# Programming by Optimisation:

## A Practical Paradigm for Computer-Aided Algorithm Design

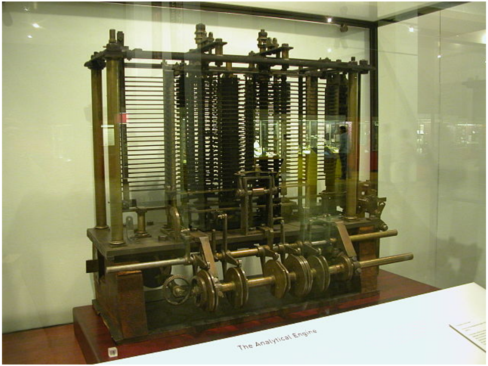
Holger H. Hoos*, Frank Hutter+, Kevin Leyton-Brown*

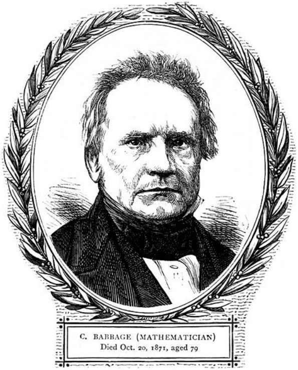
∗ Department of Computer Science
University of British Columbia
Canada

+ Department of Computer Science
University of Freiburg
Germany

IJCAI 2013
Beijing, China, 2013/08/04

# The age of machines




"As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise – by what course of calculation can these results be arrived at by the machine in the shortest time?"

(Charles Babbage, 1864)

BBC Mobile

News | Sport | Weather | Travel | TV | Radio

NEWS TECHNOLOGY

Home | US & Canada | Latin America | UK | Africa | Asia | Europe | Mid-East | Business | Health | Sci/Environment | Tech

**22 August 2011** Last updated at 20:42 ET

1.4K Share

# When algorithms control the world

**By Jane Wakefield**
Technology reporter

**If you were expecting some kind of warning when computers finally get smarter than us, then think again.**

There will be no soothing HAL 9000-type voice informing us that our human services are now surplus to requirements.

In reality, our electronic overlords are already taking control, and they are doing it in a far more subtle way than science fiction would have us believe.

Their weapon of choice - the algorithm.

Behind every smart web service is some even smarter web code. From the web retailers - calculating what books and films we might be interested in, to Facebook's friend finding and image tagging services, to the search engines that guide us around the net.

Algorithms are spreading their influence around the globe

It is these invisible computations that increasingly control how we interact with our electronic world.

At last month's TEDGlobal conference, algorithm expert Kevin Slavin delivered one of the tech show's most "sit up and take notice" speeches where he warned that the "maths that computers use to decide stuff" was infiltrating every aspect of our lives.

**Related Stories**

Are search engines skewing objectivity?

Robot reads minds to train itself

# The age of computation


BBC Mobile

News | Sport | Weather | Travel | TV | Radio

NEWS TECHNOLOGY

Home | US & Canada | Latin America | UK | Africa | Asia | Europe | Mid-East | Business | Health | Sci/Environment | Tech

22 August 2011 Last updated at 20:42 ET

1.4K Share

## When algorithms control the world

By Jane Wakefield
Technology reporter

**If you were expecting some kind of warning when computers finally get smarter than us, then think again.**

There will be no soothing HAL 9000-type voice informing us that our human services are now surplus to requirements.

In reality, our electronic overlords are already taking control, and they are doing it in a far more subtle way than science fiction would have us believe.

Their weapon of choice - the algorithm.

Behind every smart web service is some even smarter web code. From the web retailers - calculating what books and films we might be interested in, to Facebook's friend finding and image tagging services, to the search engines that guide us around the net.

Algorithms are spreading their influence around the globe

It is these invisible computations that increasingly control how we interact with our electronic world.

At last month's TEDGlobal conference, algorithm expert Kevin Slavin delivered one of the tech show's most "sit up and take notice" speeches where he warned that the "maths that computers use to decide stuff" was infiltrating every aspect of our lives.

**Related Stories**

Are search engines skewing objectivity?

Robot reads minds to train itself

"The maths[!] that computers use to decide stuff [is] infiltrating every aspect of our lives."

- financial markets
- social interactions
- cultural preferences
- artistic production
- ...

## Performance matters ...

- computation speed (time is money!)
- energy consumption (battery life, ...)
- quality of results (cost, profit, weight, ...)

## ... increasingly:

- globalised markets
- just-in-time production & services
- tighter resource constraints

## Example: Resource allocation

- resources $>$ demands $\leadsto$ many solutions, easy to find
  economically wasteful
  $\leadsto$ reduction of resources / increase of demand
- resources $<$ demands $\leadsto$ no solution, easy to demonstrate
  lost market opportunity, strain within organisation
  $\leadsto$ increase of resources / reduction of demand
- resources $\approx$ demands
  $\leadsto$ difficult to find solution / show infeasibilityresources $\approx$ demands
  $\leadsto$ difficult to find solution / show infeasibility

## This tutorial:

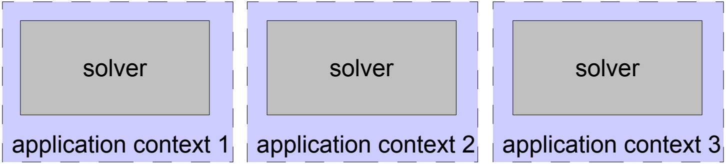
new approach to software development, leveraging ...

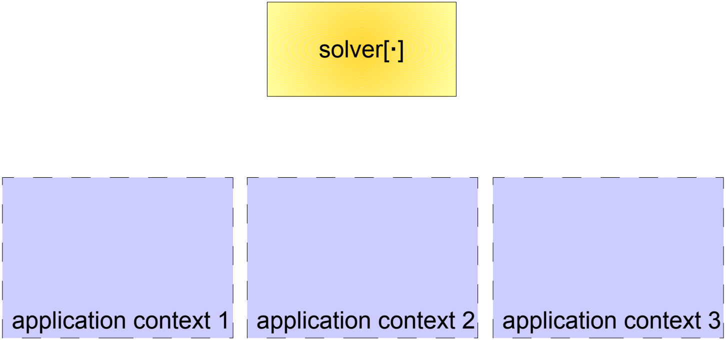
- human creativity
- optimisation & machine learning
- large amounts of computation / data

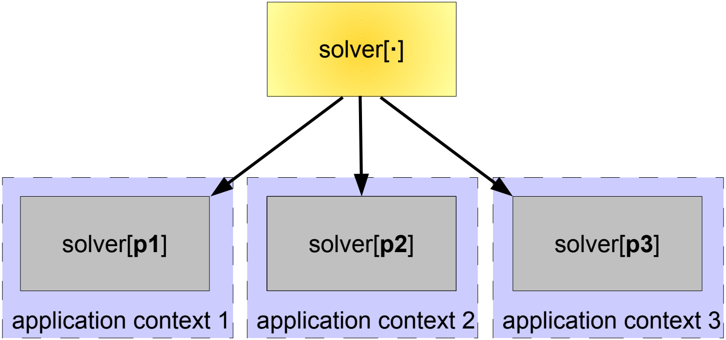
Key idea:

- program $\rightsquigarrow$ (large) space of programs
- encourage software developers to
  - avoid premature commitment to design choices
  - seek & maintain design alternatives
- automatically find performance-optimising designs for given use context(s)

$\Rightarrow$ Programming by Optimization (PbO)


solver
solver
solver
application context 1
application context 2
application context 3


solver[·]
application context 1
application context 2
application context 3


solver[·]
solver[**p1**]
application context 1
solver[**p2**]
application context 2
solver[**p3**]
application context 3

# Outline

1. Programming by Optimization: Motivation & Introduction
2. Algorithm Configuration

   Coffee Break

3. Portfolio-based Algorithm Selection
4. Software Development Support & Further Directions

# Programming by Optimization: Motivation & Introduction

## Example: SAT-based software verification

Hutter, Babić, Hoos, Hu (2007)

- **Goal:** Solve SAT-encoded software verification problems as fast as possible
- new DPLL-style SAT solver SPEAR (by Domagoj Babić)
  = highly parameterised heuristic algorithm
  (26 parameters, $\approx 8.3 \times 10^{17}$ configurations)
- manual configuration by algorithm designer
- automated configuration using ParamILS, a generic algorithm configuration procedure
  Hutter, Hoos, Stützle (2007)

## SPEAR: Performance on software verification benchmarks

| solver | num. solved | mean run-time |
|---|---|---|
| MiniSAT 2.0 | 302/302 | 161.3 CPU sec |
| SPEAR original | 298/302 | 787.1 CPU sec |
| SPEAR generic. opt. config. | 302/302 | 35.9 CPU sec |
| SPEAR specific. opt. config. | 302/302 | 1.5 CPU sec |

- $\approx$ 500-fold speedup through use automated algorithm configuration procedure (ParamILS)
- new state of the art
  (winner of 2007 SMT Competition, QF_BV category)

# Levels of PbO:

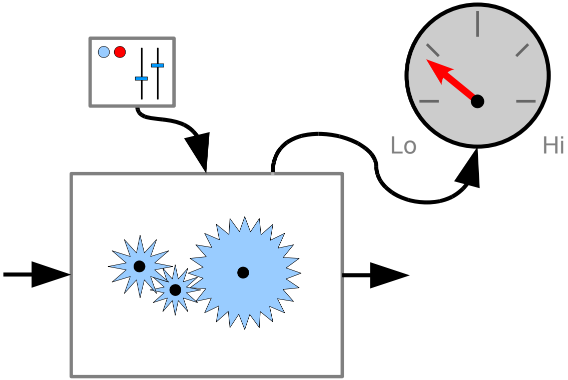
**Level 4:** Make no design choice prematurely that cannot be justified compellingly.

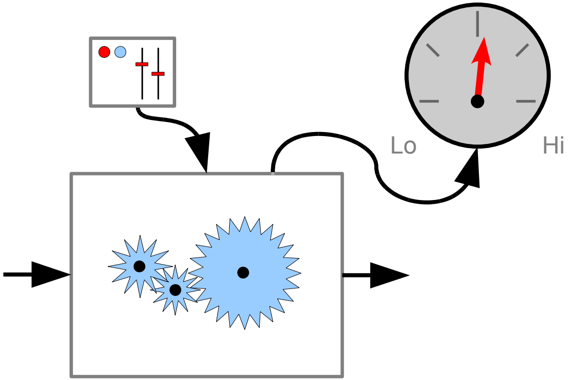
**Level 3:** Strive to provide design choices and alternatives.

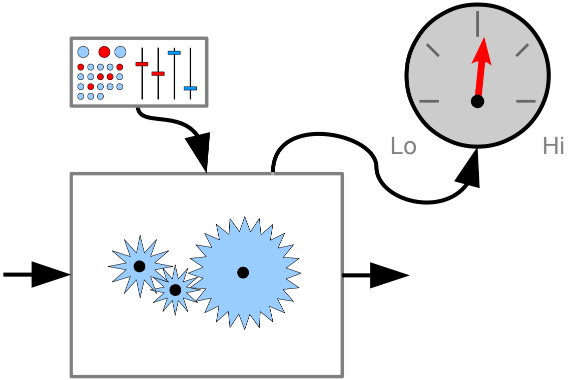
**Level 2:** Keep and expose design choices considered during software development.

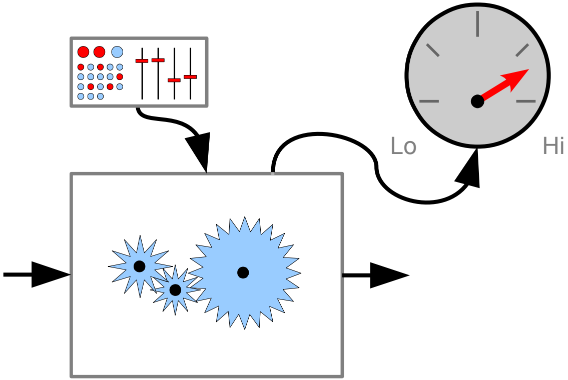
**Level 1:** Expose design choices hardwired into existing code (magic constants, hidden parameters, abandoned design alternatives).

**Level 0:** Optimise settings of parameters exposed by existing software.


Lo
Hi


Lo
Hi


Lo
Hi


Lo
Hi

## Success in optimising speed:

| Application, Design choices | Speedup | PbO level |
|---|---|---|
| SAT-based software verification (SPEAR), 41<br>Hutter, Babić, Hoos, Hu (2007) | 4.5–500 × | 2–3 |
| AI Planning (LPG), 62<br>Vallati, Fawcett, Gerevini, Hoos, Saetti (2011) | 3–118 × | 1 |
| Mixed integer programming (CPLEX), 76<br>Hutter, Hoos, Leyton-Brown (2010) | 2–52 × | 0 |

## ... and solution quality:

University timetabling, 18 design choices, PbO level 2–3
⇝ new state of the art; UBC exam scheduling
Fawcett, Chiarandini, Hoos (2009)

Machine learning / Classification, 786 design choices, PbO level 0–1
⇝ outperforms specialised model selection & hyper-parameter optimisation methods from machine learning
Thornton, Hutter, Hoos, Leyton-Brown (2012–13)

## PbO enables ...

- performance optimisation for different use contexts
  (some details later)

- adaptation to changing use contexts
  (see, *e.g.*, life-long learning – Thrun 1996)

- self-adaptation while solving given problem instance
  (*e.g.*, Battiti *et al.* 2008; Carchrae & Beck 2005; Da Costa *et al.* 2008)

- automated generation of instance-based solver selectors
  (*e.g.*, SATzilla – Leyton-Brown *et al.* 2003, Xu *et al.* 2008;
  Hydra – Xu *et al.* 2010; ISAC – Kadioglu *et al.* 2010)

- automated generation of parallel solver portfolios
  (*e.g.*, Huberman *et al.* 1997; Gomes & Selman 2001;
  Schneider *et al.* 2012)

# Cost & concerns

But what about ...

- Computational complexity?
- Cost of development?
- Limitations of scope?

# Computationally too expensive?

## SPEAR revisited:

- total configuration time on software verification benchmarks: $\approx$ 30 CPU days
- wall-clock time on 10 CPU cluster: $\approx$ 3 days
- cost on Amazon Elastic Compute Cloud (EC2): 61.20 USD (= 42.58 EUR)
- 61.20 USD pays for ...
  - 1:45 hours of average software engineer
  - 8:26 hours at minimum wage

# Too expensive in terms of development?

## Design and coding:

- tradeoff between performance/flexibility and overhead
- overhead depends on level of PbO
- traditional approach: cost from manual exploration of design choices!

## Testing and debugging:

- design alternatives for individual mechanisms and components can be tested separately

$\rightsquigarrow$ effort linear (rather than exponential) in the number of design choices

# Limited to the "niche" of NP-hard problem solving?

## Some PbO-flavoured work in the literature:

- computing-platform-specific performance optimisation of linear algebra routines
  (Whaley *et al.* 2001)
- optimisation of sorting algorithms using genetic programming
  (Li *et al.* 2005)
- compiler optimisation
  (Pan & Eigenmann 2006, Cavazos *et al.* 2007)
- database server configuration
  (Diao *et al.* 2003)

# Overview

- Programming by Optimization (PbO):
  Motivation and Introduction

- Algorithm Configuration
  - Methods (components of algorithm configuration)
  - Systems (that instantiate these components)
  - Demo & Practical Issues
  - Case Studies

- Portfolio-Based Algorithm Selection

- Software Development Support & Further Directions

# The Algorithm Configuration Problem

## Definition

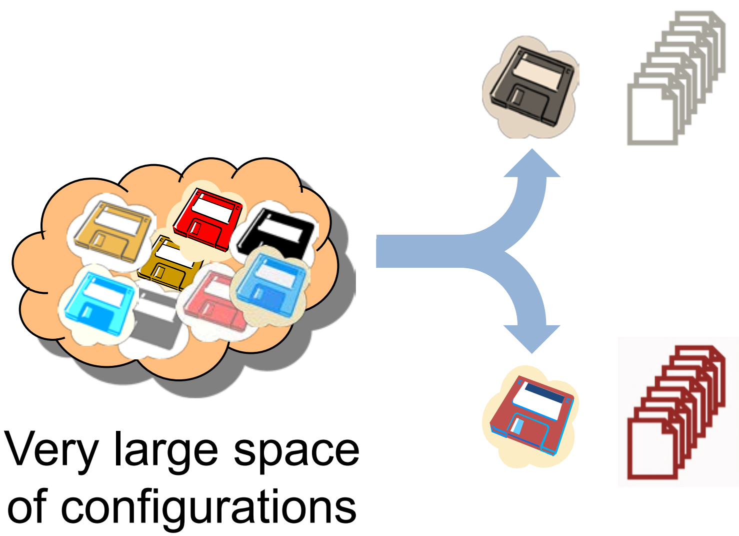
- Given:
  - Runnable algorithm *A* with configuration space $\boldsymbol{\Theta} = \Theta_1 \times \cdots \times \Theta_n$
  - Distribution *D* over problem instances $\Pi$
  - Performance metric $m : \boldsymbol{\Theta} \times \Pi \rightarrow \mathbb{R}$
- Find:

$$\boxed{\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \mathbb{E}_{\pi \sim D}[m(\boldsymbol{\theta}, \pi)]}$$

## Motivation

Customize versatile algorithms for different application domains

- Fully automated improvements
- Optimize speed, accuracy, memory, energy consumption, …



# Algorithm Parameters

## Parameter types

- Continuous, integer, ordinal
- **Categorical**: finite domain, unordered, e.g. {a,b,c}

## Parameter space has **structure**

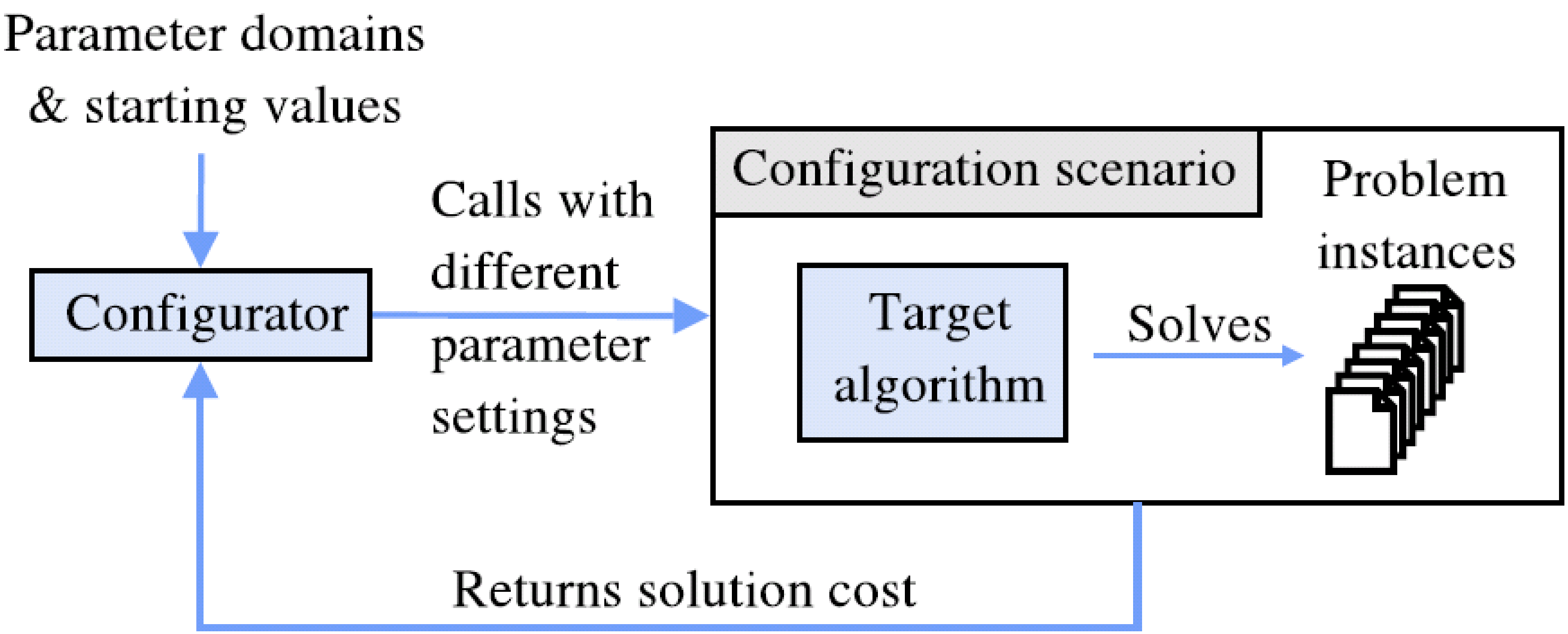
- E.g. parameter C of heuristic A is only active if A is used
- In this case, we say C is a **conditional parameter** with parent A

## Parameters give rise to a **structured space of algorithms**

- Many **configurations** (e.g. $10^{47}$)
- Configurations often yield qualitatively different behaviour

$\rightarrow$ Algorithm configuration (as opposed to “parameter tuning”)

# The Algorithm Configuration Process



# Recall the Spear Example

## SAT solver for formal verification

- 26 user-specifiable parameters
- 7 categorical, 3 Boolean, 12 continuous, 4 integer

## Objective: minimize runtime on software verification instance set

## Issues:

- Many possible settings ($8.34 \times 10^{17}$ after discretization)
- Evaluating performance of a configuration is expensive
  - Instances vary in hardness
    - Some take milliseconds, other days (for the default)
  - Improvement on a few instances might not mean much

# Configurators have Two Key Components

- Component 1: which configuration to evaluate next?
  - Out of a large combinatorial search space
- Component 2: how to evaluate that configuration?
  - Avoiding the expense of evaluating on all instances
  - Generalizing to new problem instances

# Automated Algorithm Configuration: Outline

- Methods (components of algorithm configuration)
- Systems (that instantiate these components)

- Demo & Practical Issues
- Case Studies

# Component 1: Which Configuration to Evaluate?

- For this component, we can consider a simpler problem:

  **Blackbox function optimization** $\boxed{\min_{\theta\in\Theta} f(\theta)}$

  - **Only mode of interaction: query $f(\theta)$ at arbitrary $\theta\in\Theta$**

$$\theta \rightarrow \blacksquare \rightarrow f(\theta)$$

  - Abstracts away the complexity of multiple instances
  - $\Theta$ is still a structured space
    - Mixed continuous/discrete
    - Conditional parameters
    - Still more general than "standard" continuous BBO [e.g., Hansen et al.]

# The Simplest Search Strategy: Random Search

- Select configurations uniformly at random
  - Completely uninformed
  - Global search, won't get stuck in a local region
  - At least it's better than grid search:

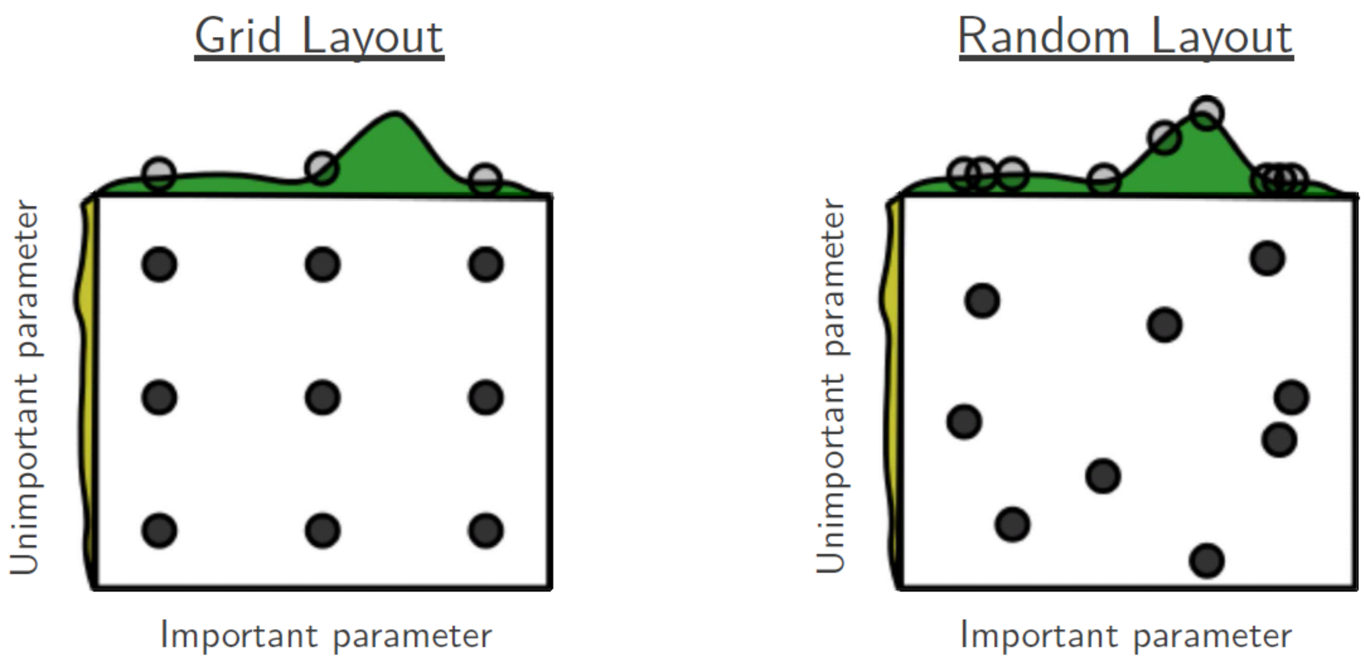



Image source: Bergstra et al, Random Search for Hyperparameter Optimization, JMLR 2012

# The Other Extreme: Gradient Descent

(aka hill climbing)

*Start with some configuration*

**repeat**

**Modify a single parameter**

**if** *performance on a benchmark set degrades* **then**

*undo modification*

**until** *no more improvement possible*
*(or "good enough")*

# Stochastic Local Search

[e.g., Hoos and Stützle, 2005]

- **Balance intensification and diversification**
  - Intensification: gradient descent
  - Diversification: restarts, random steps, perturbations, …

