Algorithm Configuration:

Learning in the Space of Algorithm Designs

Kevin Leyton-Brown

University of British Columbia
Canada CIFAR AI Chair, Amii

Frank Hutter

University of Freiburg and
Bosch Center for Artificial Intelligence

This Tutorial

High-Level Outline

Introduction, Technical Preliminaries, and a Case Study (Kevin)

Practical Methods for Algorithm Configuration (Frank)

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

Beyond Static Configuration: Related Problems and Emerging Directions (Frank)

Follow along: http://bit.ly/ACTutorial

This Tutorial

Section Outline

Follow along: http://bit.ly/ACTutorial

Algorithm Configuration

- **Algorithm configuration** is a powerful technique at the interface of ML and optimization
- It makes it possible to approach algorithm design as a **machine learning problem**
  - stop imagining that we have **good intuitions** about how to approach combinatorial optimization in practice!
  - instead, expose heuristic design choices as parameters, use **automatic methods** to search for good configurations
- Many **research challenges** in the development of methods
- Enormous scope for **applications** to practical problems

We should think about algorithm designs as a hypothesis space

Machine learning

Classical approach

- Features based on **expert insight**
- Model family selected by **hand**
- **Manual** tuning of hyperparameters

We should think about algorithm designs as a hypothesis space

Machine learning

Classical approach

- Features based on expert insight
- Model family selected by hand
- Manual tuning of hyperparameters

Deep learning

- Very **highly parameterized** models, using expert knowledge to identify appropriate invariances and model biases (e.g., convolutional structure)
- "deep": **many layers** of nodes, each depending on the last
- Use **lots of data** (plus e.g. dropout regularization) to avoid overfitting
- **Computationally intensive search** replaces human design

We should think about algorithm designs as a hypothesis space

Machine learning

Classical approach

- Features based on expert insight
- Model family selected by hand
- Manual tuning of hyperparameters

Deep learning

- Very highly parameterized models, using expert knowledge to identify appropriate invariances and model biases (e.g., convolutional structure)
- "deep": many layers of nodes, each depending on the last
- Use lots of data (plus e.g. dropout regularization) to avoid overfitting
- Computationally intensive search replaces human design

Discrete Optimization

Classical approach

- **Expert designs** a heuristic algorithm
- Iteratively conducts **small experiments** to improve the design

We should think about algorithm designs as a hypothesis space

Machine learning

Classical approach

- Features based on expert insight
- Model family selected by hand
- Manual tuning of hyperparameters

Deep learning

- Very highly parameterized models, using expert knowledge to identify appropriate invariances and model biases (e.g., convolutional structure)
- "deep": many layers of nodes, each depending on the last
- Use lots of data (plus e.g. dropout regularization) to avoid overfitting
- Computationally intensive search replaces human design

Discrete Optimization

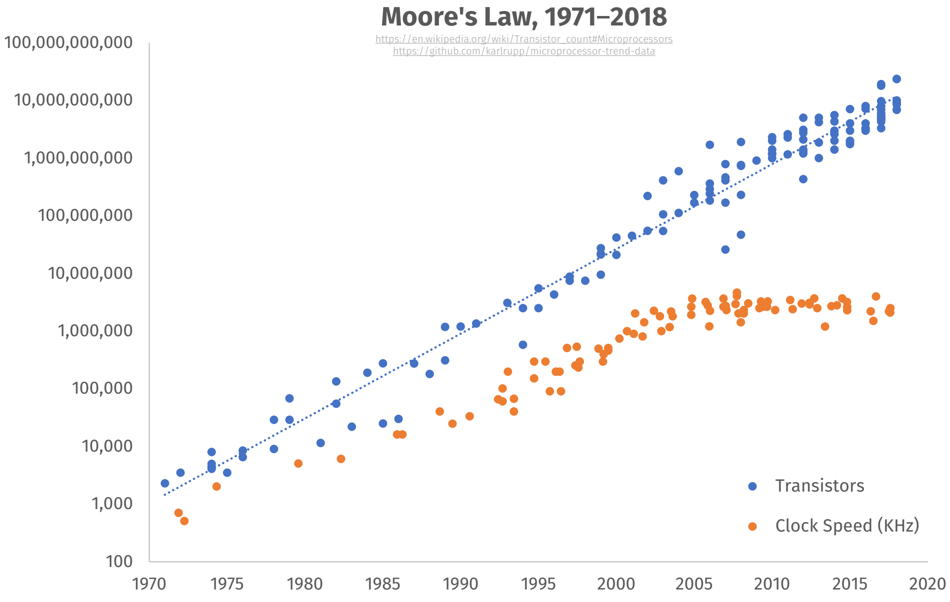
Classical approach

- Expert designs a heuristic algorithm
- Iteratively conducts small experiments to improve the design

Learning in the space of algorithm designs

- Very **highly parameterized** algorithms express a combinatorial space of heuristic design choices that make sense to an expert
- "deep": **many layers** of parameters, each depending on the last
- Use **lots of data** to characterize the distribution of interest
- **Computationally intensive search** replaces human design

Approaches that seemed crazy in 2000 make a lot of sense today...



Algorithm design in a world of learnable algorithms

Designers should:

- Shift from choosing heuristics they think will work to **exposing a wide variety of design elements** that might be sensible
  - This can be integrated into software engineering workflows; see Hoos [2012].
- get out of the business of **manual experimentation**, leaving this to automated procedures
  - this tutorial focuses mainly on **how these automated procedures work**
- **Reoptimize their designs** for new use cases rather than trying to identify a single algorithm to rule them all

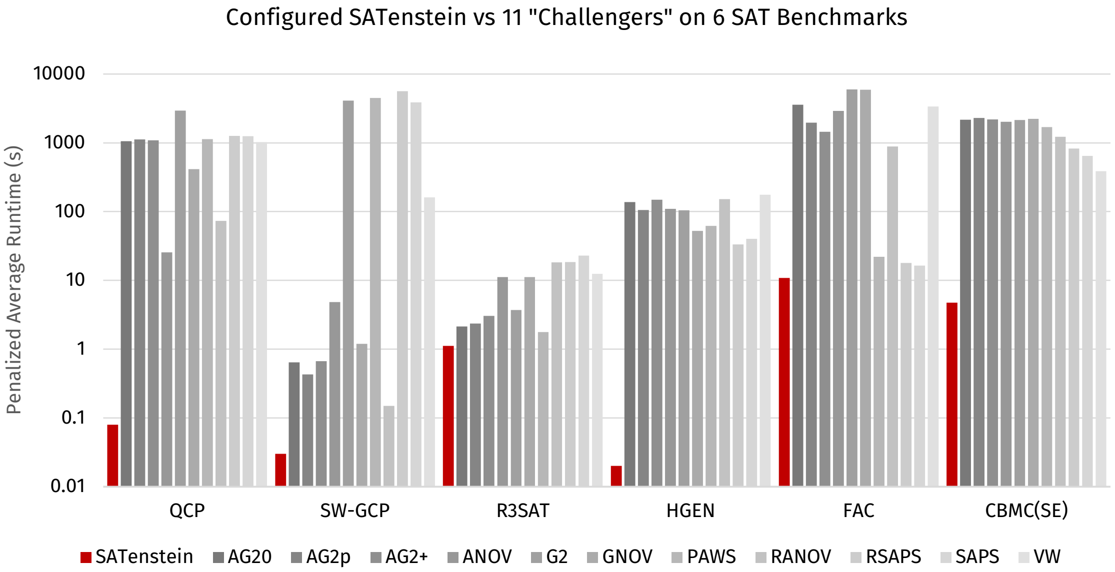
An example of how this can look: SATenstein

[Khudabukhsh, Xu, Hoos, L-B, 2009; 2016]

- **Frankenstein**'s goal:
  - Create "perfect" human being from scavenged body parts
- **SATenstein**'s goal: Create high-performance SAT solvers using components scavenged from existing solvers
  - Components drawn from or inspired by existing local search algorithms for SAT parameters determine which components are selected and how they behave (41 parameters total)
  - designed for use with algorithm configuration (3 levels of conditional params)
- SATenstein can instantiate:
  - at least **29 distinct, high-performance local-search solvers** from the literature
  - trillions of **novel solver strategies**

SATenstein outperformed the existing state of the art on each of six benchmarks

[Khudabukhsh, Xu, Hoos, L-B, 2016]



This Tutorial

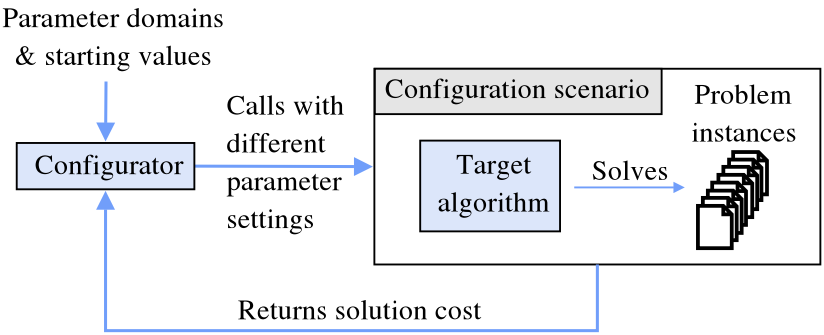
Section Outline

Introduction, Technical Preliminaries, and a Case Study (Kevin)

- Learning in the Space of Algorithm Designs
- **Defining the Algorithm Configuration Problem**
- Algorithm Runtime Prediction
- Applications and a Case Study

Follow along: http://bit.ly/ACTutorial

Algorithm Configuration Visualized



Algorithm Parameters

Parameter Types

- Continuous, integer, ordinal
- **Categorical**: finite domain, unordered, e.g., {apple, tomato, pepper}
- **Conditional**
  - allowed values of some child parameter depend on the values taken by parent parameter(s)

Parameters give rise to a structured space of configurations

- These spaces are often **huge**
  - e.g., SAT solver lingeling has 10^{947} configurations
- Changing one parameter can yield **qualitatively different behaviour**
- Overall, that's why we call it **algorithm configuration** (vs "parameter tuning")

Algorithm Configuration: General Definition

Definition (algorithm configuration)

An algorithm configuration problem is a 5-tuple $(\mathcal{A}, \boldsymbol{\Theta}, \mathcal{D}, \bar{\kappa}, m)$ where:

- $\mathcal{A}$ is a parameterized **algorithm**;
- $\boldsymbol{\Theta}$ is the parameter **configuration space** of $\mathcal{A}$;
- $\mathcal{D}$ is a **distribution over problem instances** with domain Π;
- $\bar{\kappa} < \infty$ is a **cutoff time**, after which each run of $\mathcal{A}$ will be terminated
- $m : \boldsymbol{\Theta} \times \Pi \rightarrow \mathbb{R}$ is a function that measures the cost incurred by $\mathcal{A}(\boldsymbol{\theta})$ on an instance $\pi \in \Pi$

Optimal configuration $\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \mathbb{E}_{\pi \sim \mathcal{D}}(m(\boldsymbol{\theta}, \pi))$ minimizes expected cost

Algorithm Configuration: Definition with Runtime Objective

Definition (algorithm configuration)

An algorithm configuration problem is a 5-tuple $(\mathcal{A}, \mathbf{\Theta}, \mathcal{D}, \bar{\kappa}, R_{\bar{\kappa}})$ where:

- $\mathcal{A}$ is a parameterized **algorithm**;
- $\mathbf{\Theta}$ is the parameter **configuration space** of $\mathcal{A}$;
- $\mathcal{D}$ is a **distribution over problem instances** with domain Π;
- $\bar{\kappa} < \infty$ is a **cutoff time**, after which each run of $\mathcal{A}$ will be terminated
- $R_{\bar{\kappa}} : \mathbf{\Theta} \times \Pi \to \mathbb{R}$ is a function that measures the **time it takes to run $\mathcal{A}(\boldsymbol{\theta})$ with cutoff time $\bar{\kappa}$** on instance $\pi \in \Pi$

Optimal configuration $\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta} \in \mathbf{\Theta}} \mathbb{E}_{\pi \sim \mathcal{D}}(R_{\bar{\kappa}}(\boldsymbol{\theta}, \pi))$ minimizes expected **runtime**

Beyond Runtime Optimization

Algorithm configuration methods can also be applied to objectives other than runtime optimization (though not the focus of this tutorial).

Black-Box Optimization

Optimize a function to which the algorithm **only has query access**.

Hyperparameter Optimization

Find **hyperparameters of a model** that minimize validation set loss.

This Tutorial

Section Outline

Introduction, Technical Preliminaries, and a Case Study (Kevin)

Learning in the Space of Algorithm Designs

Defining the Algorithm Configuration Problem

Algorithm Runtime Prediction

Applications and a Case Study

Follow along: http://bit.ly/ACTutorial

Algorithm Runtime Prediction

A key enabling technology will be the ability to solve the following problem.

A pretty vanilla application of regression?

Predict **how long an algorithm will take to run**, given:

- A set of instances D
- For each instance $i \in D$, a vector x_i of feature values
- For each instance $i \in D$ a runtime observation y_i We want a mapping $f(x) \to y$ that accurately predicts y_i given x_i

In other words, find a mapping $f(x) \to y$ that accurately predicts y_i given x_i.

Algorithm Runtime Prediction

A key enabling technology will be the ability to solve the following problem.

A pretty vanilla application of regression?

Predict **how long an algorithm will take to run**, given:

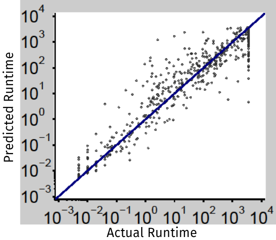
- A set of instances D
- For each instance $i \in D$, a vector x_i of feature values
- For each instance $i \in D$ a runtime observation y_i We want a mapping $f(x) \to y$ that accurately predicts y_i given x_i

In other words, find a mapping $f(x) \to y$ that accurately predicts y_i given x_i.

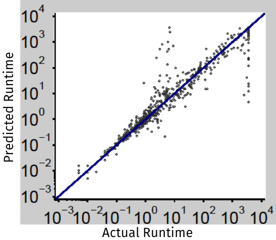
But, **is it really possible** to use supervised learning to predict the empirical behavior of an exponential-time algorithm on held-out problem inputs?

Algorithm Runtime is Surprisingly Predictable

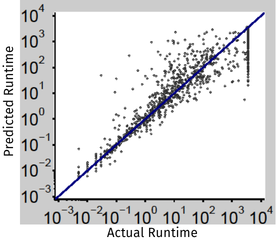
SAT Competition (Random + Handmade + Industrial) data, MINISAT solver
Random Forest (RMSE=0.47)



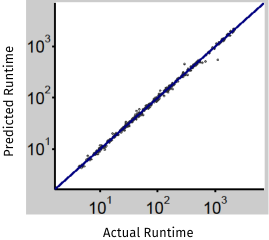
SAT: IBM hardware verification data, SPEAR solver
Random Forest (RMSE=0.38)



MIPLIB data, CPLEX 12.1 solver
Random Forest (RMSE=0.63)



Red Crested Woodpecker habitat data, CPLEX 12.1 solver
Random Forest (RMSE=0.02)



[H, Xu, L-B, Hoos, 2014]

That's Not All, Folks

[H, Xu, L-B, Hoos, 2014]

We've found that that **algorithm runtime is consistently predictable**, across:

- Four **problem domains**:
  - Satisfiability (SAT)
  - Mixed Integer Programming (MIP)
  - Travelling Salesman Problem (TSP)
  - Combinatorial Auctions
- Dozens of **solvers**, including:
  - state of the art solvers in each domain
  - black-box, commercial solvers
- Dozens of **instance distributions**, including:
  - major benchmarks (SAT competitions; MIPLIB; ...)
  - real-world data (hardware verification, computational sustainability, ...)

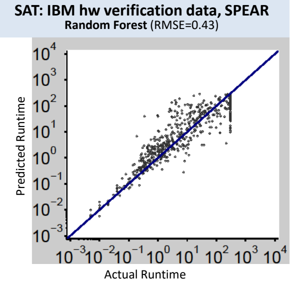
What About Modeling Algorithm Parameters, Too?

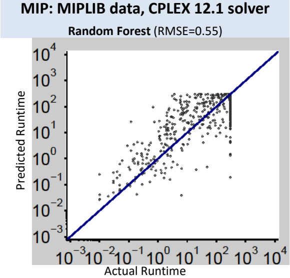
- So far we've considered the runtime of **single, black box algorithms**
- Our goal in this tutorial is understanding algorithm performance as a function of an **algorithm's parameters**
  - with the ultimate aim of optimizing this function
- Can we predict the performance of **parameterized algorithm families**?

