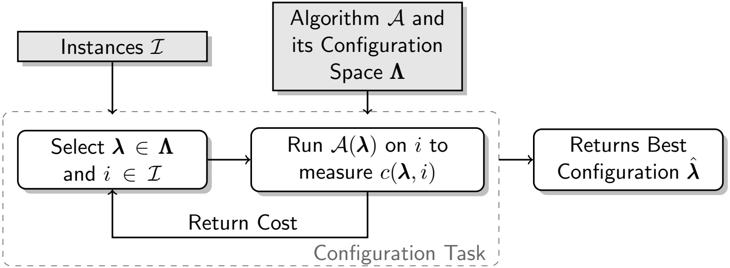
Algorithm Configuration – in More Detail



Definition

Given a parameterized algorithm $\mathcal{A}$ with possible (hyper-)parameter settings $\boldsymbol{\Lambda}$, a set of training problem instances $\mathcal{I}$, and a cost metric $c : \boldsymbol{\Lambda} \times \mathcal{I} \rightarrow \mathbb{R}$,

Algorithm Configuration – in More Detail



Definition

Given a parameterized algorithm $\mathcal{A}$ with possible (hyper-)parameter settings $\boldsymbol{\Lambda}$, a set of training problem instances $\mathcal{I}$, and a cost metric $c : \boldsymbol{\Lambda} \times \mathcal{I} \to \mathbb{R}$, the algorithm configuration problem is to find a parameter configuration $\boldsymbol{\lambda}^* \in \boldsymbol{\Lambda}$ that minimizes c across the instances in $\mathcal{I}$.

Algorithm Configuration – Full Formal Definition

Definition

An instance of the algorithm configuration problem is a 5-tuple $(\mathcal{A}, \mathbf{\Lambda}, \mathcal{D}, \kappa, c)$ where:

- $\mathcal{A}$ is a parameterized algorithm;
- $\mathbf{\Lambda}$ is the (hyper-)parameter configuration space of $\mathcal{A}$;
- $\mathcal{D}$ is a distribution over problem instances with domain $\mathcal{I}$;

Algorithm Configuration – Full Formal Definition

Definition

An instance of the algorithm configuration problem is a 5-tuple $(\mathcal{A}, \mathbf{\Lambda}, \mathcal{D}, \kappa, c)$ where:

- $\mathcal{A}$ is a parameterized algorithm;
- $\mathbf{\Lambda}$ is the (hyper-)parameter configuration space of $\mathcal{A}$;
- $\mathcal{D}$ is a distribution over problem instances with domain $\mathcal{I}$;
- $\kappa < \infty$ is a cutoff time, after which each run of $\mathcal{A}$ will be terminated if still running

Algorithm Configuration – Full Formal Definition

Definition

An instance of the algorithm configuration problem is a 5-tuple $(\mathcal{A}, \mathbf{\Lambda}, \mathcal{D}, \kappa, c)$ where:

- $\mathcal{A}$ is a parameterized algorithm;
- $\mathbf{\Lambda}$ is the (hyper-)parameter configuration space of $\mathcal{A}$;
- $\mathcal{D}$ is a distribution over problem instances with domain $\mathcal{I}$;
- $\kappa < \infty$ is a cutoff time, after which each run of $\mathcal{A}$ will be terminated if still running
- $c : \mathbf{\Lambda} \times \mathcal{I} \to \mathbb{R}$ is a function that measures the observed cost of running $\mathcal{A}(\boldsymbol{\lambda})$ on an instance $i \in \mathcal{I}$ with cutoff time κ

Algorithm Configuration – Full Formal Definition

Definition

An instance of the algorithm configuration problem is a 5-tuple $(\mathcal{A}, \boldsymbol{\Lambda}, \mathcal{D}, \kappa, c)$ where:

- $\mathcal{A}$ is a parameterized algorithm;
- $\boldsymbol{\Lambda}$ is the (hyper-)parameter configuration space of $\mathcal{A}$;
- $\mathcal{D}$ is a distribution over problem instances with domain $\mathcal{I}$;
- $\kappa < \infty$ is a cutoff time, after which each run of $\mathcal{A}$ will be terminated if still running
- $c : \boldsymbol{\Lambda} \times \mathcal{I} \to \mathbb{R}$ is a function that measures the observed cost of running $\mathcal{A}(\boldsymbol{\lambda})$ on an instance $i \in \mathcal{I}$ with cutoff time κ

The cost of a candidate solution $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ is $f(\boldsymbol{\lambda}) = \mathbb{E}_{i \sim \mathcal{D}}(c(\boldsymbol{\lambda}, i))$.
The goal is to find $\boldsymbol{\lambda}^* \in \arg\min_{\boldsymbol{\lambda} \in \boldsymbol{\Lambda}} f(\boldsymbol{\lambda})$.

Distribution of Instances

We usually have a finite number of instances from a given application

- We want to do well on that type of instances
- Future instances of this type should be solved well

Distribution of Instances

We usually have a finite number of instances from a given application

- We want to do well on that type of instances
- Future instances of this type should be solved well

Like in machine learning

- We split the instances into a training set and a test set
- We configure algorithms on the training instances
- We only use the test instances afterwards

  $\rightarrow$ unbiased estimate of generalization performance for unseen instances

Challenges of Algorithm Configuration

- Structured high-dimensional parameter space
  - categorical vs. continuous parameters
  - conditionals between parameters

Challenges of Algorithm Configuration

- Structured high-dimensional parameter space
  - categorical vs. continuous parameters
  - conditionals between parameters

- Stochastic optimization
  - Randomized algorithms: optimization across various seeds
  - Distribution of benchmark instances (often wide range of hardness)
  - Subsumes so-called *multi-armed bandit problem*

Challenges of Algorithm Configuration

- Structured high-dimensional parameter space
  - categorical vs. continuous parameters
  - conditionals between parameters

- Stochastic optimization
  - Randomized algorithms: optimization across various seeds
  - Distribution of benchmark instances (often wide range of hardness)
  - Subsumes so-called *multi-armed bandit problem*

- Generalization across instances
  - apply algorithm configuration to homogeneous instance sets
  - Instance sets can also be heterogeneous,
    i.e., no single configuration performs well on all instances

  $\rightsquigarrow$ combination of algorithm configuration and selection

Challenges of Algorithm Configuration

- Structured high-dimensional parameter space
  - categorical vs. continuous parameters
  - conditionals between parameters
- Stochastic optimization
  - Randomized algorithms: optimization across various seeds
  - Distribution of benchmark instances (often wide range of hardness)
  - Subsumes so-called *multi-armed bandit problem*
- Generalization across instances
  - apply algorithm configuration to homogeneous instance sets
  - Instance sets can also be heterogeneous,
    i.e., no single configuration performs well on all instances

  $\leadsto$ combination of algorithm configuration and selection

$\leadsto$ Hyperparameter optimization is a subproblem of algorithm configuration

[Eggensperger et al. 2019]

AutoML: Beyond AutoML

Racing for Algorithm Configuration

Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

State-of-the-art Algorithm Configuration

SMAC: Sequential Model-based Algorithm Configuration [Hutter et al. 2011]

- Bayesian Optimization +
- aggressive racing +
- adaptive capping (for optimizing runtime)

SMAC: Overview [Hutter et al. 2011]

Algorithm 1 SMAC

Input : instance set $\mathcal{I}$, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration $\boldsymbol{\lambda}_0$, performance metric c, Configuration budget b

run history $\mathcal{D}_{\text{Hist}} \leftarrow$ initial design based on $\boldsymbol{\lambda}_0$; // $\mathcal{D}_{\texttt{Hist}} = (\boldsymbol{\lambda}, i, c(i, \boldsymbol{\lambda}))_i$

while b *remains* **do**

SMAC: Overview [Hutter et al. 2011]

Algorithm 2 SMAC

Input : instance set $\mathcal{I}$, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration $\boldsymbol{\lambda}_0$, performance metric c, Configuration budget b

run history $\mathcal{D}_{\text{Hist}} \leftarrow$ initial design based on $\boldsymbol{\lambda}_0$; // $\mathcal{D}_{\texttt{Hist}} = (\boldsymbol{\lambda}, i, c(i, \boldsymbol{\lambda}))_i$

while b *remains* **do**

  $\hat{c} \leftarrow$ train empirical performance model based on run history $\mathcal{D}_{\text{Hist}}$;

SMAC: Overview [Hutter et al. 2011]

Algorithm 3 SMAC

Input : instance set $\mathcal{I}$, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration $\boldsymbol{\lambda}_0$, performance metric c, Configuration budget b

run history $\mathcal{D}_{\text{Hist}} \leftarrow$ initial design based on $\boldsymbol{\lambda}_0$; // $\mathcal{D}_{\texttt{Hist}} = (\boldsymbol{\lambda}, i, c(i, \boldsymbol{\lambda}))_i$

while b *remains* **do**

  $\hat{c} \leftarrow$ train empirical performance model based on run history $\mathcal{D}_{\text{Hist}}$;

  $\mathbf{\Lambda}_{challengers} \leftarrow$ select configurations based on $\hat{c}$;

SMAC: Overview [Hutter et al. 2011]

Algorithm 4 SMAC

Input : instance set $\mathcal{I}$, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration $\boldsymbol{\lambda}_0$, performance metric c, Configuration budget b

run history $\mathcal{D}_{\text{Hist}} \leftarrow$ initial design based on $\boldsymbol{\lambda}_0$; // $\mathcal{D}_{\texttt{Hist}} = (\boldsymbol{\lambda}, i, c(i, \boldsymbol{\lambda}))_i$

while b *remains* **do**

- $\hat{c} \leftarrow$ train empirical performance model based on run history $\mathcal{D}_{\text{Hist}}$;
- $\mathbf{\Lambda}_{challengers} \leftarrow$ select configurations based on $\hat{c}$;
- $\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\text{Hist}} \leftarrow$ intensify($\mathbf{\Lambda}_{challengers}$, $\hat{\boldsymbol{\lambda}}$); // `racing and capping`

SMAC: Overview [Hutter et al. 2011]

Algorithm 5 SMAC

Input : instance set $\mathcal{I}$, Algorithm $\mathcal{A}$ with configuration space $\mathbf{\Lambda}$, Initial configuration $\boldsymbol{\lambda}_0$, performance metric c, Configuration budget b

run history $\mathcal{D}_{\text{Hist}} \leftarrow$ initial design based on $\boldsymbol{\lambda}_0$; // $\mathcal{D}_{\texttt{Hist}} = (\boldsymbol{\lambda}, i, c(i, \boldsymbol{\lambda}))_i$

while b *remains* **do**

  $\hat{c} \leftarrow$ train empirical performance model based on run history $\mathcal{D}_{\text{Hist}}$;

  $\mathbf{\Lambda}_{challengers} \leftarrow$ select configurations based on $\hat{c}$;

  $\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\text{Hist}} \leftarrow$ intensify($\mathbf{\Lambda}_{challengers}$, $\hat{\boldsymbol{\lambda}}$); // racing and capping

return $\hat{\boldsymbol{\lambda}}$

Comparisons on N instances [Hutter et al. 2009]

- Basic(N) uses a pretty basic comparison: *better*$_N(\boldsymbol{\lambda}', \boldsymbol{\lambda})$:
  - Compare $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}$ based on N instances

Comparisons on N instances [Hutter et al. 2009]

- Basic(N) uses a pretty basic comparison: *better*$_N(\boldsymbol{\lambda}', \boldsymbol{\lambda})$:
  - Compare $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}$ based on N instances
  - How does this relate to cross-validation?