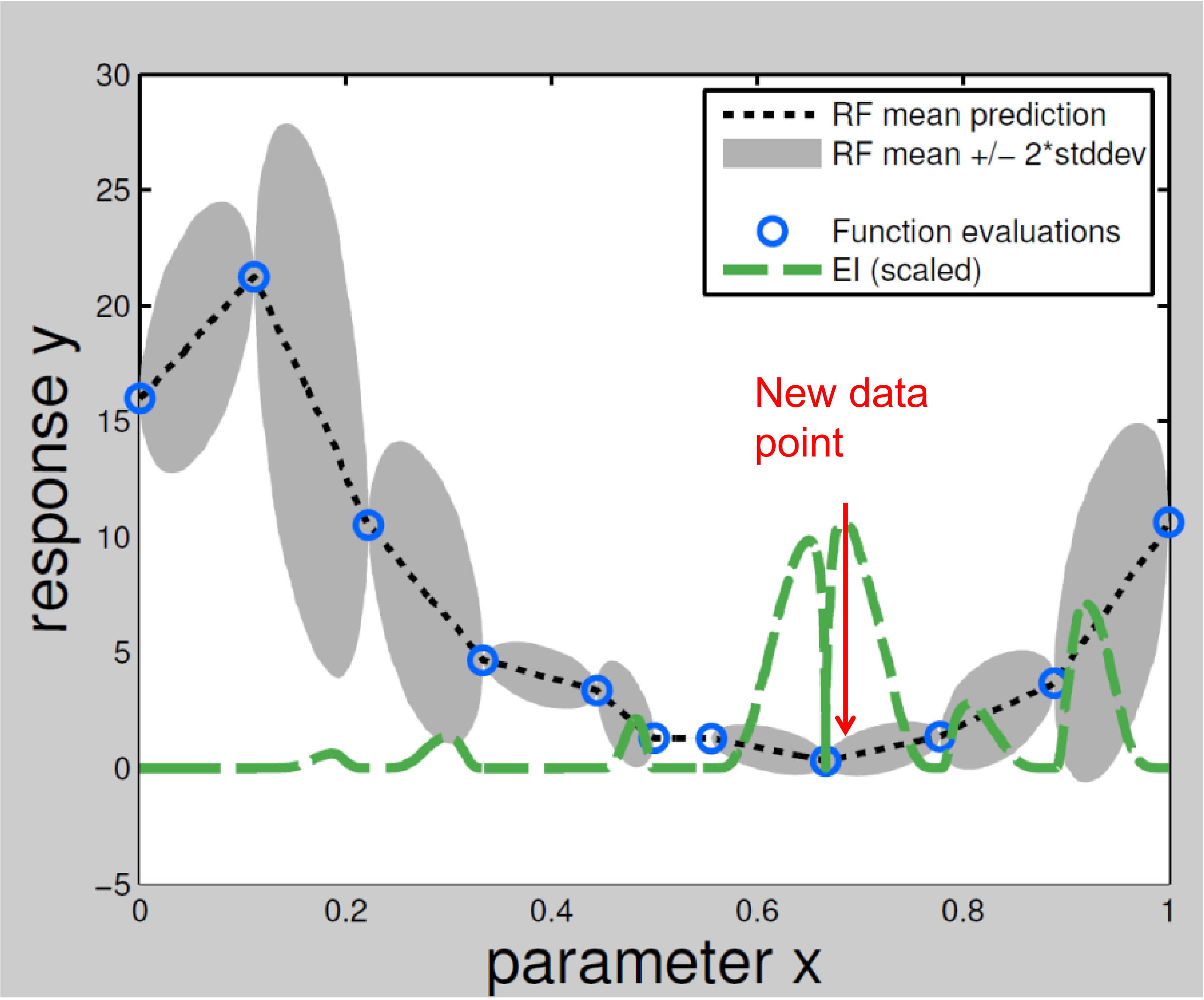
- Prominent general methods
  - Taboo search [Glover, 1986]
  - Simulated annealing [Kirkpatrick, Gelatt, C. D.; Vecchi, 1983]
  - Iterated local search [Lourenço, Martin & Stützle, 2003]

# Population-based Methods

- Population of configurations
  - Global + local search via population
  - Maintain **population fitness & diversity**

- Examples
  - Genetic algorithms [e.g., Barricelli, ’57, Goldberg, ’89]
  - Evolutionary strategies [e.g., Beyer & Schwefel, ’02]
  - Ant colony optimization [e.g., Dorigo & Stützle, ’04]
  - Particle swarm optimization [e.g., Kennedy & Eberhart, ’95]

# Sequential Model-Based Optimization



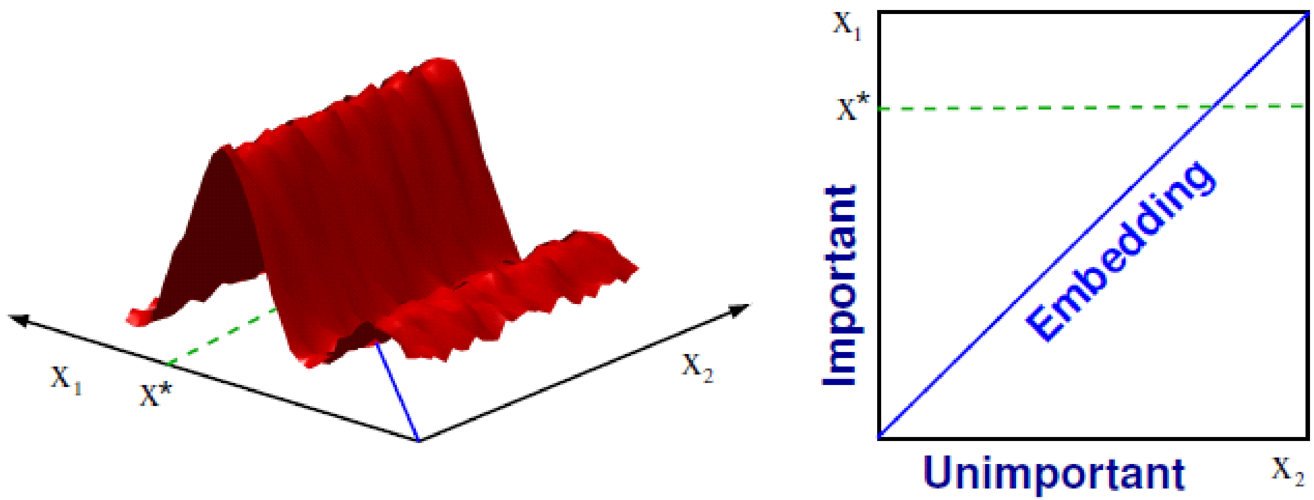
# Sequential Model-Based Optimization

- Popular approach in statistics
  to minimize expensive blackbox functions [e.g., Mockus, '78]

- Recent progress in the machine learning literature:
  **global convergence rates** for continuous optimization
  [Srinivas et al, ICML 2010]
  [Bull, JMLR 2011]
  [Bubeck et al., JMLR 2011]
  [de Freitas, Smola, Zoghi, ICML 2012]

# Exploiting Low Effective Dimensionality

- Often, not all parameters are equally important
- Can search in an embedded lower-dimensional space




- For details, see:
  - Bayesian Optimization in High Dimensions via Random Embeddings, Tuesday, 13:30, 201CD [Wang et al, IJCAI 2013]

# Summary 1: Which Configuration to Evaluate?

- Need to balance diversification and intensification
- The extremes
  - Random search
  - Hillclimbing
- Stochastic local search (SLS)
- Population-based methods
- Sequential Model-Based Optimization
- Exploiting low effective dimensionality

# Component 2: How to Evaluate a Configuration?

Back to general algorithm configuration

- Given:
  - Runnable algorithm $\mathcal{A}$ with configuration space $\boldsymbol{\Theta} = \Theta_1 \times \cdots \times \Theta_n$
  - Distribution D over problem instances $\Pi$
  - Performance metric $m : \boldsymbol{\Theta} \times \Pi \rightarrow \mathbb{R}$
- Find:

$$\boxed{\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \mathbb{E}_{\pi \sim D}[m(\boldsymbol{\theta}, \pi)]}$$

Recall the Spear example

- Instances vary in hardness
  - Some take milliseconds, other days (for the default)
  - Thus, improvement on a few instances might not mean much

# Simplest Solution: Use Fixed N Instances

- Effectively treat the problem as a blackbox function optimization problem

- **Issue: how large to choose N?**
  – Too small: overtuning
  – Too large: every function evaluation is slow

- General principle
  – Don’t waste time on bad configurations
  – Evaluate good configurations more thoroughly

# Racing Algorithms

[Maron & Moore, NIPS 1994]

[Birattari, Stützle, Paquete & Varrentrapp, GECCO 2002]

- Compare two or more algorithms against each other
  - Perform one run for each configuration at a time
  - **Discard configurations when dominated**

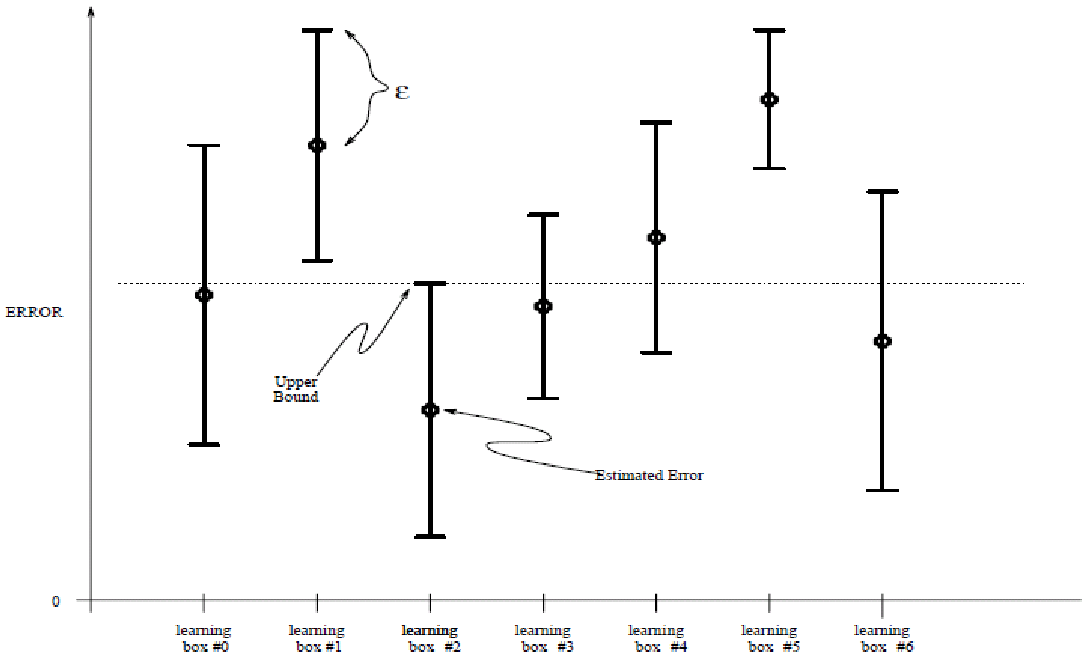



Image source: Maron & Moore, Hoeffding Races, NIPS 1994

# Saving Time: Aggressive Racing

[Hutter, Hoos & Stützle, AAAI 2007]

- Race new configurations against the best known
  - Discard poor new configurations quickly
  - **No requirement for statistical domination**

- Search component should allow to return to configurations discarded because they were “unlucky”

# Saving More Time: Adaptive Capping

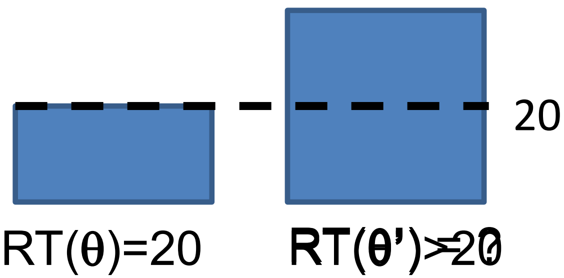
[Hutter, Hoos, Leyton-Brown & Stützle, JAIR 2009]

(only when minimizing algorithm runtime)

**Can terminate runs for poor configurations θ' early:**

– Is θ' better than θ?

- Example:




- Can terminate evaluation of θ' once guaranteed to be worse than θ

# Summary 2: How to Evaluate a Configuration?

- Simplest: fixed set of N instances
- General principle
  - Don't waste time on bad configurations
  - Evaluate good configurations more thoroughly
- Instantiations of principle
  - Racing
  - Aggressive racing
  - Adaptive capping

# Automated Algorithm Configuration: Outline

- Methods (components of algorithm configuration)
- Systems (that instantiate these components)

- Demo & Practical Issues
- Case Studies

# Overview: Algorithm Configuration Systems

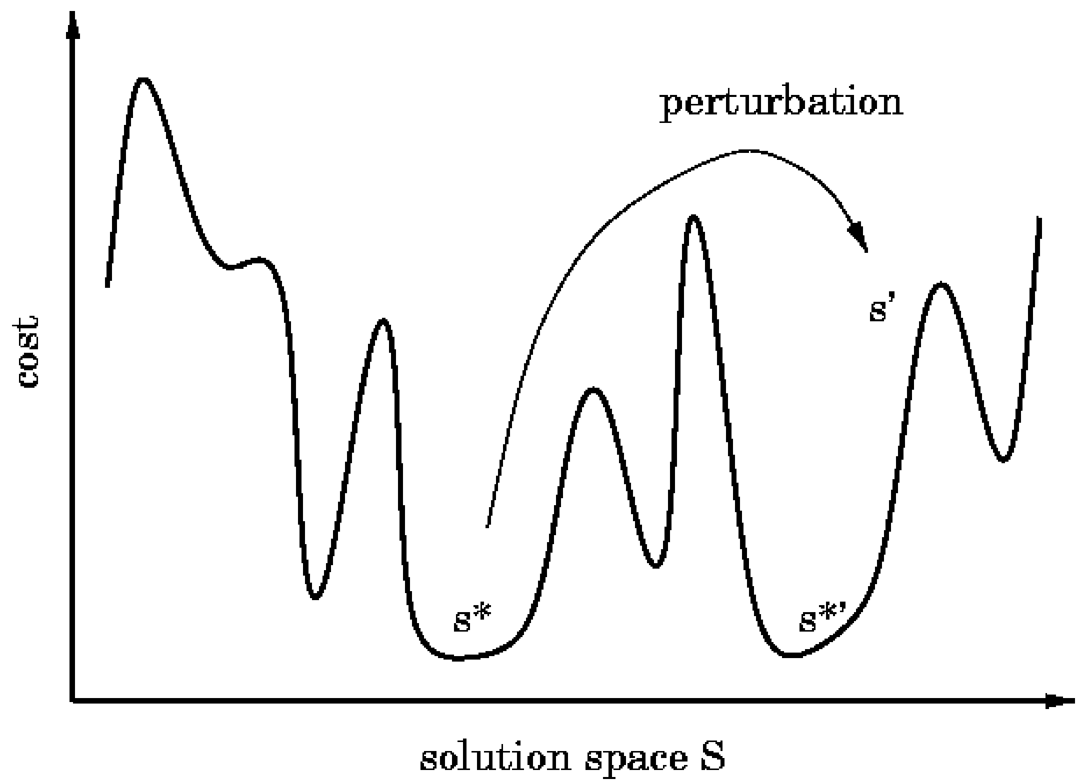
- Continuous parameters, single instances (blackbox opt)
  - Covariance adaptation evolutionary strategy (CMA-ES) [Hansen et al, since ’06]
  - Sequential Parameter Optimization (SPO) [Bartz-Beielstein et al, ’06]
  - Random Embedding Bayesian optimization (REMBO) [Wang et al, ’13]

- General algorithm configuration methods
  - ParamILS [Hutter et al, ’07 and ’09]
  - Gender-based Genetic Algorithm (GGA) [Ansotegui et al, ’09]
  - Iterated F-Race [Birattari et al, ’02 and ‘10]
  - Sequential Model-based Algorithm Configuration (SMAC) [Hutter et al, since ’11]
  - Distributed SMAC [Hutter et al, since ’12]

# The ParamILS Framework

[Hutter, Hoos, Leyton-Brown & Stützle, AAAI 2007 & JAIR 2009]

Iterated Local Search in parameter configuration space:



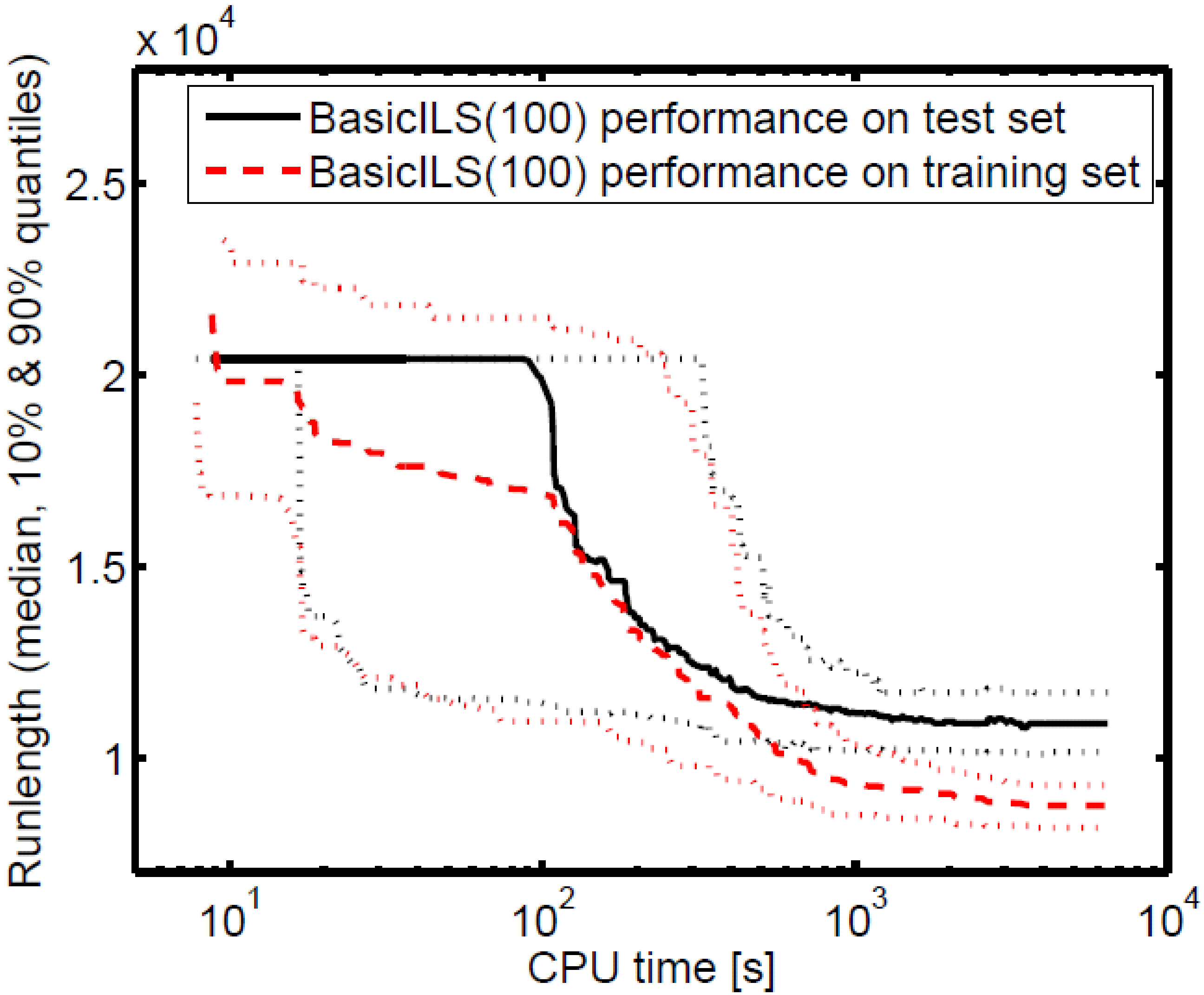

→ Performs **biased random walk over local optima**

# The BasicILS(N) algorithm

- Instantiates the ParamILS framework
- **Uses a fixed number of N runs for each evaluation**
  - Sample N instance from given set (with repetitions)
  - Same instances (and seeds) for evaluating all configurations
  - Essentially treats the problem as blackbox optimization

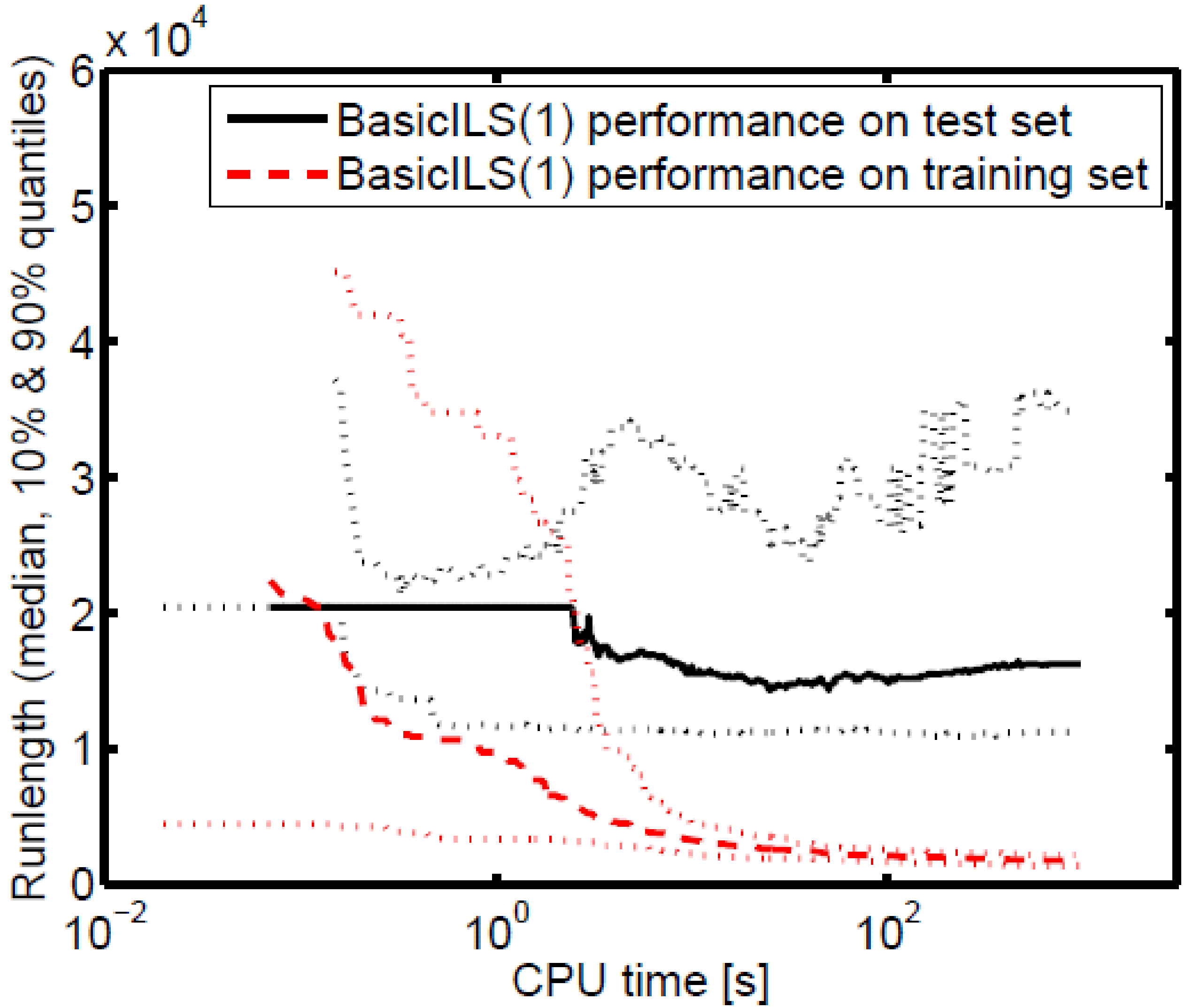
- **How to choose N?**
  - Too high: evaluating a configuration is expensive
    - $\rightarrow$ Optimization process is slow
  - Too low: noisy approximations of true cost
    - $\rightarrow$ Poor generalization to test instances / seeds

# Generalization to Test set, Large N (N=100)



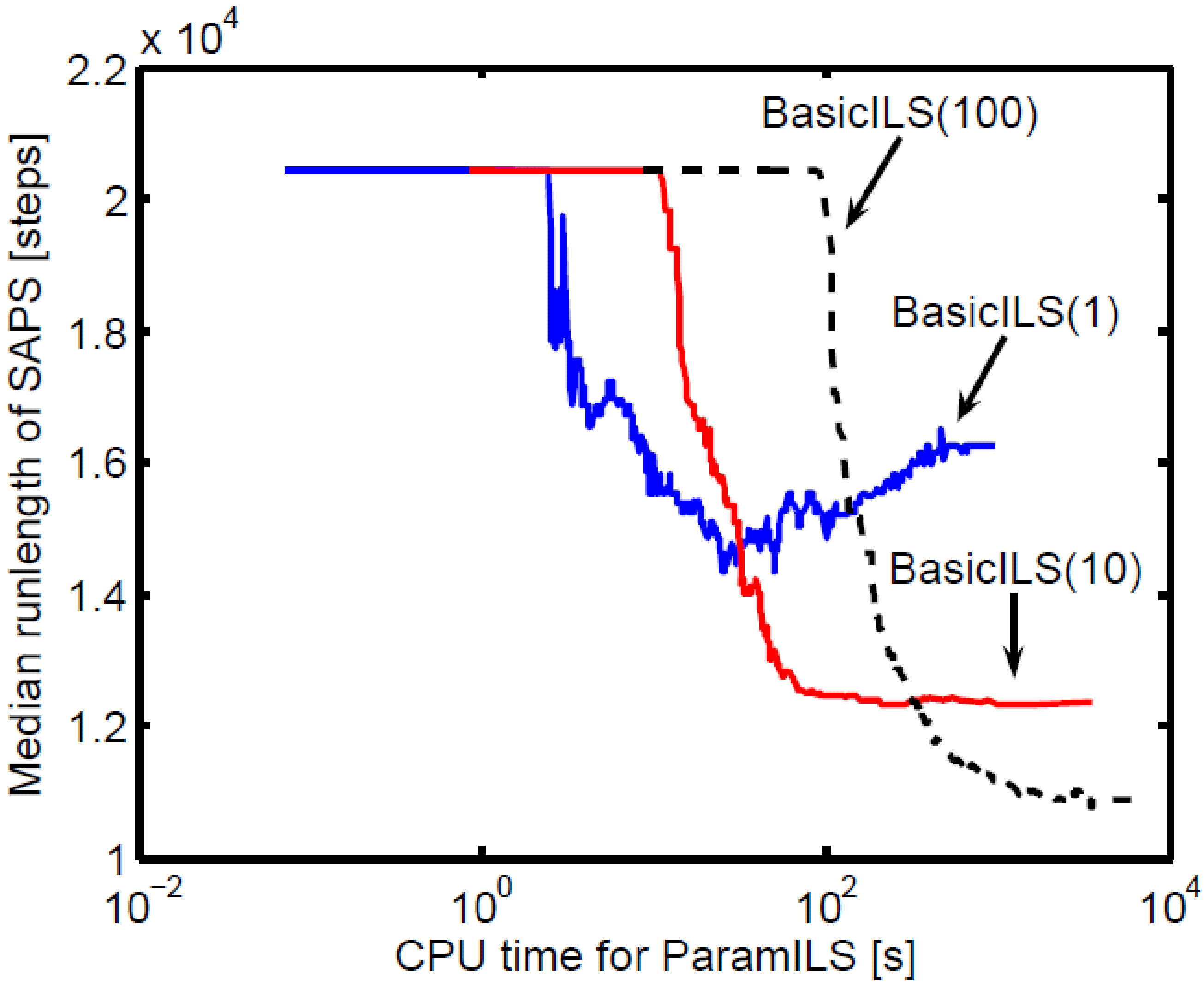

SAPS on a single QWH instance
(same instance for training & test; only difference: seeds)

# Generalization to Test Set, Small N (N=1)




SAPS on a single QWH instance
(same instance for training & test; only difference: seeds)

# BasicILS: Tradeoff Between Speed & Generalization




Test performance of SAPS on a single QWH instance

# The FocusedILS Algorithm

## Aggressive racing: more runs for good configurations

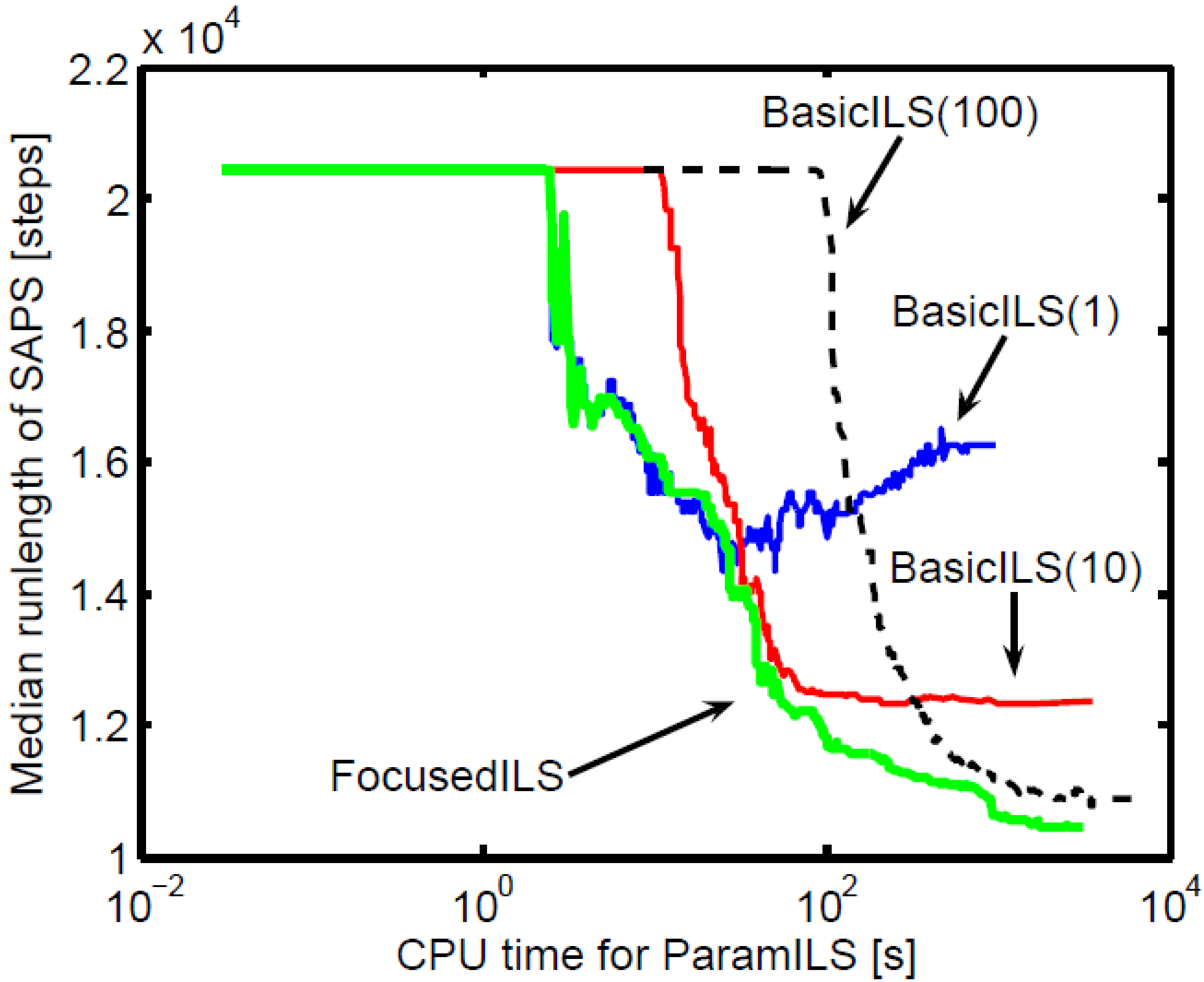
- Start with $N(\theta) = 0$ for all configurations
- Increment $N(\theta)$ whenever the search visits $\theta$
- "Bonus" runs for configurations that win many comparisons

## Theorem

As the number of FocusedILS iterations $\rightarrow \infty$,
it converges to the true optimal conguration

- Key ideas in proof:
  1. The underlying ILS eventually reaches any configuration
  2. For $N(\theta) \rightarrow \infty$, the error in cost approximations vanishes

# FocusedILS: Tradeoff Between Speed & Generalization




Test performance of SAPS on a single QWH instance

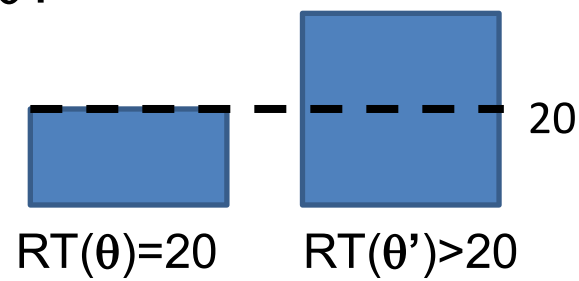
# Speeding up ParamILS

[Hutter , Hoos, Leyton-Brown, and Stützle, JAIR 2009]

## Standard adaptive capping

– Is θ' better than θ?