What About Modeling Algorithm Parameters, Too?

- So far we've considered the runtime of **single, black box algorithms**

- Our goal in this tutorial is understanding algorithm performance as a function of an **algorithm's parameters**
  - with the ultimate aim of optimizing this function
- Can we predict the performance of **parameterized algorithm families**?
  - Performance is worse than before, but we're generalizing simultaneously to **unseen problem instances** and **unseen parameter configurations**
    - On average, correct within roughly **half an order of magnitude**
  - Despite discontinuities, an algorithm's performance is well approximated by a **relatively simple function** of its parameters





So, how does it work?

In fact, it's a somewhat trickier regression problem than initially suggested

- mixed continuous/**discrete**
- **high-dimensional**, though often with low effective dimensionality
- **very noisy** response variable (e.g., exponential runtime distribution)

Plus there are some extra features that will be nice to have

- compatibility with **censored observations**
- ability to offer **uncertainty estimates** at test time

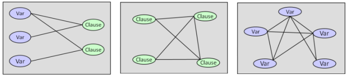
We've tried a lot of different approaches

- linear/ridge/lasso/polynomial; SVM; MARS; Gaussian processes; deep nets; ...

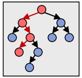
...to date, we've had the most success with **random forests of regression trees**

It's most important to get features right. For example, in SAT:

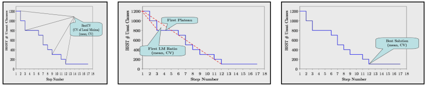
- Problem **Size** (clauses, variables, clauses/variables, ...)
- **Syntactic** properties (e.g., positive/negative clause ratio)
- Statistics of various **constraint graphs**
  - factor graph
  - clause–clause graph
  - variable–variable graph



- Knuth's **search space size** estimate



- Cumulative # of **unit propagations** at different depths



- **Local search probing**
- **Linear programming** relaxation

$$\begin{aligned}\text{maximize: } & \sum_{k\in C}\left(\sum_{i\in L,\, i\in k} v_i + \sum_{j\in \overline{L},\, i\in k}(1-v_j)\right) & \\ \text{subject to: } & \sum_{i\in k,\, i\in L} v_i + \sum_{j\in k,\, j\in \overline{L}}(1-v_j) \geq 1 & \forall k\in C \\ & v_i\in\{0,1\} & \forall i\end{aligned}$$

This Tutorial

Section Outline

Introduction, Technical Preliminaries, and a Case Study (Kevin)

- Learning in the Space of Algorithm Designs
- Defining the Algorithm Configuration Problem
- Algorithm Runtime Prediction
- **Applications and a Case Study**

Follow along: http://bit.ly/ACTutorial

Algorithm Configuration: Many Applications

FCC spectrum auction

[Auction]omics

Mixed integer programming

IBM ILOG

Analytics & Optimization

ORTEC OPTIMIZE YOUR WORLD

Social gaming

zynga

Scheduling and Resource Allocation

actenum

Applications by Colleagues

- Exam timetabling
- Motion, person tracking
- RNA sequence-structure alignment
- Protein Folding

Algorithm Competitions

- SAT, MIP, TSP, AI planning, ASP, SMT, timetabling, protein folding, ...

Applications by Others

- Kidney exchange
- Linear algebra subroutines
- Java garbage collection
- Computer GO
- Linear algebra subroutines
- Evolutionary Algorithms
- ML: Classification

A Case Study

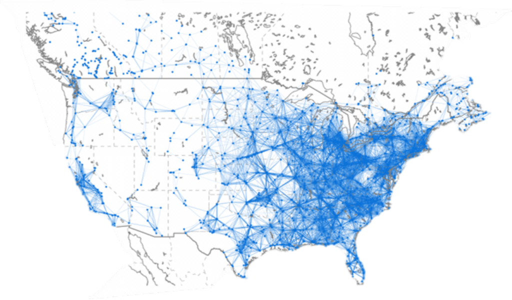
[L-B, Milgrom & Segal, 2017; Newman, Fréchette & L-B, 2017]

Over 13 months in 2016–17 the FCC held an "incentive auction" to **repurpose radio spectrum** from broadcast television to wireless internet

In total, the auction yielded **$19.8 billion**

- over $10 billion was paid to 175 broadcasters for **voluntarily relinquishing their licenses** across 14 UHF channels (84 MHz)
- Stations that continued broadcasting were assigned **potentially new channels** to fit as densely as possible into the channels that remained
- The government **netted over $7 billion** (used to pay down the national debt) after covering costs

Feasibility Testing

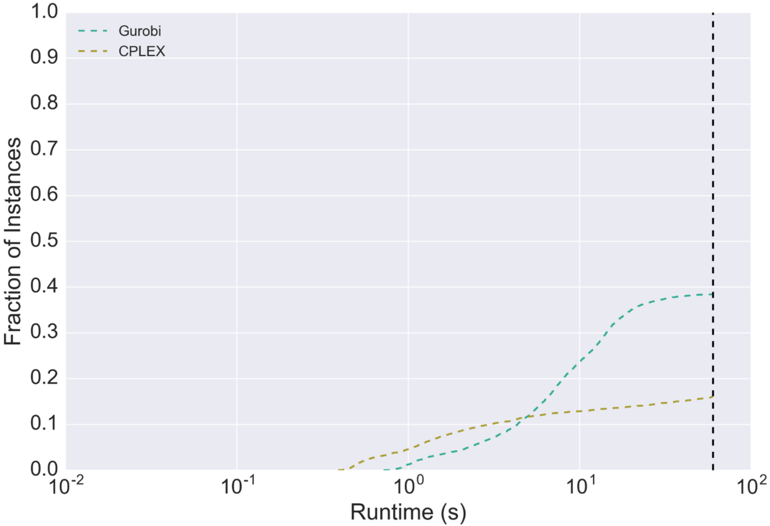


- A key subproblem in the auction:
  - asking "could station x **leave the auction** and go back on-air into the reduced band of spectrum, alongside all other stations X who have already done the same?
  - about 100K such problems arise per auction
  - about 20K are nontrivial
- A hard **graph-colouring problem**
  - 2990 stations (nodes)
  - 2.7 million interference constraints (channel-specific interference)
  - Initial skepticism about whether this problem could be solved exactly at a national scale
- What happens when we can't solve an instance:
  - Needed a minimum of two price decrements per 8h business day
  - each feasibility check was allowed a maximum of one minute
  - Treat **unsolved problems as infeasible**, raising the amount they're paid

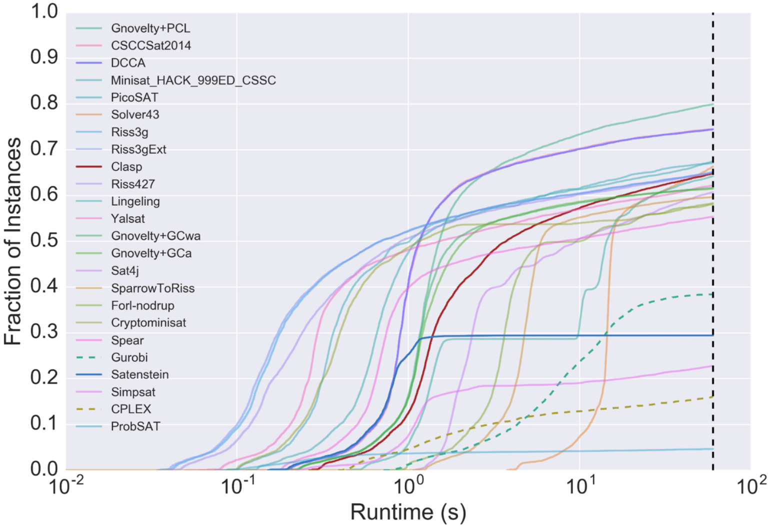
First, We Need Some Data

- We wrote a full **reverse auction simulator** (open source)
- **Generated valuations** by sampling from a model due to Doraszelski et al. [2016]
- Assumptions:
  - 84 MHz clearing target
  - stations participated when their private value for continuing to broadcast was smaller than their opening offer for going off-air
  - 1 min timeout given to SATFC
- 20 simulated auctions ⇒ **60,057 instances**
  - 2,711–3,285 instances per auction
  - all not solvable by directly augmenting the previous solution
  - about 3% of the problems encountered in full simulations
- Our goal: solve problems within a **one-minute cutoff**

The Incumbent Solution: MIP Encoding



What about trying SAT solvers?



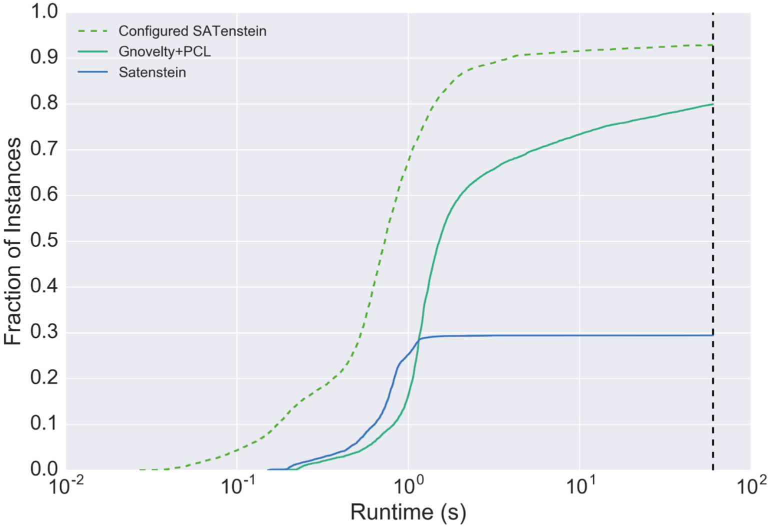
Setting Up an Algorithm Design Hypothesis Space

- Choice of complete or local-search **solver**
  - with which solver parameters
    - and, depending on solver, conditional subparameters?

- Various **problem-specific speedups**
  (each of which furthermore had parameters of its own)
  - reusing previous solutions
  - problem decomposition
  - caching similar solutions
  - removing underconstrained stations

- And further **problem-independent heuristics**
  - constraint propagation preprocessor
  - different SAT encodings

Algorithm Configuration to the Rescue



Algorithm Portfolios

[L-B, Nudelman, Shoham, 2002-2009; Xu, Hutter, Hoos, L-B, 2007-12]

Often **different solvers perform well on different instances**

- Idea: build an **algorithm portfolio**, consisting of different algorithms that can work together to solve a problem
- **SATzilla**: state-of-the-art portfolio developed by my group
  - machine learning to choose algorithm on a per-instance basis
- Or, just run all the algorithms together in parallel

Algorithm Portfolios

[L-B, Nudelman, Shoham, 2002-2009; Xu, Hutter, Hoos, L-B, 2007-12]

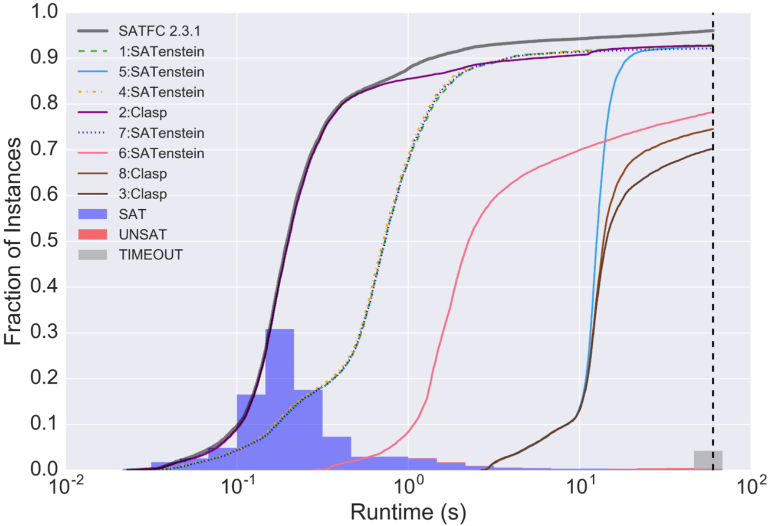
Often **different solvers perform well on different instances**

- Idea: build an **algorithm portfolio**, consisting of different algorithms that can work together to solve a problem
- **SATzilla**: state-of-the-art portfolio developed by my group
  - machine learning to choose algorithm on a per-instance basis
- Or, just run all the algorithms together in parallel

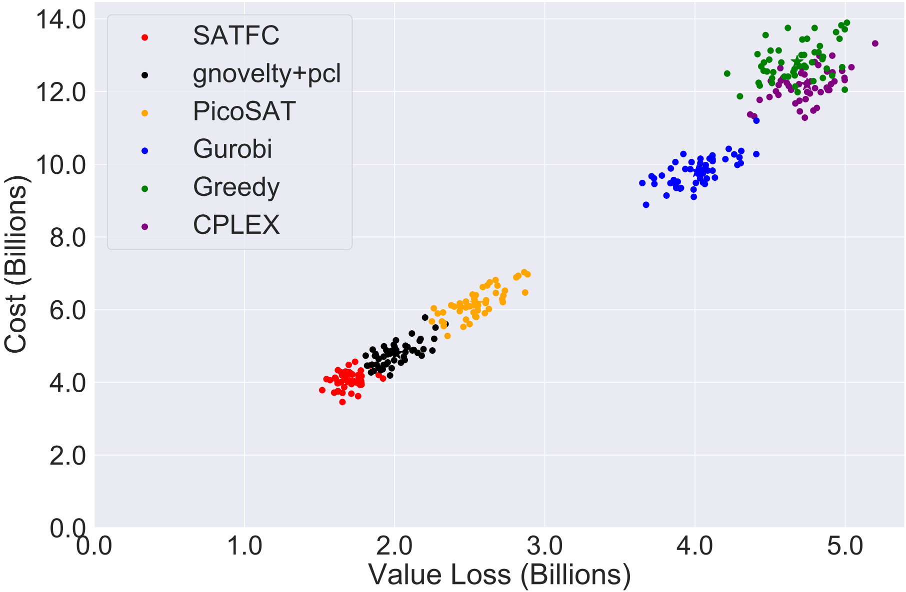
Hydra: use algorithm configuration to **learn a portfolio** of complementary algorithms

- augment an additional portfolio P by targeting instances on which P performs poorly
- Give the algorithm configuration method a dynamic performance metric:
  - performance of alg s when s outperforms P; performance of P

Performance of the Algorithm Portfolio



Economic Impact of a Stronger Solver



This Tutorial

High-Level Outline

Introduction, Technical Preliminaries, and a Case Study (Kevin)

Practical Methods for Algorithm Configuration (Frank)

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

Beyond Static Configuration: Related Problems and Emerging Directions (Frank)

Follow along: http://bit.ly/ACTutorial

This Tutorial

Section Outline

Follow along: http://bit.ly/ACTutorial

The basic components of algorithm configuration methods

Recall the core of the algorithm configuration definition

Find: $\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \mathbb{E}_{\pi\sim\mathcal{D}}(m(\boldsymbol{\theta}, \pi))$.

The two components of algorithm configuration methods

- How to select a new configuration to evaluate?
- How to compare this configuration to the best so far?

