Learning how to solve it
– faster, better and cheaper

Holger H. Hoos

LIACS	CS Department
Universiteit Leiden	University of British Columbia
The Netherlands	Canada

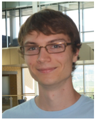
OR 2018
Bruxelles (Belgium)
2018/09/14


Frank Hutter
UBC


Lin Xu
UBC


Zongxu Mu
UBC


Chris Thornton
UBC


Yasha Pushak
UBC


Marius Schneider
U. Potsdam


Lars Kotthoff
UBC


Thomas Stützle
U. Libre de Bruxelles


Kevin Leyton-Brown
UBC


Yoav Shoham
Stanford U.

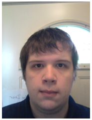

Eugene Nudelman
Stanford U.


Alan Hu
UBC


Domagoj Babić
UBC


Torsten Schaub
U. Potsdam


Benjamin Kaufmann
U. Potsdam


James Styles
UBC

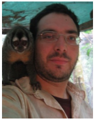

Chris Fawcett
UBC


Alfonso Gerevini
U. di Brescia


Alessandro Saetti
U. di Brescia


Mauro Vallati
U. di Brescia


Matle Helmert
U. Basel


Erez Karpas
Technion


Gabriele Röger
U. Freiburg


Jendrik Seipp
U. Freiburg


Chuan Luo
U. Leiden


Chris Cameron
UBC


Jérémy Dubois-Lacoste
U. Libre de Bruxelles


Pascal Kerschke
U. Münster


Jakob Bossek
U. Münster


Bernd Bischl
LMU München

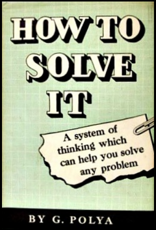

Heike Trautmann
U. Münster

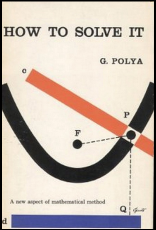

Donald Knuth
Stanford U.

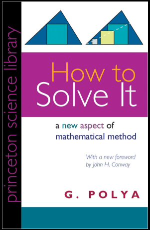
HOW TO SOLVE IT

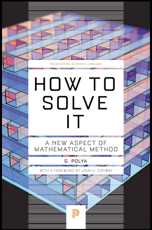
A system of thinking which can help you solve any problem

BY G. POLYA









published in 1945; $> 1\,000\,000$ copies sold

Marvin Minsky: "everyone should know the work of George Pólya
on how to solve problems"

highly praised by Zhores Ivanovich Alferov (2000 Nobel prize in physics)

How to solve it:

1. Understand the problem.
2. Devise a plan (translate).
3. Carry out the plan (solve).
4. Look back (check and reinterpret).

[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

For now, I only consider optimising this step of Polya's approach.

How to solve it:

1. Understand the problem.

2. Devise a plan (translate).

3. Carry out the plan (solve).

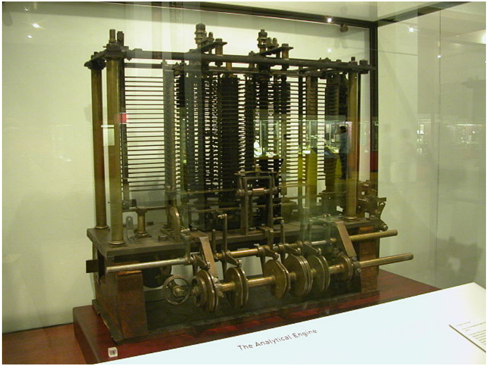
4. Look back (check and reinterpret).

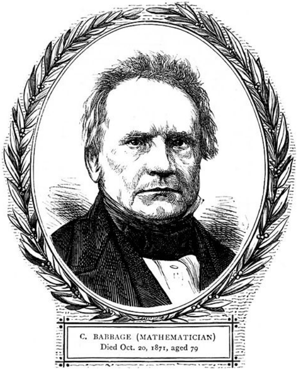
The nature of computation

Clear, precise instructions – flawlessly executed

$\rightsquigarrow$ algorithm

The age of machines





"As soon as an Analytical Engine exists, it will necessarily guide the future course of the science.

(Charles Babbage, 1864)

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
- ...

[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

Today, machine learning is "the next big thing". But some tend to forget that this, too, has a long history. Even multi-layer neural networks have been around for a long time ...

Machine learning is old ...