- Example:




- Can terminate evaluation of θ' once guaranteed to be worse than θ

## Theorem

Early termination of poor configurations does not change ParamILS's trajectory

– Often yields substantial speedups

# Gender-based Genetic Algorithm (GGA)

[Ansotegui, Sellmann & Tierney, CP 2009]

- Genetic algorithm
  - **Genome = parameter configuration**
  - Combine genomes of 2 parents to form an offspring

- **Two genders in the population**
  - Selection pressure only on one gender
  - Preserves diversity of the population

# Gender-based Genetic Algorithm (GGA)

[Ansotegui, Sellmann & Tierney, CP 2009]

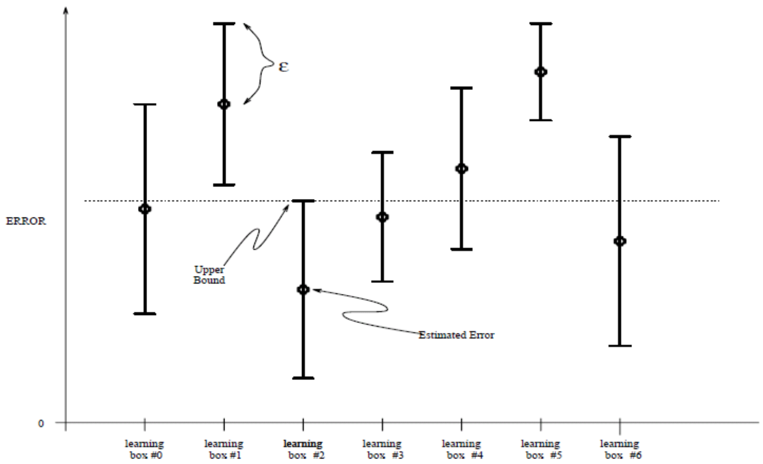
- Use N instances to evaluate configurations
  - Increase N in each generation
  - **Linear increase from $N_{start}$ to $N_{end}$**
    - User specifies #generations ahead of time

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

# F-Race and Iterated F-Race

[Birattari et al, GECCO 2002 and book chapter 2010]

- **F-Race**
  - Standard racing framework
  - F-test to establish that some configuration is dominated
  - Followed by pairwise t tests if F-test succeeds




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

# F-Race and Iterated F-Race

[Birattari et al, GECCO 2002 and book chapter 2010]

- **Can use parallel resources**
  - Simply do the k runs of each iteration in parallel
  - But does not support adaptive capping

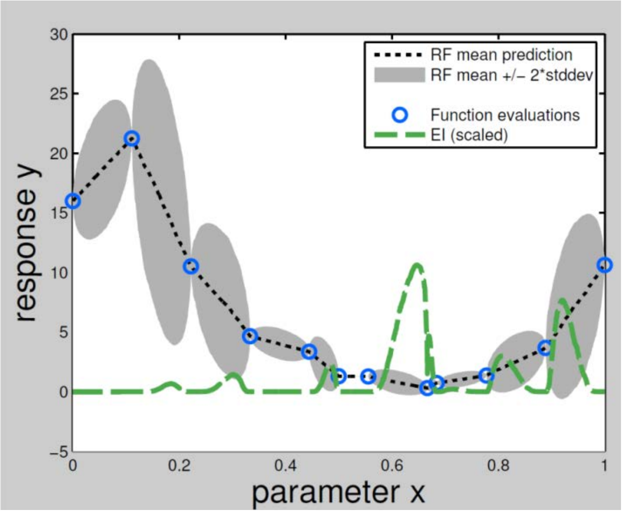
- Expected performance
  - Strong when the key challenge are reliable comparisons between configurations
  - Less good when the search component is the challenge

# SMAC

[Hutter, Hoos & Leyton-Brown, LION 2011]

## SMAC: Sequential Model-Based Algorithm Configuration

– Sequential Model-Based Optimization
  & aggressive racing




**repeat**

- construct a model to predict performance
- use that model to select promising configurations
- compare each selected configuration against the best known

**until** time budget exhausted

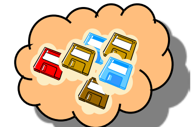
# SMAC: Aggressive Racing

- More runs for good configurations
- Increase #runs for incumbent over time
- **Theorem** for discrete configuration spaces:

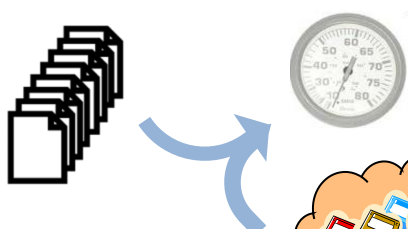
  As SMAC's overall time budget $\rightarrow \infty$,
  it converges to the optimal configuration

# SMAC: Performance Models Across Instances

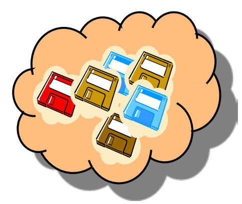
Given:

- Configuration space $\boldsymbol{\Theta} = \Theta_1 \times \cdots \times \Theta_n$ 
- For each problem instance $i$: $\mathbf{x}_i$, a vector of feature values 
- Observed algorithm runtime data: $(\theta_1, \mathbf{x}_1, y_1), \ldots, (\theta_n, \mathbf{x}_n, y_n)$ 

Find: a **mapping** $m$: $[\theta, \boldsymbol{x}] \mapsto y$ predicting A's performance



$\approx m\,(\boldsymbol{\theta}, x)$



- Rich literature on such performance prediction problems
  [see, e.g, Hutter, Xu, Hoos, Leyton-Brown, AIJ 2013, for an overview]
- Here: use a model $m$ based on **random forests**

# Regression Trees: Fitting to Data

- In each internal criterion used
- In each leaf: sto

| 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 |

$feature_2 > 3.5$

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

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

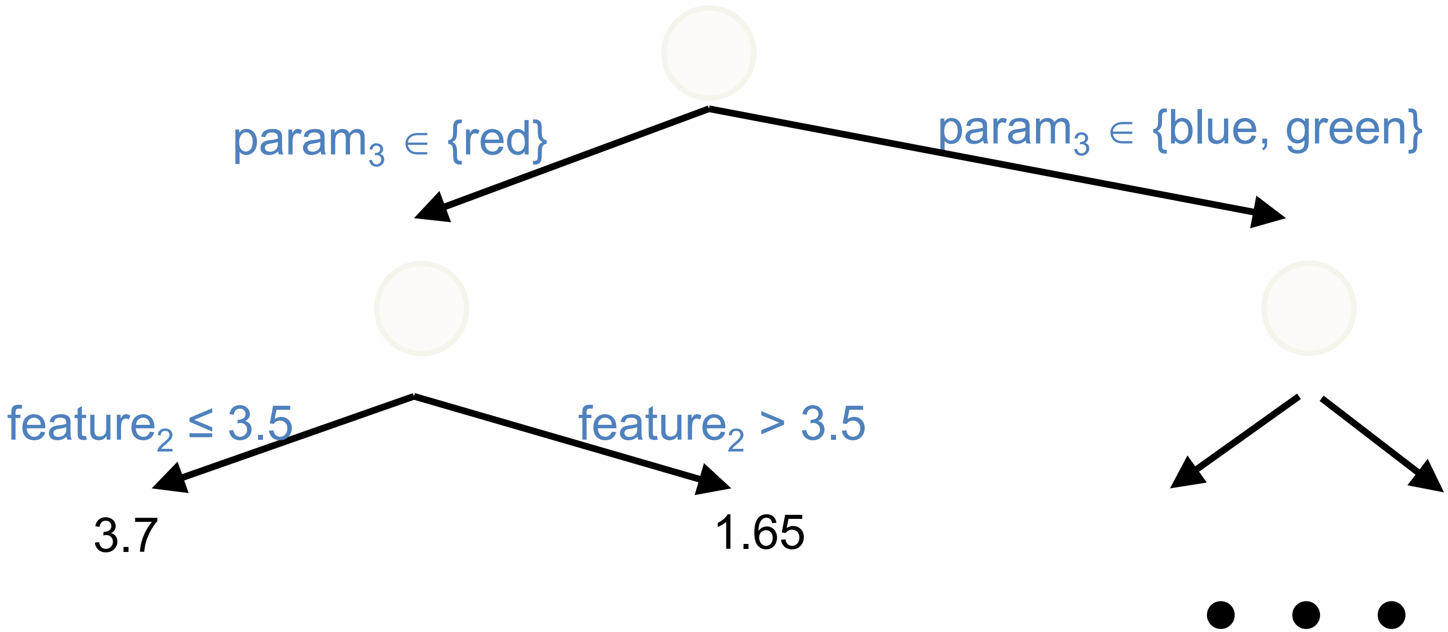
| param 1 | feature 2 | param 3 | runtime |
|---|---|---|---|
| false | 2.5 | blue | 20 |
| false | 5.5 | blue | 25 |
| true | 4.5 | green | 19 |
| true | 4 | blue | 12 |
| true | 3.5 | green | 17 |

• • •

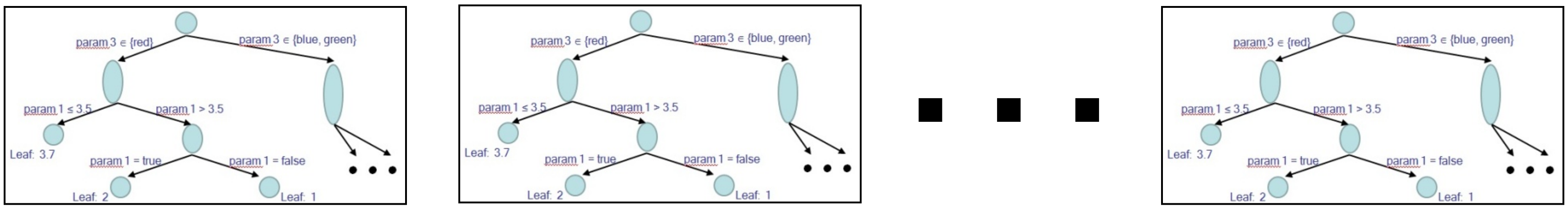
# Regression Trees: Predictions for New Inputs

E.g. $x_{n+1}$ = (true, 4.7, red)

– Walk down tree, return mean runtime stored in leaf $\Rightarrow$ 1.65



# Random Forests: Sets of Regression Trees




## Training

- **Subsample** the data T times (with repetitions)
- For each subsample, fit a **randomized** regression tree
- Complexity for N data points: $O(T\ N \log^2 N)$

## Prediction

- Predict with each of the T trees
- Return empirical **mean and variance** across these T predictions
- Complexity for N data points: $O(T \log N)$

# SMAC: Benefits of Random Forests

## Robustness

- No need to optimize hyperparameters
- Already good predictions with few training data points

## Automated selection of important input dimensions

- Continuous, integer, and categorical inputs
- Up to 138 features,  76 parameters
- Can identify important feature and parameter subsets
  - Sometimes 1 feature and 2 parameters are enough

[Hutter, Hoos, Leyton-Brown, LION 2013]

# SMAC: Models Across Multiple Instances

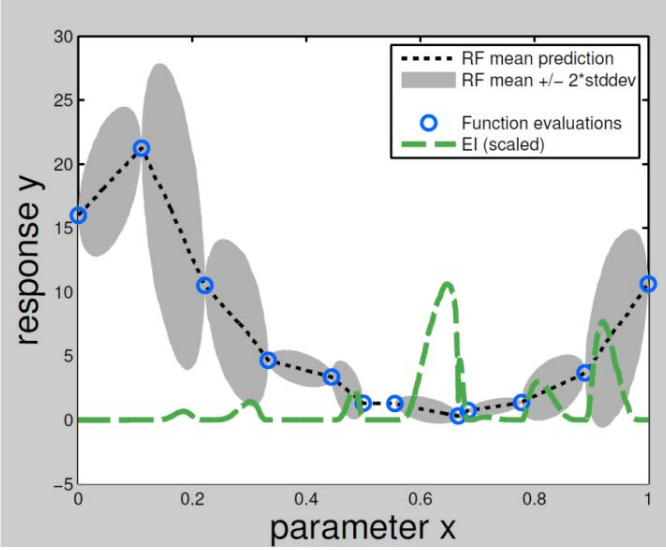
- Fit a random forest model $m : \boldsymbol{\Theta} \times \Pi \to \mathbb{R}$

- Aggregate over instances by marginalization

$$f(\boldsymbol{\theta}) := \mathbb{E}_{\pi \sim D}[m(\boldsymbol{\theta}, \pi)]$$

  – Intuition: predict for each instance and take the average
  – More efficient implementation in random forests

# SMAC: Putting it all Together




Initialize with a single run for the default

**repeat**

- learn a RF model from data so far: $m : \boldsymbol{\Theta} \times \Pi \rightarrow \mathbb{R}$
- Aggregate over instances: $f(\boldsymbol{\theta}) := \mathbb{E}_{\pi \sim D}[m(\boldsymbol{\theta}, \pi)]$
- use model *f* to select promising configurations
- compare each selected configuration against the best known

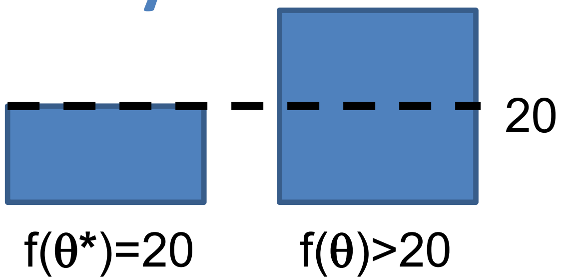
**until** time budget exhausted

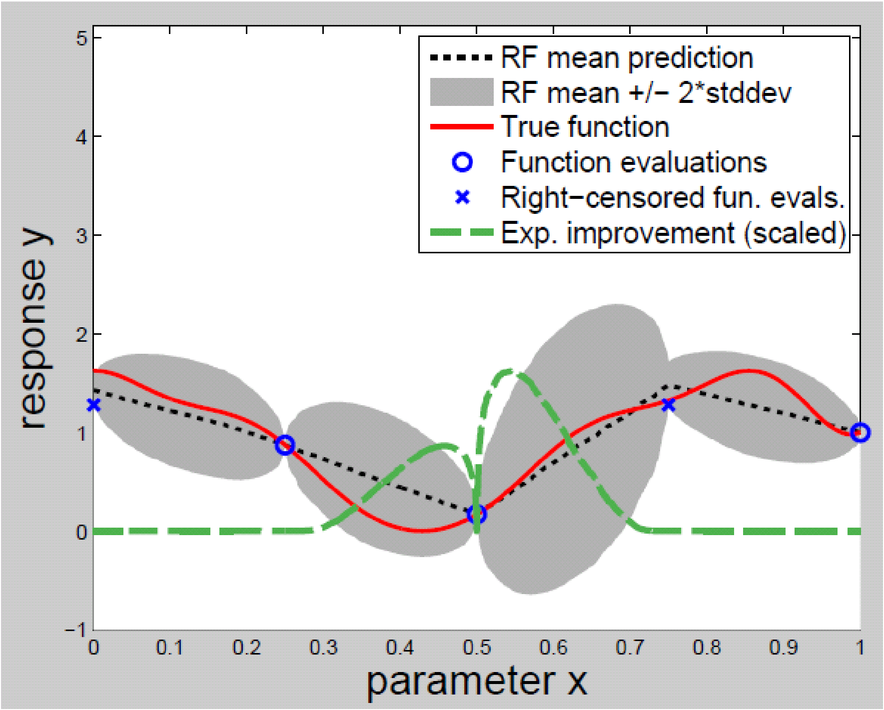
# SMAC: Adaptive Capping

[Hutter, Hoos & Leyton-Brown, NIPS 2011]

## Terminate runs for poor configurations θ early:

– Lower bound on runtime
  → right-censored data point



# Distributed SMAC

[Hutter, Hoos & Leyton-Brown, LION 2012]

[Ramage, Hutter, Hoos & Leyton-Brown, in preparation]

- **Distribute target algorithm runs across workers**
  - Maintain queue of promising configurations
  - Compare these to $\theta^*$ on distributed worker cores
- **Wallclock speedups**
  - Almost perfect speedups with up to 16 parallel workers
  - Up to 50-fold speedups with 64 workers
    - Reductions in wall clock time: 5h $\rightarrow$ 6 min - 15min<br>2 days $\rightarrow$ 40min - 2h

# Summary: Algorithm Configuration Systems

- ParamILS
- Gender-based Genetic Algorithm (GGA)
- Iterated F-Race
- Sequential Model-based Algorithm Configuration (SMAC)
- Distributed SMAC

- Which one is best?
  - First configurator competition to come in 2014 (coorganized by leading groups on algorithm configuration, co-chairs: Frank Hutter & Yuri Malitsky)

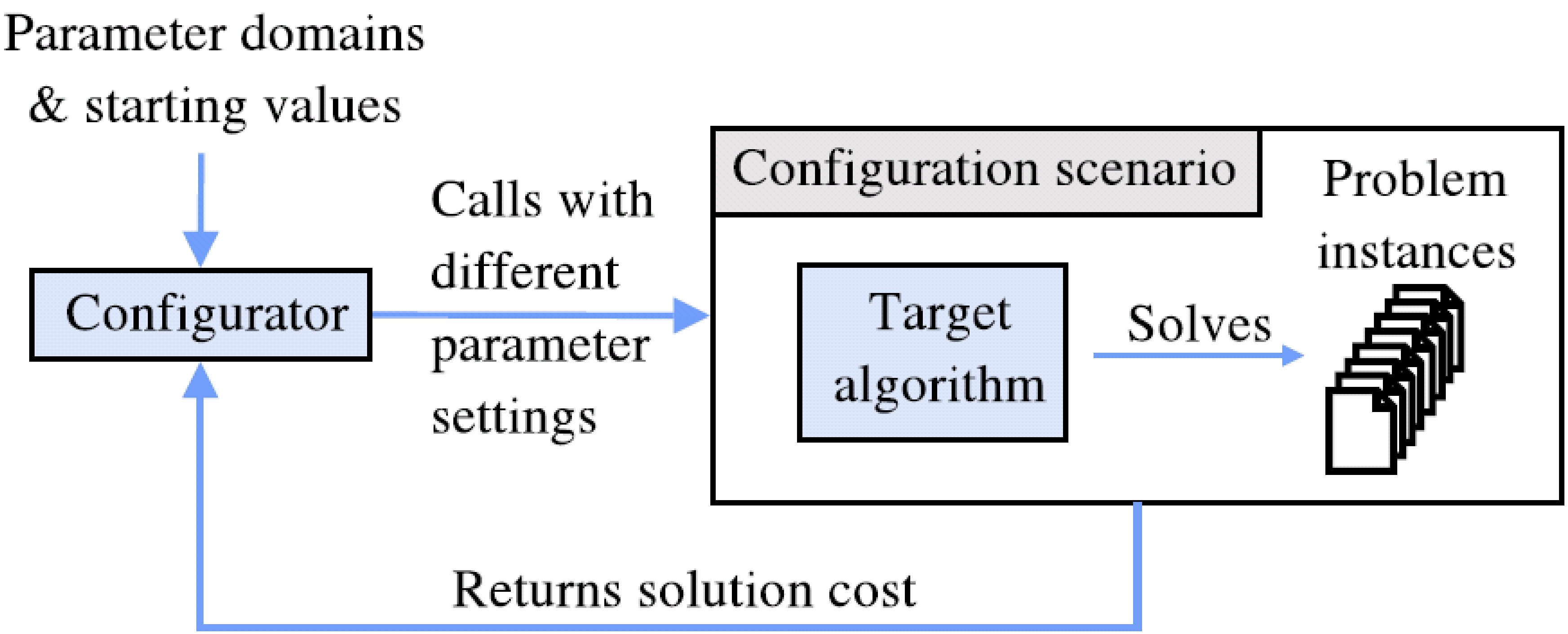
# Automated Algorithm Configuration: Outline

- Methods (components of algorithm configuration)
- Systems (that instantiate these components)

➡ Demo & Practical Issues

- Case Studies

# The Algorithm Configuration Process




What the user has to provide

Parameter space declaration file

preproc {none, simple, expensive} [simple]
alpha [1,5] [2]
beta [0.1,1] [0.5]

Wrapper for command line call

./wrapper –inst X –timeout 30
-preproc none -alpha 3 -beta 0.7
→ e.g. "successful after 3.4 seconds"

# Example: Running SMAC

```
wget http://www.cs.ubc.ca/labs/beta/Projects/SMAC/smac-v2.04.01-master-447.tar.gz

tar xzvf smac-v2.04.01-master-447.tar.gz

cd smac-v2.04.01-master-447

./smac –seed 0  --scenarioFile
example_spear/scenario-Spear-QCP-sat-small-train-small-test-mixed.txt
```

Scenario file holds:
- Location of parameter file, wrapper & instances
- Objective function (here: minimize avg. runtime)
- Configuration budget (here: 30s)
- Maximal captime per target run (here: 5s)

# Output of a SMAC run

```
[…]

[INFO ] *****Runtime Statistics*****
 Iteration: 12
 Incumbent ID: 11 (0x27CA0)
 Number of Runs for Incumbent: 26
 Number of Instances for Incumbent: 5
 Number of Configurations Run: 25
 Performance of the Incumbent: 0.05399999999999999
 Total Number of runs performed: 101
 Configuration time budget used: 30.020000000000034 s
[INFO ] **********************************************

[INFO ] Total Objective of Final Incumbent 13 (0x30977) on training set:
0.05399999999999999; on test set: 0.055

[INFO ] Sample Call for Final Incumbent 13 (0x30977)
cd /global/home/hutter/ac/smac-v2.04.01-master-447/example_spear; ruby spear_wrapper.rb example_data/QCP-
instances/qcplin2006.10422.cnf 0 5.0 2147483647 2897346 -sp-clause-activity-inc '1.3162094350513607' -sp-
clause-decay '1.739666995554204' -sp-clause-del-heur '1' -sp-first-restart '846' -sp-learned-clause-sort-heur '10' -sp-
learned-clauses-inc '1.395279056466624' -sp-learned-size-factor '0.6071142792450034' -sp-orig-clause-sort-heur '7'
-sp-phase-dec-heur '5' -sp-rand-phase-dec-freq '0.005' -sp-rand-phase-scaling '0.8863796134762909' -sp-rand-var-
dec-freq '0.01' -sp-rand-var-dec-scaling '0.6433957166060014' -sp-resolution '0' -sp-restart-inc
'1.7639087832223321' -sp-update-dec-queue '1' -sp-use-pure-literal-rule '0' -sp-var-activity-inc
'0.7825881046949665' -sp-var-dec-heur '3' -sp-variable-decay '1.0374907487192533'
```

# Decision #1: Configuration Budget & Max. Captime

- **Configuration budget**
  - Dictated by your resources & needs
    - E.g., start the configurator before leaving work on Friday
  - The longer the better (but diminishing returns)
    - Rough rule of thumb: at least enough time for 1000 target runs

- **Maximal captime per target run**
  - Dictated by your needs (typical instance hardness, etc)
  - Too high: slow progress
  - Too low: possible overtuning to easy instances
  - For SAT etc, often use 300 CPU seconds

# Decision #2: Choosing the Training Instances

- **Representative instances, moderately hard**
  - Too hard: won't solve many instances, no traction
  - Too easy: will results generalize to harder instances?
  - Rule of thumb: mix of hardness ranges
    - Roughly 75% instances solvable by default in maximal captime

- **Enough instances**
  - The more training instances the better
  - Very homogeneous instance sets: 50 instances might suffice
  - Prefer $\geq$ 300 instances, better $\geq$ 1000 instances

# Decision #2: Choosing the Training Instances

- **Split instance set into training and test sets**
  - Configure on the training instances $\rightarrow$ configuration $\theta^*$
  - Run $\theta^*$ on the test instances
    - Unbiased estimate of performance

**Pitfall: configuring on your test instances**

That's from the dark ages

**Fine practice: do multiple configuration runs and pick the $\theta^*$ with best training performance**

Not (!!) the best on the test set

# Decision #2: Choosing the Training Instances

- **Works much better on homogeneous benchmarks**
  - Instances that have something in common
    - E.g., come from the same problem domain
    - E.g., use the same encoding
  - One configuration likely to perform well on all instances

**Pitfall: configuration on too heterogeneous sets**

There often is no single great overall configuration
(but see algorithm selection etc, second half of the tutorial)

# Decision #3: How Many Parameters to Expose?

- Suggestion: all parameters you don't know to be useless
  - More parameters $\rightarrow$ larger gains possible
  - More parameters $\rightarrow$ harder problem
  - Max. #parameters tackled so far: 768
    [Thornton, Hutter, Hoos & Leyton-Brown, KDD'13]
    - With more time you can search a larger space

**Pitfall: including parameters that change the problem**

E.g., optimality threshold in MIP solving
E.g., how much memory to allow the target algorithm

# Decision #4: How to Wrap the Target Algorithm

- Do not trust any target algorithm
  - Will it terminate in the time you specify?
  - Will it correctly report its time?
  - Will it never use more memory than specified?
  - Will it be correct with all parameter settings?