Comparisons on N instances [Hutter et al. 2009]

- Basic(N) uses a pretty basic comparison: *better*$_N(\boldsymbol{\lambda}', \boldsymbol{\lambda})$:
  - Compare $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}$ based on N instances
  - How does this relate to cross-validation?

- Problem: How to set N? Problems of large N? Small N?

Comparisons on N instances [Hutter et al. 2009]

- Basic(N) uses a pretty basic comparison: *better*$_N(\boldsymbol{\lambda}', \boldsymbol{\lambda})$:
  - Compare $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}$ based on N instances
  - How does this relate to cross-validation?

- Problem: How to set N? Problems of large N? Small N?
  - Problem of large N: evaluations are slow
  - Problem of small N: overfitting to a small set of instances

  $\leadsto$ Tradeoff: Choose N of moderate size

Comparisons on N instances [Hutter et al. 2009]

Question: Which N instances should we use?

1. N different instances for each configuration
2. The same set of N instances for the entire run

Comparisons on N instances [Hutter et al. 2009]

Question: Which N instances should we use?

1. N different instances for each configuration
2. The same set of N instances for the entire run

Answer: the same N instances, so that we compare apples with apples

(but: using the same instances can also yield overtuning)

Comparisons on N instances [Hutter et al. 2009]

Question: Which N instances should we use?

1. N different instances for each configuration
2. The same set of N instances for the entire run

Answer: the same N instances, so that we compare apples with apples

(but: using the same instances can also yield overtuning)

If we sampled different instances for each configuration:

- Some configurations would randomly get easier instances
- Those configurations would look better than they really are

Comparisons on N instances [Hutter et al. 2009]

Question: For randomized algorithms, how should we set the seeds?

1. Sample a new seed for each algorithm run
2. Fix the seeds together with the instances

Comparisons on N instances [Hutter et al. 2009]

Question: For randomized algorithms, how should we set the seeds?

1. Sample a new seed for each algorithm run
2. Fix the seeds together with the instances

Answer: just like for instances, fix them to compare apples to apples

Comparisons on N instances [Hutter et al. 2009]

Question: For randomized algorithms, how should we set the seeds?

1. Sample a new seed for each algorithm run
2. Fix the seeds together with the instances

Answer: just like for instances, fix them to compare apples to apples

In summary, for each run of Basic(N):
pick N (instance, seed) pairs and use them for evaluating each $\boldsymbol{\lambda}$.

Comparisons on N instances [Hutter et al. 2009]

Question: For randomized algorithms, how should we set the seeds?

1. Sample a new seed for each algorithm run
2. Fix the seeds together with the instances

Answer: just like for instances, fix them to compare apples to apples

In summary, for each run of Basic(N):
pick N (instance, seed) pairs and use them for evaluating each $\boldsymbol{\lambda}$.
(Different Basic(N) runs can use different instances and seeds.)

The concept of overtuning

Very related to overfitting in machine learning

- Performance improves on the training set
- Performance does not improve on the test set, and may even degrade

The concept of overtuning

Very related to overfitting in machine learning

- Performance improves on the training set
- Performance does not improve on the test set, and may even degrade

More pronounced for heterogeneous benchmark sets

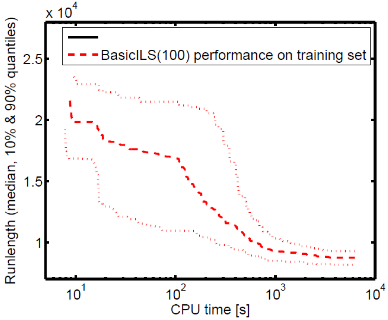
- But it even happens for very homogeneous sets
- Indeed, one can even overfit on a single instance, to the seeds used for training

Overtuning Visualized

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=100:
  average runlengths across 100 runs with different seeds
- Test cost of $\hat{\boldsymbol{\lambda}}$ here based on 1000 new seeds

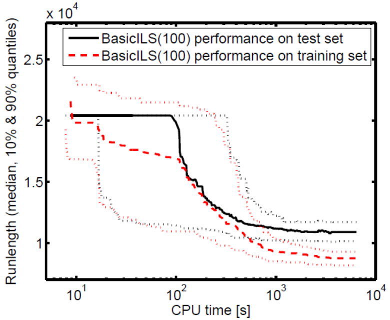
Overtuning Visualized

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=100:
  average runlengths across 100 runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds



Overtuning Visualized

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=100:
  average runlengths across 100 runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds

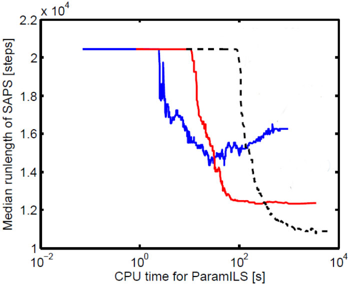


Basic(N) Test Results with Various N

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\boldsymbol{\lambda}}$ here based on 1000 new seeds

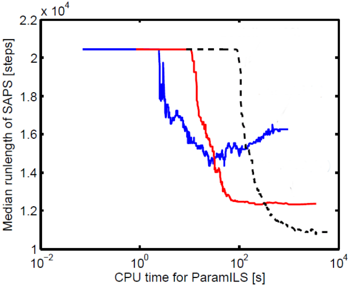
Basic(N) Test Results with Various N

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds



Basic(N) Test Results with Various N

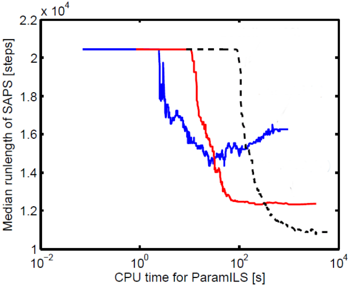
- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds



Which of these results corresponds to $N = 1$, $N = 10$, and $N = 100$?

Basic(N) Test Results with Various N

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\boldsymbol{\lambda}}$ here based on 1000 new seeds

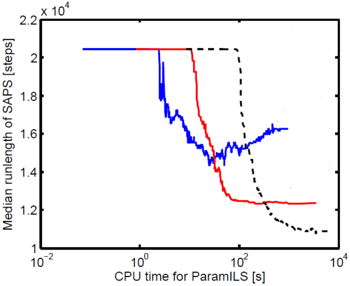


Which of these results corresponds to $N = 1$, $N = 10$, and $N = 100$?

1. N=1: blue, N=10: red, N=100 dashed black
2. N=1: dashed black, N=10: red, N=100 blue

Basic(N) Test Results with Various N

- Example: minimizing SLS solver runlengths for a single SAT instance
- Training cost, e.g., with N=?:
  average runlengths across N runs with different seeds
- Test cost of $\hat{\lambda}$ here based on 1000 new seeds



Which of these results corresponds to $N = 1$, $N = 10$, and $N = 100$?

1. N=1: blue, N=10: red, N=100 dashed black
2. N=1: dashed black, N=10: red, N=100 blue

Correct Answer: 1

Aggressive Racing (inspired by FocusedILS) [Hutter et al. 2009]

Intuition: get the best of both worlds

- Perform more runs for good configurations
  - to avoid overtuning
- Quickly reject poor configurations
  - to make progress more quickly

Aggressive Racing (inspired by FocusedILS) [Hutter et al. 2009]

Intuition: get the best of both worlds

- Perform more runs for good configurations
  - to avoid overtuning
- Quickly reject poor configurations
  - to make progress more quickly

Definition: $N(\boldsymbol{\lambda})$ and $c_N(\boldsymbol{\lambda})$

$N(\boldsymbol{\lambda})$ denotes the number of runs executed for $\boldsymbol{\lambda}$ so far.
$\hat{c}_N(\boldsymbol{\lambda})$ denotes the cost estimate of $\boldsymbol{\lambda}$ based on N runs.

Aggressive Racing (inspired by FocusedILS) [Hutter et al. 2009]

Intuition: get the best of both worlds

- Perform more runs for good configurations
  - to avoid overtuning
- Quickly reject poor configurations
  - to make progress more quickly

Definition: $N(\boldsymbol{\lambda})$ and $c_N(\boldsymbol{\lambda})$

$N(\boldsymbol{\lambda})$ denotes the number of runs executed for $\boldsymbol{\lambda}$ so far.
$\hat{c}_N(\boldsymbol{\lambda})$ denotes the cost estimate of $\boldsymbol{\lambda}$ based on N runs.

In the beginning: $N(\boldsymbol{\lambda}) = 0$ for every configuration $\boldsymbol{\lambda}$

Aggressive Racing (inspired by FocusedILS) [Hutter et al. 2009]

Definition: domination

$\boldsymbol{\lambda}^{(1)}$ dominates $\boldsymbol{\lambda}^{(2)}$ if

- $N(\boldsymbol{\lambda}^{(1)}) \geq N(\boldsymbol{\lambda}^{(2)})$ and
- $\hat{c}_{N(\boldsymbol{\lambda}^{(2)})}(\boldsymbol{\lambda}^{(1)}) \leq \hat{c}_{N(\boldsymbol{\lambda}^{(2)})}(\boldsymbol{\lambda}^{(2)})$.