Sequential Model-based AC (SMAC): high-level overview

Algorithm 1: SMAC (high-level overview)

Learn a model $\hat{m}$ from performance data so far: $\hat{m} : \mathbf{\Theta} \times \Pi \rightarrow \mathbb{R}$

Use model $\hat{m}$ to select promising configurations $\mathbf{\Theta}_{new}$

Sequential Model-based AC (SMAC): high-level overview

Algorithm 1: SMAC (high-level overview)

Learn a model $\hat{m}$ from performance data so far: $\hat{m} : \mathbf{\Theta} \times \Pi \rightarrow \mathbb{R}$

Use model $\hat{m}$ to select promising configurations $\mathbf{\Theta}_{new}$

Compare $\mathbf{\Theta}_{new}$ against best configuration so far by executing new algorithm runs

Sequential Model-based AC (SMAC): high-level overview

Algorithm 1: SMAC (high-level overview)

Initialize by executing some runs and collecting their performance data

repeat

  Learn a model $\hat{m}$ from performance data so far: $\hat{m} : \mathbf{\Theta} \times \Pi \rightarrow \mathbb{R}$

  Use model $\hat{m}$ to select promising configurations $\mathbf{\Theta}_{new}$

  Compare $\mathbf{\Theta}_{new}$ against best configuration so far by executing new algorithm runs

until *time budget exhausted*

Sequential Model-based AC (SMAC): high-level overview

Algorithm 1: SMAC (high-level overview)

Initialize by executing some runs and collecting their performance data

repeat

  Learn a model $\hat{m}$ from performance data so far: $\hat{m} : \mathbf{\Theta} \times \Pi \rightarrow \mathbb{R}$

  Use model $\hat{m}$ to select promising configurations $\mathbf{\Theta}_{new}$ $\rightsquigarrow$ **Bayesian optimization**

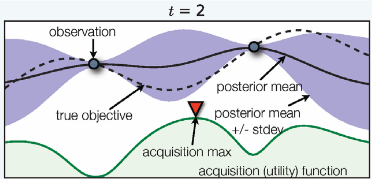
  Compare $\mathbf{\Theta}_{new}$ against best configuration so far by executing new algorithm runs

until *time budget exhausted*

Bayesian Optimization

General approach

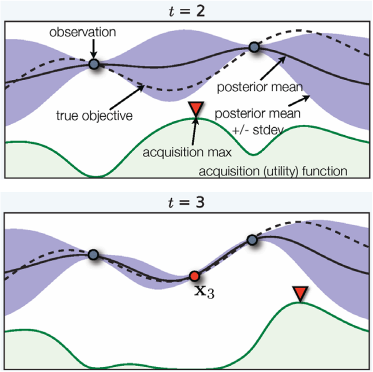
- Fit a probabilistic model to the collected function samples $\langle \boldsymbol{\theta}, f(\boldsymbol{\theta}) \rangle$
- Use the model to guide optimization, trading off exploration *vs* exploitation



Bayesian Optimization

General approach

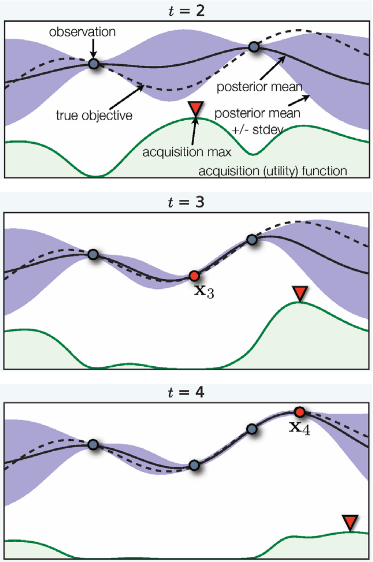
- Fit a probabilistic model to the collected function samples $\langle \boldsymbol{\theta}, f(\boldsymbol{\theta}) \rangle$
- Use the model to guide optimization, trading off exploration *vs* exploitation



Bayesian Optimization

General approach

- Fit a probabilistic model to the collected function samples $\langle \boldsymbol{\theta}, f(\boldsymbol{\theta}) \rangle$
- Use the model to guide optimization, trading off exploration *vs* exploitation



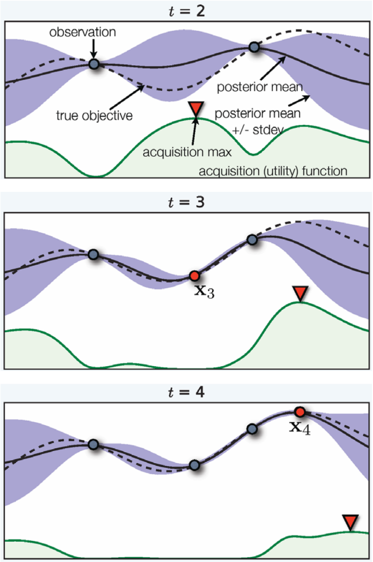
Bayesian Optimization

General approach

- Fit a probabilistic model to the collected function samples $\langle \boldsymbol{\theta}, f(\boldsymbol{\theta}) \rangle$
- Use the model to guide optimization, trading off exploration *vs* exploitation

Popular in the statistics literature [since Mockus, 1978]

- Efficient in # function evaluations
- Works when objective is nonconvex, noisy, has unknown derivatives, etc
- Recent convergence results [Srinivas et al, 2010; Bull 2011; de Freitas et al, 2012; Kawaguchi et al, 2015]



Sequential Model-based AC (SMAC): high-level overview

Algorithm 1: SMAC (high-level overview)

Initialize by executing some runs and collecting their performance data

repeat

Learn a model $\hat{m}$ from performance data so far: $\hat{m} : \mathbf{\Theta} \times \Pi \rightarrow \mathbb{R}$

Use model $\hat{m}$ to select promising configurations $\mathbf{\Theta}_{new}$

$\rightsquigarrow$ **Bayesian optimization with random forests**

Compare $\mathbf{\Theta}_{new}$ against best configuration so far by executing new algorithm runs

$\rightsquigarrow$ **How many instances to evaluate for $\theta \in \mathbf{\Theta}_{new}$?**

until *time budget exhausted*

How many instances to evaluate per configuration?

Performance on individual instances often does not generalize

- Instance hardness varies (from milliseconds to hours)
- Aim to minimize cost in expectation over instances: $c(\boldsymbol{\theta}) = \mathbb{E}_{\pi\sim\mathcal{D}}(m(\boldsymbol{\theta}, \pi))$

How many instances to evaluate per configuration?

Performance on individual instances often does not generalize

- Instance hardness varies (from milliseconds to hours)
- Aim to minimize cost in expectation over instances: $c(\boldsymbol{\theta}) = \mathbb{E}_{\pi \sim \mathcal{D}}(m(\boldsymbol{\theta}, \pi))$

Simplest, suboptimal solution: use N instances for each evaluation

- Treats the problem as a blackbox function optimization problem
- Issue: how large to choose N?
  - too small: overtuning (equivalent to over-fitting)
  - too large: every function evaluation is slow

SMAC's racing approach: focus on configurations that might beat the incumbent

- Race new configurations against the best known **incumbent configuration $\hat{\theta}$**
  - Use same instances (and seeds) as previously used for $\hat{\boldsymbol{\theta}}$
  - Aggressively discard new configuration $\boldsymbol{\theta}$ if it performs worse than $\hat{\boldsymbol{\theta}}$ on shared runs

SMAC's racing approach: focus on configurations that might beat the incumbent

- Race new configurations against the best known **incumbent configuration $\hat{\theta}$**
  - Use same instances (and seeds) as previously used for $\hat{\boldsymbol{\theta}}$
  - Aggressively discard new configuration $\boldsymbol{\theta}$ if it performs worse than $\hat{\boldsymbol{\theta}}$ on shared runs
    - No requirement for statistical domination
      (this would be inefficient since there are exponentially many bad configurations)
    - Search component allows to return to $\boldsymbol{\theta}$ even if it is discarded based on current runs

SMAC's racing approach: focus on configurations that might beat the incumbent

- Race new configurations against the best known **incumbent configuration $\hat{\theta}$**
  - Use same instances (and seeds) as previously used for $\hat{\boldsymbol{\theta}}$
  - Aggressively discard new configuration $\boldsymbol{\theta}$ if it performs worse than $\hat{\boldsymbol{\theta}}$ on shared runs
    - No requirement for statistical domination (this would be inefficient since there are exponentially many bad configurations)
    - Search component allows to return to $\boldsymbol{\theta}$ even if it is discarded based on current runs
  - Add more runs for $\hat{\boldsymbol{\theta}}$ over time $\rightsquigarrow$ build up confidence in $\hat{\boldsymbol{\theta}}$

SMAC's racing approach: focus on configurations that might beat the incumbent

- Race new configurations against the best known **incumbent configuration $\hat{\theta}$**
  - Use same instances (and seeds) as previously used for $\hat{\boldsymbol{\theta}}$
  - Aggressively discard new configuration $\boldsymbol{\theta}$ if it performs worse than $\hat{\boldsymbol{\theta}}$ on shared runs
    - No requirement for statistical domination
      (this would be inefficient since there are exponentially many bad configurations)
    - Search component allows to return to $\boldsymbol{\theta}$ even if it is discarded based on current runs
  - Add more runs for $\hat{\boldsymbol{\theta}}$ over time $\rightsquigarrow$ build up confidence in $\hat{\boldsymbol{\theta}}$

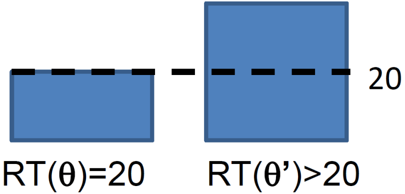
Observation

Let $\boldsymbol{\Theta}$ be finite. Then, the **probability that SMAC finds the true optimal parameter configuration $\theta^* \in \boldsymbol{\Theta}$ approaches 1** as the number of executed runs goes to infinity.

Saving More Time: Adaptive Capping

When minimizing algorithm runtime,
we can terminate runs for poor configurations $\boldsymbol{\theta}'$ early:

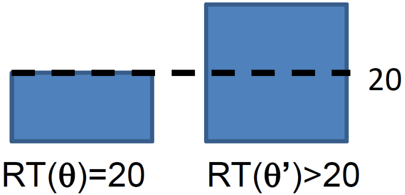
- Is $\boldsymbol{\theta}'$ better than $\boldsymbol{\theta}$?
  - Example:



Saving More Time: Adaptive Capping

When minimizing algorithm runtime,
we can terminate runs for poor configurations $\boldsymbol{\theta}'$ early:

- Is $\boldsymbol{\theta}'$ better than $\boldsymbol{\theta}$?
  - Example:

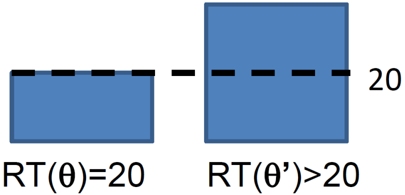


- Can terminate evaluation of $\boldsymbol{\theta}'$ once it is guaranteed to be worse than $\boldsymbol{\theta}$

Saving More Time: Adaptive Capping

When minimizing algorithm runtime,
we can terminate runs for poor configurations $\boldsymbol{\theta}'$ early:

- Is $\boldsymbol{\theta}'$ better than $\boldsymbol{\theta}$?
  - Example:



- Can terminate evaluation of $\boldsymbol{\theta}'$ once it is guaranteed to be worse than $\boldsymbol{\theta}$

Observation

Let $\boldsymbol{\Theta}$ be finite. Then, the **probability that SMAC with adaptive capping finds the true optimal parameter configuration $\theta^* \in \boldsymbol{\Theta}$ approaches 1** the number of executed runs goes to infinity.

Sequential Model-based AC (SMAC): summary

Algorithm 1: SMAC

Initialize by executing some runs and collecting their performance data

repeat

Learn a model $\hat{m}$ from performance data so far: $\hat{m} : \mathbf{\Theta} \times \Pi \rightarrow \mathbb{R}$

Use model $\hat{m}$ to select promising configurations $\mathbf{\Theta}_{new}$

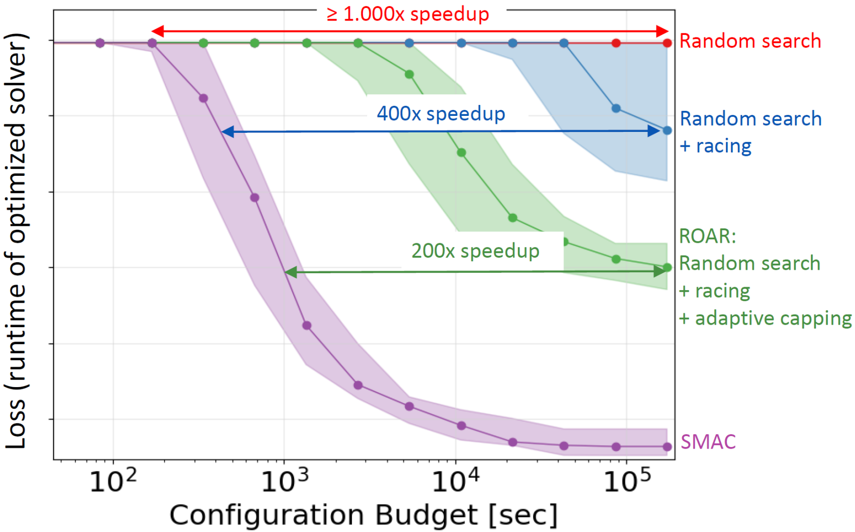
$\rightsquigarrow$ **Bayesian optimization with random forests**

Compare $\mathbf{\Theta}_{new}$ against best configuration so far by executing new algorithm runs

$\rightsquigarrow$ **Aggressive racing and adaptive capping**

until *time budget exhausted*

All of SMAC's components matter for performance



Example: Optimizing CPLEX on combinatorial auctions (Regions 100)

This Tutorial

Section Outline

Follow along: http://bit.ly/ACTutorial

AC poses many non-standard challenges to Bayesian optimization

Complex parameter space

- High dimensionality (low effective dimensionality) [Wang et al, 2013; Garnett et al., 2013]
- Mixed continuous/discrete parameters [H., 2009; H. et al, 2014]
- Conditional parameters [Swersky et al, 2013; H. & Osborne, 2013; Levesque et al., 2017]

AC poses many non-standard challenges to Bayesian optimization

Complex parameter space

- High dimensionality (low effective dimensionality) [Wang et al, 2013; Garnett et al., 2013]
- Mixed continuous/discrete parameters [H., 2009; H. et al, 2014]
- Conditional parameters [Swersky et al, 2013; H. & Osborne, 2013; Levesque et al., 2017]

Non-standard noise

- Non-Gaussian noise [Williams et al, 2000; Shah et al, 2018; Martinez-Cantinet al, 2018]
- Heteroscedastic noise [Le et al, Wang & Neal, 2012]

AC poses many non-standard challenges to Bayesian optimization

Complex parameter space

- High dimensionality (low effective dimensionality) [Wang et al, 2013; Garnett et al., 2013]
- Mixed continuous/discrete parameters [H., 2009; H. et al, 2014]
- Conditional parameters [Swersky et al, 2013; H. & Osborne, 2013; Levesque et al., 2017]

Non-standard noise

- Non-Gaussian noise [Williams et al, 2000; Shah et al, 2018; Martinez-Cantinet al, 2018]
- Heteroscedastic noise [Le et al, Wang & Neal, 2012]

Efficient use in off-the-shelf Bayesian optimization

- Robustness of the model [Malkomes and Garnett, 2018]
- Model overhead [Quiñonero-Candela & Rasmussen, 2005; Bui et al, 2018; H. et al, 2010; Snoek et al, 2015]

AC poses many non-standard challenges to Bayesian optimization

Complex parameter space

- High dimensionality (low effective dimensionality) [Wang et al, 2013; Garnett et al., 2013]
- Mixed continuous/discrete parameters [H., 2009; H. et al, 2014]
- Conditional parameters [Swersky et al, 2013; H. & Osborne, 2013; Levesque et al., 2017]

Non-standard noise

- Non-Gaussian noise [Williams et al, 2000; Shah et al, 2018; Martinez-Cantinet al, 2018]
- Heteroscedastic noise [Le et al, Wang & Neal, 2012]

Efficient use in off-the-shelf Bayesian optimization

- Robustness of the model [Malkomes and Garnett, 2018]
- Model overhead [Quiñonero-Candela & Rasmussen, 2005; Bui et al, 2018; H. et al, 2010; Snoek et al, 2015]

We'll use random forests to address all these; but we need **uncertainty estimates**

Adaptation of regression trees: storing empirical variance in every leaf

param 1	feature 2	param 3	runtime
false	2	red	3.7
false	2.5	blue	20
true	5.5	red	2.1
false	5.5	blue	25
false	5	red	1.2
true	4.5	green	19
true	4	blue	12
true	3.5	green	17

$param_3 \in \{red\}$

param 1	feature 2	param 3	runtime
false	2	red	3.7
true	5.5	red	2.1
false	5	red	1.2

$feature_2 \leq 3.5$

param 1	feature 2	param 3	runtime
false	2	red	3.7

$\mu = 3.7$

$\sigma^2 = 0$

$feature_2 > 3.5$

param 1	feature 2	param 3	runtime
true	5.5	red	2.1
false	5	red	1.2

$\mu = 1.65$

$\sigma^2 \approx 0.20$

$param_3 \in \{blue, green\}$

param 1	feature 2	param 3	runtime
false	2.5	blue	20
false	5.5	blue	25
true	4.5	green	19
true	4	blue	12
true	3.5	green	17

$\mu = 18.6$

$\sigma^2 = 4.72$

Random Forests with Uncertainty Predictions

- Random forest as a **mixture model** of T trees [H. et al., 2014]
- Predict with each of the forest's trees: μ_t and σ_t^2 for tree t
- Predictive distribution: $\mathcal{N}(\mu, \sigma^2)$ with

$$\mu = \frac{1}{T}\sum_{t=1}^{T}\mu_t$$

$$\sigma^2 = \underbrace{\left(\frac{1}{T}\sum_{t=1}^{T}\sigma_t^2\right)}_{\text{mean of the variances}} + \underbrace{\frac{1}{T}\left(\sum_{t=1}^{T}\mu_t^2\right) - \mu^2}_{\text{variance of the means}}$$