**Good practice: wrap target runs with tool controlling time and memory (e.g., runsolver** [Roussel et al, '11]**)**

**Good practice: verify correctness of target runs**

Detect crashes & penalize them

**Pitfall: blindly minimizing target algorithm runtime**

Typically, you will minimize the time to crash

# Automated Algorithm Configuration: Outline

- Methods (components of algorithm configuration)
- Systems (that instantiate these components)

- Demo & Practical Issues

Case Studies

# Back to the Spear Example

[Hutter, Babic, Hu & Hoos, FMCAD 2007]

Spear [Babic, 2007]

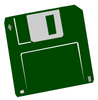
- 26 parameters
- $8.34 \times 10^{17}$ configurations

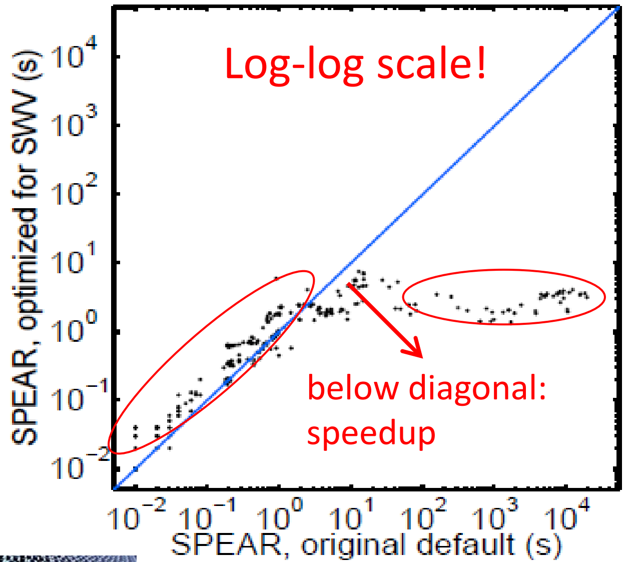
Ran ParamILS, 2 to 3 days × 10 machines

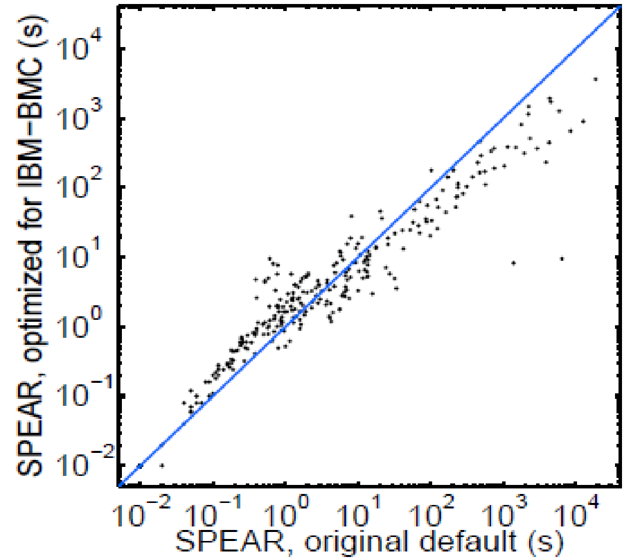
- On a training set from each of 2 distributions

Compared to default (1 week of manual tuning)

- On a disjoint test set from each distribution






500-fold speedup ⇒ won QF_BV category in 2007 SMT competition

4.5-fold speedup

# Other Examples of PbO for SAT

- SATenstein [KhudaBukhsh, Xu, Hoos & Leyton-Brown, IJCAI 2009]
  - Combined ingredients from existing solvers
  - 54 parameters, over $10^{12}$ configurations
  - Speedup factors: 1.6x to 218x

- Captain Jack [Tompkins & Hoos, SAT 2011]
  - Explored a completely new design space
  - 58 parameters, over $10^{50}$ configurations
  - After configuration: best known solver for 3sat10k and IL50k

# Configurable SAT Solver Competition (CSSC) 2013

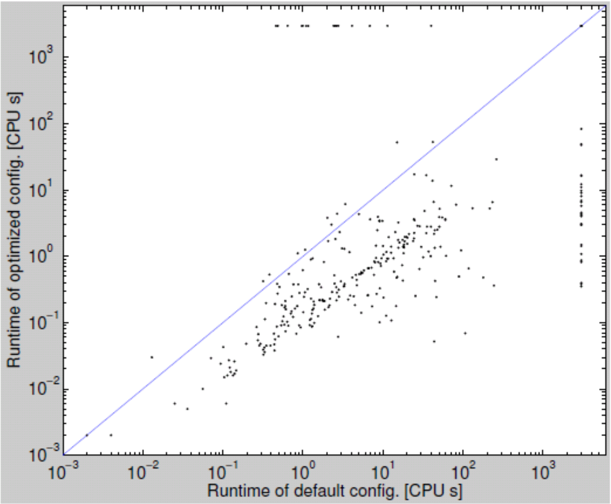
[Hutter, Balint, Bayless, Hoos & Leyton-Brown 2013]

- Annual SAT competition
  - Scores SAT solvers by their performance across instances
  - Medals for best average performance with solver defaults
    - Misleading results: implicitly highlights solvers with good defaults
- CSSC 2013
  - Better reflect an application setting: homogeneous instances $\rightarrow$ can automatically optimize parameters
  - Medals for best performance after configuration

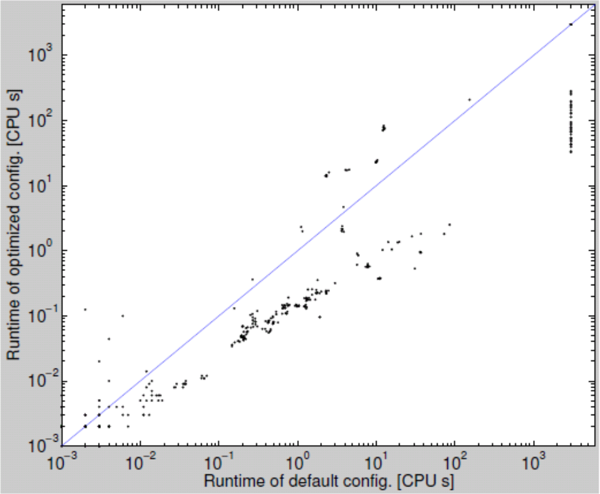
# CSSC 2013 Result #1

[Hutter, Balint, Bayless, Hoos & Leyton-Brown 2013]

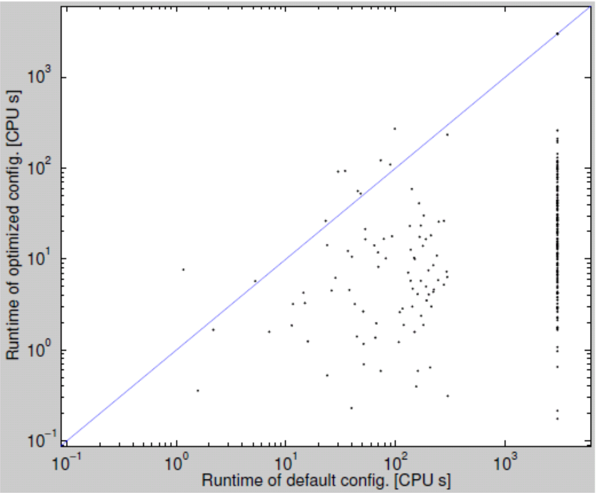
- Performance often improved a lot:




Riss3gExt on BMC08
Timeouts: 32 → 20




Clasp on graph isomorphism
Timeouts: 42 → 6




gNovelty+Gca on 5SAT 500
Timeouts: 163 → 4

# CSSC 2013 Result #2

[Hutter, Balint, Bayless, Hoos & Leyton-Brown 2013]

- Automated configuration changed algorithm rankings
  - Example: random SAT+UNSAT category

| 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 |

# Configuration of a Commercial MIP solver

[Hutter, Hoos & Leyton-Brown, CPAIOR 2010]

## Mixed Integer Programming (MIP)

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

## Commercial MIP solver: IBM ILOG CPLEX

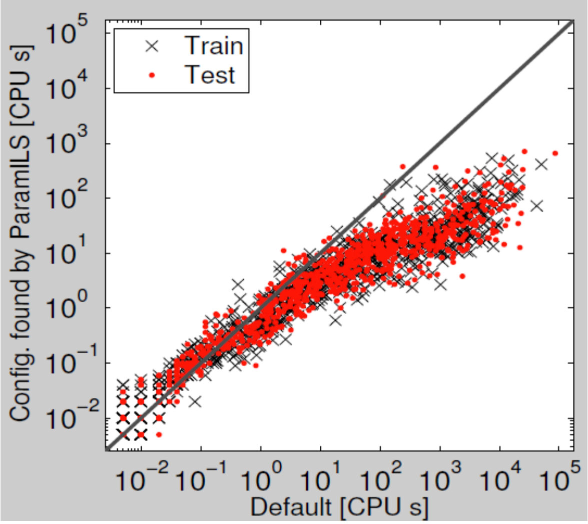
- Leading solver for the last 15 years
- Licensed by  over 1 000 universities and 1 300 corporations
- 76 parameters, $10^{47}$ configurations

## Minimizing runtime to optimal solution

- Speedup factor: 2× to 50×
- Later work: speedups up to 10,000×

## Minimizing optimality gap reached

- Gap reduction factor: 1.3× to 8.6×



# Comparison to CPLEX Tuning Tool

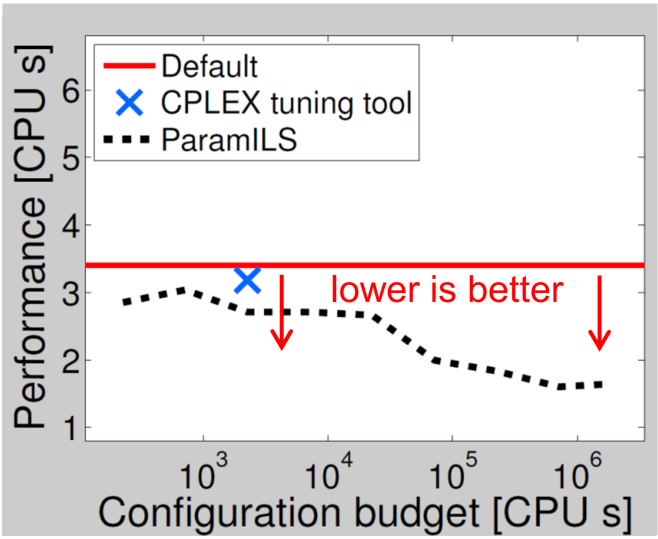
[Hutter, Hoos & Leyton-Brown, CPAIOR 2010]

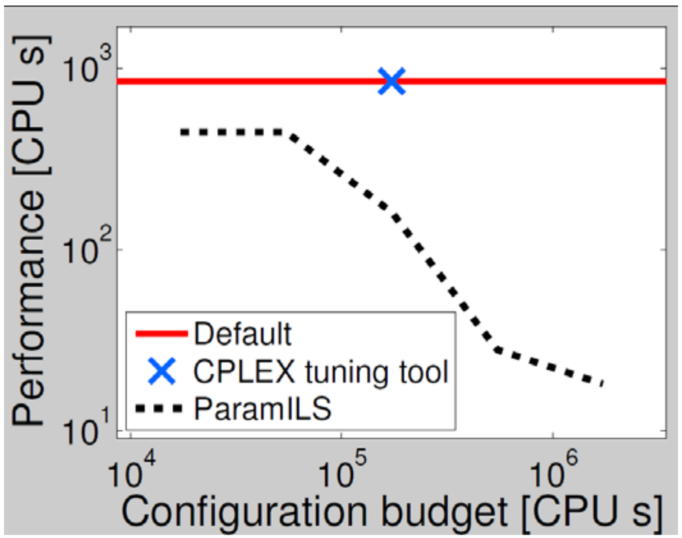
## CPLEX tuning tool

- 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




2-fold speedup
(our worst result)

50-fold speedup
(our best result)

# Machine Learning Application: Auto-WEKA

[Thornton, Hutter, Hoos & Leyton-Brown, KDD 2013]

**WEKA**: most widely used off-the-shelf machine learning package (>18,000 citations on Google Scholar)

**Different methods work best on different data sets**

– 30 base classifiers (with up to 8 parameters each)
– 14 meta-methods
– 3 ensemble methods
– 3 feature search methods & 8 feature evaluators
– Want a **true off-the-shelf solution:** Learn

# Machine Learning Application: Auto-WEKA

[Thornton, Hutter, Hoos & Leyton-Brown, KDD 2013]

- Combined model selection & hyperparameter optimization
  - All hyperparameters are conditional on their model being used
  - WEKA's configuration space: 786 parameters
  - Optimize cross-validation (CV) performance

- Results
  - SMAC yielded **best CV performance** on 19/21 data sets
  - **Best test performance** for most sets; especially in **8 largest**

- Auto-WEKA is online: http://www.cs.ubc.ca/labs/beta/Projects/autoweka/

# Applications of Algorithm Configuration

Mixed integer programming

Scheduling and Resource Allocation

Exam Timetabling since 2010

Spam filters

**Helped win Competitions**

SAT: since 2009

IPC: since 2011

Time-tabling: 2007

SMT: 2007

**Other Academic Applications**

Protein Folding

Game Theory: Kidney Exchange

Computer GO

Linear algebra subroutines

Evolutionary Algorithms

Machine Learning: Classification

Coffee Break

# Overview

- Programming by Optimization (PbO): Motivation and Introduction
- Algorithm Configuration
- **Portfolio-Based Algorithm Selection**
  - **SATzilla: a framework for algorithm selection**
  - **Comparing simple and complex algorithm selection methods**
  - **Evaluating component solver contributions**
  - **Hydra: automatic portfolio construction**
- Software Development Tools and Further Directions

# SATZILLA:
# A FRAMEWORK FOR ALGORITHM SELECTION

[Nudelman, Leyton-Brown, Andrew,  Gomes, McFadden, Selman, Shoham; 2003];
[Nudelman, Leyton-Brown, Devkar, Shoham, Hoos; 2004];
[Xu, Hutter, Hoos, Leyton-Brown; 2007, 2008, 2012]

all self-citations can be followed at http://cs.ubc.ca/~kevinlb

# SAT Solvers

What if I want to solve an NP-complete problem?

- **theory:** unless P=NP, some instances will be intractably hard
- **practice:** can do surprisingly well, but much care required

SAT is a useful testbed, on which researchers have worked to develop high-performance solvers for decades.

- There are **many high performance SAT solvers**
  - indeed, for years a biannual international competition has received >20 submissions in each of 9 categories

- However, **no solver is dominant**
  - different solvers work well on different problems
    - hence the different categories
  - even within a category, the best solver varies by instance

# Portfolio-Based Algorithm Selection

- We advocate building an **algorithm portfolio** to leverage the power of all available algorithms
  - indeed, an idea that has been floating around since Rice [1976]
  - lately, achieving top performance
- In particular, I'll describe **SATzilla**:
  - an algorithm portfolio constructed from all available state-of-the-art complete and incomplete SAT solvers
  - very successful in competitions
    - we've done much evaluation, but I'll focus on competition data
    - methods work beyond SAT, but I'll focus on that domain
  - in recent years, **many other portfolios in the same vein**
    - SATzilla embodies many of the core ideas that make them all successful

# Recently, many portfolios with strong practical performance

*[*]Algorithm Selection [†]Sequential Execution [‡]Parallel Execution*

- **Satisfiability:**
  - SATzilla[*†] [various coauthors, cited earlier; 2003—ongoing]
  - 3S[*†] [Sellmann, 2011]
  - ppfolio[‡] [Roussel, 2011]
  - claspfolio[*] [Gebser, Kaminski, Kaufmann, Schaub, Schneider, Ziller, 2011]
  - aspeed[†‡] [Kaminski, Hoos, Schaub, Schneider, 2012]
- **Constraint Satisfaction:**
  - CPHydra[*†] [O'Mahony, Hebrard, Holland, Nugent, O'Sullivan, 2008]
- **Planning:**
  - FD Stone Soup[†] [Helmert, Röger, Karpas, 2011]
- **Mixed Integer Programming:**
  - ISAC[*] [Kadioglu, Malitsky, Sellmann, Tierney, 2010]
  - MIPzilla[*†] [Xu, Hutter, Hoos, Leyton-Brown, 2011]
- **..and this is just the tip of the iceberg:**
  - http://dl.acm.org/citation.cfm?id=1456656 [Smith-Miles, 2008]
  - http://4c.ucc.ie/~larsko/assurvey [Kotthoff, 2012]

# SATzilla: Results from SAT Competitions

- **2003:** **first portfolio** entered in a SAT competition
  - requirement to submit only source code: a monstrous mess!
  - 2 silver, 1 bronze (out of 9 tracks, as below)
- **2004:** 2 bronze
- **2007:** **3 gold**, 1 silver, 1 bronze
- **2009:** 3 gold, 2 silver
- **2011:** Entered the **Evaluation Track** (more later)
- **2012:** SAT Challenge (strong performance; many portfolios entered)
- **2013:** Portfolios now a **victim of their own success**?
  - "The emphasis of SAT Competition 2013 is on evaluation of core solvers:" single-core portfolios of >2 solvers not eligible

# 2012 SAT Challenge: Application

| Rank | Solver | % solved | # solved |
|---|---|---|---|
| | **VBS** | **94.7** | **568** |
| 1 | SATzilla2012 APP | 88.5 | 531 |
| 2 | SATzilla2012 ALL | 85.8 | 515 |
| 3 | Industrial SAT Solver | 83.2 | 499 |
| 4 | interactSAT | 80.0 | 480 |
| 5 | glucose | 79.2 | 475 |
| 6 | SINN | 78.7 | 472 |
| 7 | ZENN | 78.0 | 468 |
| 8 | Lingeling | 77.8 | 467 |

* Interacting multi-engine solvers: like portfolios, but richer interaction between solvers

# 2012 SAT Challenge: Hard Combinatorial

| Rank | Solver | % solved | # solved |
|---|---|---|---|
| | **VBS** | **88.2** | **529** |
| 1 | SATzilla2012 COMB | 79.3 | 476 |
| 2 | SATzilla2012 ALL | 78.8 | 473 |
| 3 | ppfolio2012 | 70.3 | 422 |
| 4 | interactSAT_c | 79.5 | 417 |
| 5 | pfolioUZK | 66.8 | 401 |
| 6 | aspeed-crafted | 61.7 | 370 |
| 7 | clasp-crafted | 61.2 | 367 |
| 8 | claspfolio-crafted | 58.7 | 352 |

# SAT Challenge 2012: Random

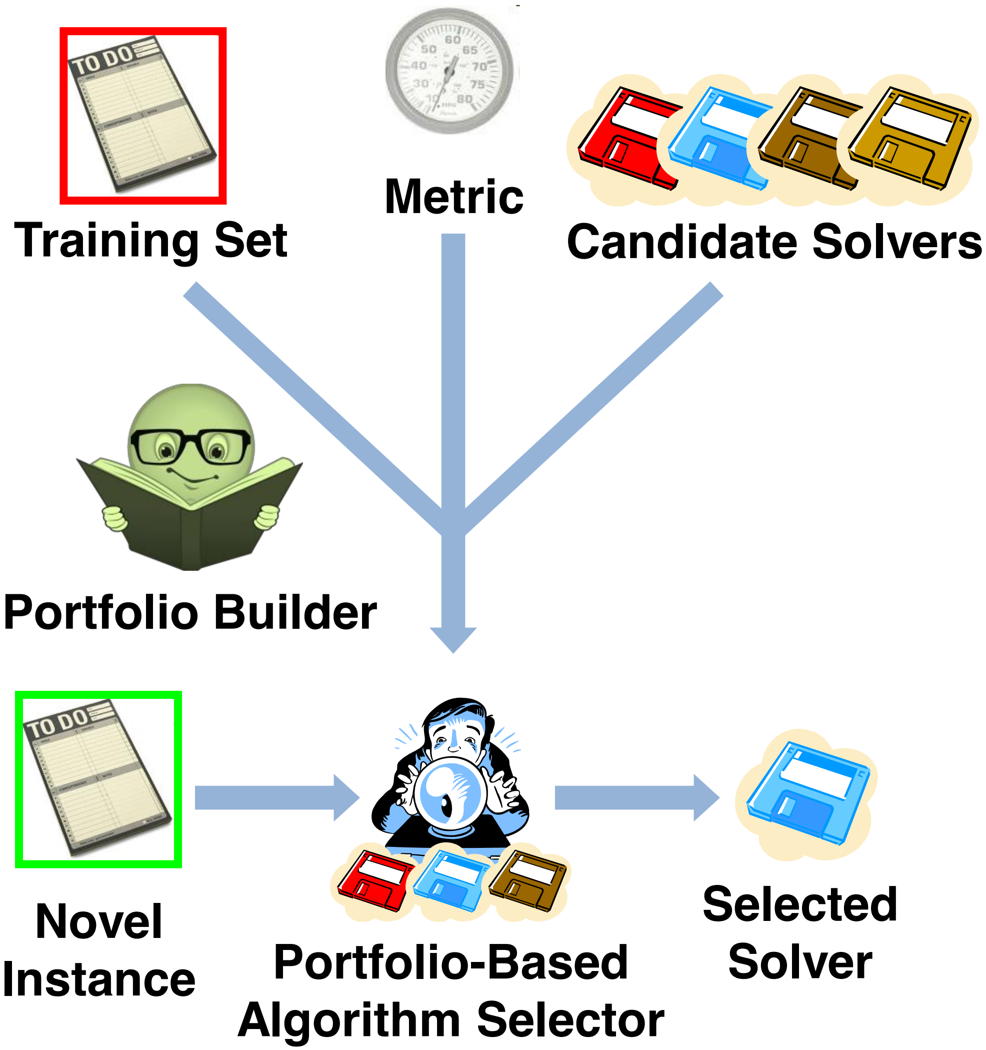
| Rank | Solver | % solved | # solved |
|---|---|---|---|
| | **VBS** | **93.0** | **558** |
| 1 | CCASat | 70.5 | 423 |
| 2 | SATzilla2012 RAND | 53.5 | 321 |
| 3 | SATzilla2012 ALL | 51.0 | 306 |
| 4 | sattime2012 | 44.8 | 269 |
| 5 | ppfolio2012 | 42.2 | 253 |
| 6 | pfolioUZK | 38.3 | 230 |
| 7 | ssa | 25.0 | 150 |
| 8 | gNovelty+PCL | 20.5 | 123 |

# 2012 SAT Challenge: Sequential Portfolio

| Rank | Solver | % solved | # solved |
|---|---|---|---|
| | **VBS** | **80.7** | **484** |
| 1 | SATzilla2012 ALL | 72.2 | 433 |
| 2 | ppfolio2012 | 61.7 | 370 |
| 3 | pfolioUZK | 60.3 | 362 |

- 3S **deserves mention**, though isn't compared here
  [Kadioglu, Malitsky, Sabharwal, Samulowitz, Sellmann, 2011]
  - Disqualified on a technicality
    - chose a buggy solver that returned an incorrect result
    - an occupational hazard for portfolios!
  - Overall performance **nearly as strong as SATzilla**

# SATzilla (stylized version)




- **Given:**
  - **training set of instances**
  - **performance metric**
  - **candidate solvers**
  - **portfolio builder** ***(incl. instance features)***
- Training:
  - collect performance data
  - learn a model for selecting among solvers
- At Runtime:
  - evaluate model
  - run selected solver

# SATzilla Methodology (offline)

1. Identify a target **instance distribution**
2. Select a set of candidate **solvers**
3. Identify a set of instance **features**

}  *SATzilla's input*

4. On a training set, **compute features and solver runtimes**
5. Identify a set of "**presolvers**" and a schedule for running them. Discard data that they can solve within a given cutoff time
6. Identify a "**backup solver**": the best on remaining data
7. **Learn models** for selecting among solvers from step (2)
8. Choose a subset of the solvers to **include in the portfolio**: those for which the portfolio obtained in step (7) has best performance on instances from a distinct validation set

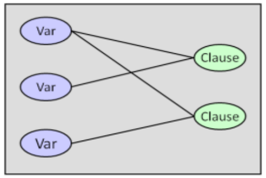
# SATzilla Methodology (online)

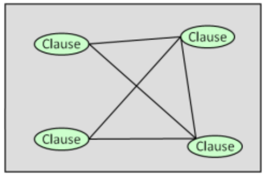
9. Sequentially **run each presolver** until its cutoff time
   - if the instance is solved, terminate
10. Compute **features**
    - if there's an error, run the backup solver
    - potentially, predict which features will be cheap and compute only them
11. **Evaluate models** to determine which solver to run
    - potentially, evaluate different models depending on which features were computed
12. **Run the selected algorithm**
    - if it crashes, etc., run the next-best algorithm

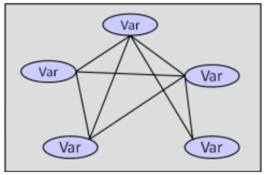
# SAT Instance Features (2003—2013)

Over 100 features. Some illustrative examples from SAT:

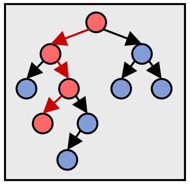
- Problem **Size** (clauses, variables, clauses/variables, …)
- **Syntactic** properties (e.g., positive/negative clause ratio)
- Statistics of various **constraint graphs**
  - factor graph
  - clause–clause graph
  - variable–variable graph



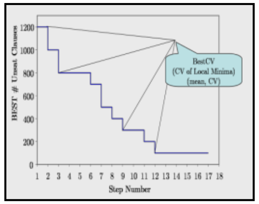



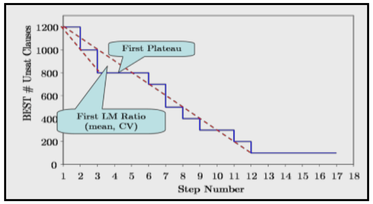



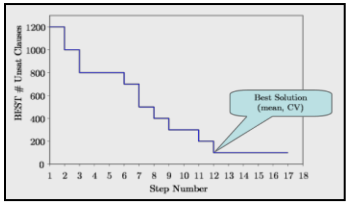

- Knuth's **search space size** estimate



- Cumulative number of **unit propagations** at different depths (SATz heuristic)
- **Local search probing**




- **Linear programming** relaxation

maximize: $\sum_{k \in C} \left( \sum_{i \in L,\, i \in k} v_i + \sum_{j \in \bar{L},\, j \in k} (1 - v_j) \right)$

subject to: $\sum_{i \in k,\, i \in L} v_i + \sum_{j \in k,\, j \in \bar{L}} (1 - v_j) \geq 1 \quad \forall k \in C$

$v_i \in \{0, 1\} \quad \forall i$

# Presolvers and Subset Selection

- **Presolvers**
  - Consider discrete set of exponentially increasing time amounts
  - For every choice of two presolvers + captimes for each, run the entire SATzilla pipeline and evaluate overall performance
  - Keep the choice that yields best performance

- **Subset selection**
  - Consider every subset of the given solver set
    - omitting a weak solver prevents models from accidentally choosing it
    - conditioned on choice of presolvers
    - computationally cheap: models decompose across solvers
  - Keep the subset that achieves the best performance

# How is SATzilla an example of PbO?

- SATzilla **builds a new meta-algorithm** out of a given set of existing solvers
- Two senses in which this involves automatically choosing among candidate algorithm designs via optimization:
  1. fitting the **machine learning models**, which govern the meta-algorithm's behavior
     - machine learning *is* optimization
  2. determining **properties of the meta-algorithm**:
     - pre-solver schedule
     - solver subset selection
     - backup solver

# Try it yourself!

- SATzilla is **freely available online**

  http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/

- You can try it for your problem
  - we have features for SAT, MIP and TSP
  - you need to provide features for other domains
    - in many cases, the general idea between our existing features
    - can also make features by reducing your problem to e.g. SAT and computing the SAT features

# COMPARING SIMPLE AND COMPLEX ALGORITHM SELECTION METHODS

[Xu, Hutter, Hoos, Leyton-Brown, ongoing work]

# Methods

How should SATzilla **choose among candidate solvers**?