I.e.: we have at least as many runs for $\boldsymbol{\lambda}^{(1)}$ and its cost is at least as low.

Aggressive Racing (inspired by FocusedILS) [Hutter et al. 2009]

Definition: domination

$\boldsymbol{\lambda}^{(1)}$ dominates $\boldsymbol{\lambda}^{(2)}$ if

- $N(\boldsymbol{\lambda}^{(1)}) \geq N(\boldsymbol{\lambda}^{(2)})$ and
- $\hat{c}_{N(\boldsymbol{\lambda}^{(2)})}(\boldsymbol{\lambda}^{(1)}) \leq \hat{c}_{N(\boldsymbol{\lambda}^{(2)})}(\boldsymbol{\lambda}^{(2)})$.

I.e.: we have at least as many runs for $\boldsymbol{\lambda}^{(1)}$ and its cost is at least as low.

better$(\boldsymbol{\lambda}', \hat{\boldsymbol{\lambda}})$ in a nutshell

- $\hat{\boldsymbol{\lambda}}$ is the current configuration to beat (incumbent)

Aggressive Racing (inspired by FocusedILS) [Hutter et al. 2009]

Definition: domination

$\boldsymbol{\lambda}^{(1)}$ dominates $\boldsymbol{\lambda}^{(2)}$ if

- $N(\boldsymbol{\lambda}^{(1)}) \geq N(\boldsymbol{\lambda}^{(2)})$ and
- $\hat{c}_{N(\boldsymbol{\lambda}^{(2)})}(\boldsymbol{\lambda}^{(1)}) \leq \hat{c}_{N(\boldsymbol{\lambda}^{(2)})}(\boldsymbol{\lambda}^{(2)})$.

I.e.: we have at least as many runs for $\boldsymbol{\lambda}^{(1)}$ and its cost is at least as low.

better($\boldsymbol{\lambda}', \hat{\boldsymbol{\lambda}}$) in a nutshell

- $\hat{\boldsymbol{\lambda}}$ is the current configuration to beat (incumbent)
- Perform runs of $\boldsymbol{\lambda}'$ until either
  - $\hat{\boldsymbol{\lambda}}$ dominates $\boldsymbol{\lambda}'$ $\rightsquigarrow$ reject $\boldsymbol{\lambda}'$, or
  - $\boldsymbol{\lambda}'$ dominates $\hat{\boldsymbol{\lambda}}$ $\rightsquigarrow$ change current incumbent ($\hat{\boldsymbol{\lambda}} \leftarrow \boldsymbol{\lambda}'$)

Aggressive Racing (inspired by FocusedILS) [Hutter et al. 2009]

Definition: domination

$\boldsymbol{\lambda}^{(1)}$ dominates $\boldsymbol{\lambda}^{(2)}$ if

- $N(\boldsymbol{\lambda}^{(1)}) \geq N(\boldsymbol{\lambda}^{(2)})$ and
- $\hat{c}_{N(\boldsymbol{\lambda}^{(2)})}(\boldsymbol{\lambda}^{(1)}) \leq \hat{c}_{N(\boldsymbol{\lambda}^{(2)})}(\boldsymbol{\lambda}^{(2)})$.

I.e.: we have at least as many runs for $\boldsymbol{\lambda}^{(1)}$ and its cost is at least as low.

better($\boldsymbol{\lambda}', \hat{\boldsymbol{\lambda}}$) in a nutshell

- $\hat{\boldsymbol{\lambda}}$ is the current configuration to beat (incumbent)
- Perform runs of $\boldsymbol{\lambda}'$ until either
  - $\hat{\boldsymbol{\lambda}}$ dominates $\boldsymbol{\lambda}'$ $\rightsquigarrow$ reject $\boldsymbol{\lambda}'$, or
  - $\boldsymbol{\lambda}'$ dominates $\hat{\boldsymbol{\lambda}}$ $\rightsquigarrow$ change current incumbent ($\hat{\boldsymbol{\lambda}} \leftarrow \boldsymbol{\lambda}'$)
- Over time: perform extra runs of $\hat{\boldsymbol{\lambda}}$ to gain more confidence in it

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10
$\boldsymbol{\lambda}'$			

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10
$\boldsymbol{\lambda}'$	2		

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10
$\boldsymbol{\lambda}'$	2	10	

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10
$\boldsymbol{\lambda}'$	2	10	
	$\rightarrow$ reject, since $\hat{c}_2(\boldsymbol{\lambda}') = 6 > \hat{c}_2(\hat{\boldsymbol{\lambda}}) = 2.5$		

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10
$\boldsymbol{\lambda}'$	2	10	
		$\rightarrow$ reject, since $\hat{c}_2(\boldsymbol{\lambda}') = 6 > \hat{c}_2(\hat{\boldsymbol{\lambda}}) = 2.5$	
$\boldsymbol{\lambda}''$	3		

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10
$\boldsymbol{\lambda}'$	2	10	
	$\rightarrow$ reject, since $\hat{c}_2(\boldsymbol{\lambda}') = 6 > \hat{c}_2(\hat{\boldsymbol{\lambda}}) = 2.5$		
$\boldsymbol{\lambda}''$	3	1	

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10
$\boldsymbol{\lambda}'$	2	10	
	$\rightarrow$ reject, since $\hat{c}_2(\boldsymbol{\lambda}') = 6 > \hat{c}_2(\hat{\boldsymbol{\lambda}}) = 2.5$		
$\boldsymbol{\lambda}''$	3	1	5

Toy Example

- Let $\hat{\boldsymbol{\lambda}}$ be the incumbent (evaluated on $i^{(1)}, i^{(2)}, i^{(3)}$)
- We'll look at challengers $\boldsymbol{\lambda}'$ and $\boldsymbol{\lambda}''$

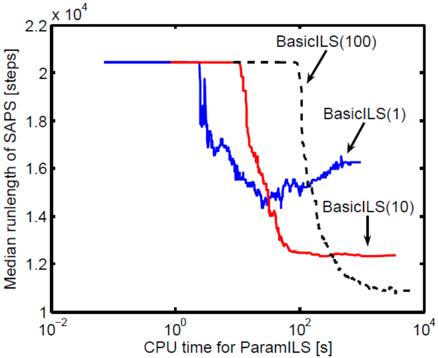
	$i^{(1)}$	$i^{(2)}$	$i^{(3)}$
$\hat{\boldsymbol{\lambda}}$	3	2	10
$\boldsymbol{\lambda}'$	2	10	
	$\rightarrow$ reject, since $\hat{c}_2(\boldsymbol{\lambda}') = 6 > \hat{c}_2(\hat{\boldsymbol{\lambda}}) = 2.5$		
$\boldsymbol{\lambda}''$	3	1	5

- new incumbent: $\hat{\boldsymbol{\lambda}} \leftarrow \boldsymbol{\lambda}''$
- Perform an additional run for new $\hat{\boldsymbol{\lambda}}$ to increase confidence over time

Racing achieves the best of both worlds

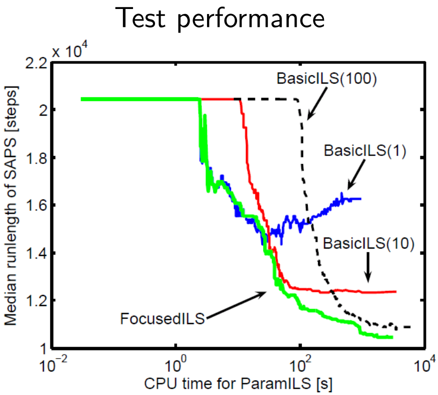
Aggressive racing (aka FocusedILS): Fast progress and no overtuning

Test performance



Racing achieves the best of both worlds

Aggressive racing (aka FocusedILS): Fast progress and no overtuning



Overview of Racing

Input : candidate configurations $\boldsymbol{\Lambda}_{new}$, cutoff κ_{max}, previously evaluated runs $\mathcal{D}_{\mathsf{Hist}}$, budget T, incumbent $\hat{\boldsymbol{\lambda}}$

while $\boldsymbol{\Lambda}_{new}$ *not empty* **do**

  $\boldsymbol{\lambda}^{(t)} \leftarrow$ getNext($\boldsymbol{\Lambda}_{new}$);

Overview of Racing

Input : candidate configurations $\boldsymbol{\Lambda}_{new}$, cutoff κ_{max}, previously evaluated runs $\mathcal{D}_{\text{Hist}}$, budget T, incumbent $\hat{\boldsymbol{\lambda}}$

while $\boldsymbol{\Lambda}_{new}$ *not empty* **do**

  $\boldsymbol{\lambda}^{(t)} \leftarrow$ getNext($\boldsymbol{\Lambda}_{new}$);

  $i, s \leftarrow$ instance and seed drawn uniformly at random;

  $c \leftarrow$ EvaluateRun($\hat{\boldsymbol{\lambda}}, i, s, \kappa_{max}$);

  $\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\hat{\boldsymbol{\lambda}}, i, s, c)$;

Overview of Racing

Input : candidate configurations $\mathbf{\Lambda}_{new}$, cutoff κ_{max}, previously evaluated runs $\mathcal{D}_{\text{Hist}}$, budget T, incumbent $\hat{\boldsymbol{\lambda}}$

while $\mathbf{\Lambda}_{new}$ *not empty* **do**

$\boldsymbol{\lambda}^{(t)} \leftarrow$ getNext($\mathbf{\Lambda}_{new}$);

$i, s \leftarrow$ instance and seed drawn uniformly at random;

$c \leftarrow$ EvaluateRun($\hat{\boldsymbol{\lambda}}, i, s, \kappa_{max}$);

$\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\hat{\boldsymbol{\lambda}}, i, s, c)$;

while *true* **do**

$\mathcal{I}^{+}, \mathbf{s}^{+} \leftarrow$getAlreadyEvaluatedOn($\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\text{Hist}}$);