- Alan Turing (1950): Computing machinery and intelligence
- Farley and Clark (1954): Simulation of Self-Organizing Systems by Digital Computer
- Arthur Samuel (1959): Some Studies in Machine Learning Using the Game of Checkers
- Paul Werbos (1974): Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences
- ...

[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

Of course, there has been much progress recently. Still, much of it still fits into a few categories of approaches that have been studied for a while.

Traditional machine learning:

- supervised classification / regression:

  Given: set of training data with correct labels
  $T := \{(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_k}, y_k)\}$
  Want: function mapping $\mathbf{x}$ to y minimising error on T

- unsupervised learning
- semi-supervised learning
- reinforcement learning

NB: learning = optimisation over family of functions ('models')

Generalised machine learning:

- optimisation over family of algorithms for given problem **P**
  e.g., TSP solvers
- **Goal:** maximise performance
  e.g., expected time for finding optimal solution
- **Given:** set of problem instances $T := \{i_1, \ldots i_k\}$
- **Want:** algorithm with maximum performance on T

The machine learning revolution

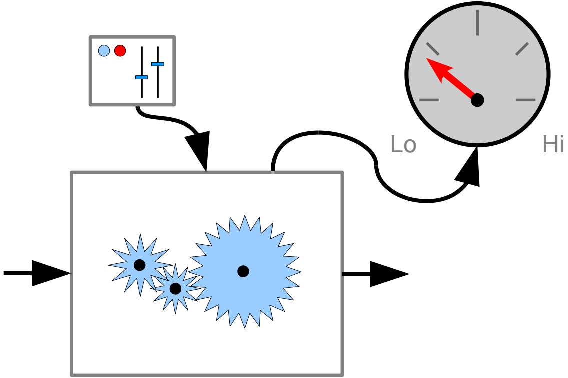
manually constructed algorithms

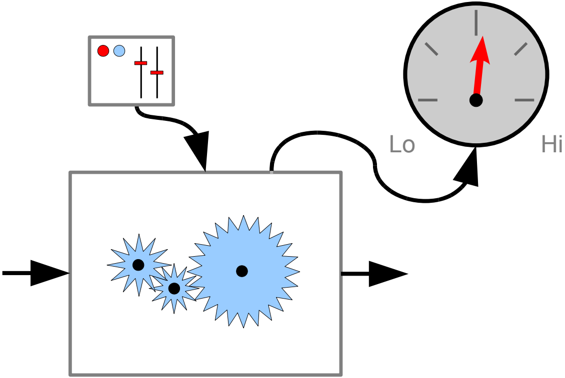
$\rightsquigarrow$

automatic adaptation to given set / distribution of inputs
through optimisation of performance metric (loss minimisation)

machine learning procedures
= meta-algorithms (procedures for optimising algorithm)

2


Lo
Hi


Lo
Hi

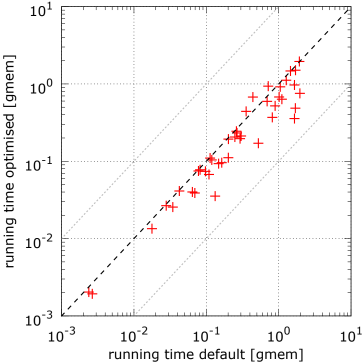
"Parameter optimization for general broad-spectrum use is a daunting task [...]

How could then *any* set of defaults be recommended, without an enormous expense of time and money? Fortunately, there's a way out of this dilemma, thanks to advances in the theory of learning."

Donald Knuth, The Art of Computer Programming,
Vol. 4, Fascicle 6 (Satisfiability), p. 125

Knuth's sat13 (7.2.2.2C) on diverse set of instances

(very easy to medium; trained on very easy only)

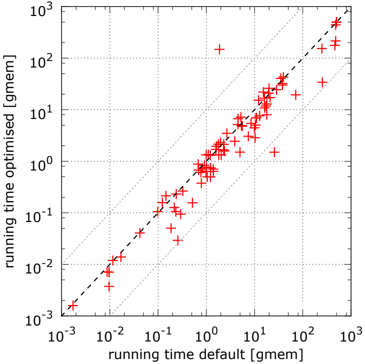


mean running time 0.572 → 0.402 gmems;

geometric average speedup: 1.414-fold

Knuth's sat13 (7.2.2.2C) on diverse set of instances

(TAOCP testing instances; trained on very easy)