- Runtime prediction
- Pairwise classification
- Cost-sensitive classification

Is this better than some **simple alternatives**?

- Best single solver
- Time slicing
- Sequential scheduling

Recall: the best we can hope for is the **virtual best solver**

- choose the best solver on a per-instance basis

# Methods: Runtime Prediction

- How it works
  - Build an "**empirical hardness model**" predicting the amount of time each solver will take to run on each instance
  - oddly enough, this is possible to do
- A **regression problem**:
  - linear regression
  - quadratic ridge regression
  - random forests of regression trees
- Evaluate the model for each solver, and choose the solver **predicted to be fastest**
  - advantage: implicitly penalizes big mispredictions more than small mispredictions (RMSE)
  - disadvantage: solves a harder problem than necessary
- *The method used by SATzilla 2003—2009*

# Methods: Pairwise Classification

- How it works:
  - Build a classifier to determine which algorithm to prefer between **each pair of algorithms** in the portfolio
  - Loss function: 0-1 error
- A **classification problem**:
  - support vector machines
  - decision forests
- Classifiers **vote for different algorithms**; the algorithm with the most votes is selected
  - Advantage: selection is a classification problem
  - Disadvantage: big and small errors treated the same
- *We tried this method back in 2003-4, opted against it*

# Methods: Cost Sensitive Classification

- How it works:
  - Build a classifier to determine which algorithm to prefer between each pair of algorithms in the portfolio
  - Loss function: **cost of misclassification**
- Both decision forests and support vector machines have **cost-sensitive variants**
- Classifiers vote for different algorithms; the algorithm with the most votes is selected
  - Advantage: selection is a classification problem
  - Advantage: big and small errors treated differently
- *The method used by SATzilla since 2011*

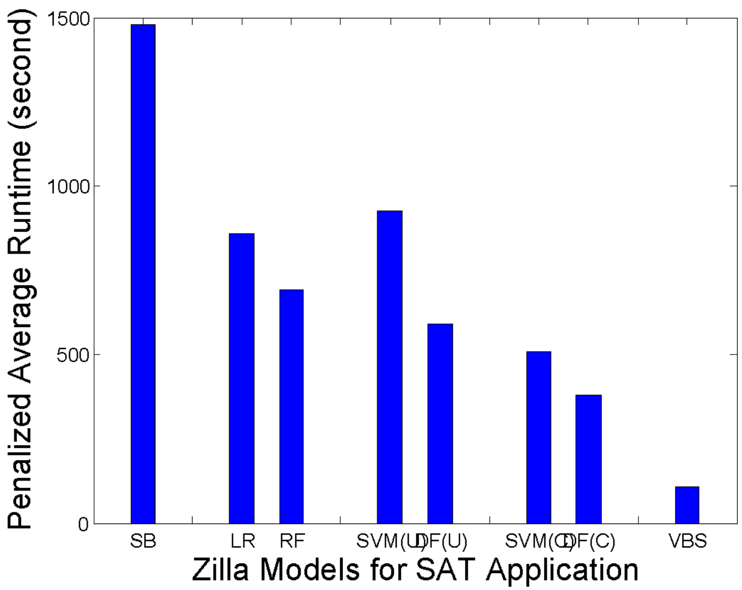
# Methods: Time Slicing (ppfolio)

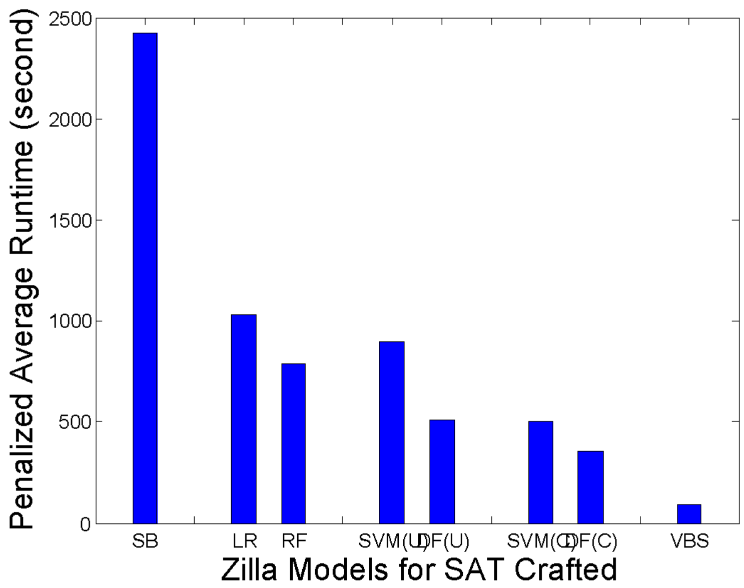
- Don't build a model
  - thus, **no features are needed**
- Run **all algorithms in parallel**
  - with one processor, time slicing
  - $k$ solvers: runtime is $k$ times minimum runtime across solvers on every given instance
- **Solver selection**: keep the set of $k$ solvers that maximizes a performance metric on a training set
  - we approximated this optimization greedily

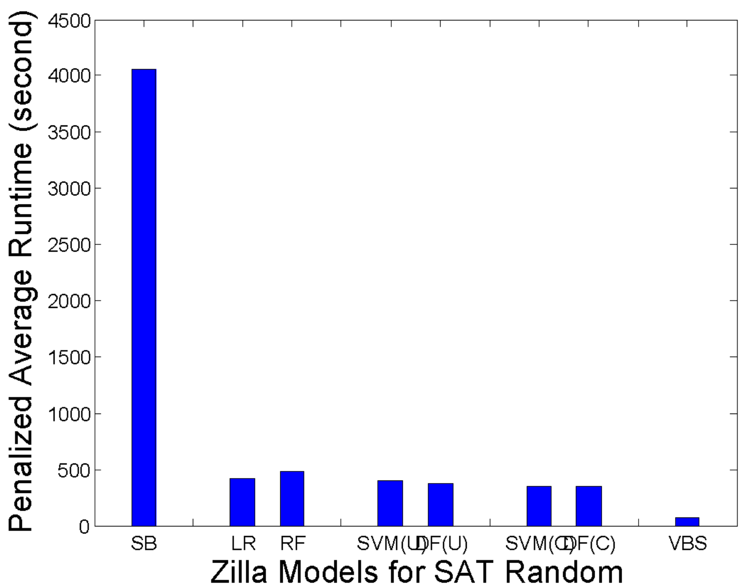
# Methods: Simple Sequential Portfolios

- Pick a **sequence of solvers and time budgets**
- What we did:
  - For **every permutation** of 4 solvers from the 7 candidate solvers that constitute the best VBS in terms of PAR10, consider all assignments of solvers to time budgets having total length $\leq \mathrm{T}$ and calculate out their performance
  - budgets: $\{0, 10^{0\mathrm{t}}, 10^{\mathrm{t}}, 10^{2\mathrm{t}}, \dots, 10^{30\mathrm{t}}\}$, $\mathrm{t} = \log_{10}\left(\frac{\mathrm{captime}}{30}\right)$
  - Add a 5th solver to the **end of the sequence**:
    - Pick the solver that achieves the best performance on the remaining unsolved instances within the remaining time
    - Set the time budget to be the remaining time

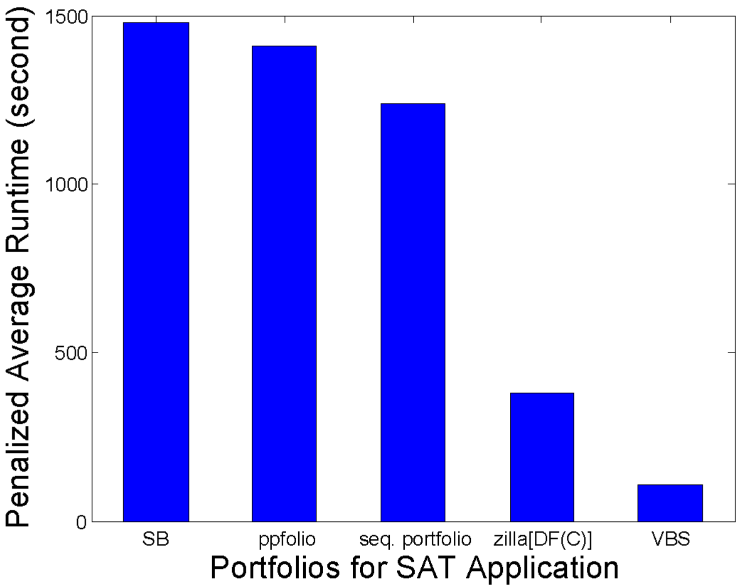
# SAT: SATzilla Variants

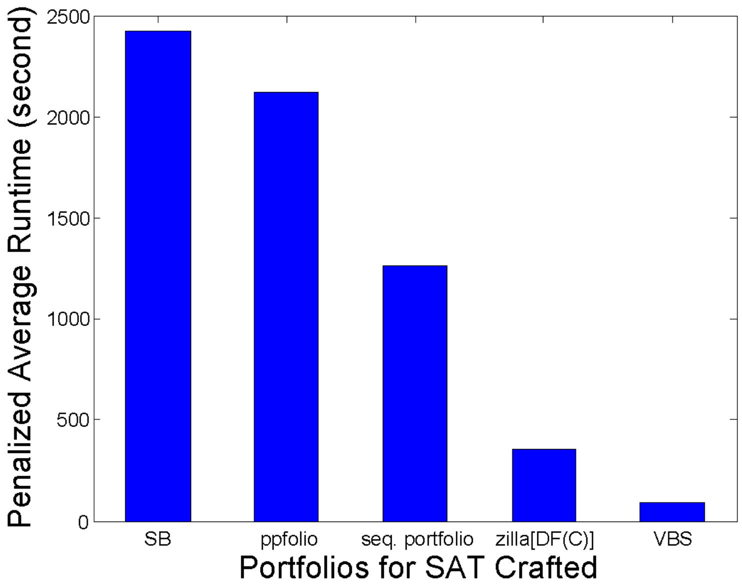


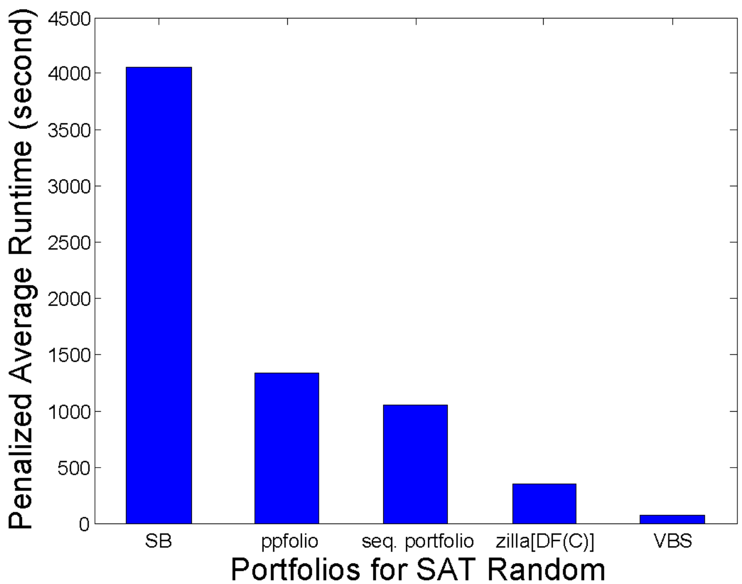



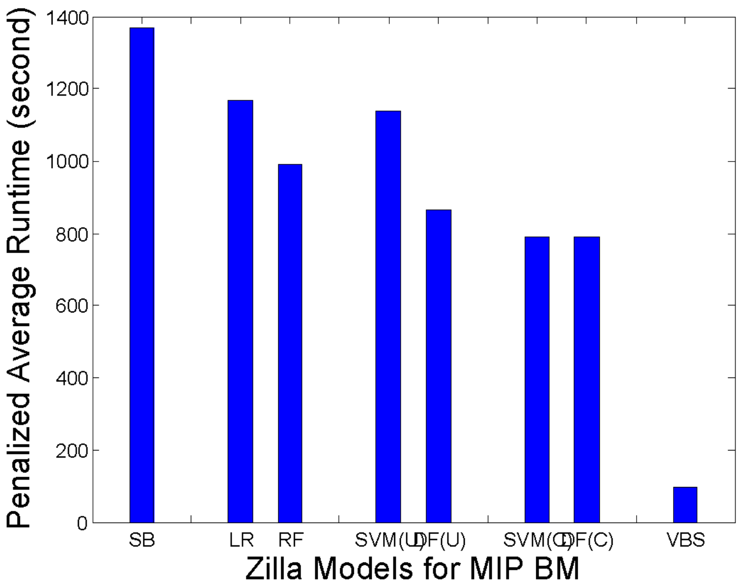
# SAT: SATzilla vs Baselines

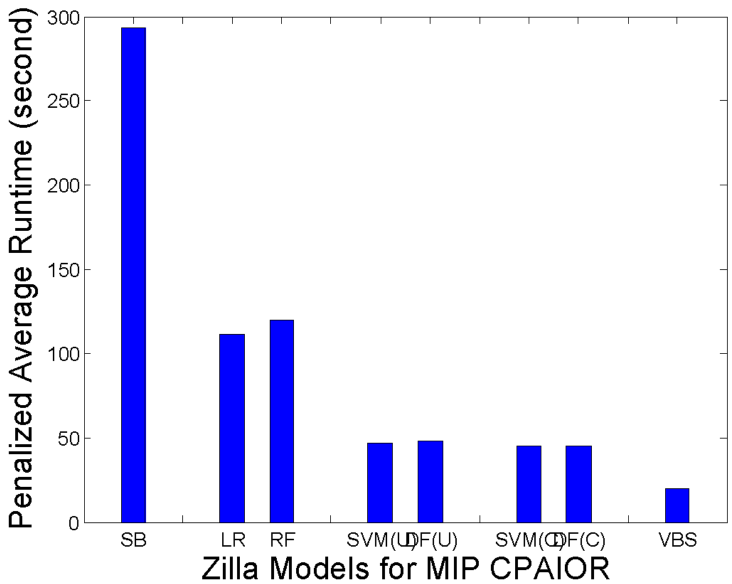


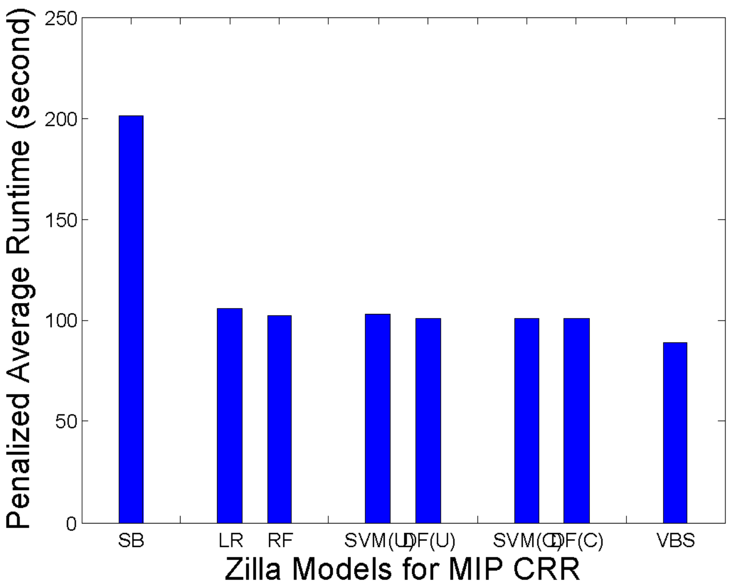



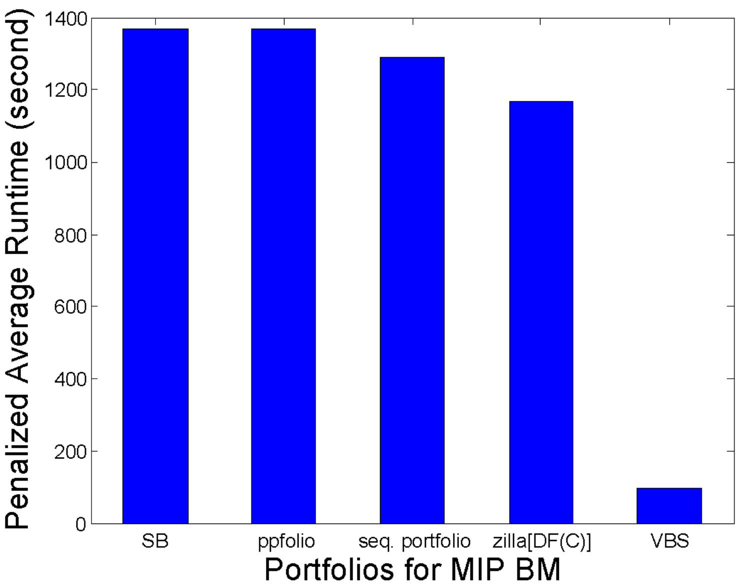
# MIP: MIPzilla Variants

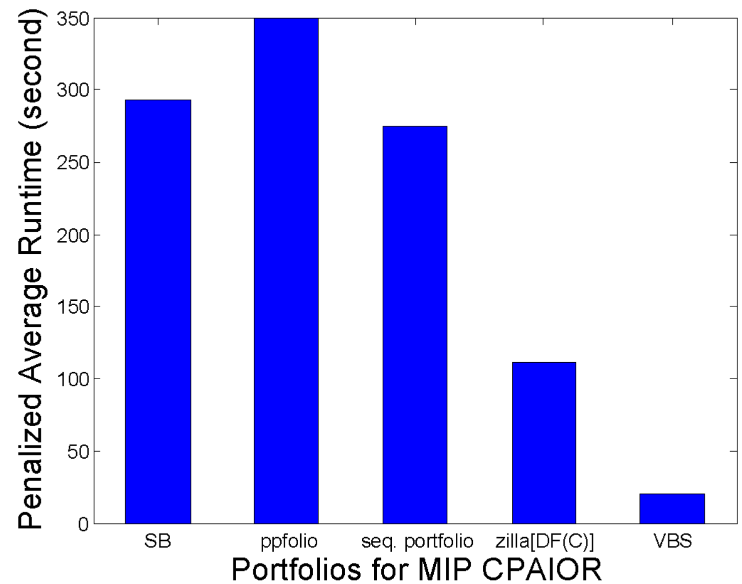


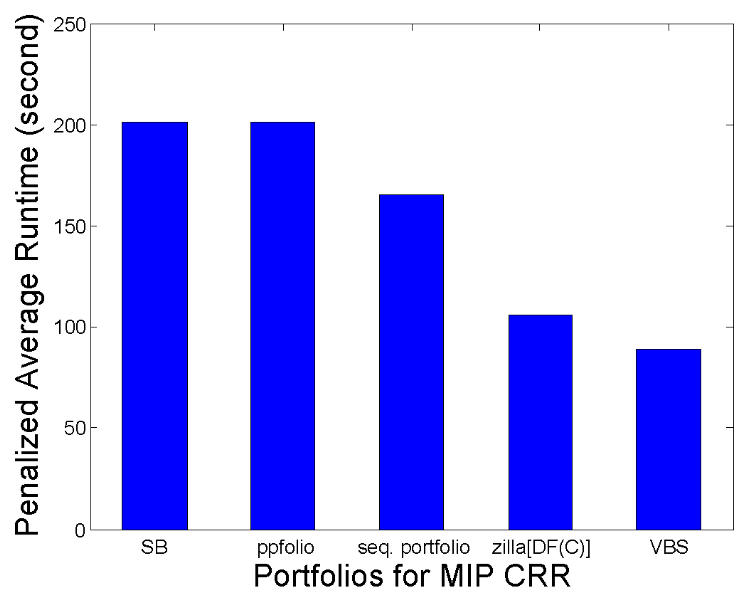



# MIP: MIPzilla vs Baselines



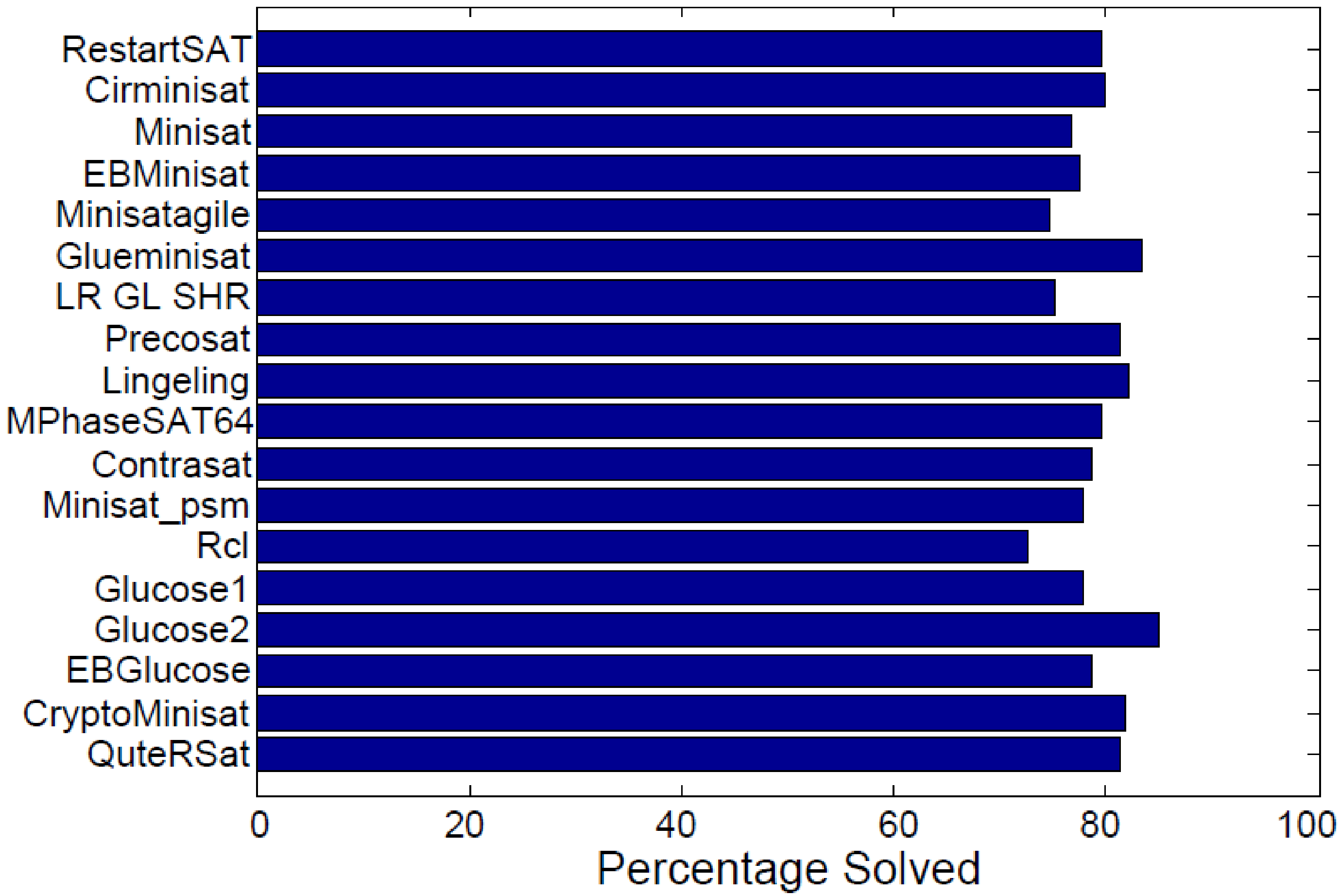
# EVALUATING COMPONENT SOLVER CONTRIBUTIONS

[Xu, Hutter, Hoos, Leyton-Brown, 2012]

# Evaluation Track for SAT Competition 2011

- Goal: use portfolios to **study the solvers submitted** to the 2011 SAT Competition
  - We considered all instances from 2011 SAT Competition: 300 Application; 300 Crafted; 300 Random
- Candidate solvers from 2011 SAT Competition:
  - for building SATzilla:
    - all sequential, non-portfolio solvers from Phase 2:
    - 18 Application; 15 Crafted; 9 Random
  - for determining VBS and SBS:
    - all solvers from Phase 2 of competition:
    - 31 Application; 25 Crafted; 17 Random
- How should we **assess the value of a solver**?
  - One option: look at its overall performance

# Performance of Individual Solvers (Application)




5000 CPU sec cutoff

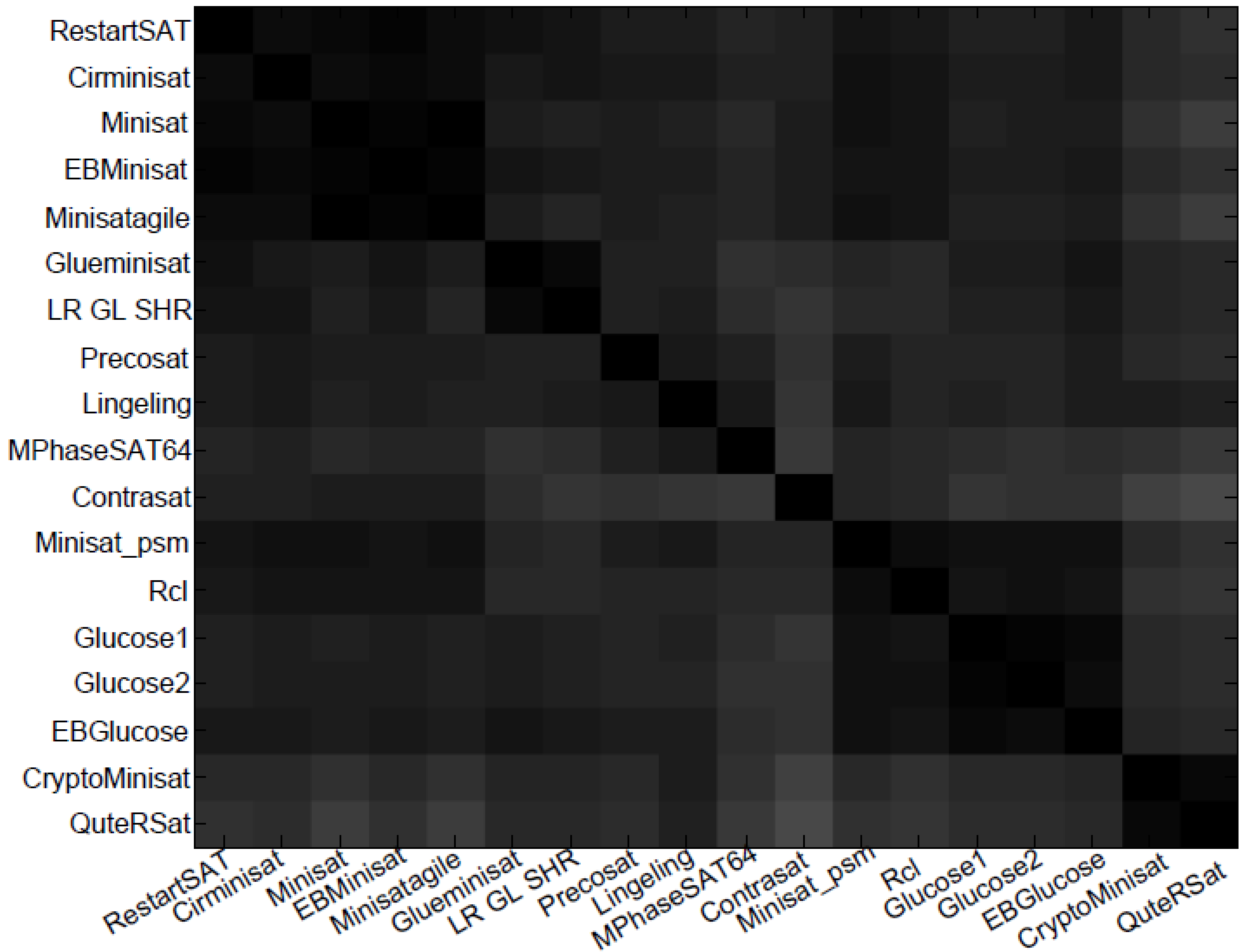
# Assessing Solver Quality

- How should we assess the value of a solver?
  - One option: look at its overall performance
- However, **portfolio-based methods** consistently outperform individual solvers, and so arguably represent the current state of the art

| *Solver* | *Application* | *Crafted* | *Random* |
|---|---|---|---|
| VBS | 84.7% | 76.3% | 82.2% |
| SATzilla 2011 | 75.3% | 66.0% | 80.8% |
| SATzilla 2009 | 70.3% | 63.0% | 80.3% |
| Gold medalist (SBS) | 71.7% | 54.3% | 68.0% |