$\mathcal{I}^{(t)}, \mathbf{s}^{(t)} \leftarrow$getAlreadyEvaluatedOn($\boldsymbol{\lambda}^{(t)}, \mathcal{D}_{\text{Hist}}$);

$i^{(t)}, s^{(t)} \leftarrow$ drawn uniformly at random from $\mathcal{I}^{+} \setminus \mathcal{I}^{(t)}$ and $\mathbf{s}^{+} \setminus \mathbf{s}^{(\mathbf{t})}$;

Overview of Racing

Input : candidate configurations $\mathbf{\Lambda}_{new}$, cutoff κ_{max}, previously evaluated runs $\mathcal{D}_{\text{Hist}}$, budget T, incumbent $\hat{\boldsymbol{\lambda}}$

while $\mathbf{\Lambda}_{new}$ *not empty* **do**

$\boldsymbol{\lambda}^{(t)} \leftarrow$ getNext($\mathbf{\Lambda}_{new}$);

$i, s \leftarrow$ instance and seed drawn uniformly at random;

$c \leftarrow$ EvaluateRun($\hat{\boldsymbol{\lambda}}, i, s, \kappa_{max}$);

$\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\hat{\boldsymbol{\lambda}}, i, s, c)$;

while *true* **do**

$\mathcal{I}^{+}, \mathbf{s}^{+} \leftarrow$getAlreadyEvaluatedOn($\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\text{Hist}}$);

$\mathcal{I}^{(t)}, \mathbf{s}^{(t)} \leftarrow$getAlreadyEvaluatedOn($\boldsymbol{\lambda}^{(t)}, \mathcal{D}_{\text{Hist}}$);

$i^{(t)}, s^{(t)} \leftarrow$ drawn uniformly at random from $\mathcal{I}^{+} \setminus \mathcal{I}^{(t)}$ and $\mathbf{s}^{+} \setminus \mathbf{s}^{(\mathbf{t})}$;

$c_i \leftarrow$ EvaluateRun($\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, \kappa_{max}$);

$\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, c^{(t)})$;

Overview of Racing

Input : candidate configurations $\mathbf{\Lambda}_{new}$, cutoff κ_{max}, previously evaluated runs $\mathcal{D}_{\text{Hist}}$, budget T, incumbent $\hat{\boldsymbol{\lambda}}$

while $\mathbf{\Lambda}_{new}$ *not empty* **do**

$\boldsymbol{\lambda}^{(t)} \leftarrow \text{getNext}(\mathbf{\Lambda}_{new})$;

$i, s \leftarrow$ instance and seed drawn uniformly at random;

$c \leftarrow \text{EvaluateRun}(\hat{\boldsymbol{\lambda}}, i, s, \kappa_{max})$;

$\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\hat{\boldsymbol{\lambda}}, i, s, c)$;

while *true* **do**

$\mathcal{I}^{+}, \mathbf{s}^{+} \leftarrow \text{getAlreadyEvaluatedOn}(\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\text{Hist}})$;

$\mathcal{I}^{(t)}, \mathbf{s}^{(t)} \leftarrow \text{getAlreadyEvaluatedOn}(\boldsymbol{\lambda}^{(t)}, \mathcal{D}_{\text{Hist}})$;

$i^{(t)}, s^{(t)} \leftarrow$ drawn uniformly at random from $\mathcal{I}^{+} \setminus \mathcal{I}^{(t)}$ and $\mathbf{s}^{+} \setminus \mathbf{s}^{(\mathbf{t})}$;

$c_i \leftarrow \text{EvaluateRun}(\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, \kappa_{max})$;

$\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, c^{(t)})$;

if *average cost of* $\boldsymbol{\lambda}^{(t)} >$ *average cost of* $\hat{\boldsymbol{\lambda}}$ *across* $\mathcal{I}^{(t)}$ *and* $\mathbf{s}^{(t)}$ **then**

break;

Overview of Racing

Input : candidate configurations $\mathbf{\Lambda}_{new}$, cutoff κ_{max}, previously evaluated runs $\mathcal{D}_{\text{Hist}}$, budget T, incumbent $\hat{\boldsymbol{\lambda}}$

while $\mathbf{\Lambda}_{new}$ *not empty* **do**

$\boldsymbol{\lambda}^{(t)} \leftarrow$ getNext($\mathbf{\Lambda}_{new}$);

$i, s \leftarrow$ instance and seed drawn uniformly at random;

$c \leftarrow$ EvaluateRun($\hat{\boldsymbol{\lambda}}, i, s, \kappa_{max}$);

$\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\hat{\boldsymbol{\lambda}}, i, s, c)$;

while *true* **do**

$\mathcal{I}^{+}, \mathbf{s}^{+} \leftarrow$getAlreadyEvaluatedOn($\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\text{Hist}}$);

$\mathcal{I}^{(t)}, \mathbf{s}^{(t)} \leftarrow$getAlreadyEvaluatedOn($\boldsymbol{\lambda}^{(t)}, \mathcal{D}_{\text{Hist}}$);

$i^{(t)}, s^{(t)} \leftarrow$ drawn uniformly at random from $\mathcal{I}^{+} \setminus \mathcal{I}^{(t)}$ and $\mathbf{s}^{+} \setminus \mathbf{s}^{(\mathbf{t})}$;

$c_i \leftarrow$ EvaluateRun($\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, \kappa_{max}$);

$\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, c^{(t)})$;

if *average cost of* $\boldsymbol{\lambda}^{(t)} >$ *average cost of* $\hat{\boldsymbol{\lambda}}$ *across* $\mathcal{I}^{(t)}$ *and* $\mathbf{s}^{(t)}$ **then**

break;

else if *average cost of* $\boldsymbol{\lambda}^{(t)} <$ *average cost of* $\hat{\boldsymbol{\lambda}}$ *and* $\mathcal{I}^{+} = \mathcal{I}^{(t)}$ *and* $\mathbf{s}^{(t)} = \mathbf{s}^{+}$ **then**

$\hat{\boldsymbol{\lambda}} \leftarrow \boldsymbol{\lambda}^{(t)}$;

Overview of Racing

Input : candidate configurations $\mathbf{\Lambda}_{new}$, cutoff κ_{max}, previously evaluated runs $\mathcal{D}_{\text{Hist}}$, budget T, incumbent $\hat{\boldsymbol{\lambda}}$

while $\mathbf{\Lambda}_{new}$ *not empty* **do**

- $\boldsymbol{\lambda}^{(t)} \leftarrow$ getNext($\mathbf{\Lambda}_{new}$);
- $i, s \leftarrow$ instance and seed drawn uniformly at random;
- $c \leftarrow$ EvaluateRun($\hat{\boldsymbol{\lambda}}, i, s, \kappa_{max}$);
- $\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\hat{\boldsymbol{\lambda}}, i, s, c)$;
- **while** *true* **do**
  - $\mathcal{I}^{+}, \mathbf{s}^{+} \leftarrow$getAlreadyEvaluatedOn($\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\text{Hist}}$);
  - $\mathcal{I}^{(t)}, \mathbf{s}^{(t)} \leftarrow$getAlreadyEvaluatedOn($\boldsymbol{\lambda}^{(t)}, \mathcal{D}_{\text{Hist}}$);
  - $i^{(t)}, s^{(t)} \leftarrow$ drawn uniformly at random from $\mathcal{I}^{+} \setminus \mathcal{I}^{(t)}$ and $\mathbf{s}^{+} \setminus \mathbf{s}^{(\mathbf{t})}$;
  - $c_i \leftarrow$ EvaluateRun($\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, \kappa_{max}$);
  - $\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, c^{(t)})$;
  - **if** *average cost of* $\boldsymbol{\lambda}^{(t)} >$ *average cost of* $\hat{\boldsymbol{\lambda}}$ *across* $\mathcal{I}^{(t)}$ *and* $\mathbf{s}^{(t)}$ **then**
    - break;
  - **else if** *average cost of* $\boldsymbol{\lambda}^{(t)} <$ *average cost of* $\hat{\boldsymbol{\lambda}}$ *and* $\mathcal{I}^{+} = \mathcal{I}^{(t)}$ *and* $\mathbf{s}^{(t)} = \mathbf{s}^{+}$ **then**
    - $\hat{\boldsymbol{\lambda}} \leftarrow \boldsymbol{\lambda}^{(t)}$;
- **if** *time spent exceeds* T *or* $\mathbf{\Lambda}_{new}$ *is empty* **then**
  - **return** $\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\mathit{Hist}}$

AutoML: Beyond AutoML

Capping of Runtimes

Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

Adaptive capping [Hutter et al. 2009]

- Assumptions
  - optimization of runtime
  - each configuration run has a time limit (e.g., 300 sec)

Adaptive capping [Hutter et al. 2009]

- Assumptions
  - optimization of runtime
  - each configuration run has a time limit (e.g., 300 sec)
- E.g., $\hat{\boldsymbol{\lambda}}$ needed 1 sec to solve i_1
  - Do we need to run $\boldsymbol{\lambda}'$ for 300 sec?
  - Terminate evaluation of $\boldsymbol{\lambda}'$ once guaranteed to be worse than $\hat{\boldsymbol{\lambda}}$

Adaptive capping [Hutter et al. 2009]

- Assumptions
  - optimization of runtime
  - each configuration run has a time limit (e.g., 300 sec)
- E.g., $\hat{\boldsymbol{\lambda}}$ needed 1 sec to solve i_1
  - Do we need to run $\boldsymbol{\lambda}'$ for 300 sec?
  - Terminate evaluation of $\boldsymbol{\lambda}'$ once guaranteed to be worse than $\hat{\boldsymbol{\lambda}}$

$\rightsquigarrow$ To compare against $\hat{\boldsymbol{\lambda}}$ based on N runs,
we can terminate evaluation of $\boldsymbol{\lambda}'$ after time $\sum_{k=1}^{N} c(\hat{\boldsymbol{\lambda}}, i_k)$

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$		

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$	3	

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$	3	300

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$	3	300

$\rightarrow$ reject $\boldsymbol{\lambda}'$ (cost: 303)