Another recent variant for uncertainty in random forests: Mondrian forests

[Lakshminarayanan, Roy & Teh, 2015; Lakshminarayanan, Roy & Teh, 2016]

A key modification of random forests: sampling split points

param 1	param 2	param 3	runtime
false	2	red	3.7
true	5.5	red	2.1
false	5	red	1.2

$\text{param}_2 \leq 3.5$

param 1	param 2	param 3	runtime
false	2	red	3.7

$\text{param}_2 > 3.5$

param 1	param 2	param 3	runtime
true	5.5	red	2.1
false	5	red	1.2

- To obtain this split, the split point should be somewhere between **L=2**, **U=5**
- Standard: split at mid-point $\frac{1}{2}(L+U) = 3.5$
- Now instead: **sample split point from Uniform [L,U]**

A key modification of random forests: sampling split points

param 1	param 2	param 3	runtime
false	2	red	3.7
true	5.5	red	2.1
false	5	red	1.2

$\text{param}_2 \leq 3.5$

param 1	param 2	param 3	runtime
false	2	red	3.7

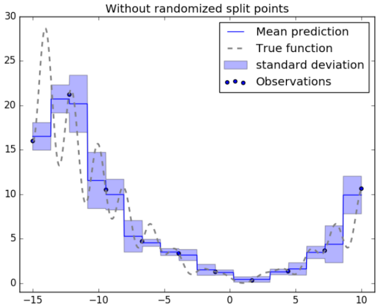
$\text{param}_2 > 3.5$

param 1	param 2	param 3	runtime
true	5.5	red	2.1
false	5	red	1.2

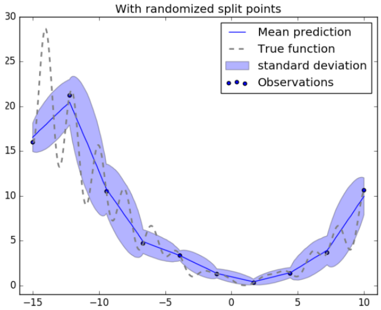
- To obtain this split, the split point should be somewhere between **L=2**, **U=5**
- Standard: split at mid-point $\frac{1}{2}(L+U) = 3.5$
- Now instead: **sample split point from Uniform [L,U]**

Random forests with better uncertainty estimates

- Sampling split points is crucial to obtain smooth uncertainty estimates



Split at mid point 0.5 (L+U)

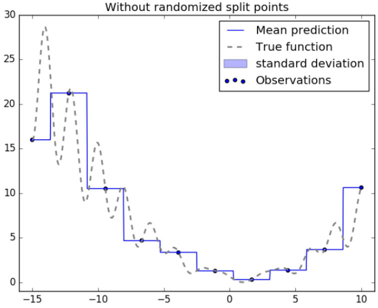


Split at sample from Uniform[L, U]

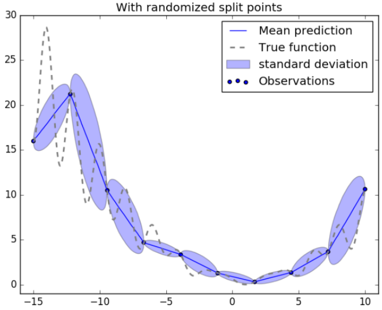
1000 trees, min. number of points per leaf = 1; with bootstrapping

Random forests with better uncertainty estimates

- Sampling split points is crucial to obtain smooth uncertainty estimates



Split at mid point 0.5 (L+U)



Split at sample from Uniform[L, U]

1000 trees, min. number of points per leaf = 1; without bootstrapping

Aggregating Model Predictions Across Multiple Instances

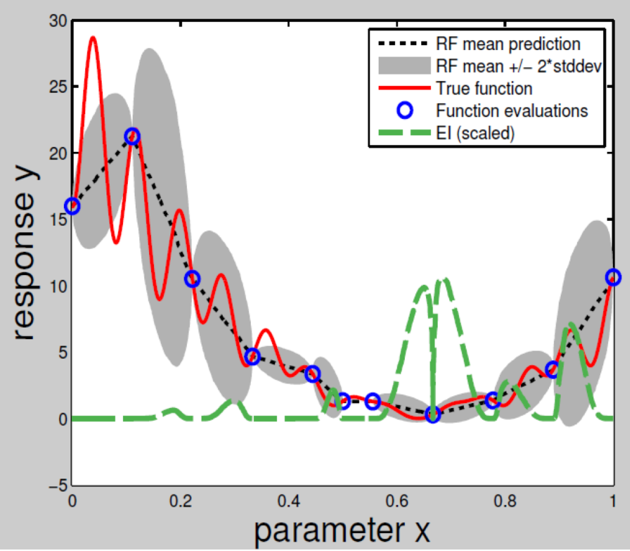
Problem

- Model $\hat{m} : \boldsymbol{\Theta} \times \Pi \to \mathbb{R}$ predicts for one instance at a time
- We want a model that marginalizes over instances: $\hat{f}(\boldsymbol{\theta}) = \mathbb{E}_{\pi \sim \mathcal{D}}(\hat{m}(\boldsymbol{\theta}, \pi))$

Solution

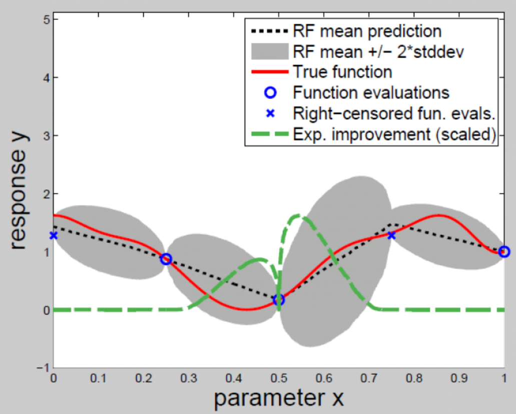
- Intuition: predict for each instance and then average
- More efficient implementation in random forests
  - Keep track of fraction of instances compatible with each leaf
  - Weight the predictions of the leaves accordingly

Bayesian optimization with random forests



Bayesian optimization with censored data

- Terminating poor runs early yields **censored** data points
  - ⇝ we only know a **lower bound** for some data points
- Use an EM-style approach to fill in censored values [Schmee & Hahn, 1979; H. et al, 2013]



Handling of conditional parameters in random forests

- Only split on a parameter if it's guaranteed to be active in the current node
  - Splits higher up in the tree must guarantee parent parameters to have right values

Handling of conditional parameters in random forests

- Only split on a parameter if it's guaranteed to be active in the current node
  - Splits higher up in the tree must guarantee parent parameters to have right values
- Empirically, both GPs and RFs have their advantages [Eggensperger et al, 2013]

Low-dimensional, continuous

Experiment	#evals	SMAC Valid. loss	Spearmint Valid. loss	TPE Valid. loss
branin (0.398)	200	0.655±0.27	**0.398**±0.00	0.526± 0.13
har6 (-3.322)	200	-2.977±0.11	**-3.133**±0.41	-2.823±0.18
Log.Regression	100	8.6±0.9	**7.3**±0.2	8.2±0.6
LDA ongrid	50	**1269.6**±2.9	1272.6±10.3	1271.5±3.5
SVM ongrid	100	**24.1**±0.1	24.6±0.9	24.2±0.0

Handling of conditional parameters in random forests

- Only split on a parameter if it's guaranteed to be active in the current node
  - Splits higher up in the tree must guarantee parent parameters to have right values
- Empirically, both GPs and RFs have their advantages [Eggensperger et al, 2013]

Low-dimensional, continuous

Experiment	#evals	SMAC Valid. loss	Spearmint Valid. loss	TPE Valid. loss
branin (0.398)	200	0.655±0.27	**0.398**±0.00	0.526± 0.13
har6 (-3.322)	200	-2.977±0.11	**-3.133**±0.41	-2.823±0.18
Log.Regression	100	8.6±0.9	**7.3**±0.2	8.2±0.6
LDA ongrid	50	**1269.6**±2.9	1272.6±10.3	1271.5±3.5
SVM ongrid	100	**24.1**±0.1	24.6±0.9	24.2±0.0
HP-NNET convex	200	**18.3**±1.9	20.0±0.9	18.5±1.4
HP-NNET MRBI	200	**48.3**±1.80	51.4±3.2	48.9±1.4
HP-DBNET convex	200	**15.4**±0.8	17.45±5.6	16.1±0.5
Auto-WEKA	30h	**27.5**±4.9	40.64±7.2	35.5±2.9

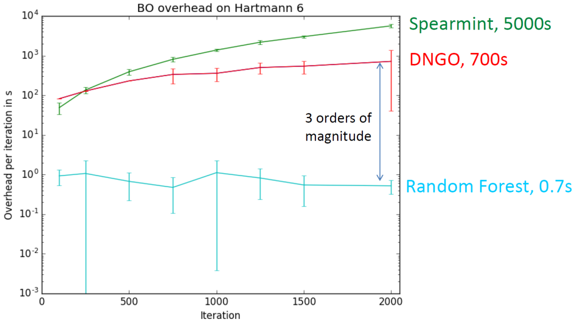
High-dimensional, conditional

Computational efficiency of random forests and standard Gaussian processes

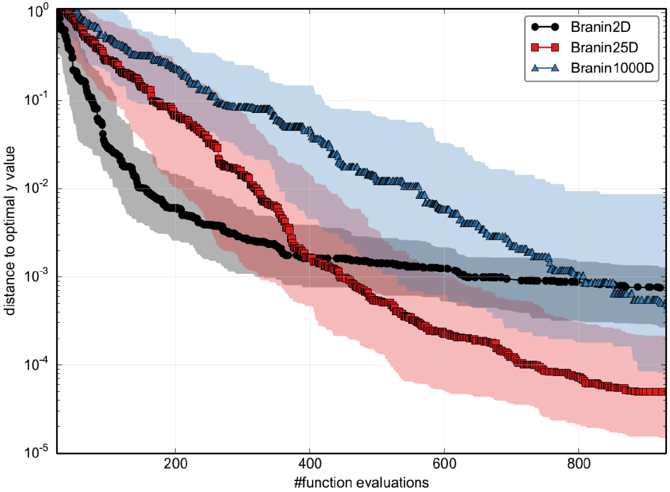
Computational complexity for N data points (and T trees in a forest)

	Random forests	Standard GPs
Training	$O(TN\log^2 N)$	$O(N^3)$
Prediction	$O(T\log N)$	$O(N^2)$

Empirical scaling of runtime with the number of data points:



Scaling with high dimensions (low effective dimensionality)



2 important dimensions (Branin test function)

+ additional unimportant dimensions, following Wang et al [2013]

This Tutorial

Section Outline

Practical Methods for Algorithm Configuration (Frank)

- Sequential Model-Based Algorithm Configuration (SMAC)
- Details on the Bayesian Optimization in SMAC
- **Other Algorithm Configuration Methods**
- Case Studies and Evaluation

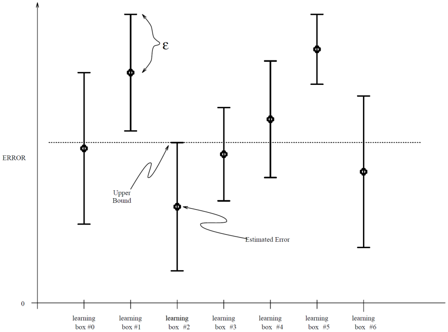
Follow along: http://bit.ly/ACTutorial

There are many continuous blackbox optimization methods

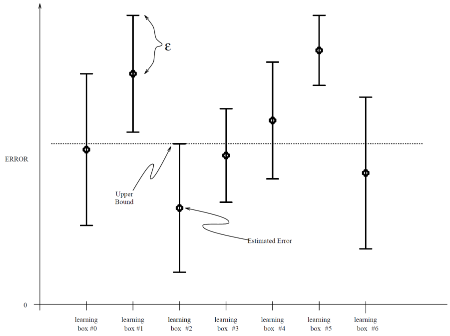
- Evolutionary strategies, e.g., CMA-ES [Hansen & Ostermeier, 2001; Hansen, 2016]
  - Strong results for **continuous hyperparameter optimization** [Friedrichs & Igel, 2004], especially with parallel resources [Loshchilov & H., 2016]
  - Also strong results for **optimizing NN parameters**, especially when only approximate gradients are available (RL) [Salimans et al, 2017; Conti et al, 2018, Chrabaszcz et al, 2018]
- Differential evolution [Storn and Price, 1997]
- Particle swarm optimization [Kennedy & Eberhart, 1995]

⇝ For continuous parameter spaces, these could be used instead of Bayesian optimization

There are many approaches for model selection



There are many approaches for model selection



- E.g., Hoeffding races [Maron & Moore, 1993]
- To compare a set of configurations (or algorithms):
  - Use Hoeffding's bound to compute a confidence band for each configuration
  - Stop evaluating configuration when its lower bound is above another's upper bound

F-race and Iterated F-race

F-race [Birattari et al, 2002]

- Similar idea as Hoeffding races
- But uses a statistical test instead to check whether $\boldsymbol{\theta}$ is inferior
  - Namely, the F-test, followed by pairwise t-tests

Iterated F-Race [López-Ibáñez et al, 2016]

- Maintain a probability distribution over which configurations are good
- Sample k configurations from that distribution & race them with F-race
- Update distributions with the results of the race

$\rightsquigarrow$ Focus on solution quality optimization

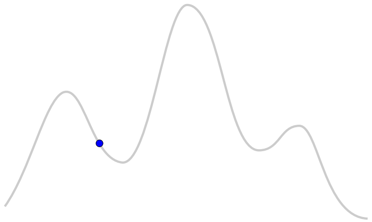
The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

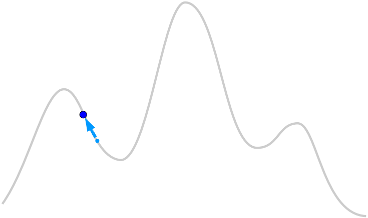


Initialisation

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

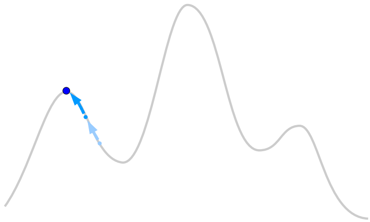


Local Search

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

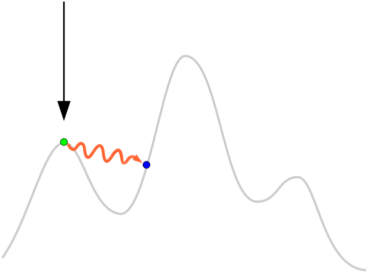


Local Search

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

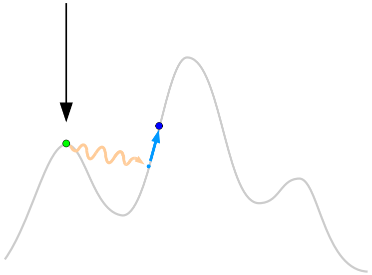


Perturbation

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

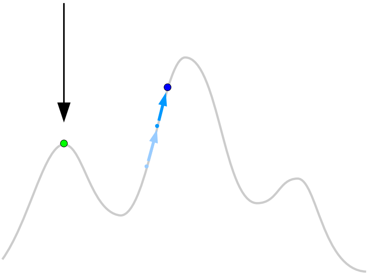


Local Search

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

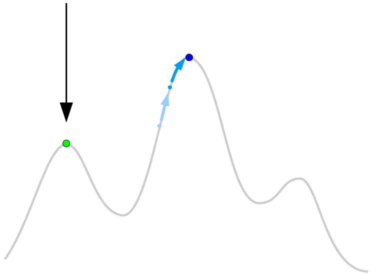


Local Search

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

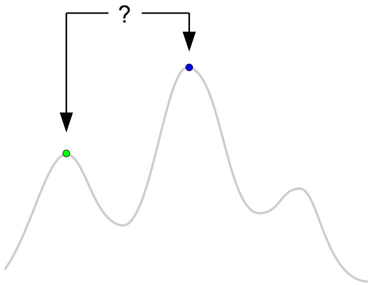


Local Search

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

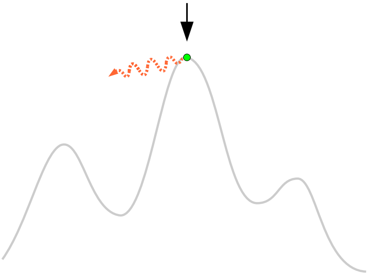


Selection (using Acceptance Criterion)