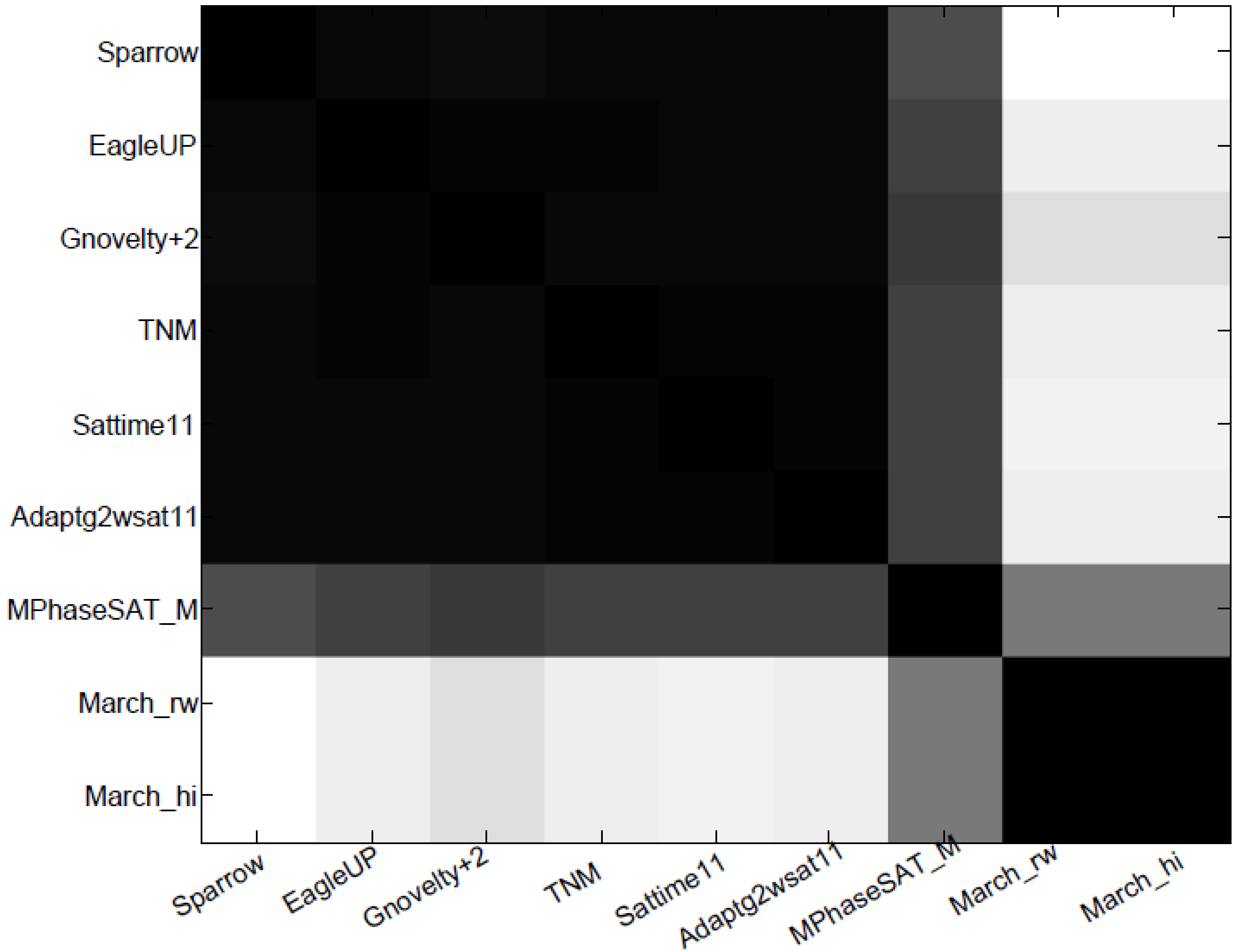
- The success of a portfolio-based solver ultimately depends on the **strength of its component solvers**
- How should we assess component solvers' **contributions to a portfolio**?
  1. their degree of **correlation**

# Correlation of Solver Performance (Application)




darker = higher Spearman correlation coefficient

# Correlation of Solver Performance (Random)

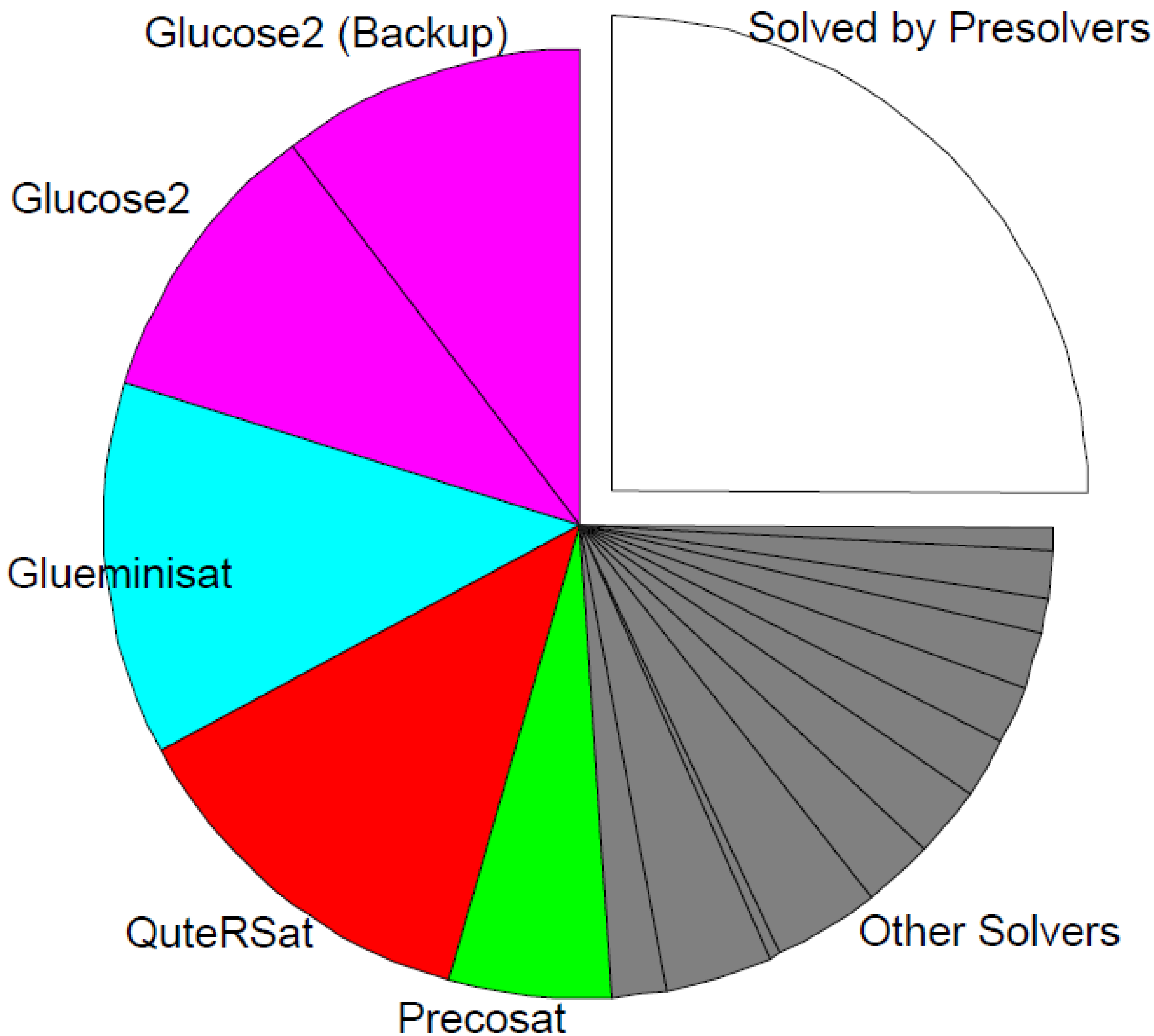



darker = higher Spearman correlation coefficient

# Assessing Solver Contributions

- The success of a portfolio-based solver ultimately depends on the **strength of its component solvers**
- How should we assess component solvers' **contributions to a portfolio**?
    1. their degree of **correlation**
    2. the **frequency with which they are selected** by the portfolio

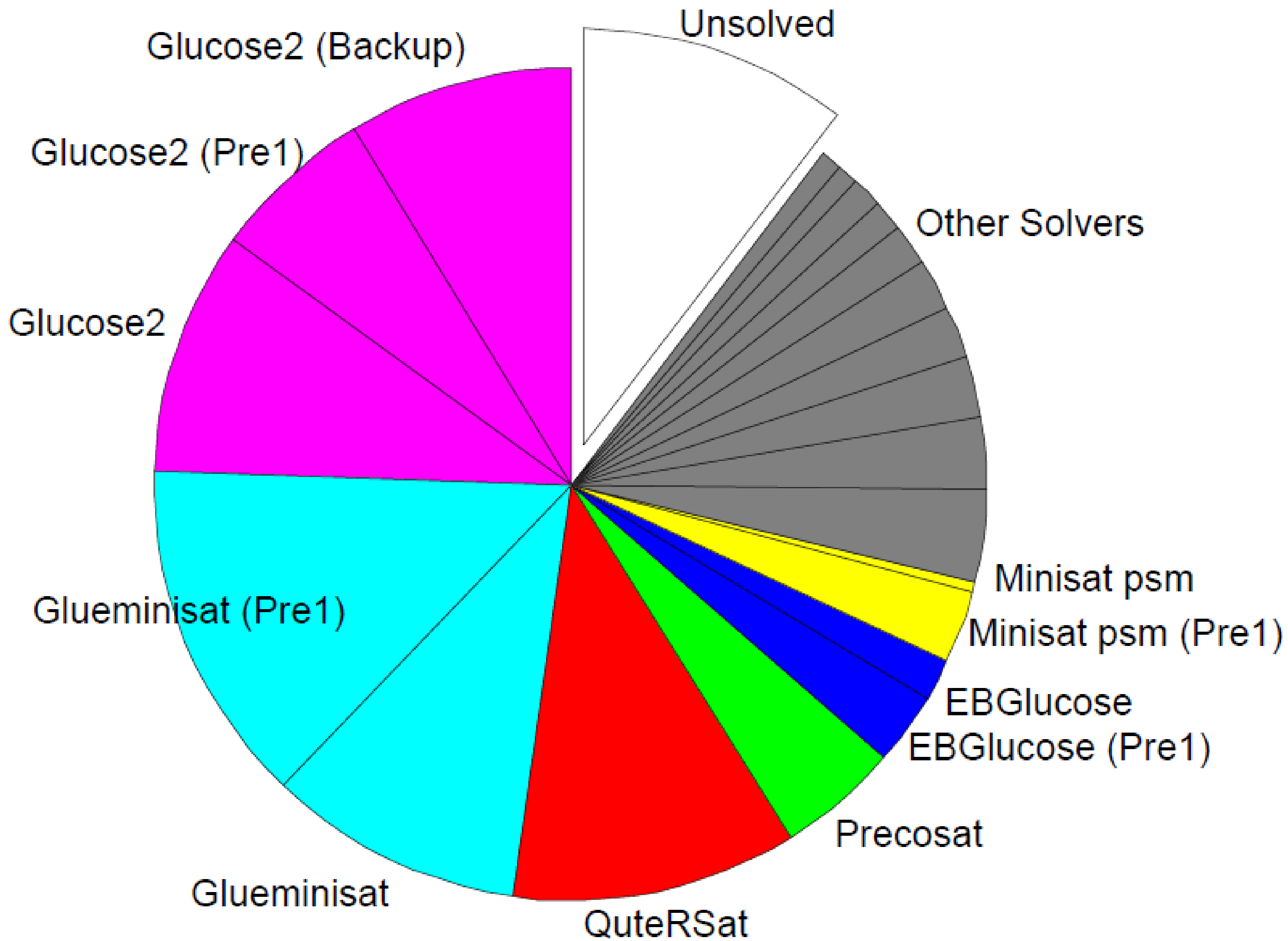
# Selection Frequency in SATzilla2011 (Application)



# Assessing Solver Contributions

- The success of a portfolio-based solver ultimately depends on the **strength of its component solvers**
- How should we assess component solvers' **contributions to a portfolio**?
  1. their degree of **correlation**
  2. the **frequency with which they are selected** by the portfolio
  3. the **fraction of instances** they're responsible for solving

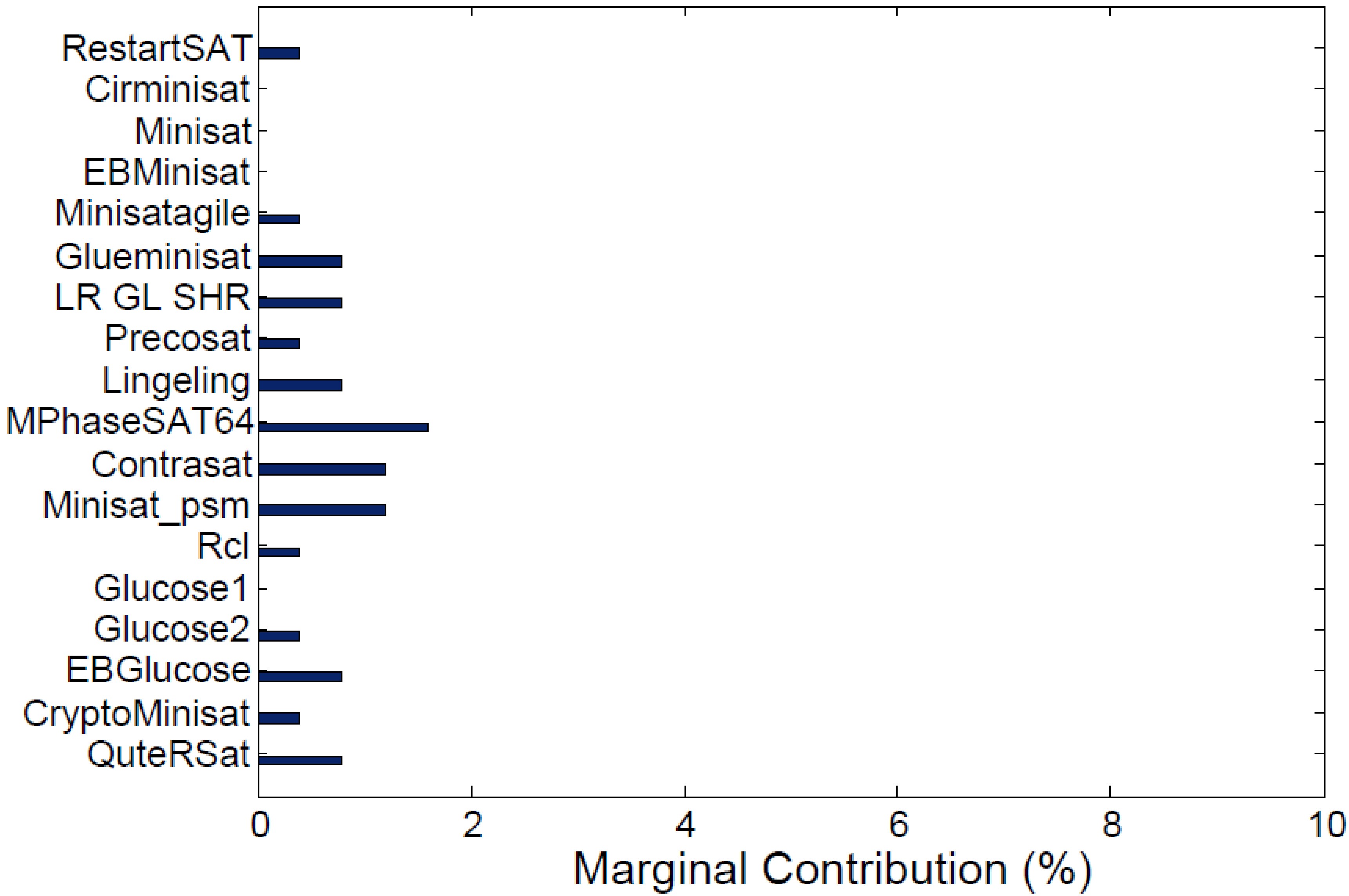
# Instances Solved by SATzilla2011 Components (Application)



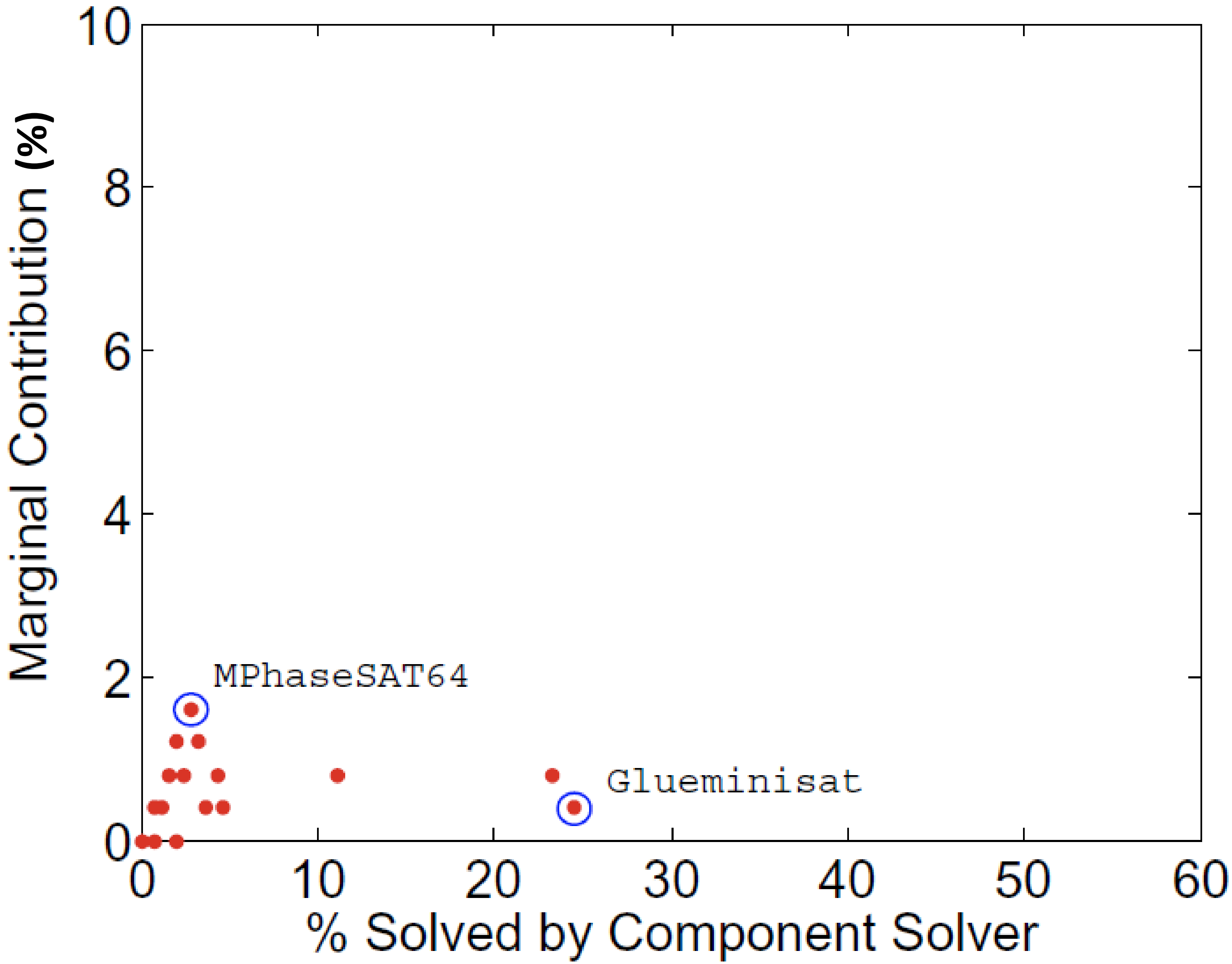
# Assessing Solver Contributions

- The success of a portfolio-based solver ultimately depends on the **strength of its component solvers**
- How should we assess component solvers' **contributions to a portfolio**?
  1. their level of **correlation**
  2. the **frequency with which they are selected** by the portfolio
  3. the **fraction of instances** they're responsible for solving
  4. their **marginal contribution to portfolio performance**

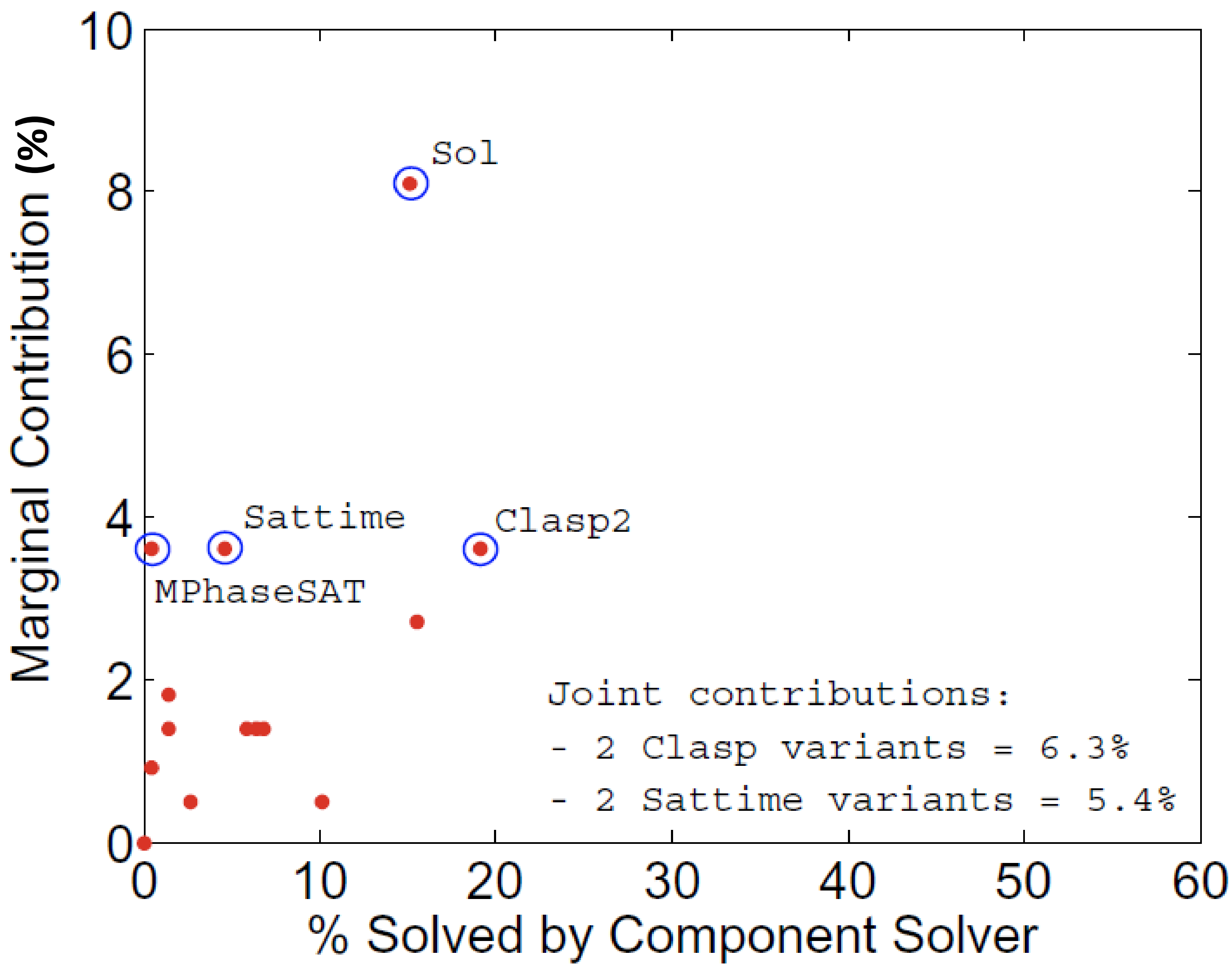
# Marginal Contribution of Components (Application)



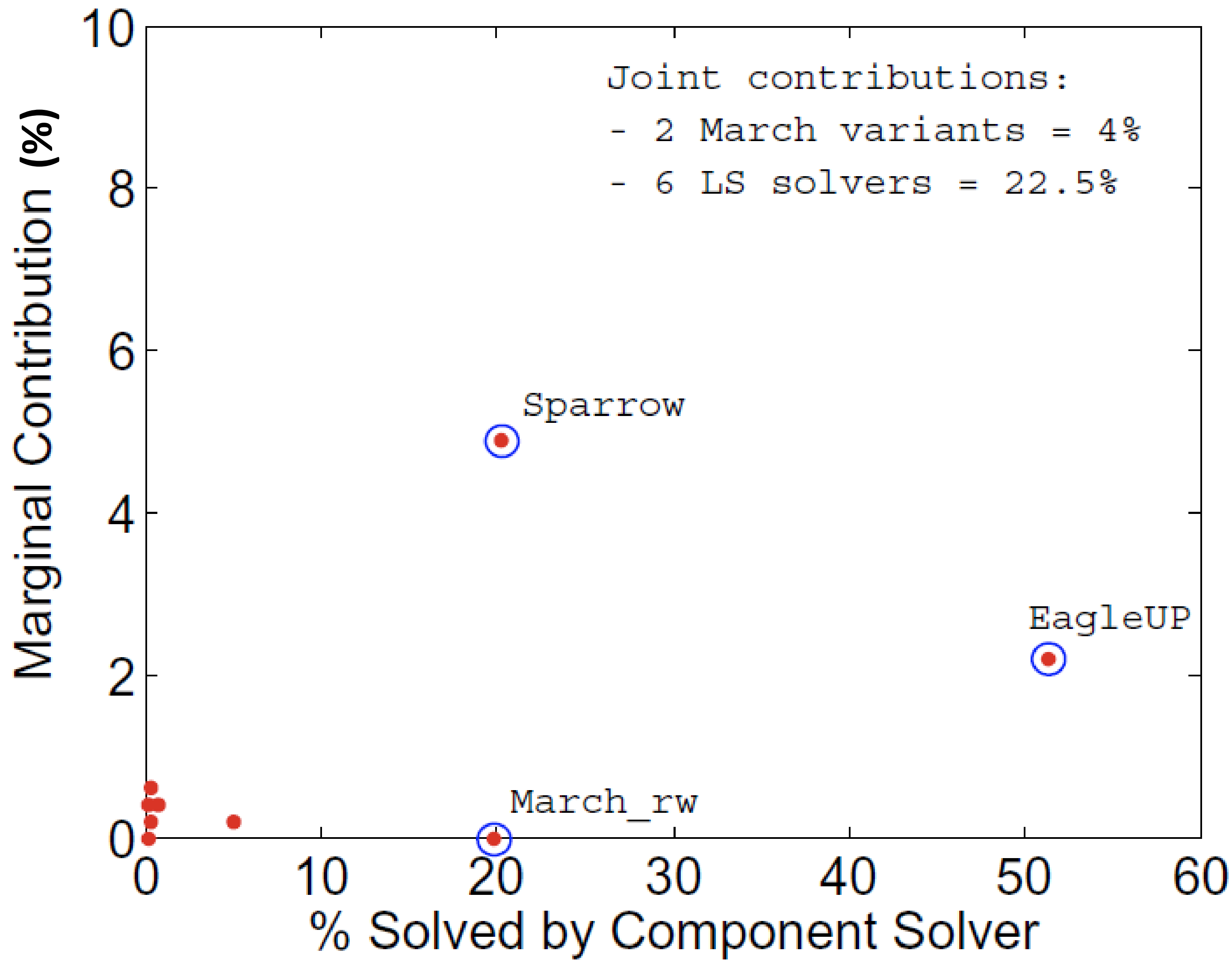
# Instances Solved vs Marginal Contribution (Application)



# Instances Solved vs Marginal Contribution (Crafted)



# Instances Solved vs Marginal Contribution (Random)



# HYDRA: AUTOMATIC PORTFOLIO CONSTRUCTION

[Leyton-Brown, Nudelman, Andrew, McFadden, Shoham, 2003];
[Leyton-Brown, Nudelman, Shoham, 2009]
[KhudaBukhsh, Xu, Hoos, Leyton-Brown, 2009]
[Xu, Hoos, Leyton-Brown, 2010]
[Xu, Hutter, Hoos, Leyton-Brown, 2011]

# Motivation

- What about situations where we **don't start out with a set of strong solvers** to choose among?