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$	3	300
		$\rightarrow$ reject $\boldsymbol{\lambda}'$ (cost: 303)
With adaptive capping		
$\boldsymbol{\lambda}'$		

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$	3	300
		$\rightarrow$ reject $\boldsymbol{\lambda}'$ (cost: 303)
With adaptive capping		
$\boldsymbol{\lambda}'$	3	

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$	3	300
		$\rightarrow$ reject $\boldsymbol{\lambda}'$ (cost: 303)
With adaptive capping		
$\boldsymbol{\lambda}'$	3	300

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$	3	300
		$\rightarrow$ reject $\boldsymbol{\lambda}'$ (cost: 303)
With adaptive capping		
$\boldsymbol{\lambda}'$	3	300
	$\rightarrow$ cut off after $\kappa = 4$ seconds, reject $\boldsymbol{\lambda}'$ (cost: 7)	

Toy-Example: Adaptive capping

runtime cutoff $\kappa = 300$, comparison based on 2 instances

	i_1	i_2
$\hat{\boldsymbol{\lambda}}$	4	2
Without adaptive capping		
$\boldsymbol{\lambda}'$	3	300
		$\rightarrow$ reject $\boldsymbol{\lambda}'$ (cost: 303)
With adaptive capping		
$\boldsymbol{\lambda}'$	3	300
	$\rightarrow$ cut off after $\kappa = 4$ seconds, reject $\boldsymbol{\lambda}'$ (cost: 7)	

Note: To combine adaptive capping with BO, we need to impute the censored observations caused by adaptive capping. [Hutter et al. 2011]

Overview of Racing and Adaptive Capping

Input : candidate configurations $\mathbf{\Lambda}_{new}$, cutoff κ_{max}, previously evaluated runs $\mathcal{D}_{\text{Hist}}$, budget T, incumbent $\hat{\boldsymbol{\lambda}}$

while $\mathbf{\Lambda}_{new}$ *not empty* **do**

  $\boldsymbol{\lambda}^{(t)} \leftarrow$ getNext($\mathbf{\Lambda}_{new}$);

  [... add new run for incumbent ...];

  while *true* **do**

    $\mathcal{I}^+, \mathbf{s}^+ \leftarrow$getAlreadyEvaluatedOn($\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\text{Hist}}$);

    $\mathcal{I}^{(t)}, \mathbf{s}^{(t)} \leftarrow$getAlreadyEvaluatedOn($\boldsymbol{\lambda}^{(t)}, \mathcal{D}_{\text{Hist}}$);

    $i^{(t)}, s^{(t)} \leftarrow$ drawn uniformly at random from $\mathcal{I}_+ \setminus \mathcal{I}^{(t)}$ and $\mathbf{s}^+ \setminus \mathbf{s}^{(\mathbf{t})}$;

    $\kappa^{(i)} \leftarrow$ AdaptCutoff($\kappa_{max}, \langle(\boldsymbol{\lambda}^{(j)}, c^{(j)})\rangle_{\boldsymbol{\lambda}^{(j)}=\boldsymbol{\lambda}^+}$) $\cdot\, \xi$;

    $c_i \leftarrow$ EvaluateRun($\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(i)}, \kappa^{(i)}$);

    $\mathcal{D}_{\text{Hist}} \leftarrow \mathcal{D}_{\text{Hist}} \cup (\boldsymbol{\lambda}^{(t)}, i^{(t)}, s^{(t)}, c^{(t)})$;

    if *average cost of* $\boldsymbol{\lambda}^{(t)} >$ *average cost of* $\hat{\boldsymbol{\lambda}}$ *across* $\mathcal{I}^{(t)}$ *and* $\mathbf{s}^{(t)}$ **then**

      break;

    else if *average cost of* $\boldsymbol{\lambda}^{(t)} <$ *average cost of* $\hat{\boldsymbol{\lambda}}$ *and* $\mathcal{I}^+ = \mathcal{I}^{(t)}$ *and* $\mathbf{s}^{(t)} = \mathbf{s}^+$ **then**

      $\hat{\boldsymbol{\lambda}} \leftarrow \boldsymbol{\lambda}^{(t)}$;

  if *time spent exceeds* T *or* $\mathbf{\Lambda}_{new}$ *is empty* **then**

    return $\hat{\boldsymbol{\lambda}}, \mathcal{D}_{\mathit{Hist}}$

AutoML: Beyond AutoML

Structured Procastination

Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

Structured Procrastination [Kleinberg et al. 2017]

Idea

- incumbent driven methods (such as aggressive racing with adaptive capping) provide no theoretical guarantees about runtime

Structured Procrastination [Kleinberg et al. 2017]

Idea

- incumbent driven methods (such as aggressive racing with adaptive capping) provide no theoretical guarantees about runtime
- task: for a fix set of configuration, identify the one with the best average runtime
- instead of top-down capping, use bottom up capping

Structured Procrastination [Kleinberg et al. 2017]

Idea

- incumbent driven methods (such as aggressive racing with adaptive capping) provide no theoretical guarantees about runtime
- task: for a fix set of configuration, identify the one with the best average runtime
- instead of top-down capping, use bottom up capping
- start with a minimal cap-time and increase it step by step

Structured Procrastination [Kleinberg et al. 2017]

Idea

- incumbent driven methods (such as aggressive racing with adaptive capping) provide no theoretical guarantees about runtime
- task: for a fix set of configuration, identify the one with the best average runtime
- instead of top-down capping, use bottom up capping
- start with a minimal cap-time and increase it step by step
- unsuccessful runs (with too small cap-time) are procrastinated to later

$\leadsto$ worst-case runtime guarantees

Structured Procrastination: Outline [Kleinberg et al. 2017]

Algorithm 1 Structured Procrastination

Input : finite (small) set of configurations $\mathbf{\Lambda}$, minimal cap-time κ_0, sequence of instances $i^{(1)}, \dots, i^{(N)}$

Output : best incumbent configuration $\hat{\boldsymbol{\lambda}}$

for each $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ initialize a queue $Q_{\boldsymbol{\lambda}}$ with entries $(i^{(k)}, \kappa_0)$; `// small queue in the beginning`

initialize a look-up table $R(\boldsymbol{\lambda}, i) = 0$; `// optimistic runtime estimate`

Structured Procrastination: Outline [Kleinberg et al. 2017]

Algorithm 2 Structured Procrastination

Input : finite (small) set of configurations $\boldsymbol{\Lambda}$, minimal cap-time κ_0, sequence of instances $i^{(1)}, \dots, i^{(N)}$
Output : best incumbent configuration $\hat{\boldsymbol{\lambda}}$

for each $\boldsymbol{\lambda} \in \boldsymbol{\Lambda}$ initialize a queue $Q_{\boldsymbol{\lambda}}$ with entries $(i^{(k)}, \kappa_0)$; `// small queue in the beginning`
initialize a look-up table $R(\boldsymbol{\lambda}, i) = 0$; `// optimistic runtime estimate`
while b *remains* **do**

Structured Procrastination: Outline [Kleinberg et al. 2017]

Algorithm 3 Structured Procrastination

Input : finite (small) set of configurations $\mathbf{\Lambda}$, minimal cap-time κ_0, sequence of instances $i^{(1)}, \dots, i^{(N)}$
Output : best incumbent configuration $\hat{\boldsymbol{\lambda}}$

for each $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ initialize a queue $Q_{\boldsymbol{\lambda}}$ with entries $(i^{(k)}, \kappa_0)$; `// small queue in the beginning`
initialize a look-up table $R(\boldsymbol{\lambda}, i) = 0$; `// optimistic runtime estimate`
while b *remains* **do**
    determine the best $\hat{\boldsymbol{\lambda}}$ according to $R(\boldsymbol{\lambda}, \cdot)$;

Structured Procrastination: Outline [Kleinberg et al. 2017]

Algorithm 4 Structured Procrastination

Input : finite (small) set of configurations $\mathbf{\Lambda}$, minimal cap-time κ_0, sequence of instances $i^{(1)}, \dots, i^{(N)}$
Output : best incumbent configuration $\hat{\boldsymbol{\lambda}}$

for each $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ initialize a queue $Q_{\boldsymbol{\lambda}}$ with entries $(i^{(k)}, \kappa_0)$; `// small queue in the beginning`
initialize a look-up table $R(\boldsymbol{\lambda}, i) = 0$; `// optimistic runtime estimate`
while b *remains* **do**
    determine the best $\hat{\boldsymbol{\lambda}}$ according to $R(\boldsymbol{\lambda}, \cdot)$;
    get first element $(i^{(k)}, \kappa)$ from $Q_{\hat{\boldsymbol{\lambda}}}$;

Structured Procrastination: Outline [Kleinberg et al. 2017]

Algorithm 5 Structured Procrastination

Input : finite (small) set of configurations $\mathbf{\Lambda}$, minimal cap-time κ_0, sequence of instances $i^{(1)}, \dots, i^{(N)}$
Output : best incumbent configuration $\hat{\boldsymbol{\lambda}}$

for each $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ initialize a queue $Q_{\boldsymbol{\lambda}}$ with entries $(i^{(k)}, \kappa_0)$; `// small queue in the beginning`
initialize a look-up table $R(\boldsymbol{\lambda}, i) = 0$; `// optimistic runtime estimate`
while b *remains* **do**
  determine the best $\hat{\boldsymbol{\lambda}}$ according to $R(\boldsymbol{\lambda}, \cdot)$;
  get first element $(i^{(k)}, \kappa)$ from $Q_{\hat{\boldsymbol{\lambda}}}$;
  Run $\hat{\boldsymbol{\lambda}}$ on $i^{(k)}$ capped at κ;
  if *terminates* **then**
    $R(\hat{\boldsymbol{\lambda}}, i^{(k)}) :=$ t;

Structured Procrastination: Outline [Kleinberg et al. 2017]