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]

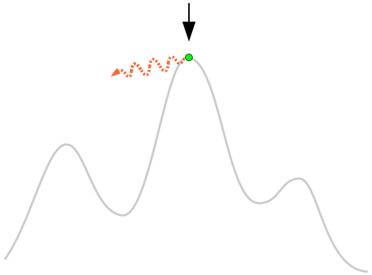


Perturbation

Animation credit: Holger Hoos

The ParamILS Framework

Iterated local search in parameter configuration space [H. et al, 2007; H. et al, 2009]



Perturbation

Animation credit: Holger Hoos

ParamILS predates SMAC; **aggressive racing & adaptive capping originate here**

Gender-based Genetic Algorithm (GGA) [Ansotegui et al, 2009]

Genetic algorithm:

- Population of individuals as genomes (i.e., solution candidates)
- Modify population by
  - Mutations (i.e., random changes)
  - Crossover (i.e., combination of 2 parents to form an offspring)

Gender-based Genetic Algorithm (GGA) [Ansotegui et al, 2009]

Genetic algorithm:

- Population of individuals as genomes (i.e., solution candidates)
- Modify population by
  - Mutations (i.e., random changes)
  - Crossover (i.e., combination of 2 parents to form an offspring)

Genetic algorithm for algorithm configuration

- **Genome = parameter configuration**
- Crossover: Combine 2 configurations to form a new configuration

Gender-based Genetic Algorithm (GGA) [Ansotegui et al, 2009]

Genetic algorithm:

- Population of individuals as genomes (i.e., solution candidates)
- Modify population by
  - Mutations (i.e., random changes)
  - Crossover (i.e., combination of 2 parents to form an offspring)

Genetic algorithm for algorithm configuration

- **Genome = parameter configuration**
- Crossover: Combine 2 configurations to form a new configuration

Two genders in the population (competitive and non-competitive)

- Selection pressure only on one gender
- Preserves diversity of the population

GGA: Racing and Capping

Can exploit parallel resources

- **Evaluate population members in parallel**
- Adaptive capping: can stop when the first k succeed

GGA: Racing and Capping

Can exploit parallel resources

- **Evaluate population members in parallel**
- Adaptive capping: can stop when the first k succeed

Use N instances to evaluate configurations

- Increase N in each generation
- Linear increase from N_{start} to N_{end}

This Tutorial

Section Outline

Practical Methods for Algorithm Configuration (Frank)

Sequential Model-Based Algorithm Configuration (SMAC)

Details on the Bayesian Optimization in SMAC

Other Algorithm Configuration Methods

Case Studies and Evaluation

Follow along: http://bit.ly/ACTutorial

Configuration of a SAT Solver for Verification [H. et al, FMCAD 2007]

SAT-encoded instances from formal verification

- Software verification [Babić & Hu; CAV '07]
- IBM bounded model checking [Zarpas; SAT '05]

State-of-the-art tree search solver for SAT-based verification

- Spear, developed by Domagoj Babić at UBC
- 26 parameters, 8.34×10^{17} configurations

Configuration of a SAT Solver for Verification [H. et al, FMCAD 2007]

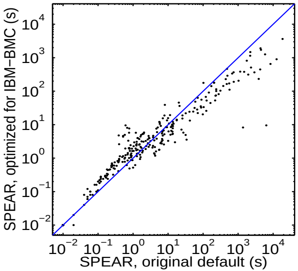
- Ran ParamILS, 2 days $\times$ 10 machines
  - On a training set from each of hardware and software verification

Configuration of a SAT Solver for Verification [H. et al, FMCAD 2007]

- Ran ParamILS, 2 days $\times$ 10 machines
  - On a training set from each of hardware and software verification
- Compared to manually-engineered default
  - 1 week of performance tuning
  - Competitive with the state of the art
  - Comparison on unseen test instances

Configuration of a SAT Solver for Verification [H. et al, FMCAD 2007]

- Ran ParamILS, 2 days $\times$ 10 machines
  - On a training set from each of hardware and software verification
- Compared to manually-engineered default
  - 1 week of performance tuning
  - Competitive with the state of the art
  - Comparison on unseen test instances

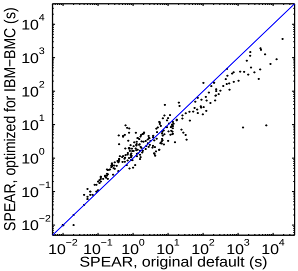


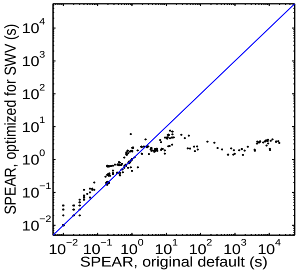
IBM Hardware verification:

4.5-fold speedup on average

Configuration of a SAT Solver for Verification [H. et al, FMCAD 2007]

- Ran ParamILS, 2 days × 10 machines
  - On a training set from each of hardware and software verification
- Compared to manually-engineered default
  - 1 week of performance tuning
  - Competitive with the state of the art
  - Comparison on unseen test instances





IBM Hardware verification: **4.5-fold speedup** on average

Software verification: **500-fold speedup** ⇝ won QF_BV category in **2007 SMT competition**

Configuration of a Commercial MIP Solver [H. et al, CPAIOR 2010]

Mixed integer programming (MIP)

$$\begin{aligned} \text{min} \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{Z} \text{ for } i \in I \end{aligned}$$

Combines efficiency of solving linear programs with representational power of integer variables

Configuration of a Commercial MIP Solver [H. et al, CPAIOR 2010]

Mixed integer programming (MIP)

$$\begin{aligned} \text{min} \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{Z} \text{ for } i \in I \end{aligned}$$

Combines efficiency of solving linear programs with representational power of integer variables

Commercial MIP Solver CPLEX

- Leading solver for 15 years (at the time)
- Licensed by over 1 000 universities and 1 300 corporations
- 76 parameters, 10^{47} configurations

Configuration of a Commercial MIP Solver [H. et al, CPAIOR 2010]

Mixed integer programming (MIP)

$$\begin{aligned} \text{min} \quad & c^{\mathsf{T}} x \\ \text{s.t.} \quad & Ax \leq b \\ & x_i \in \mathbb{Z} \text{ for } i \in I \end{aligned}$$

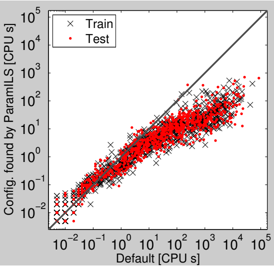
Combines efficiency of solving linear programs with representational power of integer variables

Commercial MIP Solver CPLEX

- Leading solver for 15 years (at the time)
- Licensed by over 1 000 universities and 1 300 corporations
- 76 parameters, 10^{47} configurations

Improvements by configuration with ParamILS

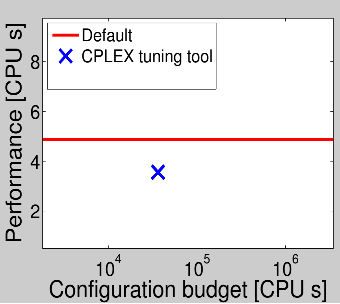
- Between $2\times$ and $50\times$ speedups to solve optimally
- Later work with CPLEX team: up to **10 000× speedups**
- Reduction of optimality gap: $1.3\times$ to $8.6\times$



Wildlife corridor instances

Comparison to CPLEX Tuning Tool [H. et al, CPAIOR 2010]

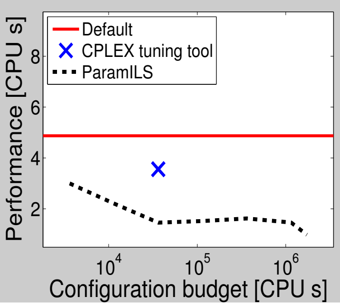
- CPLEX tuning tool
  - Introduced in version 11 (late 2007, after *ParamILS*)
  - Evaluates predefined good configurations, returns best one
  - Required runtime varies (from $< 1h$ to weeks)



CPLEX on MIK instances

Comparison to CPLEX Tuning Tool [H. et al, CPAIOR 2010]

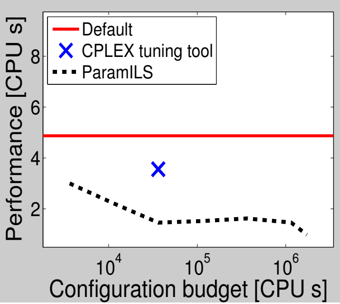
- CPLEX tuning tool
  - Introduced in version 11 (late 2007, after *ParamILS*)
  - Evaluates predefined good configurations, returns best one
  - Required runtime varies (from $< 1h$ to weeks)
- *ParamILS*: anytime algorithm
  - At each time step, keeps track of its incumbent



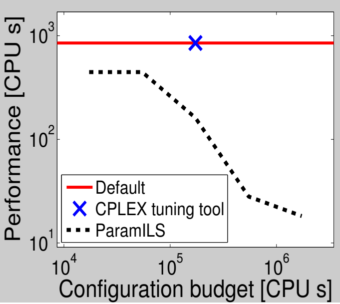
CPLEX on MIK instances

Comparison to CPLEX Tuning Tool [H. et al, CPAIOR 2010]

- CPLEX tuning tool
  - Introduced in version 11 (late 2007, after *ParamILS*)
  - Evaluates predefined good configurations, returns best one
  - Required runtime varies (from $< 1h$ to weeks)
- *ParamILS*: anytime algorithm
  - At each time step, keeps track of its incumbent



CPLEX on MIK instances



CPLEX on SUST instances

Note: lower is better

SMAC further improved performance for both of these case studies

AC scenario	GGA	ParamILS	SMAC
CPLEX on CLS	5.36	2.12	**1.77**
CPLEX on CORLAT	20.47	9.57	**5.38**
CPLEX on RCW2	63.65	54.09	**49.69**
CPLEX on Regions200	7.09	**3.04**	**3.09**
SPEAR on IBM	−−	801.32	**775.15**
SPEAR on SWV	−−	1.26	**0.87**

Configurable SAT Solver Competition (CSSC) [H. et al, AIJ 2015]

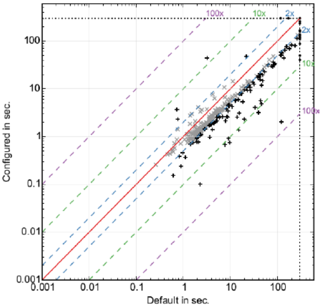
Annual SAT competition

- Scores SAT solvers by their performance across instances
- Medals for best average performance with solver defaults
- Implicitly highlights solvers with good defaults

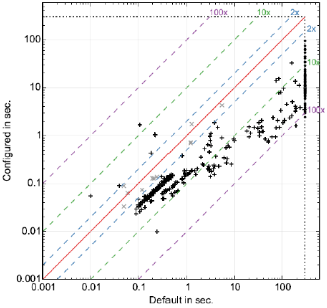
Configurable SAT Solver Challenge (CSSC)

- Better reflects an application setting: homogeneous instances
- Can automatically optimize parameters
- Medals for best **performance after configuration**
  - Based on configuration by all of SMAC, ParamILS and GGA

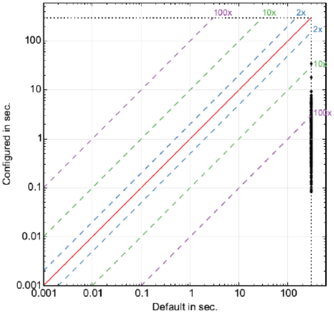
CSSC result #1: Solver performance often improved a lot



Lingeling on CircuitFuzz:
Timeouts: 119 → 107



Clasp on n-queens:
Timeouts: 211 → 102



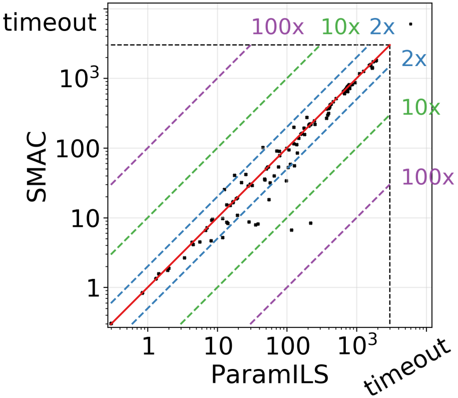
probSAT on unif rnd 5-SAT:
Timeouts: 250 → 0

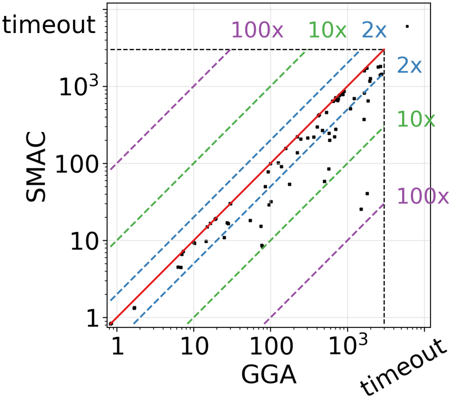
CSSC result #2: Automated configuration changed algorithm rankings

Example: random SAT+UNSAT category in 2013

Solver	CSSC ranking	Default ranking
Clasp	1	6
Lingeling	2	4
Riss3g	3	5
Solver43	4	2
Simpsat	5	1
Sat4j	6	3
For1-nodrup	7	7
gNovelty+GCwa	8	8
gNovelty+Gca	9	9
gNovelty+PCL	10	10

CSSC result #3: SMAC yielded larger speedups than ParamILS and GGA





Each dot: performance achieved by the two configurators being compared for one solver on one benchmark distribution

This Tutorial

High-Level Outline

Introduction, Technical Preliminaries, and a Case Study (Kevin)

Practical Methods for Algorithm Configuration (Frank)

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

Beyond Static Configuration: Related Problems and Emerging Directions (Frank)

Follow along: http://bit.ly/ACTutorial

Algorithm Configuration

- It's trivial to achieve **optimality in the limit**
  - what makes an algorithm configurator good is finding good configurations quickly
- So far our focus, like most of the literature, has been on empirical performance
- Let's now consider obtaining **meaningful theoretical guarantees about worst-case running time**
  - This section follows Kleinberg, L-B & Lucier [2017]
    - but uses notation consistent with the rest of this tutorial

This Tutorial

Section Outline

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

- **Technical Setup**
- Structured Procrastination (the case of few configurations)
- Extensions to Structured Procrastination (many configurations and more)
- LeapsAndBounds
- CapsAndRuns
- Structured Procrastination with Confidence
- Related Work and Further Reading

Follow along: http://bit.ly/ACTutorial

Problem Definition Redux

(Notation you'll need for this section, slide 1/2)

An **algorithm configuration problem** is defined by $(\mathcal{A}, \mathbf{\Theta}, \mathcal{D}, \bar{\kappa}, R)$:

- $\mathcal{A}$ is a parameterized **algorithm**
- $\mathbf{\Theta}$ is the parameter **configuration space** of $\mathcal{A}$
  - We use θ to identify particular configurations
- $\mathcal{D}$ is a **probability distribution over input instances** with domain Π; typically the uniform distribution over a benchmark set
  - We use π to identify (input instance, random seed) pairs, which we call *instances*
- $\bar{\kappa} < \infty$ is a **max cutoff time**, after which each run of $\mathcal{A}$ will be terminated
- $R_\kappa(\theta, \pi)$ is the **runtime** of configuration $\theta \in \mathbf{\Theta}$ on instance π, with cutoff time κ
  - $R_\kappa(\theta) = \mathbb{E}_{\pi\sim\mathcal{D}}[R_\kappa(\theta, \pi)]$ denotes **expected κ-capped running time** of θ
  - $R(\theta) = R_{\bar{\kappa}}(\theta)$ denotes **expected running time** of θ
- $\kappa_0 > 0$ is the **minimum runtime**: $R(\theta, \pi) \geq \kappa_0$ for all θ and π

Approximately Optimal Configurations

(Notation you'll need for this section, slide 2/2)

Let **OPT** $= \min_\theta\{R(\theta)\}$.

Definition (ϵ-Optimality)

Given $\epsilon > 0$, find $\theta^* \in \boldsymbol{\Theta}$ such that $R(\theta^*) \le (1+\epsilon)$OPT.

- If θ's average running time is driven by **a small set of exceedingly bad inputs that occur very rarely**, then we'd need to run θ on *many* inputs
- Implies worst-case bounds scaling **linearly with $\bar{\kappa}$** even when OPT $\ll \bar{\kappa}$

Approximately Optimal Configurations

(Notation you'll need for this section, slide 2/2)

Let **OPT** $= \min_\theta\{R(\theta)\}$.