- Solution: take a PbO approach to identifying a set of solvers that **will work together well as a portfolio**, rather than just a single solver!
  - combines **algorithm configuration** with algorithm selection
  - design space now includes lots of new choices:
    - number of solvers to include in the portfolio
    - the design of each solver
  - **PbO: make these choices via automated optimization**

# SATenstein

- **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
- A **highly parameterized, generalized SLS solver** built using UBCSAT [Tompkins & Hoos, 2004]
  - 3 categories of SLS algorithms
    - WalkSAT
    - $G^2$WSAT
    - dynamic local search algorithms
  - can instantiate 25 known algorithms
  - 41 parameters, > $10^{11}$ possible instantiations

# How does SATenstein work?

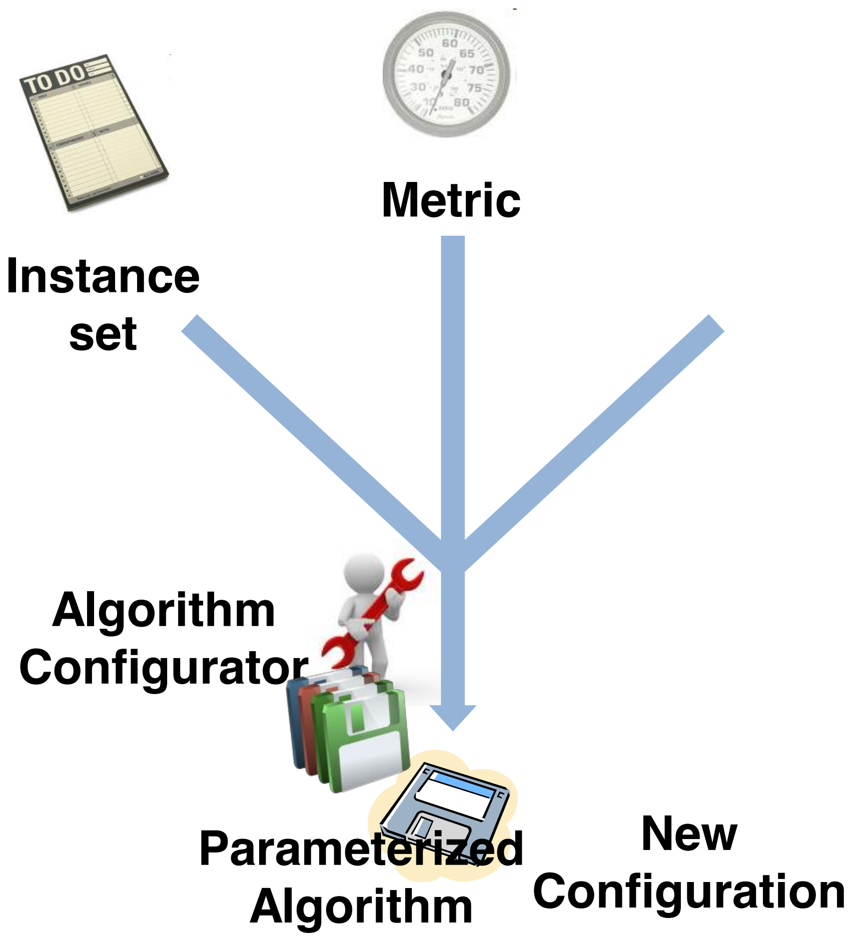
**Existing Algorithm Components**

**Domain Expert**

**Parameterized Algorithm**

- **Designer creates highly-parameterized algorithm from existing components**
- Given:
  - training set of instances
  - performance metric
  - parameterized algorithm
  - algorithm configurator
- Configure algorithm:
  - run configurator on training instances
  - output is a configuration that optimizes metric

# How does SATenstein work?

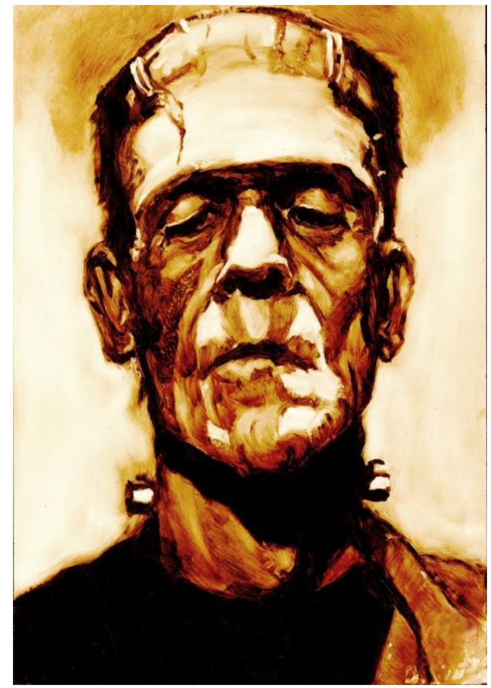



- Designer creates highly-parameterized algorithm from existing components
- Given:
  - training set of instances
  - performance metric
  - parameterized algorithm
  - algorithm configurator
- Configure algorithm:
  - run configurator on training instances
  - output is a configuration that optimizes metric

# SATenstein

**SATzilla**
*portfolio-based algorithm selection*

**SATenstein**
*algorithm design via automatic configuration*



# Advantages and Disadvantages


**Exploit per-instance variation between solvers using learned runtime models**

- **practical:** e.g., won competition medals
- **fully automated:** requires only cluster time rather than human design effort

**Key drawback:**

- requires a set of **strong, relatively uncorrelated** candidate solvers
- **can't be applied** in domains for which such solvers do not exist

# Advantages and Disadvantages

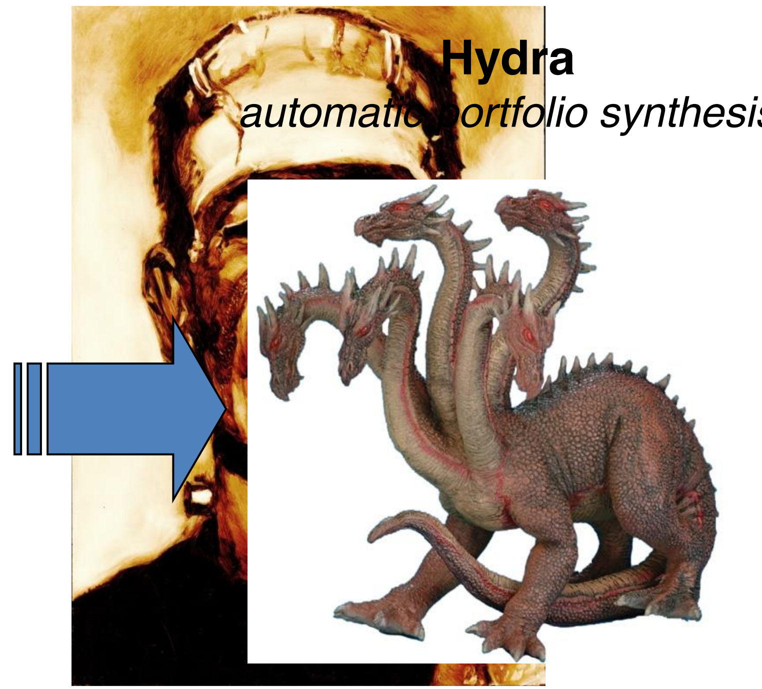
- Instead of manually exploring a design space, build a **highly parameterized algorithm** and then configure it automatically
  - as we've suggested earlier in the tutorial
- Can find **powerful, novel designs**
- But: **only produces single algorithms** designed to perform well on the entire training set

**SATenstein**

[KhudaBukhsh, Xu, Hoos, Leyton-Brown, 2009]

*algorithm design via automatic configuration*

# Hydra




Starting from a **single parameterized algorithm**, automatically find a set of **uncorrelated configurations** that can be used to build a **strong portfolio.**

# Hydra: Methodology

- Idea: augment an additional portfolio *P* by targeting instances **on which *P* performs poorly**
  - original idea: "boosting as a metaphor for algorithm design"
    [Leyton-Brown, Nudelman, Andrew, McFadden, Shoham, 2003];
    [Leyton-Brown, Nudelman, Shoham, 2009]
  - problem: the original algorithm could easily stagnate
    - indeed, same problem if you misunderstood Hydra as presented in the previous tutorial

- Avoid **stagnation** via a dynamic performance metric:
  - return performance of *s* when *s* outperforms *P*
  - return performance of *P* otherwise
- Intuitively: *s* is scored for its **marginal contribution** to *P*

- This metric is given to an off-the-shelf configurator, which optimizes it to find a new configuration *s**

- Thus, we retain the **same core idea as "boosting":**
  - build a new algorithm that explicitly aims to **improve upon an existing portfolio**

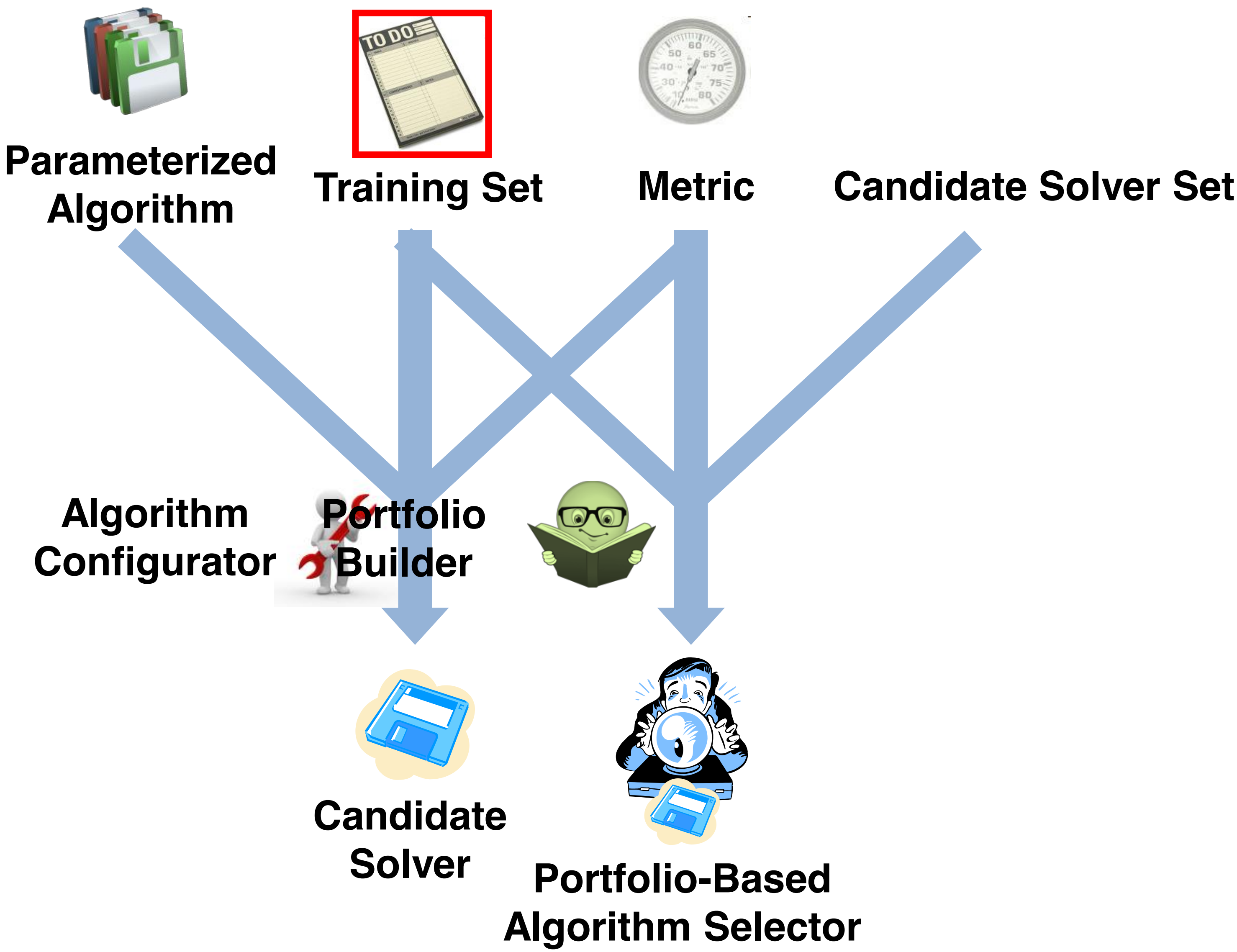
# Related Idea: ISAC

**ISAC: Instance Specific Algorithm Configuration**

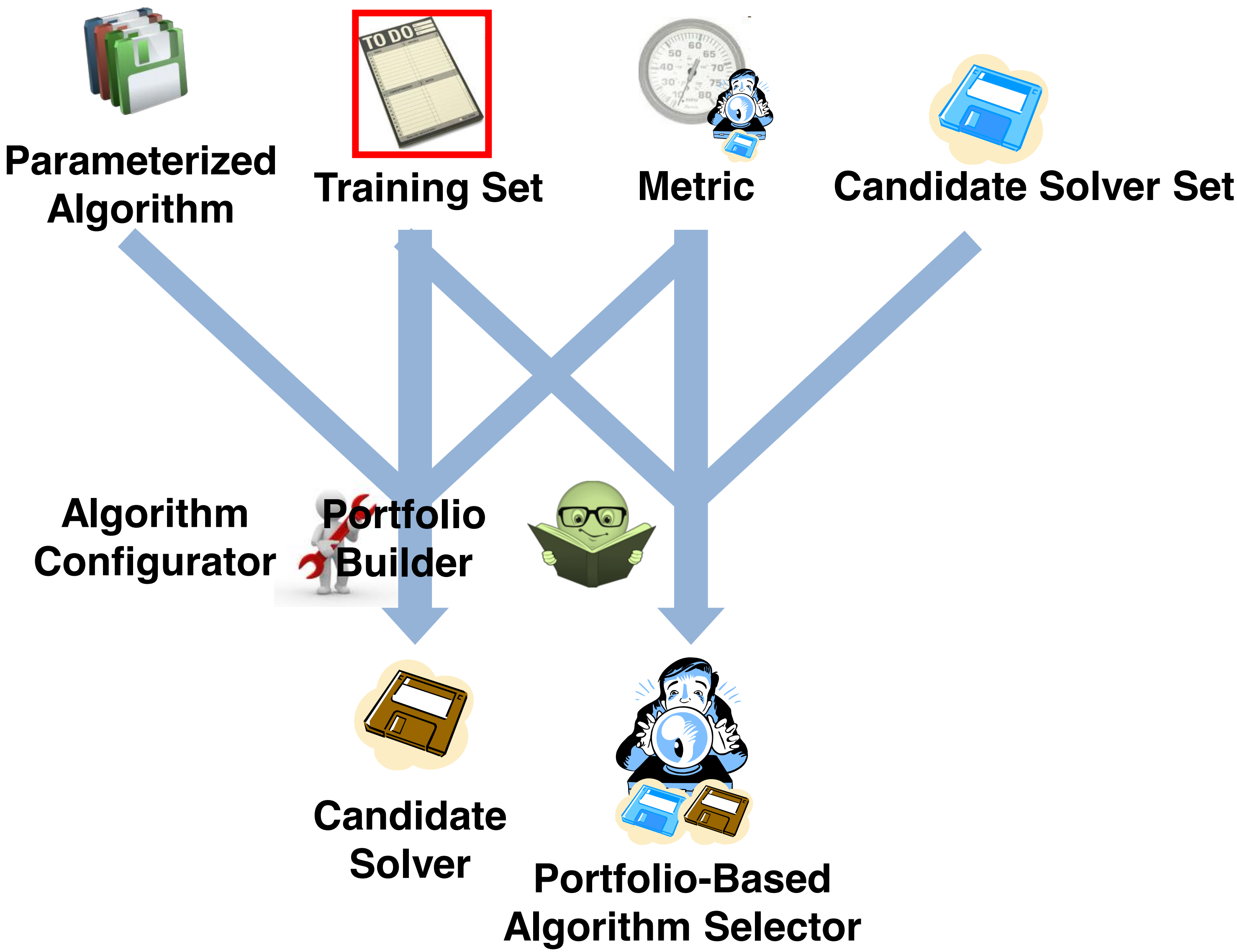
[Kadioglu, Malitsky, Sellmann, Tierney, 2010; Malitky, Sellman, 2012]

- How it works:
  - Compute features for training instances
  - **Cluster training instances** (using, e.g., k-means)
  - Configure a solver **for each cluster** of instances
  - At runtime, find the cluster whose center is closest to the features of the test instance, and run that solver
- Advantage: training **decomposes very nicely**
- Disadvantage: instance similarity may not correlate closely with runtime
  - thus solvers aren't **explicitly forced to be uncorrelated**
  - problem gets worse with uninformative features

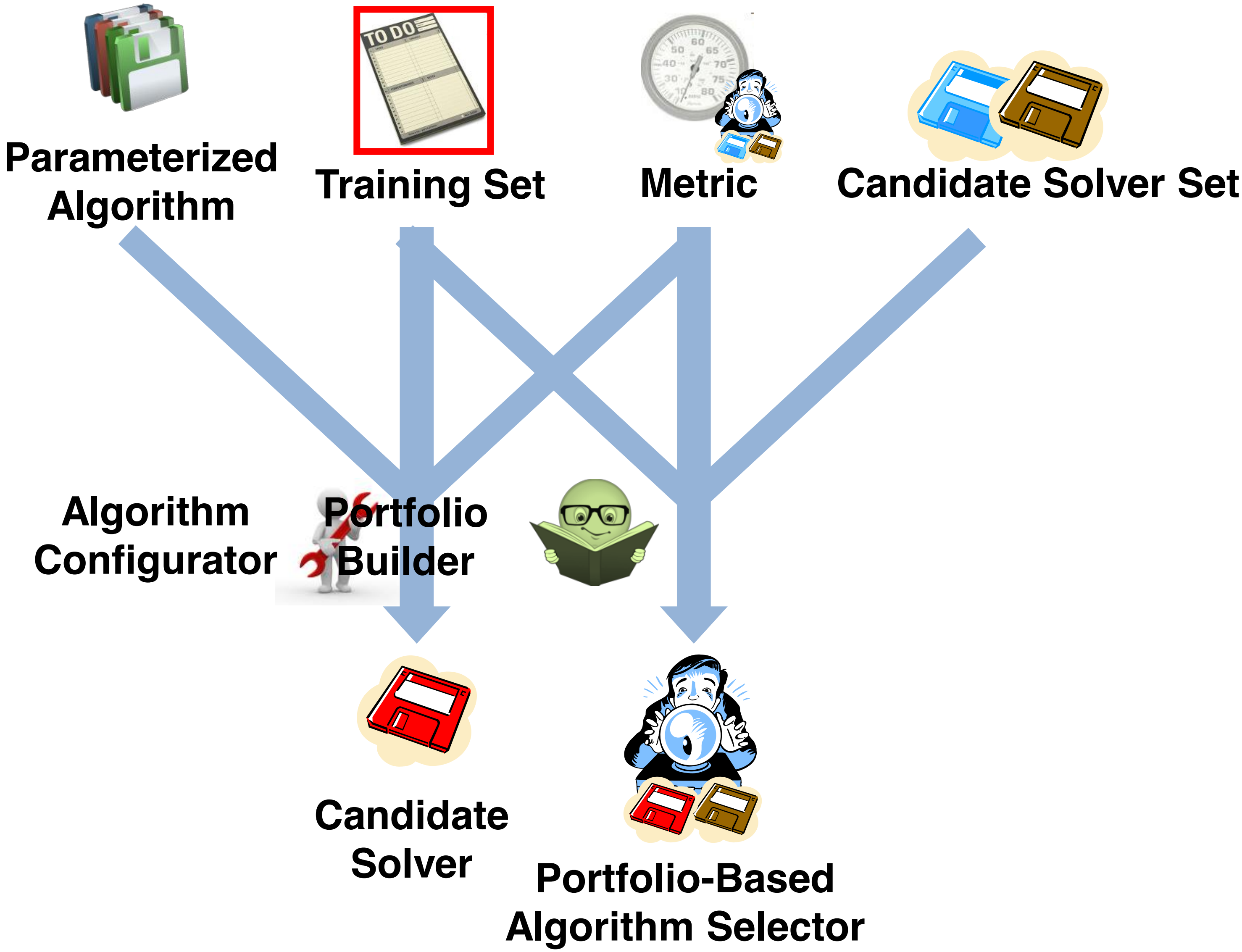
# Hydra Procedure: Iteration 1



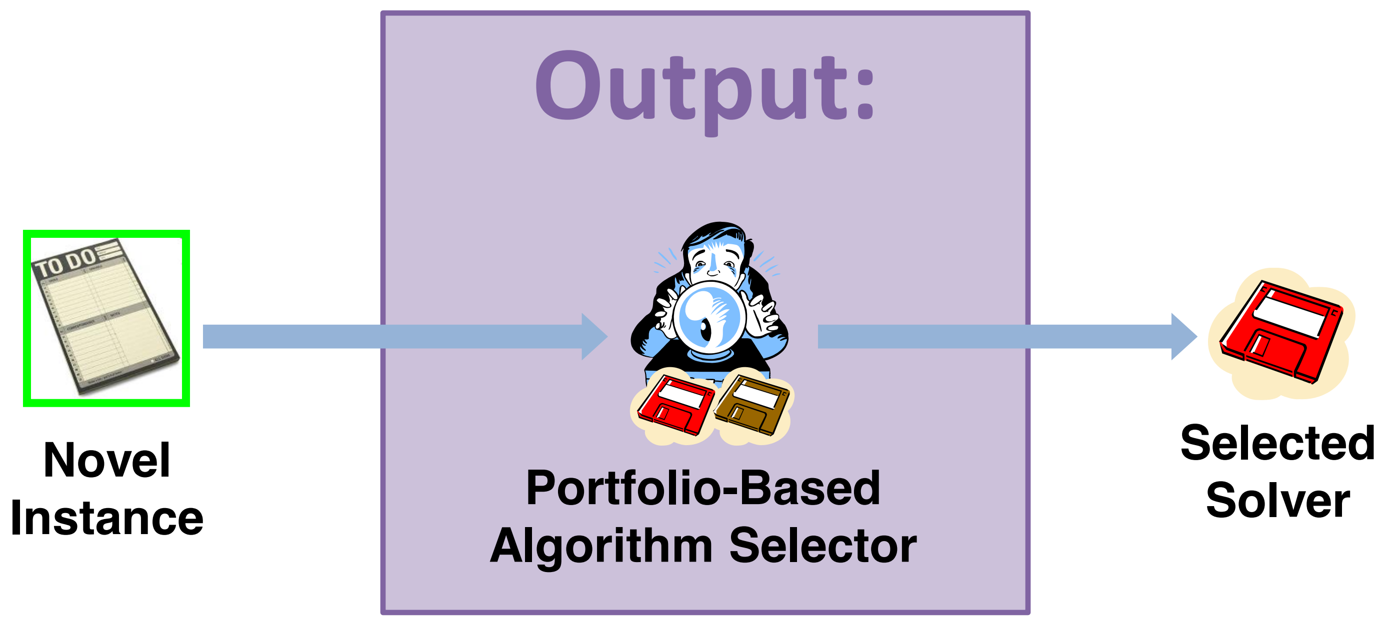
# Hydra Procedure: Iteration 2



# Hydra Procedure: Iteration 3



# Hydra Procedure: After Termination



# Another Interpretation

- Hydra can also be understood as a procedure for **building parallel algorithm portfolios**
  - obtain the min runtime across a set of solvers by **running all of them in parallel** rather than selecting only one of them
    - disadvantage: **wasted computation** on all but one core
    - advantage: automatic method for **parallelization**
    - advantage: **no need for features**
  - exactly the **same procedure** as before

# Experimental Evaluation

- Even though Hydra is **most useful in other domains**, I'll describe an **evaluation on SAT**.
- High bar for comparison
  - – strong state-of-the-art solvers
  - – portfolio-based solvers already successful
  - $\Rightarrow$ to be able to argue that Hydra does well, **we want to compare to a strong portfolio**
- Pragmatic benefits
  - – a wide variety of interesting datasets
  - – existing instance features
  - – SATenstein is a suitable configuration target

# Experimental Setup: Challengers

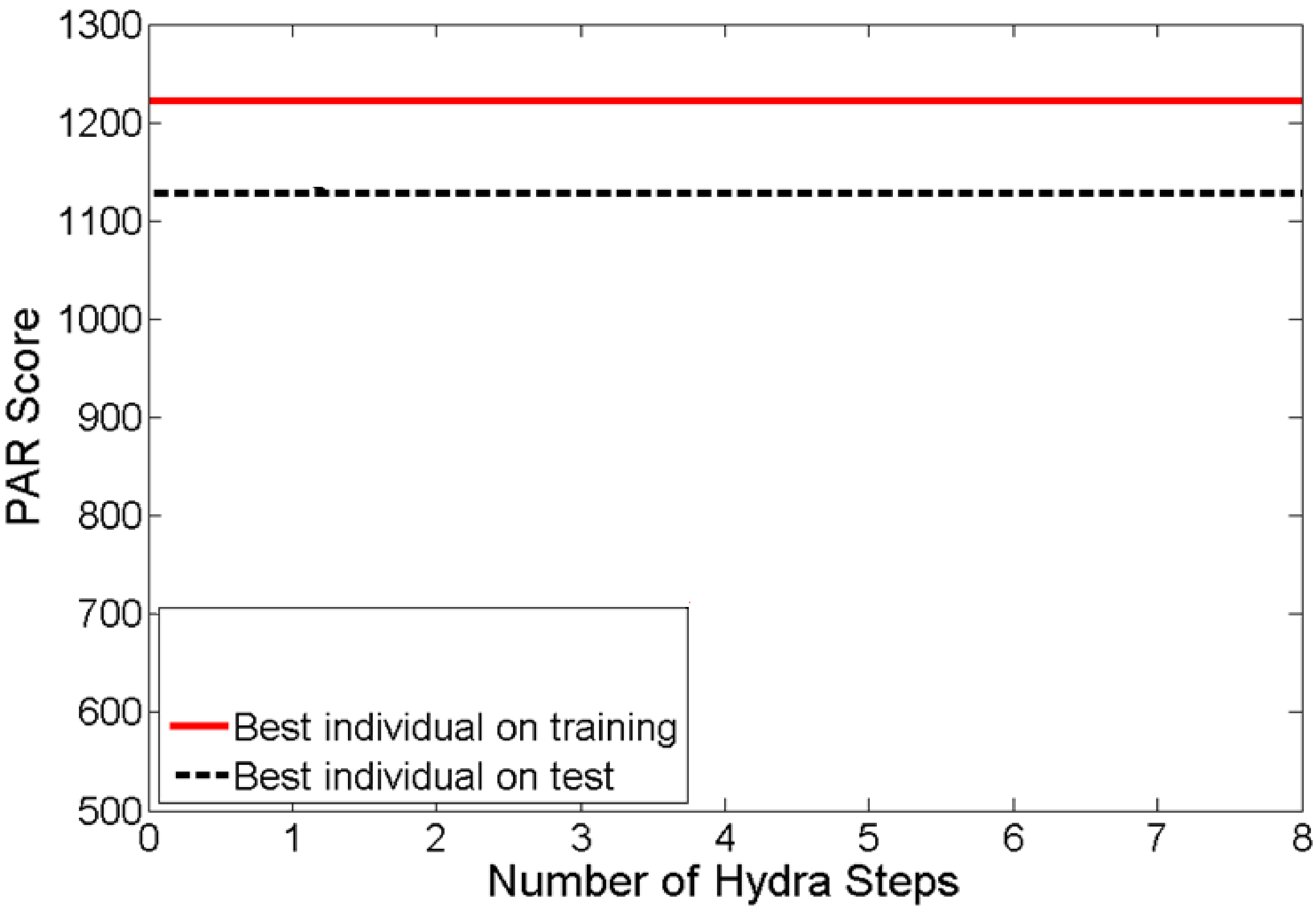
- Individual state-of-the-art solvers
  - 11 **manually-crafted** SLS solvers
    - all 7 SLS winners of any SAT competition 2002 – 2007
    - 4 other prominent solvers
  - 6 **SATenstein solvers** tuned for particular distributions
- Also considered **SATzilla portfolios of challengers**