Algorithm 6 Structured Procrastination

Input : finite (small) set of configurations $\mathbf{\Lambda}$, minimal cap-time κ_0, sequence of instances $i^{(1)}, \ldots, i^{(N)}$
Output : best incumbent configuration $\hat{\boldsymbol{\lambda}}$

for each $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ initialize a queue $Q_{\boldsymbol{\lambda}}$ with entries $(i^{(k)}, \kappa_0)$; `// small queue in the beginning`
initialize a look-up table $R(\boldsymbol{\lambda}, i) = 0$; `// optimistic runtime estimate`
while b *remains* **do**
  determine the best $\hat{\boldsymbol{\lambda}}$ according to $R(\boldsymbol{\lambda}, \cdot)$;
  get first element $(i^{(k)}, \kappa)$ from $Q_{\hat{\boldsymbol{\lambda}}}$;
  Run $\hat{\boldsymbol{\lambda}}$ on $i^{(k)}$ capped at κ;
  if *terminates* **then**
    $R(\hat{\boldsymbol{\lambda}}, i^{(k)}) :=$ t;
  else
    $R(\hat{\boldsymbol{\lambda}}, i^{(k)}) := \kappa$;
    Insert $(i^{(k)}, 2 \cdot \kappa)$ at the end of $Q_{\hat{\boldsymbol{\lambda}}}$;

Structured Procrastination: Outline [Kleinberg et al. 2017]

Algorithm 7 Structured Procrastination

Input : finite (small) set of configurations $\boldsymbol{\Lambda}$, minimal cap-time κ_0, sequence of instances $i^{(1)}, \dots, i^{(N)}$
Output : best incumbent configuration $\hat{\boldsymbol{\lambda}}$

```
for each λ ∈ Λ initialize a queue Q_λ with entries (i^(k), κ_0);      // small queue in the beginning
initialize a look-up table R(λ, i) = 0;                                // optimistic runtime estimate
while b remains do
    determine the best λ̂ according to R(λ, ·);
      get first element (i^(k), κ) from Q_λ̂;
      Run λ̂ on i^(k) capped at κ;
      if terminates then
        R(λ̂, i^(k)) := t;
    else
        R(λ̂, i^(k)) := κ;
          Insert (i^(k), 2 · κ) at the end of Q_λ̂;
    Replenish queue Q_λ̂ if too small;
```

Structured Procrastination: Outline [Kleinberg et al. 2017]

Algorithm 8 Structured Procrastination

Input : finite (small) set of configurations $\mathbf{\Lambda}$, minimal cap-time κ_0, sequence of instances $i^{(1)}, \dots, i^{(N)}$
Output : best incumbent configuration $\hat{\boldsymbol{\lambda}}$

for each $\boldsymbol{\lambda} \in \mathbf{\Lambda}$ initialize a queue $Q_{\boldsymbol{\lambda}}$ with entries $(i^{(k)}, \kappa_0)$; `// small queue in the beginning`
initialize a look-up table $R(\boldsymbol{\lambda}, i) = 0$; `// optimistic runtime estimate`
while b *remains* **do**
  determine the best $\hat{\boldsymbol{\lambda}}$ according to $R(\boldsymbol{\lambda}, \cdot)$;
  get first element $(i^{(k)}, \kappa)$ from $Q_{\hat{\boldsymbol{\lambda}}}$;
  Run $\hat{\boldsymbol{\lambda}}$ on $i^{(k)}$ capped at κ;
  if *terminates* **then**
    $R(\hat{\boldsymbol{\lambda}}, i^{(k)})$:= t;
  else
    $R(\hat{\boldsymbol{\lambda}}, i^{(k)}) := \kappa$;
    Insert $(i^{(k)}, 2 \cdot \kappa)$ at the end of $Q_{\hat{\boldsymbol{\lambda}}}$;
  Replenish queue $Q_{\hat{\boldsymbol{\lambda}}}$ if too small;
return $\hat{\boldsymbol{\lambda}} := \arg\min_{\boldsymbol{\lambda} \in \mathbf{\Lambda}} \sum_{k=1}^{N} R(\boldsymbol{\lambda}, i^{(k)})$

Extensions

- We can derive theoretical optimality guarantees with structured procrastination (SP)

Extensions

- We can derive theoretical optimality guarantees with structured procrastination (SP)
- In practice, plain SP is rather slow and requires the setting of some hyperparameters

Extensions

- We can derive theoretical optimality guarantees with structured procrastination (SP)
- In practice, plain SP is rather slow and requires the setting of some hyperparameters
- Several extensions and similar ideas:
  - [Kleinberg et al. 2019]
  - [Weisz et al. 2018]
  - [Weisz et al. 2019]

AutoML: Beyond AutoML

Best Practices for Algorithm Configuration

Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

Pitfalls & Best Practices

The goals of automated algorithm configuration include:

Pitfalls & Best Practices

The goals of automated algorithm configuration include:

1. reducing the expertise required to use an algorithm

Pitfalls & Best Practices

The goals of automated algorithm configuration include:

1. reducing the expertise required to use an algorithm
2. less human-time

Pitfalls & Best Practices

The goals of automated algorithm configuration include:

1. reducing the expertise required to use an algorithm
2. less human-time
3. tuning algorithms to the task at hand

Pitfalls & Best Practices

The goals of automated algorithm configuration include:

1. reducing the expertise required to use an algorithm
2. less human-time
3. tuning algorithms to the task at hand
4. faster development of algorithms

Pitfalls & Best Practices

The goals of automated algorithm configuration include:

1. reducing the expertise required to use an algorithm
2. less human-time
3. tuning algorithms to the task at hand
4. faster development of algorithms
5. facilitating systematic and reproducible research

BUT:

1. algorithm configuration can lead to over-tuning

Pitfalls & Best Practices

The goals of automated algorithm configuration include:

1. reducing the expertise required to use an algorithm
2. less human-time
3. tuning algorithms to the task at hand
4. faster development of algorithms
5. facilitating systematic and reproducible research

BUT:

1. algorithm configuration can lead to over-tuning
2. using algorithm configuration requires (at least some) expertise in algorithm configuration

Pitfalls & Best Practices

The goals of automated algorithm configuration include:

1. reducing the expertise required to use an algorithm
2. less human-time
3. tuning algorithms to the task at hand
4. faster development of algorithms
5. facilitating systematic and reproducible research

BUT:

1. algorithm configuration can lead to over-tuning
2. using algorithm configuration requires (at least some) expertise in algorithm configuration
3. if done wrong, waste of time and compute resources

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric
2. Define instance set

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric
2. Define instance set
3. Split your instances in training and test

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric
2. Define instance set
3. Split your instances in training and test
4. Define the configuration space

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric
2. Define instance set
3. Split your instances in training and test
4. Define the configuration space
5. Choose your preferred configurator

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric
2. Define instance set
3. Split your instances in training and test
4. Define the configuration space
5. Choose your preferred configurator
6. Implement an interface between your algorithm and the configurator

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric
2. Define instance set
3. Split your instances in training and test
4. Define the configuration space
5. Choose your preferred configurator
6. Implement an interface between your algorithm and the configurator
7. Define resource limitations of your algorithm

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric
2. Define instance set
3. Split your instances in training and test
4. Define the configuration space
5. Choose your preferred configurator
6. Implement an interface between your algorithm and the configurator
7. Define resource limitations of your algorithm
8. Run the configurator on your algorithm and the training instances

Setting up AC

9 Steps to your well-performing algorithm:

1. Define your performance metric
2. Define instance set
3. Split your instances in training and test
4. Define the configuration space
5. Choose your preferred configurator
6. Implement an interface between your algorithm and the configurator
7. Define resource limitations of your algorithm
8. Run the configurator on your algorithm and the training instances
9. Validate the eventually returned configuration on your test instances

Pitfall 1: Trust your algorithm

We have encountered algorithms that

- ignored resource limitations
- returned wrong solutions
- even returned negative runtimes

Pitfall 1: Trust your algorithm

We have encountered algorithms that

- ignored resource limitations
- returned wrong solutions
- even returned negative runtimes

Best Practice 1: Never trust your algorithm

Explicitly check and use external software to:

1. ensure resource limitations
2. terminate your algorithm
3. verify returned solutions
4. measure performance

Pitfall 2: File System

Algorithm configurators ...

- often produce quite some log files (e.g., for each algorithm run)
- are often used on HPC clusters with a shared file system

Pitfall 2: File System

Algorithm configurators ...

- often produce quite some log files (e.g., for each algorithm run)
- are often used on HPC clusters with a shared file system

$\rightsquigarrow$ It's surprisingly easy to substantially slow down a shared file system

Pitfall 2: File System

Algorithm configurators ...

- often produce quite some log files (e.g., for each algorithm run)
- are often used on HPC clusters with a shared file system

$\rightsquigarrow$ It's surprisingly easy to substantially slow down a shared file system

Best Practice 2: Don't use the Shared File System

To relieve the file system of a HPC cluster:

- design well which files are required and which are not
- use a local (SSD) disc

Pitfall 3: Over-tuning

It's easy to overtune to different aspects, incl.:

- training instances
- random seeds
- machine type

Pitfall 3: Over-tuning

It's easy to overtune to different aspects, incl.:

- training instances
- random seeds
- machine type

In practice, it can be hard to prevent over-tuning, e.g., by

- using larger instance sets
- tuning on the target hardware

Best Practice 3: Check for Over-Tuning

Check for over-tuning by validating your final configuration on

- many random seeds
- a large set of unused test instances
- a different hardware

Pitfall 4: Heterogeneous Instance Sets

Algorithm configurators...

- use some kind of racing to not evaluate each configuration on all instances

Pitfall 4: Heterogeneous Instance Sets

Algorithm configurators...

- use some kind of racing to not evaluate each configuration on all instances
- can be mislead on subsets of instances if the instance set is heterogeneous

Pitfall 4: Heterogeneous Instance Sets

Algorithm configurators...

- use some kind of racing to not evaluate each configuration on all instances
- can be mislead on subsets of instances if the instance set is heterogeneous

$\rightsquigarrow$ returned configurations often perform worse than default configurations
in the validation phase

Pitfall 4: Heterogeneous Instance Sets

Algorithm configurators...