Definition (ϵ-Optimality)

Given $\epsilon > 0$, find $\theta^* \in \mathbf{\Theta}$ such that $R(\theta^*) \leq (1+\epsilon)$OPT.

- If θ's average running time is driven by **a small set of exceedingly bad inputs that occur very rarely**, then we'd need to run θ on *many* inputs
- Implies worst-case bounds scaling **linearly with** $\bar{\kappa}$ even when OPT $\ll \bar{\kappa}$

We **relax our objective** by allowing the running time of θ^* to be *capped* at some threshold value κ for a δ fraction of (instance, seed) pairs

Definition ((ϵ, δ)-Optimality)

A configuration θ^* is (ϵ, δ)-*optimal* if there exists some threshold κ for which $R_\kappa(\theta^*) \leq (1+\epsilon)$**OPT** and $\Pr_{\pi\sim\mathcal{D}}\left(R(\theta^*, \pi) > \kappa\right) \leq \delta$.

Existing Approaches

Definition (incumbent-driven)

An algorithm configuration procedure is **incumbent-driven** if, whenever an algorithm run is performed, the captime is either $\bar{\kappa}$ or (an amount proportional to) the runtime of a previously performed algorithm run.

Existing algorithm configuration procedures are incumbent driven:

F-race [Birattari *et al.*, 2002], ParamILS [Hutter *et al.*, 2007; 2009], GGA [Ansótegui *et al.*, 2009; 2015], irace [López-Ibáñez *et al.*, 2016], ROAR and SMAC [Hutter *et al.*, 2011]

Theorem (running time lower bound)

*Any (ϵ, δ)-optimal incumbent-driven search procedure has worst-case expected runtime that scales at least **linearly with** $\bar{\kappa}$.*

This Tutorial

Section Outline

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

- Technical Setup
- **Structured Procrastination** (the case of few configurations)
- Extensions to Structured Procrastination (many configurations and more)
- LeapsAndBounds
- CapsAndRuns
- Structured Procrastination with Confidence
- Related Work and Further Reading

Follow along: http://bit.ly/ACTutorial

Structured Procrastination

- A time management scheme due to Stanford philosopher John Perry
  [Perry, 1996; 2011 Ig Nobel Prize in Literature]
  - Keep a set of **daunting tasks that you procrastinate to avoid**, thereby accomplishing other tasks
  - Eventually, replace each daunting task with **a new task that is even more daunting**, and so complete the first task

Structured Procrastination

- A time management scheme due to Stanford philosopher John Perry
  [Perry, 1996; 2011 Ig Nobel Prize in Literature]
  - Keep a set of **daunting tasks that you procrastinate to avoid**, thereby accomplishing other tasks
  - Eventually, replace each daunting task with **a new task that is even more daunting**, and so complete the first task
- Similarly, the Structured Procrastination algorithm configuration procedure
  [Kleinberg, Lucier & L-B, 2017]:
  - maintains **sets of tasks** *(for each configuration θ, a queue of runs to perform)*;
  - starts with the **easiest tasks** *(shortest captimes)*;
  - **procrastinates** when these tasks prove daunting *(puts them back on the queue)*.

Structured Procrastination

- A time management scheme due to Stanford philosopher John Perry
  [Perry, 1996; 2011 Ig Nobel Prize in Literature]
  - Keep a set of **daunting tasks that you procrastinate to avoid**, thereby accomplishing other tasks
  - Eventually, replace each daunting task with **a new task that is even more daunting**, and so complete the first task
- Similarly, the Structured Procrastination algorithm configuration procedure
  [Kleinberg, Lucier & L-B, 2017]:
  - maintains **sets of tasks** *(for each configuration θ, a queue of runs to perform)*;
  - starts with the **easiest tasks** *(shortest captimes)*;
  - **procrastinates** when these tasks prove daunting *(puts them back on the queue)*.

Key insight

Only spend a long time running a given configuration on a given instance after having failed to find any other (configuration, instance) pair that could be evaluated more quickly.

Structured Procrastination

For now we consider the case of **few configurations**; let $|\mathbf{\Theta}| = n$

Structured Procrastination

For now we consider the case of **few configurations**; let $|\mathbf{\Theta}| = n$

1. Initialize a **bounded-length queue** Q_θ of (instance, captime) pairs for each configuration θ
   – instances randomly sampled from $\mathcal{D}$ with randomly sampled seeds
   – initial captimes of κ_0
2. Calculate **approximate expected runtime** for each θ
   – zero for configurations on which no runs have yet been performed
   – else average runtimes, treating capped runs as though they finished

Structured Procrastination

For now we consider the case of **few configurations**; let $|\boldsymbol{\Theta}| = n$

1. Initialize a **bounded-length queue** Q_θ of (instance, captime) pairs for each configuration θ
2. Calculate **approximate expected runtime** for each θ
   - zero for configurations on which no runs have yet been performed
   - else average runtimes, treating capped runs as though they finished
3. Choose the task **optimistically predicted to be easiest**: the (instance, captime) pair at the head of the queue corresponding to the θ with smallest approximate expected runtime

Structured Procrastination

For now we consider the case of **few configurations**; let $|\mathbf{\Theta}| = n$

1. Initialize a **bounded-length queue** Q_θ of (instance, captime) pairs for each configuration θ
2. Calculate **approximate expected runtime** for each θ
3. Choose the task **optimistically predicted to be easiest**: the (instance, captime) pair at the head of the queue corresponding to the θ with smallest approximate expected runtime
4. If the task does not complete within its captime, **procrastinate**: **double the captime** and put the task at the tail of Q_θ
   - We'll do many other runs before we'll forecast this to be the easiest task

Structured Procrastination

For now we consider the case of **few configurations**; let $|\mathbf{\Theta}| = n$

1. Initialize a **bounded-length queue** Q_θ of (instance, captime) pairs for each configuration θ
2. Calculate **approximate expected runtime** for each θ
3. Choose the task **optimistically predicted to be easiest**: the (instance, captime) pair at the head of the queue corresponding to the θ with smallest approximate expected runtime
4. If the task does not complete within its captime, **procrastinate**: **double the captime** and put the task at the tail of Q_θ
   - We'll do many other runs before we'll forecast this to be the easiest task
5. If execution has not yet been interrupted, goto 2

Structured Procrastination

For now we consider the case of **few configurations**; let $|\boldsymbol{\Theta}| = n$

1. Initialize a **bounded-length queue** Q_θ of (instance, captime) pairs for each configuration θ
2. Calculate **approximate expected runtime** for each θ
3. Choose the task **optimistically predicted to be easiest**: the (instance, captime) pair at the head of the queue corresponding to the θ with smallest approximate expected runtime
4. If the task does not complete within its captime, **procrastinate**: **double the captime** and put the task at the tail of Q_θ
5. If execution has not yet been interrupted, goto 2
6. Return the configuration that **we spent the most total time running**
   - it might seem more intuitive to return the configuration with best approximate expected runtime, but this isn't statistically stable

Running Structured Procrastination

The user must specify

- an **algorithm configuration problem** $(\mathcal{A}, \mathbf{\Theta}, \mathcal{D}, \bar{\kappa}, R, \kappa_0)$;
- a **precision** ϵ (how far solutions can be from optimal);
- a **failure probability** ζ (max probability with which guarantees can fail to hold).

The user does not need to specify δ (the fraction of **"outlying" instances** on which running times may be capped)

- this parameter is gradually reduced as the algorithm runs
- when the algorithm is stopped, it returns the δ for which it is guaranteed to have found an (ϵ, δ)-optimal configuration

Structured Procrastination: Running Time

Theorem (worst-case running time, few configurations)

*For any $\delta > 0$, an execution of the Structured Procrastination algorithm **identifies an (ϵ, δ)-optimal configuration with probability at least $1 - \zeta$** within worst-case expected time*

$$O\left(\frac{n}{\delta\epsilon^2}\ln\left(\frac{n\ln\bar{\kappa}}{\zeta\delta\epsilon^2}\right)OPT\right).$$

Structured Procrastination: Running Time

Theorem (worst-case running time, few configurations)

For any $\delta > 0$, an execution of the Structured Procrastination algorithm ***identifies an (ϵ, δ)-optimal configuration with probability at least $1 - \zeta$*** *within worst-case expected time*

$$O\left(\frac{n}{\delta\epsilon^2} \ln\left(\frac{n \ln \bar{\kappa}}{\zeta\delta\epsilon^2}\right) OPT\right).$$

Structured Procrastination: Running Time

Theorem (worst-case running time, few configurations)

For any $\delta > 0$, an execution of the Structured Procrastination algorithm ***identifies an (ϵ, δ)-optimal configuration with probability at least $1 - \zeta$*** *within worst-case expected time*

$$O\left(\frac{n}{\delta\epsilon^2}\ln\left(\frac{n\ln\bar{\kappa}}{\zeta\delta\epsilon^2}\right)\mathit{OPT}\right).$$

Theorem (running time lower bound for few configurations)

Suppose an algorithm configuration procedure is guaranteed to select an (ϵ, δ)-optimal configuration with probability at least $\frac{1}{2}$. Its worst-case expected running time ***must be at least $\Omega\left(\frac{n}{\delta\epsilon^2}\mathit{OPT}\right)$***.

This Tutorial

Section Outline

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

- Technical Setup
- Structured Procrastination (the case of few configurations)
- **Extensions to Structured Procrastination** (many configurations and more)
- LeapsAndBounds
- CapsAndRuns
- Structured Procrastination with Confidence
- Related Work and Further Reading

Follow along: http://bit.ly/ACTutorial

The Case of Many Configurations

- We need a **different approach** if we want to handle infinitely many configurations—our current guarantees are superlinear in n
  - Relax the requirement that we find performance close to that of OPT
  - Instead, seek a configuration with performance close to **the best that remains after we exclude the γ fraction of fastest configurations** from $\boldsymbol{\Theta}$ (call this **OPT**$_\gamma$)
    - in other words, seek **a configuration within the top-performing $\lfloor 1/\gamma \rfloor$-quantile**

Definition ($(\epsilon, \delta, \gamma)$-Optimality)

A configuration θ^* is $(\epsilon, \delta, \gamma)$-*optimal* if there exists some threshold κ for which $R_\kappa(\theta^*) \leq (1+\epsilon)$ **OPT**$_\gamma$ and $\Pr_{\pi \sim \mathcal{D}} \left(R(\theta^*, \pi) > \kappa \right) \leq \delta$.

Extending Structured Procrastination to Many Configurations

We **extend the Structured Procrastination algorithm** to seek the best among a random sample of $1/\gamma$ configurations

- It **gradually reduces both δ and γ** to tighten guarantees
  - reduces γ by sampling more configurations
  - sets $\delta = \gamma^{\omega}$

Theorem

For any γ, ω and with $\delta = \gamma^{\omega}$, *an execution of the Structured Procrastination algorithm identifies an $(\epsilon, \delta, \gamma)$-optimal configuration with probability at least $1 - \zeta$ in worst-case expected time*

$$O\left(\frac{1}{\delta\gamma\epsilon^2}\ln\left(\frac{\ln\bar{\kappa}}{\zeta\delta\gamma\epsilon^2}\right)\mathit{OPT}_{\gamma}\right).$$

Extending Structured Procrastination to Many Configurations

Theorem

For any γ, ω and with $\delta = \gamma^\omega$, an execution of the Structured Procrastination algorithm identifies an $(\epsilon, \delta, \gamma)$-optimal configuration with probability at least $1 - \zeta$ in worst-case expected time

$$O\left(\frac{1}{\delta\gamma\epsilon^2}\ln\left(\frac{\ln\bar{\kappa}}{\zeta\delta\gamma\epsilon^2}\right)\mathit{OPT}_\gamma\right).$$

Theorem (running time lower bound for many configurations)

Suppose an algorithm configuration procedure is guaranteed to select an $(\epsilon, \delta, \gamma)$-optimal configuration with probability at least $\frac{1}{2}$. Its worst-case expected running time ***must be at least*** $\Omega\left(\frac{1}{\delta\gamma\epsilon^2}\mathit{OPT}_\gamma\right)$.

Practical extensions

Theorem (compatibility with Bayesian optimization & local search)

Suppose that half of the configurations sampled in Structured Procrastination are ***generated in a way that depends arbitrarily on previous observations****. Then worst-case runtime is increased by at most a factor of 2.*

Theorem (linear speedups when parallelized)

Suppose that Structured Procrastination is ***executed by p processors running in parallel****. Then, provided it is run for a sufficiently long time (linear in p), worst-case runtime decreases by at least a factor of $p - 1$.*

This Tutorial

Section Outline

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

- Technical Setup
- Structured Procrastination (the case of few configurations)
- Extensions to Structured Procrastination (many configurations and more)
- **LeapsAndBounds**
- CapsAndRuns
- Structured Procrastination with Confidence
- Related Work and Further Reading

Follow along: http://bit.ly/ACTutorial

LeapsAndBounds

- A second, **approximately optimal** algorithm configuration technique due to Weisz, György & Szepesvári [2018]
- Improves on SP's worst-case performance by:
  - **removing dependence on $\bar{\kappa}$** (replaced with OPT, usually much smaller)
  - tightening the worst-case performance bound by a log factor
- **Empirically outperforms** SP
  - based on very limited experiments, but likely true overall
- But is **not anytime**: requires both ϵ, δ as inputs

LeapsAndBounds: How it Works

The algorithm at a glance:

1. Attempt to guess an (initially) low value of OPT
2. Try to find a configuration whose mean is smaller than this guess
   - Discard configurations whose mean is large **relative to the current guess**
   - Use **fewer samples** to estimate mean runtime of configurations with **low runtime variance** across instances
3. If none, double the guess and repeat

LeapsAndBounds: Running Time

Theorem (worst-case running time)

For any $\epsilon \in (0, 1/3)$, $\delta \in (0, 1)$, an execution of LeapsAndBounds ***identifies an (ϵ, δ)-optimal configuration with probability at least $1 - \zeta$*** *within worst-case expected time*

$$O\left(\frac{n}{\delta\epsilon^2}\ln\left(\frac{n \ln \mathit{OPT}}{\zeta}\right)\mathit{OPT}\right).$$

Structured Procrastination

Compare to Structured Procrastination:

$$O\left(\frac{n}{\delta\epsilon^2}\ln\left(\frac{n \ln \bar{\kappa}}{\zeta\delta\epsilon^2}\right)\mathrm{OPT}\right).$$

LeapsAndBounds: Empirical Performance

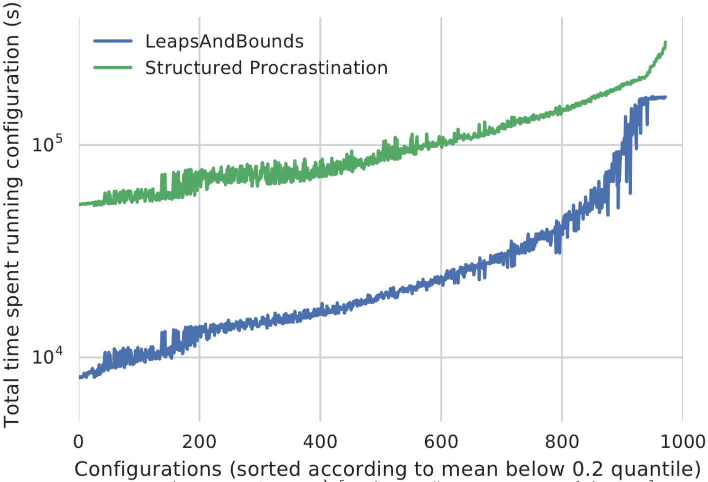
972 `minisat` configurations running on 20,118 nontrivial `CNFuzzDD` SAT problems

Time to prove $(\epsilon = 0.2, \delta = 0.2)$-optimality: **SP 1,169** CPU days; **L&B 369** CPU days

LeapsAndBounds: Empirical Performance

972 `minisat` configurations running on 20,118 nontrivial `CNFuzzDD` SAT problems

Time to prove $(\epsilon = 0.2, \delta = 0.2)$-optimality: **SP 1,169** CPU days; **L&B 369** CPU days



$(\epsilon = 0.2, \delta = 0.2)$ [Weisz, György & Szepesvári, 2018]

This Tutorial

Section Outline

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

- Technical Setup
- Structured Procrastination (the case of few configurations)
- Extensions to Structured Procrastination (many configurations and more)
- LeapsAndBounds
- **CapsAndRuns**
- Structured Procrastination with Confidence
- Related Work and Further Reading