# Performance Summary

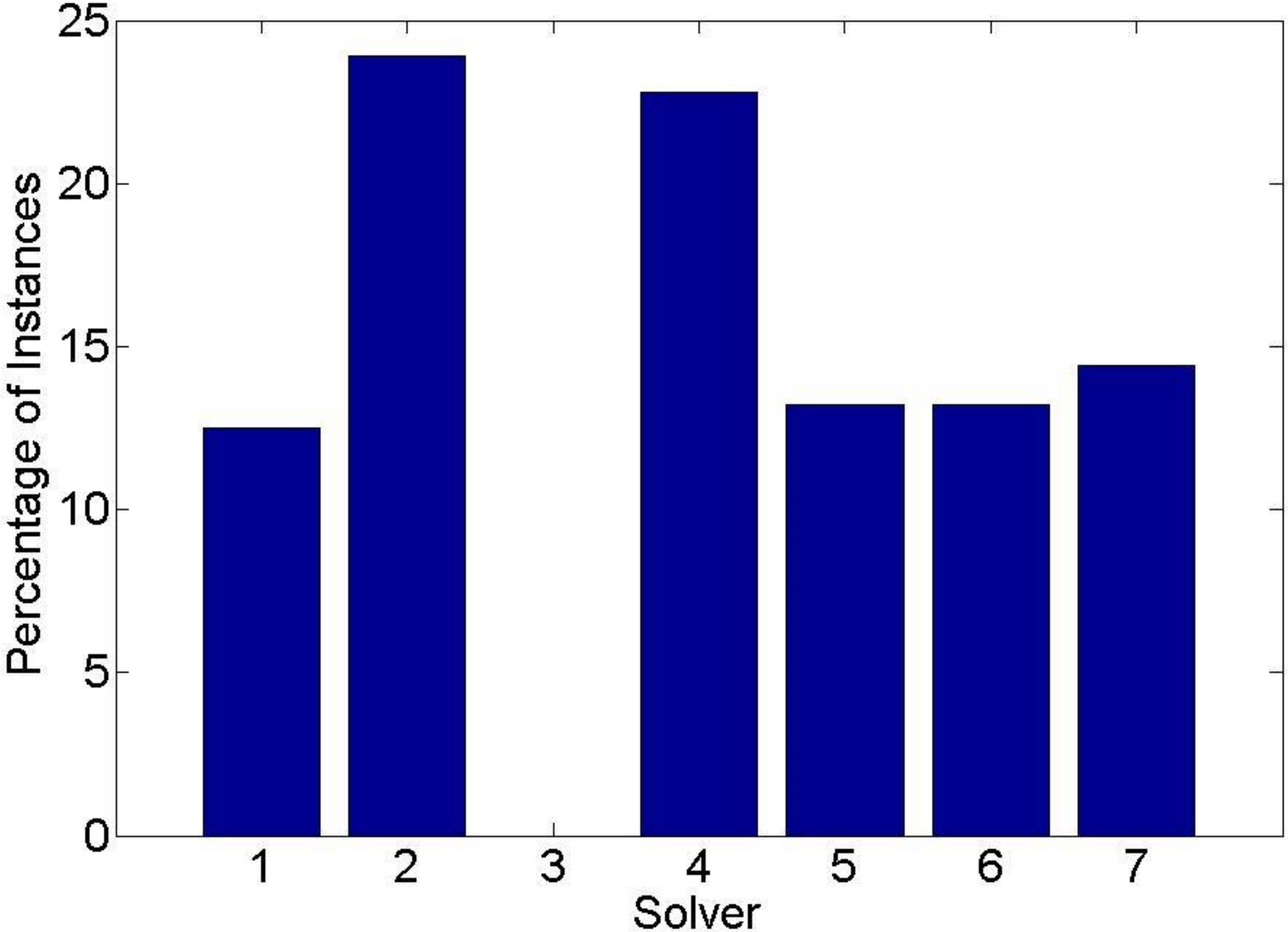
| Solver | RAND | HAND | BM | INDU |
|---|---|---|---|---|
| Best Challenger (of 17) | 1128.63 | 2960.39 | 224.53 | 11.89 |

** Statistically insignificant performance difference (sign rank test).*
*Hydra's performance was significantly better in all other pairings.*

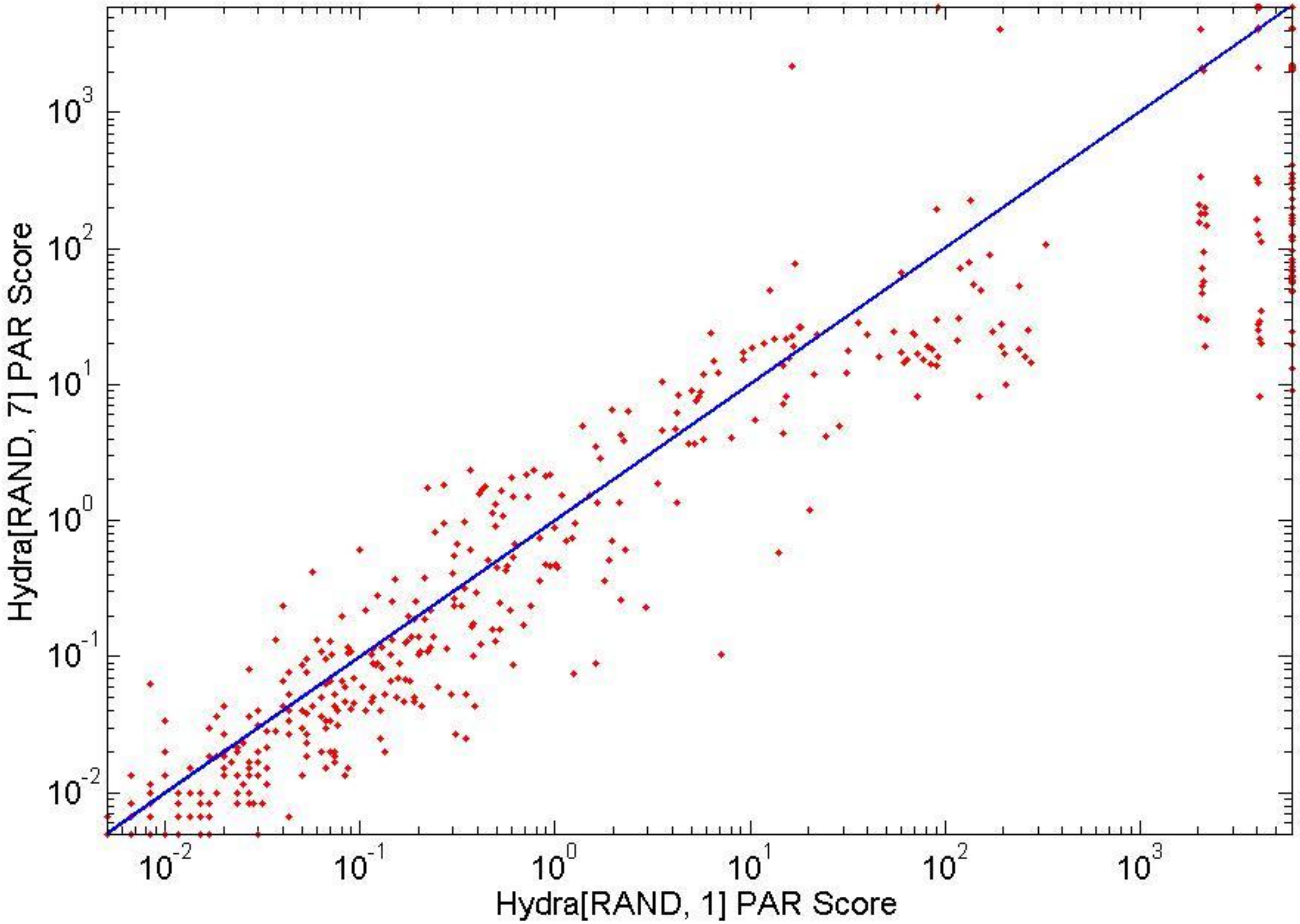
# Performance Progress, RAND



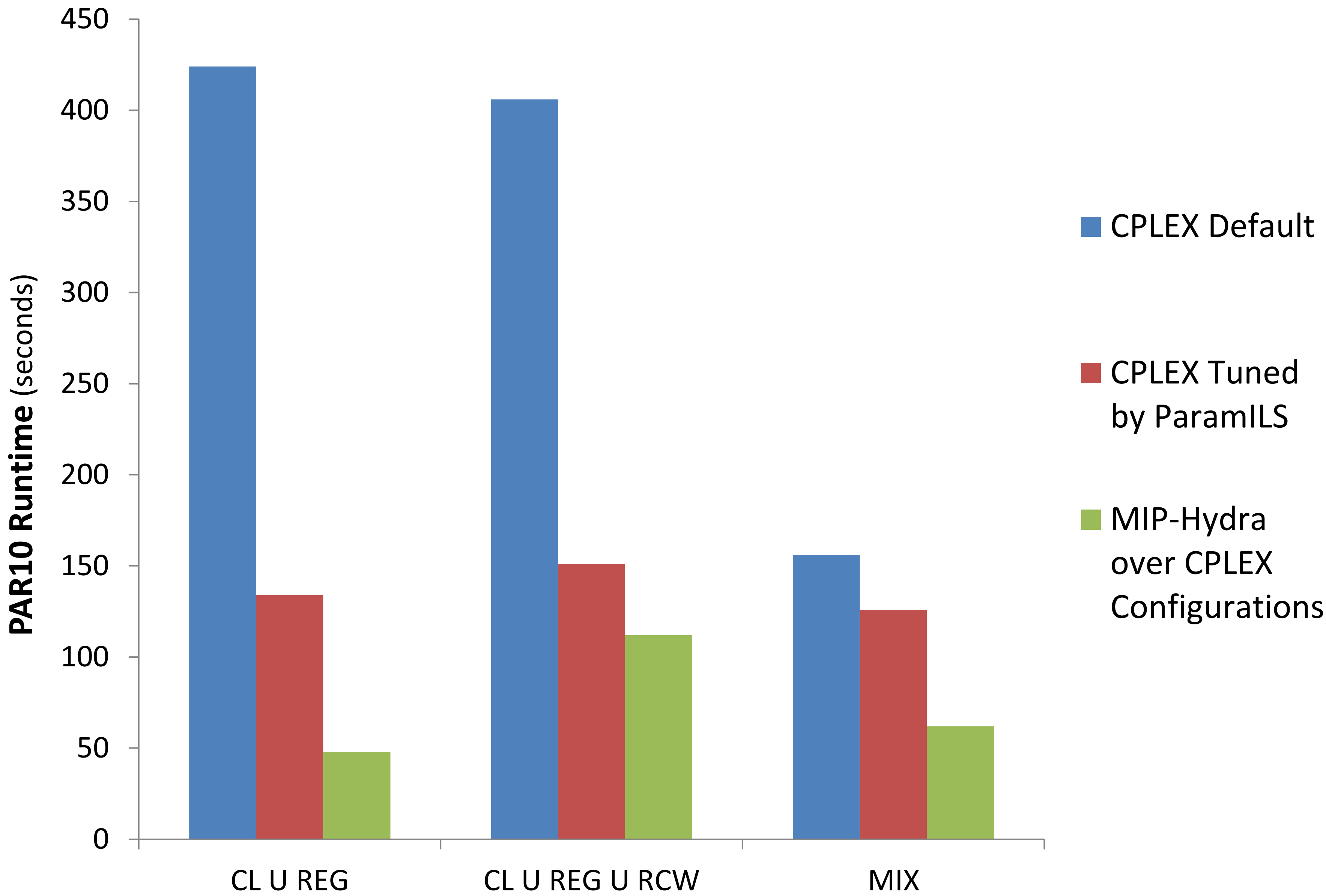
# Selection Percentages After 7 Iterations, RAND



# Improvement After 7 Iterations, RAND



# We've had success applying Hydra to MIP, too

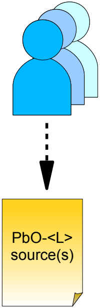


# Conclusions

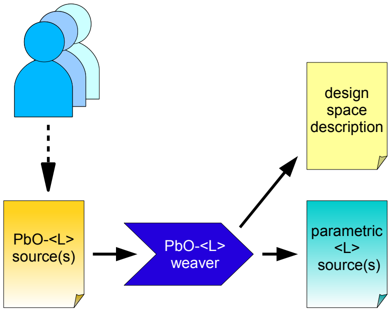
- **SATzilla: a framework for algorithm selection**
  - a robust and practically successful method for performing portfolio-based algorithm selection
  - works beyond SAT; free downloadable tools
- **Comparing simple & complex algorithm selection methods**
  - SATzilla with cost-sensitive classification is consistently best
  - but, often diminishing returns from more complex methods
    - most important thing is using portfolios rather than single solvers
- **Evaluating component solver contributions**
  - examine solvers’ marginal contributions to portfolio
  - sometimes surprising: “weak” solvers can be important
- **Hydra: automatic portfolio construction**
  - again, leverage the idea of marginal contribution to build strong portfolios, combining selection with configuration

# Software Development Support and Further Directions

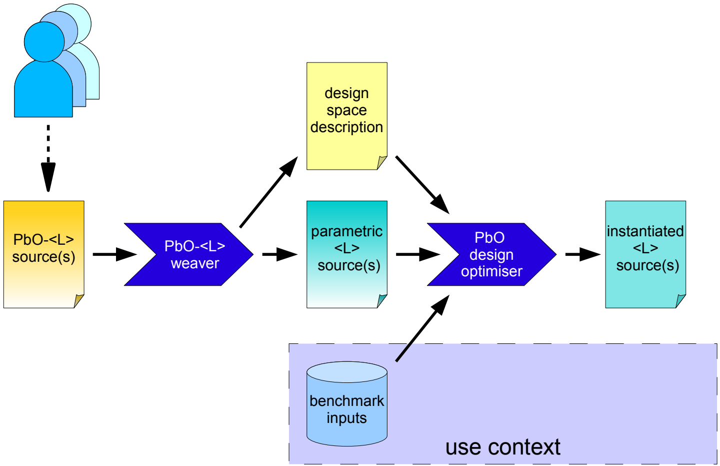
# Software development in the PbO paradigm



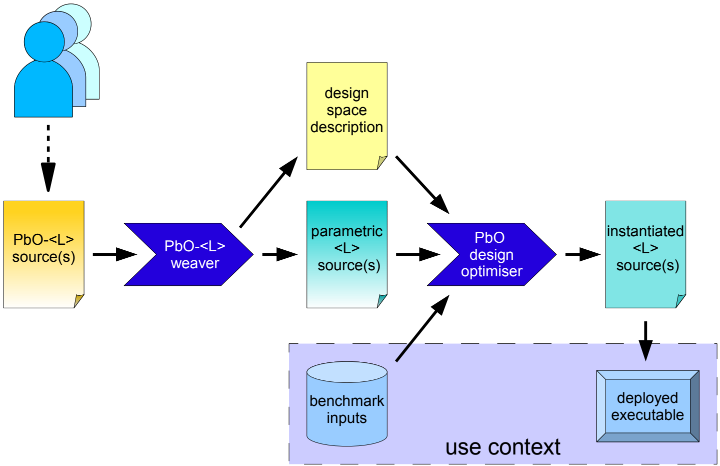
# Software development in the PbO paradigm



# Software development in the PbO paradigm



# Software development in the PbO paradigm



# Design space specification

## Option 1: use language-specific mechanisms

- command-line parameters
- conditional execution
- conditional compilation (`ifdef`)

## Option 2: generic programming language extension

Dedicated support for ...

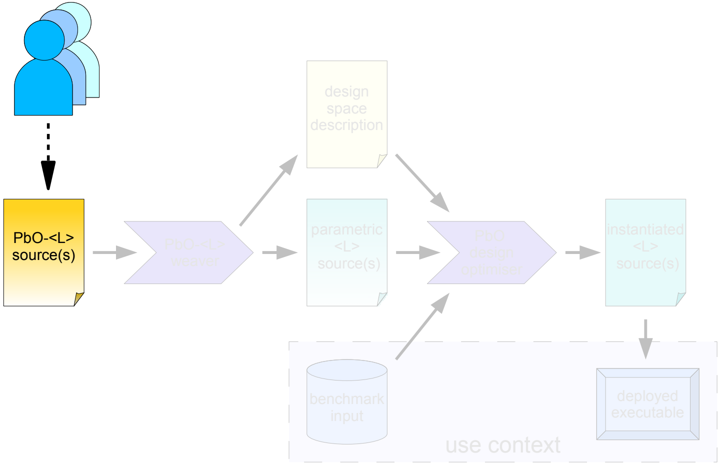
- exposing parameters
- specifying alternative blocks of code

Advantages of generic language extension:

- reduced overhead for programmer
- clean separation of design choices from other code
- dedicated PbO support in software development environments

Key idea:

- augmented sources: *PbO-Java* = Java + PbO constructs, ...
- tool to compile down into target language: *weaver*


PbO-<L>
source(s)
PbO-<L>
weaver
design
space
description
parametric
<L>
source(s)
PbO
design
optimiser
instantiated
<L>
source(s)
benchmark
input
deployed
executable
use context

## Exposing parameters

```
...
numerator -= (int) (numerator / (adjfactor+1) * 1.4);
...      ...
##PARAM(float multiplier=1.4)
numerator -= (int) (numerator / (adjfactor+1) * ##multiplier);
...
```

- parameter declarations can appear at arbitrary places (before or after first use of parameter)
- access to parameters is read-only (values can only be set/changed via command-line or config file)

## Specifying design alternatives

- **Choice:** set of interchangeable fragments of code that represent design alternatives (**instances of choice**)
- **Choice point:** location in a program at which a choice is available

```
##BEGIN CHOICE preProcessing
<block 1>
##END CHOICE preProcessing
```

## Specifying design alternatives

- **Choice:** set of interchangeable fragments of code that represent design alternatives (**instances of choice**)
- **Choice point:**
  location in a program at which a choice is available

```
##BEGIN CHOICE preProcessing=standard
<block S>
##END CHOICE preProcessing

##BEGIN CHOICE preProcessing=enhanced
<block E>
##END CHOICE preProcessing
```

## Specifying design alternatives

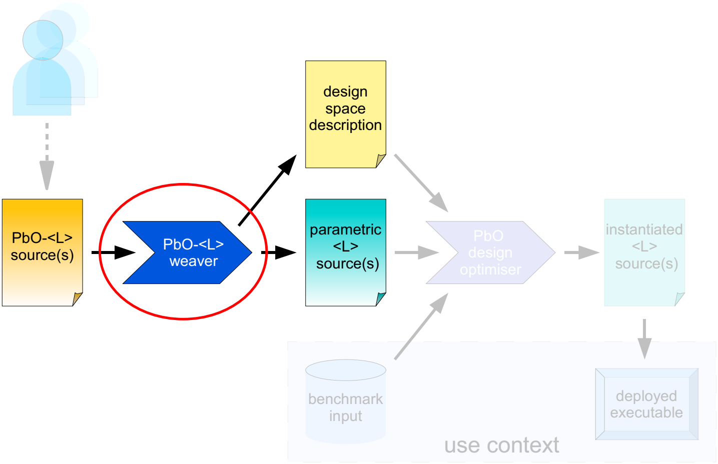
- **Choice:** set of interchangeable fragments of code that represent design alternatives (**instances of choice**)
- **Choice point:** location in a program at which a choice is available

```
##BEGIN CHOICE preProcessing
<block 1>
##END CHOICE preProcessing

...

##BEGIN CHOICE preProcessing
<block 2>
##END CHOICE preProcessing
```

## Specifying design alternatives

- **Choice:** set of interchangeable fragments of code that represent design alternatives (**instances of choice**)
- **Choice point:** location in a program at which a choice is available

```
##BEGIN CHOICE preProcessing
<block 1a>
  ##BEGIN CHOICE extraPreProcessing
  <block 2>
  ##END CHOICE extraPreProcessing
<block 1b>
##END CHOICE preProcessing
```


PbO-<L>
source(s)
PbO-<L>
weaver
design
space
description
parametric
<L>
source(s)
PbO
design
optimiser
instantiated
<L>
source(s)
benchmark
input
deployed
executable
use context

## The Weaver

transforms PbO-$<$L$>$ code into $<$L$>$ code
($<$L$>$ = Java, C++, . . . )

- parametric mode:
  - expose parameters
  - make choices accessible via (conditional, categorical) parameters
- (partial) instantiation mode:
  - hardwire (some) parameters into code (expose others)
  - hardwire (some) choices into code (make others accessible via parameters)

# The road ahead

- Support for PbO-based software development
  - Weavers for PbO-C, PbO-C++, PbO-Java
  - PbO-aware development platforms
  - Improved / integrated PbO design optimiser
  - Debugging and performance analysis tools
- Best practices
- Many further applications
- Scientific insights

## Which choices matter?

**Observation:** Some design choices matter more than others

depending on ...

- algorithm under consideration
- given use context

**Knowledge which choices / parameters matter may ...**

- guide algorithm development
- facilitate configuration

3 recent approaches:

- Forward selection based on empirical performance models
  Hutter, Hoos, Leyton-Brown (2013)
- Functional ANOVA based on empirical performance models
  Hutter, Hoos, Leyton-Brown (under review)
- Ablation analysis
  Fawcett, Hoos (2013)

# Functional ANOVA based on empirical performance models

Hutter, Hoos, Leyton-Brown (under review)

**Key idea:**

- build regression model of algorithm performance as a function of all input parameters (= design choices)

  $\rightsquigarrow$ empirical performance models (EPMs)

- analyse variance in model output (= predicted performance) due to each parameter, parameter interactions

- importance of parameter: fraction of performance variation over configuration space explained by it (main effect)

- analogous for sets of parameters (interaction effects)

## Decomposition of variance in a nutshell

For parameters $p_1, \ldots, p_n$ and a function (performance model) $y$:

$$
\begin{aligned}
y(p_1, \ldots, p_n) \quad = \quad & \mu \\
& + f_1(p_1) + f_2(p_2) + \cdots + f_n(p_n) \\
& + f_{1,2}(p_1, p_2) + f_{1,3}(p_1, p_3) + \cdots + f_{n-1,n}(p_{n-1}, p_n) \\
& + f_{1,2,3}(p_1, p_2, p_3) + \cdots \\
& + \cdots
\end{aligned}
$$

Note:

- Straightforward computation of main and interaction effects is intractable.
  (integration over combinatorial spaces of configurations)

- For random forest models, marginal performance predictions and variance decomposition (up to constant-sized interactions) can be computed exactly and efficiently.

Empirical study:

- 8 high-performance solvers for SAT, ASP, MIP, TSP
(4–85 parameters)
- 12 well-known sets of benchmark data
(random + real-world structure)
- random forest models for performance prediction,
trained on 10 000 randomly sampled configurations per solver
+ data from 25+ runs of SMAC configuration procedure

Fraction of variance explained by main effects:

| | |
|---|---|
| CPLEX on RCW (comp sust) | 70.3% |
| CPLEX on CORLAT (comp sust) | 35.0% |
| Clasp on software verificatition | 78.9% |
| Clasp on DB query optimisation | 62.5% |
| CryptoMiniSAT on bounded model checking | 35.5% |
| CryptoMiniSAT on software verification | 31.9% |

Fraction of variance explained by main + 2-interaction effects:

| | |
|---|---|
| CPLEX on RCW (comp sust) | 70.3% + 12.7% |
| CPLEX on CORLAT (comp sust) | 35.0% + 8.3% |
| Clasp on software verificatition | 78.9% + 14.3% |
| Clasp on DB query optimisation | 62.5% + 11.7% |
| CryptoMiniSAT on bounded model checking | 35.5% + 20.8% |
| CryptoMiniSAT on software verification | 31.9% + 28.5% |

Note:

may pick up variation caused by poorly performing configurations

Simple solution:

cap at default performance or quantile from distribution of randomly sampled configurations; build model from capped data.

## Ablation analysis

Fawcett, Hoos (2013)

**Key idea:**

- given two configurations, $A$ and $B$, change one parameter at a time to get from $A$ to $B$

  $\rightsquigarrow$ ablation path

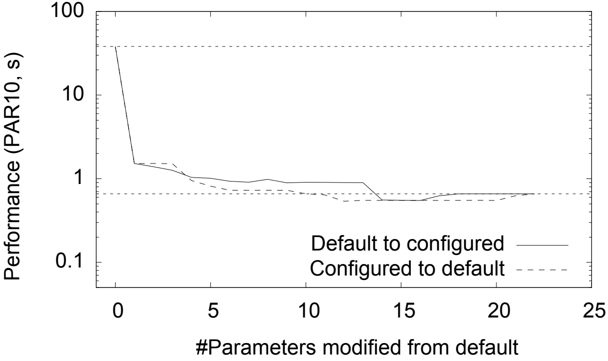
- in each step, change parameter to achieve maximal gain (or minimal loss) in performance

- for computational efficiency, use racing (F-race) for evaluating parameters considered in each step

## Empirical study:

- high-performance solvers for SAT, MIP, AI Planning (26–76 parameters), well-known sets of benchmark data (real-world structure)
- optimised configurations obtained from ParamILS (minimisation of penalised average running time; 10 runs per scenario, 48 CPU hours each)

## Ablation between default and optimised configurations:




LPG on Depots planning domain

## Which parameters are important?

LPG on `depots`:

- `cri_intermediate_levels` (43% of overall gain!)
- `triomemory`
- `donot_try_suspected_actions`
- `walkplan`
- `weight_mutex_in_relaxed_plan`

**Note:** Importance of parameters varies between planning domains

## Leveraging parallelism

- design choices in parallel programs
  (Hamadi, Jabhour, Sais 2009)
- deriving parallel programs from sequential sources
  $\rightsquigarrow$ concurrent execution of optimised designs
  (parallel portfolios)
  (Hoos, Leyton-Brown, Schaub, Schneider 2012)
- parallel design optimisers
  (*e.g.*, Hutter, Hoos, Leyton-Brown 2012)

# Take-home Message

# Programming by Optimisation ...

- leverages computational power to construct better software
- enables creative thinking about design alternatives
- produces better performing, more flexible software
- facilitates scientific insights into
  - efficacy of algorithms and their components
  - empirical complexity of computational problems

... changes how we build and use high-performance software

## More Information:

www.cs.ubc.ca/labs/beta/Projects/PbO_Tutorial

www.prog-by-opt.net

## If PbO works for you:

Make our day – let us know!

Share the joy – tell everyone else!