- use some kind of racing to not evaluate each configuration on all instances
- can be mislead on subsets of instances if the instance set is heterogeneous

$\rightsquigarrow$ returned configurations often perform worse than default configurations in the validation phase

Best Practice 4: Ensure Homogeneity

Algorithm configurators should only run on homogeneous instance sets.
Different degrees of homogeneity:

- Strong homogeneity: all instances agree on the ranking of configurations
- Weak homogeneity: all instances agree on the top-performing configurations

More Pitfalls and Best Practices

... can be found in [Eggensperger et al. 2019]

AutoML: Beyond AutoML

Per-Instance Algorithm Configuration

Bernd Bischl Frank Hutter Lars Kotthoff
Marius Lindauer Joaquin Vanschoren

Homogeneous vs. Heterogeneous Instances

Assumption of AC: Homogeneous Instance Distribution

- Algorithm configuration tools assume that the instance distribution is homogeneous (see video on "Best Practices for AC")
- Important because
  - there is a well-performing configuration for all (or most) instances
  - the racing algorithm can make educated decisions on subsets

Homogeneous vs. Heterogeneous Instances

Assumption of AC: Homogeneous Instance Distribution

- Algorithm configuration tools assume that the instance distribution is homogeneous (see video on "Best Practices for AC")
- Important because
  - there is a well-performing configuration for all (or most) instances
  - the racing algorithm can make educated decisions on subsets

Violated assumption of AC: Hetergeneous Instance Distribution

- The racing algorithm will make inconsistent (or even wrong) decisions
- There is no single well-performing configuration for all instances

Homogeneous vs. Heterogeneous Instances

Assumption of AC: Homogeneous Instance Distribution

- Algorithm configuration tools assume that the instance distribution is homogeneous (see video on "Best Practices for AC")
- Important because
  - there is a well-performing configuration for all (or most) instances
  - the racing algorithm can make educated decisions on subsets

Violated assumption of AC: Hetergeneous Instance Distribution

- The racing algorithm will make inconsistent (or even wrong) decisions
- There is no single well-performing configuration for all instances

$\rightsquigarrow$ What should we do with heterogeneous instance distributions?

Why are systems for heterogeneous instance distributions important?

1. We cannot guarantee homogeneity in practice
    1. Instances might get larger and harder
    2. The underlying task or business case might change

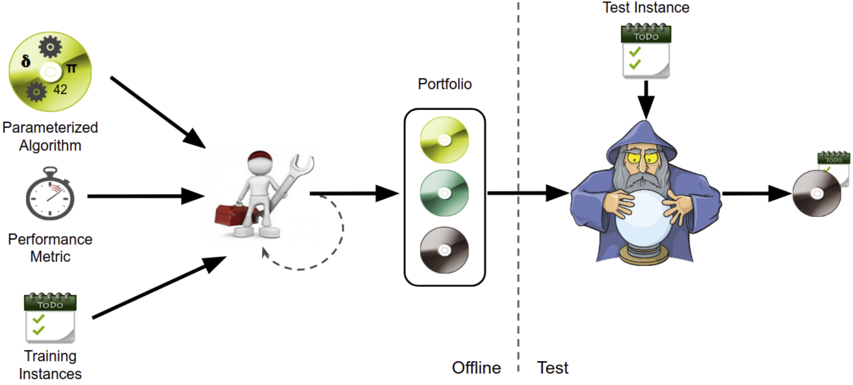
Why are systems for heterogeneous instance distributions important?

1. We cannot guarantee homogeneity in practice
    1. Instances might get larger and harder
    2. The underlying task or business case might change
2. We don't want to do algorithm configuration always from scratch

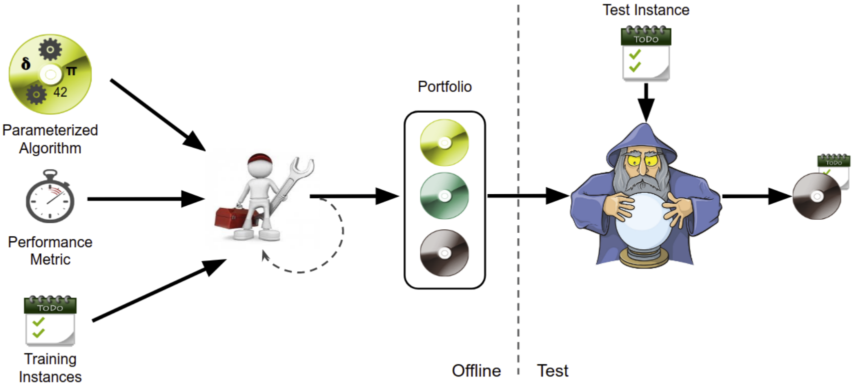
Why are systems for heterogeneous instance distributions important?

1. We cannot guarantee homogeneity in practice
   1. Instances might get larger and harder
   2. The underlying task or business case might change
2. We don't want to do algorithm configuration always from scratch
3. An adaptive configuration system would be the holy grail
   - $\leadsto$ hard to achieve

PIAC: Per-Instance Algorithm Configuration



PIAC: Per-Instance Algorithm Configuration



- You can use whichever kind of algorithm selection (wizard) you want
- Challenge: Building a portfolio
- Use case: Instances are heterogeneous

PIAC: Manual Expert Approach

Basic Assumption

Heterogeneous instance set can be divided into homogeneous subsets

PIAC: Manual Expert Approach

Basic Assumption

Heterogeneous instance set can be divided into homogeneous subsets

Manual Expert

- An expert knows the homogeneous subsets (e.g., origin of instances)
- Determine a well-performing configuration on each subset
  $\rightarrow$ portfolio of configurations
- Use Algorithm Selection to select a well-performing configuration on each instance

Instance-Specific Algorithm Configuration: ISAC [Kadioglu et al. 2010]

Idea

Training:

1. Cluster instances into homogeneous subsets
   (using g-means in the instance feature space)
2. Apply algorithm configuration (here GGA) on each instance set

Instance-Specific Algorithm Configuration: ISAC [Kadioglu et al. 2010]

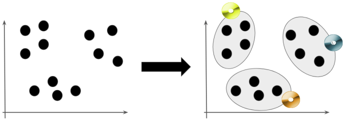
Idea

Training:

1. Cluster instances into homogeneous subsets
   (using g-means in the instance feature space)
2. Apply algorithm configuration (here GGA) on each instance set

Test:

1. Determine the nearest cluster (k-NN with $k = 1$) in feature space
2. Apply optimized configuration of this cluster



Hydra [Xu et al. 2010]

Idea

- Iteratively add configurations to a portfolio $\mathbf{P}$, start with $\mathbf{P} = \emptyset$
- In each iteration, determine configuration that is complementary to $\mathbf{P}$
  - $\leadsto$ Maximize marginal contribution to $\mathbf{P}$

Hydra [Xu et al. 2010]

Idea

- Iteratively add configurations to a portfolio $\mathbf{P}$, start with $\mathbf{P} = \emptyset$
- In each iteration, determine configuration that is complementary to $\mathbf{P}$
  - $\rightsquigarrow$ Maximize marginal contribution to $\mathbf{P}$

Marginal contribution of a configuration $\boldsymbol{\lambda}$ to a portfolio $\mathbf{P}$:

$$c(\mathbf{P}) - c(\mathbf{P} \cup \{\boldsymbol{\lambda}\})$$

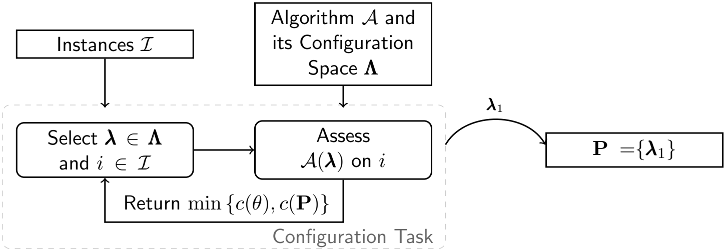
Hydra [Xu et al. 2010]

Idea

- Iteratively add configurations to a portfolio $\mathbf{P}$, start with $\mathbf{P} = \emptyset$
- In each iteration, determine configuration that is complementary to $\mathbf{P}$
  - $\rightsquigarrow$ Maximize marginal contribution to $\mathbf{P}$

Marginal contribution of a configuration $\boldsymbol{\lambda}$ to a portfolio $\mathbf{P}$:

$$c(\mathbf{P}) - c(\mathbf{P} \cup \{\boldsymbol{\lambda}\})$$



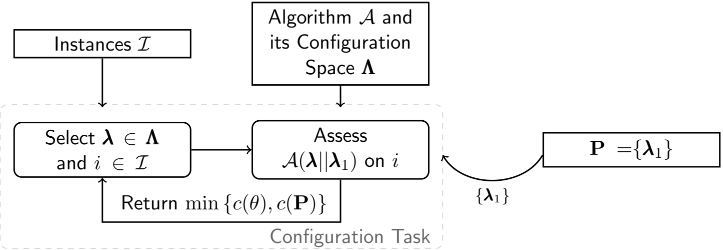
Hydra [Xu et al. 2010]

Idea

- Iteratively add configurations to a portfolio $\mathbf{P}$, start with $\mathbf{P} = \emptyset$
- In each iteration, determine configuration that is complementary to $\mathbf{P}$
  - $\leadsto$ Maximize marginal contribution to $\mathbf{P}$

Marginal contribution of a configuration $\boldsymbol{\lambda}$ to a portfolio $\mathbf{P}$:

$$c(\mathbf{P}) - c(\mathbf{P} \cup \{\boldsymbol{\lambda}\})$$



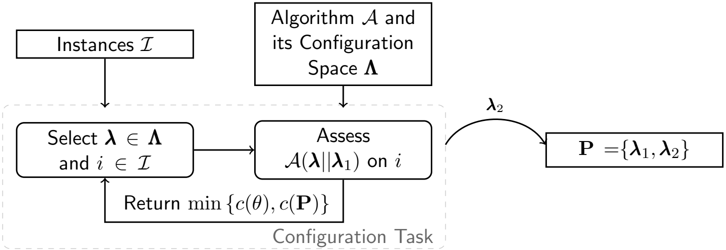
Hydra [Xu et al. 2010]

Idea

- Iteratively add configurations to a portfolio $\mathbf{P}$, start with $\mathbf{P} = \emptyset$
- In each iteration, determine configuration that is complementary to $\mathbf{P}$
  - $\rightsquigarrow$ Maximize marginal contribution to $\mathbf{P}$