Follow along: http://bit.ly/ACTutorial

CapsAndRuns

- Recent extension to LeapsAndBounds [Weisz, György & Szepesvári, ICML 2019]
  - **Tue Jun 11th 04:20–04:25 PM Room 103**

CapsAndRuns

- Recent extension to LeapsAndBounds [Weisz, György & Szepesvári, ICML 2019]
  - **Tue Jun 11th 04:20–04:25 PM Room 103**
- **Adapts to easy problem instances** by eliminating configurations that are dominated by other configurations
- Also provides an **improved bound** for non-worst-case instances
  - scales with suboptimality gap, $\frac{R(\theta)}{R(\theta)-OPT}$, instead of ϵ^{-1}
  - dependence on ϵ and δ individually, rather than product $\epsilon\delta$
- Bounds are also improved by defining (ϵ, δ)-optimality w.r.t. $OPT_{\delta/2}$, the optimal configuration when capping runs at the $\delta/2$-quantile, rather than OPT
- Still **not anytime**

CapsAndRuns: How it Works

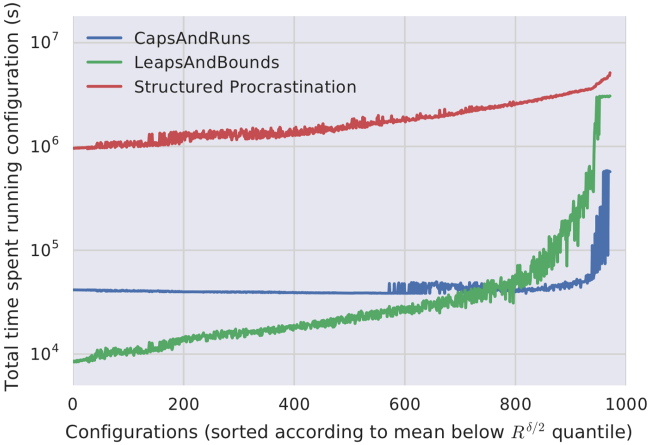
Proceeds in **two phases**:

- Phase 1: **Estimate** $(1-\delta)$**-quantile** of each configuration's runtime over $\mathcal{D}$
- Phase 2: **Estimate mean runtime** of each configuration using the **quantile** from Phase 1 as **captime**
- Return configuration with **minimum estimated mean**

CapsAndRuns: Empirical Results

972 `minisat` configurations running on 20,118 nontrivial `CNFuzzDD` SAT problems

Time to prove $(\epsilon = 0.05, \delta = 0.2)$-optimality (CPU days): **SP 20,643**; **L&B 1,451**; **C&R: 586**



$(\epsilon = 0.05, \delta = 0.2)$ [Weisz, György & Szepesvári, ICML 2019]

This Tutorial

Section Outline

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

- Technical Setup
- Structured Procrastination (the case of few configurations)
- Extensions to Structured Procrastination (many configurations and more)
- LeapsAndBounds
- CapsAndRuns
- **Structured Procrastination with Confidence**
- Related Work and Further Reading

Follow along: http://bit.ly/ACTutorial

Structured Procrastination with Confidence

- Recent extension to Structured Procrastination [Kleinberg, L-B, Lucier & Graham, arXiv 2019]
- **Adapts to easy problem instances** by maintaining confidence bounds on each configuration's runtime
- Anytime algorithm: δ is **gradually refined** during the search process
  - helpful when user can't predict the relationship between these parameters and runtime
  - also improves performance: by starting with large values of δ, SPC **eliminates bad configurations early on**
- SPC's running time matches (up to log factors) the running time of a hypothetical "optimality verification procedure" that knows the identity of the optimal configuration
  - i.e., SPC takes about as long to prove (ϵ, δ)-optimality as our hypothetical verification procedure would need to demonstrate that fact to a skeptic
  - **When verification is easy, SPC is fast**

Recall: Structured Procrastination

1. Initialize a **bounded-length queue** Q_θ of (instance, captime) pairs for each configuration θ
2. Calculate **approximate expected runtime** for each θ
3. Choose the task **optimistically predicted to be easiest**: the (instance, captime) pair at the head of the queue corresponding to the i with smallest approximate expected runtime
4. If the task does not complete within its captime, **procrastinate**: **double the captime** and put the task at the tail of Q_θ
5. If execution has not yet been interrupted, goto 2
6. Return the configuration that **we spent the most total time running**

Structured Procrastination with Confidence

1. Initialize a **bounded-length queue** Q_θ of (instance, captime) pairs for each configuration θ
2. Calculate **lower confidence bound on expected runtime** ~~**approximate expected runtime**~~ for each θ
3. Choose the task **optimistically predicted to be easiest**: the (instance, captime) pair at the head of the queue corresponding to the i with smallest approximate expected runtime
4. If the task does not complete within its captime, **procrastinate**: **double the captime** and put the task at the tail of Q_θ
5. If execution has not yet been interrupted, goto 2
6. Return the configuration that **we spent the most total time running**

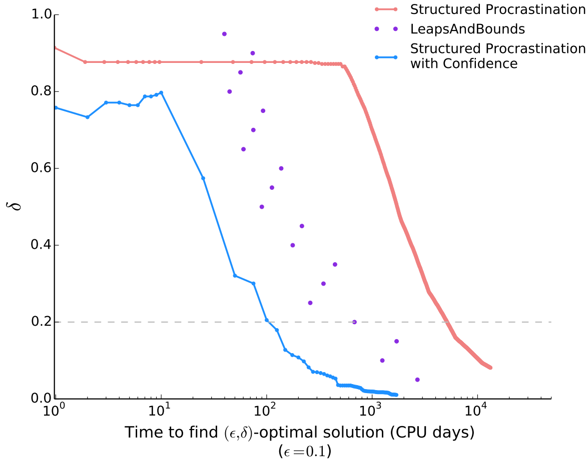
Structured Procrastination with Confidence

1. Initialize a **bounded-length queue** Q_θ of (instance, captime) pairs for each configuration θ
2. Calculate ~~**approximate expected runtime**~~ **lower confidence bound on expected runtime** for each θ
3. Choose the task **optimistically predicted to be easiest**: the (instance, captime) pair at the head of the queue corresponding to the i with smallest approximate expected runtime
4. If the task does not complete within its captime, **procrastinate**: **double the captime** and put the task at the tail of Q_θ
5. If execution has not yet been interrupted, goto 2
6. ~~Return the configuration that **we spent the most total time running**~~
   Return the configuration that either solved or attempted the greatest number of instances

Structured Procrastination with Confidence: Empirical Performance

972 `minisat` configurations running on 20,118 nontrivial `CNFuzzDD` SAT problems

Time to prove $(\epsilon = 0.1, \delta = 0.2)$-optimality: **SP 5,150**; **L&B 680**; **SPC 150** (CPU days)



[Kleinberg, L-B, Lucier & Graham, 2019]

This Tutorial

Section Outline

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

- Technical Setup
- Structured Procrastination (the case of few configurations)
- Extensions to Structured Procrastination (many configurations and more)
- LeapsAndBounds
- CapsAndRuns
- Structured Procrastination with Confidence
- **Related Work and Further Reading**

Follow along: http://bit.ly/ACTutorial

Related Work: Bandits

- Bandits:
  - Optimism in the face of uncertainty [Auer, Cesa-Bianchi & Fischer 2002, Bubeck & Cesa-Bianchi 2012]
  - bandits with **correlated arms** that scale to large experimental design settings [Kleinberg 2006; Kleinberg, Slivkins & Upfal 2008; Chaudhuri, Freund & Hsu 2009, Bubeck, Munos, Stoltz & Szepesvári 2011, Srinivas, Krause, Kakade & Seeger 2012, Cesa-Bianchi & Lugosi 2012, Munos 2014]
- However, our runtime minimization objective is **crucially different** from more general objective functions targeted in most bandits literature:
  - cost of pulling an arm **measured in the same units** as the minimization objective function
  - freedom to **set a maximum amount** κ we are willing to pay in pulling an arm; if true cost exceeds κ, we pay only κ but learn only that true cost was higher
- Beyond the assumption that **all arms involve the same, fixed cost**:
  - **Variable costs** and a fixed overall budget, but no capping [Guha & Munagala 2007, Tran-Thanh, Chapman, Rogers & Jennings 2012, Badanidiyuru, Kleinberg, & Slivkins 2013]
  - The algorithm can *specify* a **maximum cost to be paid** when pulling an arm, but never pays less than that amount [Kandasamy, Dasarathy, Poczos & Schneider 2016]
  - Observations are **censored if they exceed a given budget** [Ganchev, Nevmyvaka, Kearns & Vaughan 2010]

Other Important Related Work

- **Hyperparameter optimization**
  - Key initial work [Bergstra, Bardenet, Bengio & Kégl 2011,Thornton, H, Hoos & L-B 2013]
  - Hyperband: uses similar theoretical tools [Li, Jamieson, DeSalvo, Rostamizadeh, & Talwalkar 2016]
- **Learning-theoretic foundations**
  - Gupta & Roughgarden [2017]: framed configuration and selection in terms of learning theory
  - Sample-efficient, special-purpose algorithms for particular classes of problems
    - combinatorial partitioning problems (clustering, max-cut, etc) [Balcan, Nagarajan, Vitercik & White 2017]
    - branching strategies in tree search [Balcan, Dick, Sandholm & Vitercik 2018]
    - various algorithm selection problems [Balcan, Dick & Vitercik 2018]

This Tutorial

High-Level Outline

Introduction, Technical Preliminaries, and a Case Study (Kevin)

Practical Methods for Algorithm Configuration (Frank)

Algorithm Configuration Methods with Theoretical Guarantees (Kevin)

Beyond Static Configuration: Related Problems and Emerging Directions (Frank)

Follow along: http://bit.ly/ACTutorial

This Tutorial

Section Outline

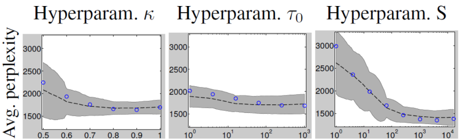
Beyond Static Configuration: Related Problems and Emerging Directions (Frank)

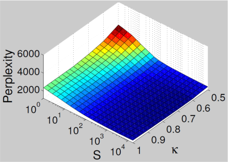
- **Parameter Importance**
- Algorithm Selection
- End-to-End Learning of Combinatorial Solvers
- Integrating ML and Combinatorial Optimization

Follow along: http://bit.ly/ACTutorial

Global effect of a parameter

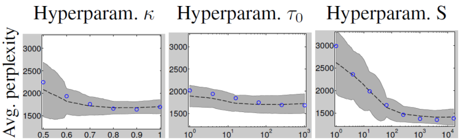
- To quantify the global effect of one or more parameters, we can **marginalize predicted performance across all settings of all other parameters** [H., Hoos & L-B, 2014]

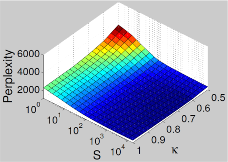




Global effect of a parameter

- To quantify the global effect of one or more parameters, we can **marginalize predicted performance across all settings of all other parameters** [H., Hoos & L-B, 2014]





In regression trees, we can do this **efficiently**:

$$\mathrm{avg}(v) = \sum_{\theta_2 \in \Theta_2} \cdots \sum_{\theta_n \in \Theta_n} \frac{1}{|\Theta_2|} \cdots \frac{1}{|\Theta_n|} f(v, \theta_2, \ldots, \theta_n)$$

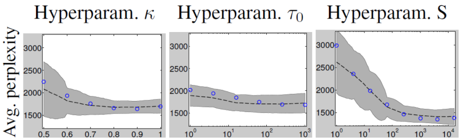
$$\approx \sum_{\theta_2 \in \Theta_2} \cdots \sum_{\theta_n \in \Theta_n} \frac{1}{|\Theta_2|} \cdots \frac{1}{|\Theta_n|} \hat{f}(v, \theta_2, \ldots, \theta_n)$$

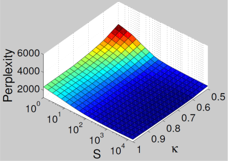
$$= \sum_{\theta_2 \in \Theta_2} \cdots \sum_{\theta_n \in \Theta_n} \frac{1}{|\Theta_2|} \cdots \frac{1}{|\Theta_n|} \sum_{P_i \in \mathcal{P}} \mathbb{I}(\langle v, \theta_2, \ldots, \theta_n \rangle \in P_i) \cdot c(P_i)$$

$$= \sum_{P_i \in \mathcal{P}} \frac{|\Theta_2{}^{(i)}| \cdots |\Theta_n{}^{(i)}|}{|\Theta_2| \cdots |\Theta_n|} \cdot \mathbb{I}(v \in \Theta_1{}^{(i)}) \cdot c(P_i)$$

Linear time computation

Functional analysis of variance (fANOVA) [H., Hoos & L-B, 2014]





- By definition, the variance of predictor $\hat{f}$ across its domain $\mathbf{\Theta}$ is:

$$\mathbb{V} = \frac{1}{||\mathbf{\Theta}||} \int (\hat{f}(\boldsymbol{\theta}) - \hat{f}_0)^2 d\boldsymbol{\theta}$$

- Functional ANOVA [Sobol, 1993] **decomposes this variance into components** due to each subset of the parameters N:

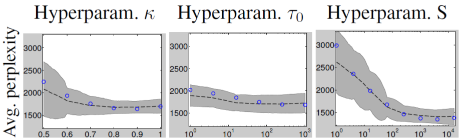
$$\mathbb{V} = \sum_{U \subset N} \mathbb{V}_U, \text{ where } \mathbb{V}_U = \frac{1}{||\mathbf{\Theta}_U||} \int \hat{f}_U^2(\boldsymbol{\theta}_U) d\mathbf{\Theta}_U$$

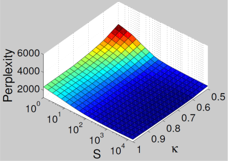
Functional analysis of variance (fANOVA) [H., Hoos & L-B, 2014]

"Main effect" S explains 65% of variance

"Interaction effect" of S&κ explains another 18%

Computing this took milliseconds





- By definition, the variance of predictor $\hat{f}$ across its domain $\mathbf{\Theta}$ is:

$$\mathbb{V} = \frac{1}{||\mathbf{\Theta}||} \int (\hat{f}(\boldsymbol{\theta}) - \hat{f}_0)^2 d\boldsymbol{\theta}$$

- Functional ANOVA [Sobol, 1993] **decomposes this variance into components** due to each subset of the parameters N:

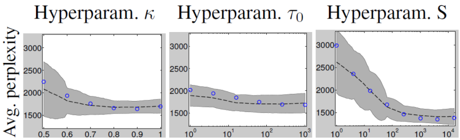
$$\mathbb{V} = \sum_{U \subset N} \mathbb{V}_U, \text{ where } \mathbb{V}_U = \frac{1}{||\mathbf{\Theta}_U||} \int \hat{f}_U^2(\boldsymbol{\theta}_U) d\mathbf{\Theta}_U$$

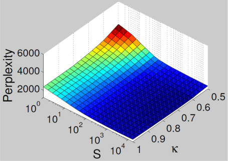
Functional analysis of variance (fANOVA) [H., Hoos & L-B, 2014]

"Main effect" S explains 65% of variance

"Interaction effect" of S&κ explains another 18%

Computing this took milliseconds





- By definition, the variance of predictor $\hat{f}$ across its domain $\mathbf{\Theta}$ is:

$$\mathbb{V} = \frac{1}{||\mathbf{\Theta}||} \int (\hat{f}(\boldsymbol{\theta}) - \hat{f}_0)^2 d\boldsymbol{\theta}$$

- Functional ANOVA [Sobol, 1993] **decomposes this variance into components** due to each subset of the parameters N:

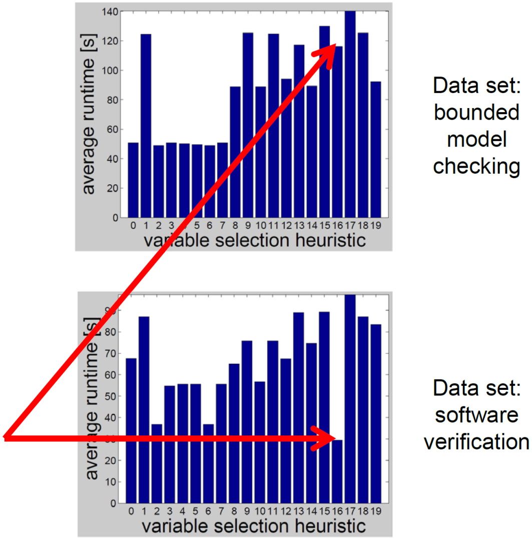
$$\mathbb{V} = \sum_{U \subset N} \mathbb{V}_U, \text{ where } \mathbb{V}_U = \frac{1}{||\mathbf{\Theta}_U||} \int \hat{f}_U^2(\boldsymbol{\theta}_U) d\mathbf{\Theta}_U$$

Theorem

In regression trees, main effects can be computed in linear time.