Marginal contribution of a configuration $\boldsymbol{\lambda}$ to a portfolio $\mathbf{P}$:

$$c(\mathbf{P}) - c(\mathbf{P} \cup \{\boldsymbol{\lambda}\})$$



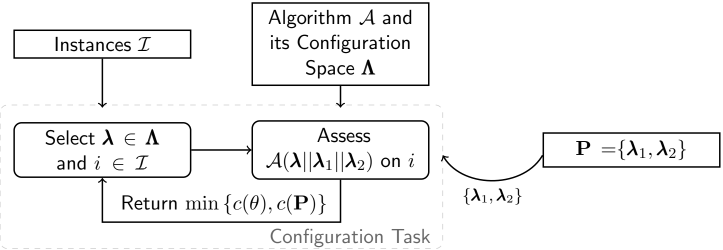
Hydra [Xu et al. 2010]

Idea

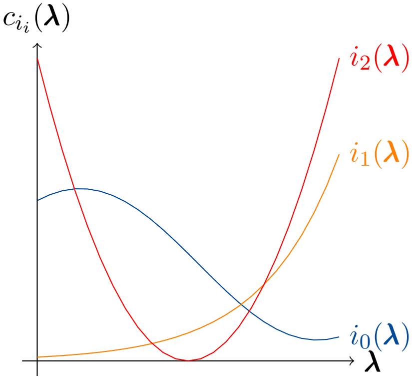
- Iteratively add configurations to a portfolio $\mathbf{P}$, start with $\mathbf{P} = \emptyset$
- In each iteration, determine configuration that is complementary to $\mathbf{P}$
  - $\leadsto$ Maximize marginal contribution to $\mathbf{P}$

Marginal contribution of a configuration $\boldsymbol{\lambda}$ to a portfolio $\mathbf{P}$:

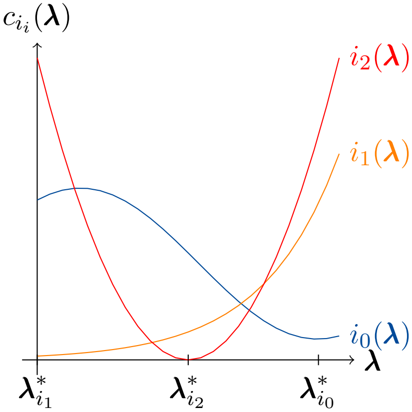
$$c(\mathbf{P}) - c(\mathbf{P} \cup \{\boldsymbol{\lambda}\})$$



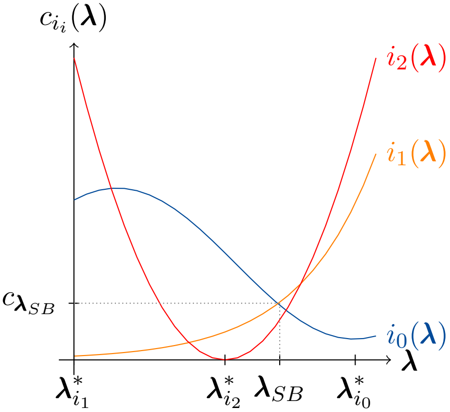
Heterogeneous example



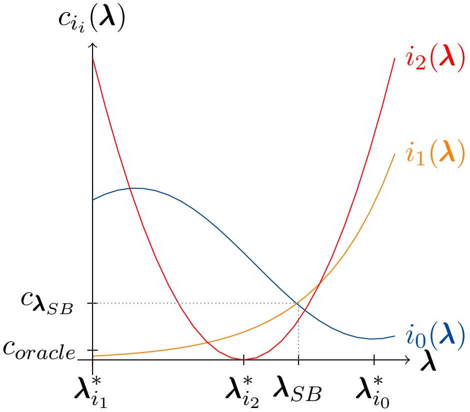
Heterogeneous example



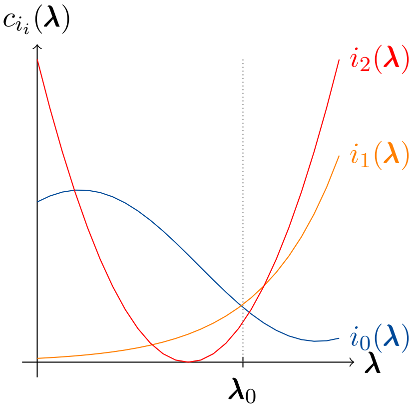
Heterogeneous example



Heterogeneous example

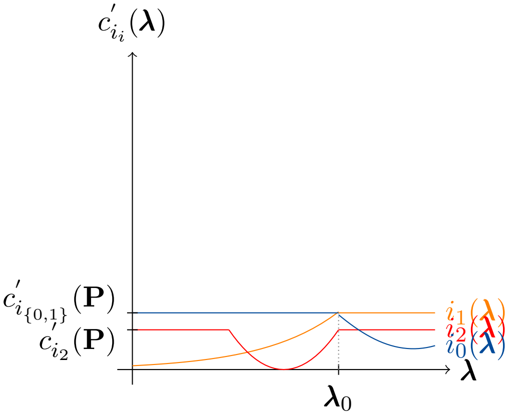


Hydra: Iteration 1



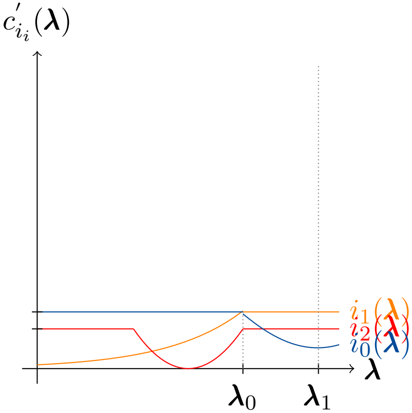
Search initial well performing configuration. Add $\boldsymbol{\lambda}_0$ to $\mathbf{P}$

Hydra: Iteration 1



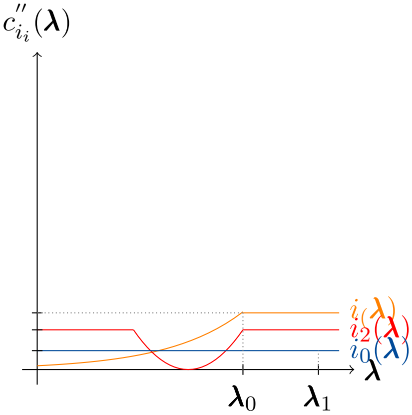
Update metric

Hydra: Iteration 2



Search well performing configuration complementary to $\mathbf{P}$.
Add $\boldsymbol{\lambda}_1$ to $\mathbf{P}$.

Hydra: Iteration 2



Update metric

Cedalion [Seipp et al. 2015]

Idea

- Optimize a schedule of configurations with algorithm configuration

Cedalion [Seipp et al. 2015]

Idea

- Optimize a schedule of configurations with algorithm configuration

Approach

- Iteratively add a configuration with a time slot t to a schedule $\mathcal{S} \oplus \langle \boldsymbol{\lambda}, t \rangle$

Cedalion [Seipp et al. 2015]

Idea

- Optimize a schedule of configurations with algorithm configuration

Approach

- Iteratively add a configuration with a time slot t to a schedule $\mathcal{S} \oplus \langle \boldsymbol{\lambda}, t \rangle$
- In each iteration, only optimize on instances not solved so far
- The time slot is a further parameter in the configuration space

Cedalion [Seipp et al. 2015]

Idea

- Optimize a schedule of configurations with algorithm configuration

Approach

- Iteratively add a configuration with a time slot t to a schedule $\mathcal{S} \oplus \langle \boldsymbol{\lambda}, t \rangle$
- In each iteration, only optimize on instances not solved so far
- The time slot is a further parameter in the configuration space
- Optimize marginal contribution per time spent:

$$\frac{c(\mathcal{S}) - c(\mathcal{S} \oplus \langle \boldsymbol{\lambda}, t \rangle)}{t}$$

Submodularity

Observation

- Performance metrics of Hydra and Cedalion are submodular
  - Family of functions
  - Adding an element to a set reduces the function value
  - Diminishing returns: decrease of the value reduction over time

Submodularity

Observation

- Performance metrics of Hydra and Cedalion are submodular
  - Family of functions
  - Adding an element to a set reduces the function value
  - Diminishing returns: decrease of the value reduction over time

Definition (Submodularity of f)

For every $X, Y \subseteq Z$ with $X \subseteq Y$ and every $x \in Z - Y$ we have that
$f(X \cup \{x\}) - f(X) \geq f(Y \cup \{x\}) - f(Y)$

Submodularity

Observation

- Performance metrics of Hydra and Cedalion are submodular
  - Family of functions
  - Adding an element to a set reduces the function value
  - Diminishing returns: decrease of the value reduction over time

Definition (Submodularity of f)

For every $X, Y \subseteq Z$ with $X \subseteq Y$ and every $x \in Z - Y$ we have that
$f(X \cup \{x\}) - f(X) \geq f(Y \cup \{x\}) - f(Y)$

Advantage

We can bound the error of the portfolio/schedule:
At most away from optimum by factor of 0.63 (see [Streeter and Golovin. 2008])

Dynamic Instance Grouping [Liu et al. 2018]

Idea

- Similar to ISAC: group instances into clusters
- Similar to Hydra: refine clusters and configurations over several iterations

Dynamic Instance Grouping [Liu et al. 2018]

Idea

- Similar to ISAC: group instances into clusters
- Similar to Hydra: refine clusters and configurations over several iterations

Main Idea

1. Group instances randomly into clusters
2. run AC on each cluster
3. Update clusters based on performance (estimates)
4. Go to 2. if budget is not empty
5. Consider all configurations ever found to create final portfolio

Dynamic Instance Grouping [Liu et al. 2018]

Idea

- Similar to ISAC: group instances into clusters
- Similar to Hydra: refine clusters and configurations over several iterations

Main Idea

1. Group instances randomly into clusters
2. run AC on each cluster
3. Update clusters based on performance (estimates)
4. Go to 2. if budget is not empty
5. Consider all configurations ever found to create final portfolio

- increase the configuration budget in each iteration
  - first clusterings will have a poor quality $\rightarrow$ small configuration time
  - later clusterings will be better $\rightarrow$ more configuration time