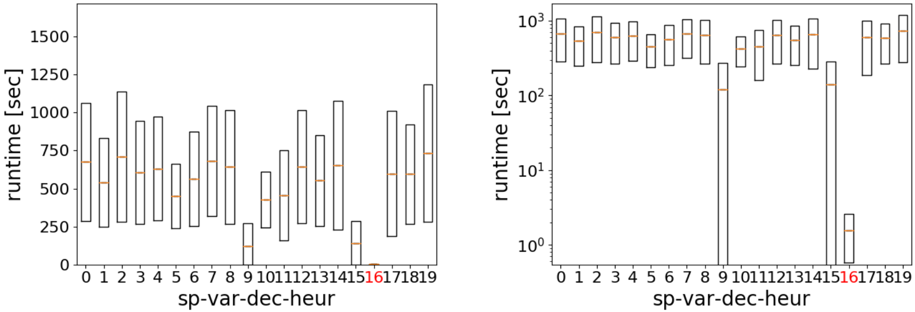
Functional ANOVA example for SAT solver Spear [H., Hoos & L-B, 2014]

- SAT solver Spear:
  26 parameters
- Posthoc analysis of data gathered from optimization with SMAC
- **93% of variation in runtimes is due to a single parameter**: the **variable selection heuristic**.
- Analysis took seconds



Local parameter importance (LPI): changing each parameter around the incumbent

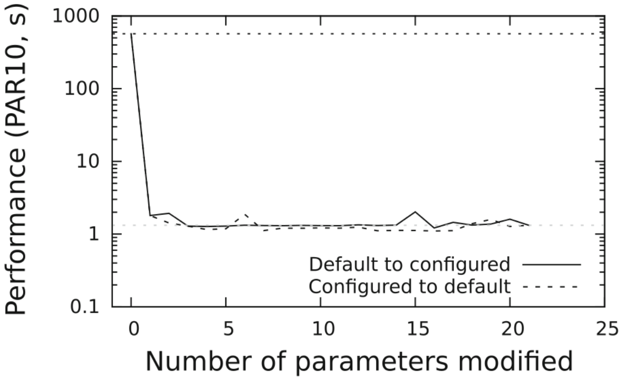
- What is the **local effect of varying one parameter of the incumbent**?
  - Use relative changes to quantify local parameter importance
  - Can also be done based on the predictive model of algorithm performance [Biedenkapp et al, 2018]



Results for Spear on SWV

Ablation between default and incumbent configuration

- Greedily change the parameter that improves performance most [Fawcett et al. 2013]
  - Can also be done based on the predictive model of algorithm performance [Biedenkapp et al, 2017]



Results for Spear on SWV

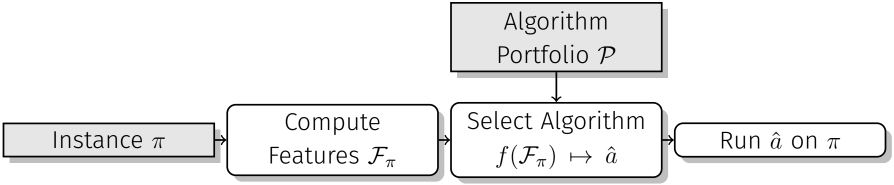
This Tutorial

Section Outline

Follow along: http://bit.ly/ACTutorial

Algorithm selection

- In this tutorial, we focussed on finding a single configuration that performs well on average: $\arg\min_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \mathbb{E}_{\pi\sim\mathcal{D}}(m(\boldsymbol{\theta},\pi))$
- We can also learn a function that picks the best configuration $\boldsymbol{\theta}\in\boldsymbol{\Theta}$ or algorithm $a\in\mathcal{P}$ per instance π with features $\mathcal{F}_\pi$: $\arg\min_{f:\Pi\to\boldsymbol{\Theta}} \mathbb{E}_{\pi\sim\mathcal{D}}(m(f(\mathcal{F}_\pi),\pi))$



- There is a rich literature on this **algorithm selection** problem [L-B et al, 2003 Xu et al, 2008; Smith-Miles, 2009; Xu et al, 2012; Kotthoff, 2014; Malitsky et al, 2013; Lindauer et al, 2015; Lorregia et al, 2016]

Example SAT Challenge 2012

Rank	RiG	Solver	#solved
–	–	Virtual Best Solver (VBS)	568
1	1	SATzilla2012 APP	531
2	2	SATzilla2012 ALL	515
3	1	Industrial SAT Solver	499
–	–	lingeling (SAT Competition 2011 Bronze)	488
4	2	interactSAT	480
5	1	glucose	475
6	2	SINN	472
7	3	ZENN	468
8	4	Lingeling	467
9	5	linge_dyphase	458
10	6	simpsat	453

The VBS (virtual best solver) is an oracle algorithm selector of competition entries.
(pink: algorithm selectors, blue: portfolios, green: single-engine solvers)

Automated construction of portfolios from a single algorithm

- **Algorithm Configuration**
  - Strength: find a single configuration with strong performance for a given cost metric
  - Weakness: for heterogeneous instance sets, there is often no configuration that performs great for all instances
- **Algorithm Selection**
  - Strength: works well for heterogeneous instance sets due to per-instance selection
  - Weakness: in standard algorithm selection, the set of algorithms $\mathcal{P}$ to choose from typically only contains a few algorithms
- **Putting the two together** [Kadioglu et al, 2010; Xu et al, 2010]
  - Use algorithm configuration to determine useful configurations
  - Use algorithm selection to select from them based on instance characteristics

Warmstarting of algorithm configuration [Lindauer & H., 2018]

- Humans often **don’t start from scratch** when tuning an algorithm‘s parameters
  - They use their previous experience
  - E.g., tuning CPLEX for a few applications tells you which parameters tend to be important

Warmstarting of algorithm configuration [Lindauer & H., 2018]

- Humans often **don't start from scratch** when tuning an algorithm's parameters
  - They use their previous experience
  - E.g., tuning CPLEX for a few applications tells you which parameters tend to be important
- We would also like to make use of previous AC runs on other distributions
  - Option 1: initialize from **strong previous configurations**
  - Option 2: **reuse the previous models** (weighted by how useful they are)
  - Combination of 1+2 often works best

Warmstarting of algorithm configuration [Lindauer & H., 2018]

- Humans often **don't start from scratch** when tuning an algorithm's parameters
  - They use their previous experience
  - E.g., tuning CPLEX for a few applications tells you which parameters tend to be important
- We would also like to make use of previous AC runs on other distributions
  - Option 1: initialize from **strong previous configurations**
  - Option 2: **reuse the previous models** (weighted by how useful they are)
  - Combination of 1+2 often works best
- Results
  - Can yield large speedups ($> 100\times$) when similar configurations work well
  - Does not substantially slow down the search if misleading
  - On average: $4\times$ speedups over running SMAC from scratch

This Tutorial

Section Outline

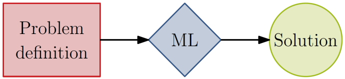
Beyond Static Configuration: Related Problems and Emerging Directions (Frank)

- Parameter Importance
- Algorithm Selection
- **End-to-End Learning of Combinatorial Solvers**
- Integrating ML and Combinatorial Optimization

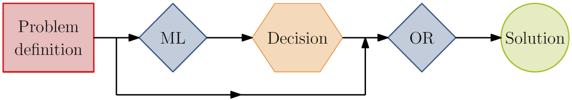
Follow along: http://bit.ly/ACTutorial

Categorization of ML for Combinatorial Optimization / Operations Research (OR)

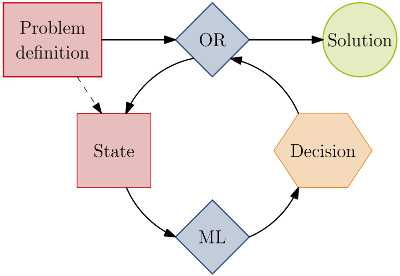
- Recent survey article [Yoshua Bengio, Andrea Lodi and Antoine Prouvost, 2018]
  - Define three categories of combinining ML and OR



ML acts alone to solve the problem



ML augments OR with valuable information



Integrating ML into OR; OR algorithm repeatedly calls the same model to make decisions

End-to-end learning of algorithms (in general)

Learn a neural network with parameters ϕ that defines an algorithm

- The network's parameters ϕ are trained to optimize the true objective (or a proxy)
- The network is queried for each action of the algorithm

End-to-end learning of algorithms (in general)

Learn a neural network with parameters ϕ that defines an algorithm

- The network's parameters ϕ are trained to optimize the true objective (or a proxy)
- The network is queried for each action of the algorithm

Examples

- Learning to learn with gradient descent [Andrychowicz et al, 2016] / learning to optimize [Li & Malik, 2017]: parameterize an update rule for base-level NN parameters $\mathbf{w}$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + g(\nabla f(\mathbf{w}_t), \phi)$$

End-to-end learning of algorithms (in general)

Learn a neural network with parameters ϕ that defines an algorithm

- The network's parameters ϕ are trained to optimize the true objective (or a proxy)
- The network is queried for each action of the algorithm

Examples

- Learning to learn with gradient descent [Andrychowicz et al, 2016] / learning to optimize [Li & Malik, 2017]: parameterize an update rule for base-level NN parameters $\mathbf{w}$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + g(\nabla f(\mathbf{w}_t), \phi)$$

- Learning a gradient-free optimizer's update rule [Chen et al, 2017]
- Learning unsupervised learning rules [Metz et al, 2019]
- AlphaZero [Silver et al, 2018], etc

End-to-end learning of combinatorial problems

Learning to solve Euclidean TSP

- Pointer networks [Vinyals et al, 2015]
  - RNN to encode TSP instance
  - Another RNN with attention-like mechanism to predict probability distribution over next node
  - Trained with supervised learning, using optimal solutions to TSP instances
- Reinforcement learning avoids need for optimal solutions
  - Train an RNN [Bello et al, 2017] or a graph neural network [Kool et al, 2019]
- Directly predict the permutation [Emami & Ranka, 2018; Nowak et al, 2017]
- Learn a greedy heuristic to choose next node [Dai et al, 2018]

End-to-end learning of combinatorial problems

Learning to solve SAT

- NeuroSAT [Selsam et al, 2019]
  - Use permutation invariant graph neural network
  - Learn a message passing algorithm for solving new instances
- SATNet [Wang et al, 2019]
  - Differentiable approximate MaxSAT solver
  - Can be integrated as a component of a deep learning system (e.g., “visual Sudoku”)
- Learning to predict satisfiability [Cameron et al, 2019]
  - Even at the phase transition, with 80% accuracy
  - Using exchangeable deep networks

This Tutorial

Section Outline

Beyond Static Configuration: Related Problems and Emerging Directions (Frank)

Parameter Importance

Algorithm Selection

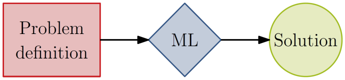
End-to-End Learning of Combinatorial Solvers

Integrating ML and Combinatorial Optimization

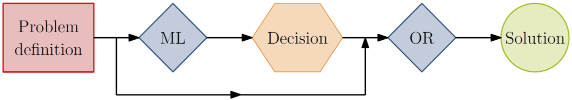
Follow along: http://bit.ly/ACTutorial

Categorization of ML for Combinatorial Optimization / Operations Research (OR)

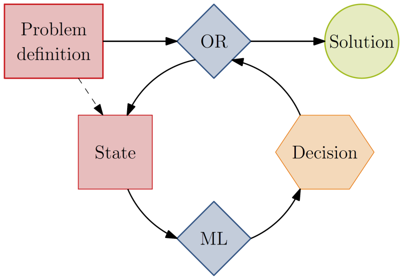
- Recent survey article [Yoshua Bengio, Andrea Lodi and Antoine Prouvost, 2018]
  - Defines three categories of combinining ML and OR



ML acts alone to solve the problem



ML augments OR with valuable information



Integrating ML into OR; OR algorithm repeatedly calls the same model to make decisions

Learning to make simple decisions online

Dynamic restart policies

- For a randomized algorithm
- Based on an initial observation window of a run, predict whether this run is good or bad (and thus whether to restart) [Kautz et al, 2002; Horvitz et al, 2001]

Learning to make simple decisions online

Dynamic restart policies

- For a randomized algorithm
- Based on an initial observation window of a run, predict whether this run is good or bad (and thus whether to restart) [Kautz et al, 2002; Horvitz et al, 2001]

Dynamic algorithm portfolios

- Run several algorithms in parallel
- Decide time shares adaptively based on algorithms' progress

[Carchrae & Beck, 2014; Gagliolo & Schmidhuber, 2006]

Learning to make simple decisions online

Dynamic restart policies

- For a randomized algorithm
- Based on an initial observation window of a run, predict whether this run is good or bad (and thus whether to restart) [Kautz et al, 2002; Horvitz et al, 2001]

Dynamic algorithm portfolios

- Run several algorithms in parallel
- Decide time shares adaptively based on algorithms' progress

[Carchrae & Beck, 2014; Gagliolo & Schmidhuber, 2006]

Learning in which search nodes to apply primal heuristics

- Primal heuristics can find feasible solutions in branch-and-bound
- Too expensive to apply in every node $\rightsquigarrow$ learn when to apply [Khalil et al, 2017]

Learning to select/switch between algorithms online

Learning to select a sorting algorithm at each node

- Keep track of a state (e.g., length of sequence left to be sorted recursively)
- Choose algorithm to use for subtree based on state using RL [Lagoudakis & Littmann, 2000]
  - E.g., QuickSort for long sequences, InsertionSort for short ones

Learning to select branching rules for DPLL in SAT solving

- Keep track of a backtracking state
- Choose branching rule based on state using RL [Lagoudakis & Littmann, 2001]

Parameter control

Adapting algorithm parameters online

- A strict **generalization of algorithm configuration**
  - just pick a fixed setting and never change it

Parameter control

Adapting algorithm parameters online

- A strict **generalization of algorithm configuration**
  - just pick a fixed setting and never change it
- A strict **generalization of per-instance algorithm configuration (PIAC)**
  - just select configuration once in the beginning per instance, never change

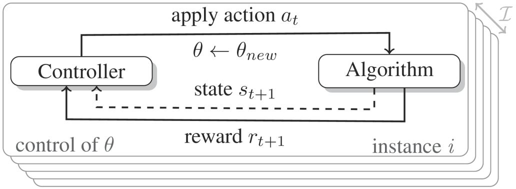
Parameter control

Adapting algorithm parameters online

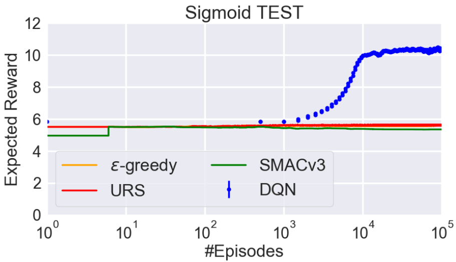
- A strict **generalization of algorithm configuration**
  - just pick a fixed setting and never change it
- A strict **generalization of per-instance algorithm configuration (PIAC)**
  - just select configuration once in the beginning per instance, never change
- A strict **generalization of algorithm selection** (finite set of algorithms $\mathcal{P}$)
  - special case of PIAC with one categorical parameter with domain $\mathcal{P}$

Parameter control: a reinforcement learning problem

- Formulation of the single-instance case as an MDP [Adriaensen & Nowe, 2016]
  - But a strong policy for a single instance may not generalize
- Formulation of the general problem as a **contextual MDP** to learn to generalize across instances [Biedenkapp et al, 2019]



Shared state & action spaces
Different transition and reward functions



First promising results on toy functions

Conclusions

Summary

- Algorithm configuration: **learning in the space of algorithm designs**
- **Practical AC methods** are very mature; often able to speed up state-of-the-art algorithms by orders of magnitude
- Much recent progress on **AC with worst-case runtime guarantees**; likely to impact practice soon
- **Related problems:** parameter importance; algorithm selection; end-to-end learning; other ways of integrating ML with combinatorial optimization

Further resources

- **Code** available for SMAC, CAVE (parameter importance), Auto-WEKA, Auto-sklearn
- See http://automl.org for **more material**; also, we're hiring: http://automl.org/jobs