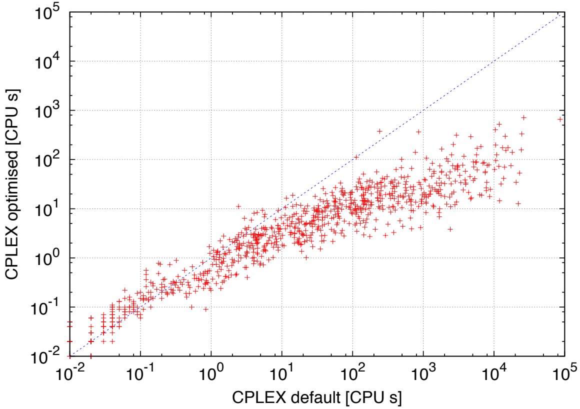


mean running time 47.4 $\rightarrow$ 36.9 gmems;

geometric average speedup: 1.357-fold

CPLEX on Wildlife Corridor Design

Hutter, HH, Leyton-Brown (2010)



$\leadsto$ 52.3 $\times$ speedup on average!

The algorithm configuration problem

Given:

- parameterised target algorithm A with configuration space C
- set of (training) inputs I
- performance metric m (w.l.o.g. to be minimised)

Want: $c^* \in \arg\min_{c \in C} m(A[c], I)$

Algorithm configuration is challenging:

- size of configuration space
- discrete / categorical parameters
- parameter interactions
- conditional parameters
- performance varies across inputs (problem instances)
- evaluating configurations can be very costly
- censored algorithm runs

$\rightsquigarrow$ standard optimisation methods are insufficient

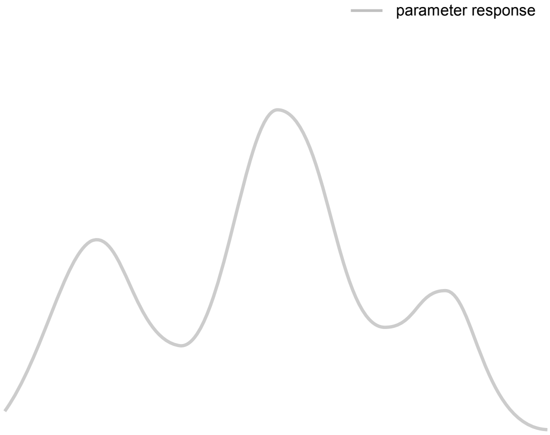
Algorithm configuration approaches:

- Advanced sampling methods
  (*e.g.*, REVAC, REVAC++ – Nannen & Eiben 2006–09)
- Racing
  (*e.g.*, F-Race – Birattari, Stützle, Paquete, Varrentrapp 2002;
  Iterative F-Race – Balaprakash, Birattari, Stützle 2007;
  irace package – López-Ibáñez, Dubois-Lacoste, Stützle, Birattari 2011;
  irace+capping – Péres-Cáceres, López-Ibáñez, HH, Stützle, Birattari 2017)
- Model-free search
  (*e.g.*, ParamILS – Hutter, HH, Stützle 2007;
  Hutter, HH, Leyton-Brown, Stützle 2009)
- Sequential model-based optimisation
  (*e.g.*, SPO – Bartz-Beielstein 2006; SMAC – Hutter, HH, Leyton-Brown 2011–12)

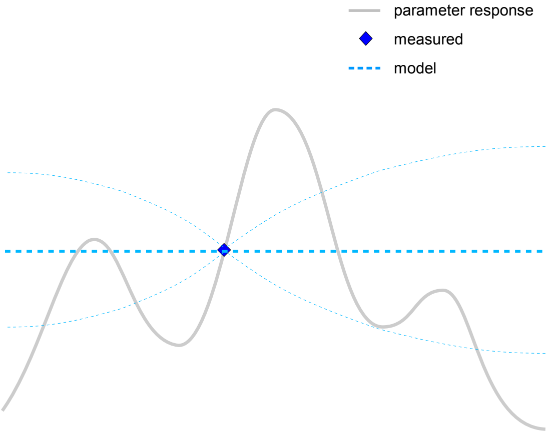
Algorithm configuration approaches:

- Advanced sampling methods
  (*e.g.*, REVAC, REVAC++ – Nannen & Eiben 2006–09)
- Racing
  (*e.g.*, F-Race – Birattari, Stützle, Paquete, Varrentrapp 2002;
  Iterative F-Race – Balaprakash, Birattari, Stützle 2007;
  irace package – López-Ibáñez, Dubois-Lacoste, Stützle, Birattari 2011;
  irace+capping – Péres-Cáceres, López-Ibáñez, HH, Stützle, Birattari 2017)
- Model-free search
  (*e.g.*, ParamILS – Hutter, HH, Stützle 2007;
  Hutter, HH, Leyton-Brown, Stützle 2009)
- Sequential model-based optimisation
  (*e.g.*, SPO – Bartz-Beielstein 2006; SMAC – Hutter, HH, Leyton-Brown 2011–12)

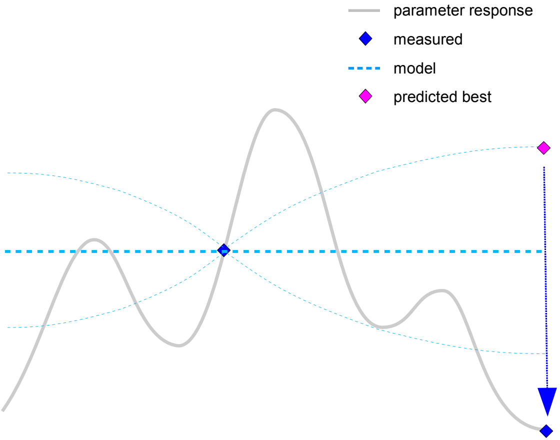
Sequential Model-based Optimisation



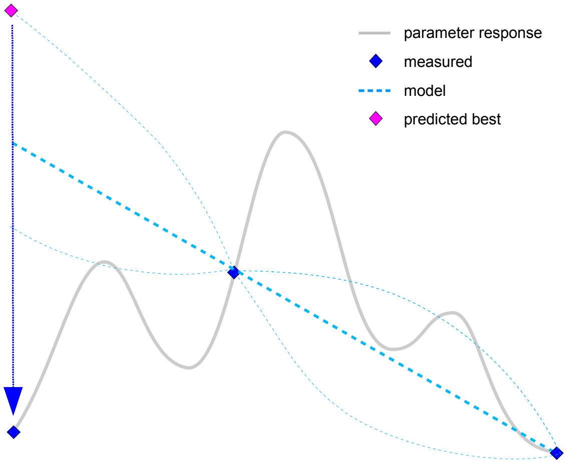
Sequential Model-based Optimisation



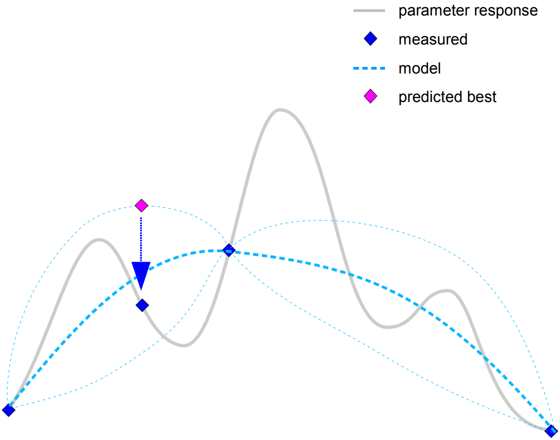
Sequential Model-based Optimisation



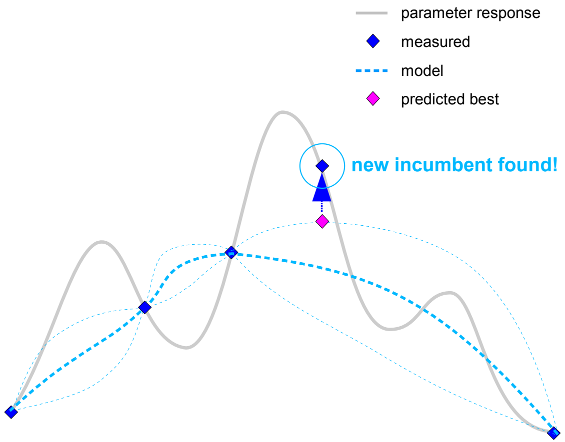
Sequential Model-based Optimisation



Sequential Model-based Optimisation



Sequential Model-based Optimisation



Sequential Model-based Algorithm Configuration (SMAC)

Hutter, HH, Leyton-Brown (2011)

- uses *random forest model* to predict performance of parameter configurations
- predictions based on algorithm parameters and instance features, aggregated across instances
- finds promising configurations based on *expected improvement criterion*, using multi-start local search and random sampling
- initialisation with single configuration (algorithm default or randomly chosen)

Excellent results on widely studied problems:

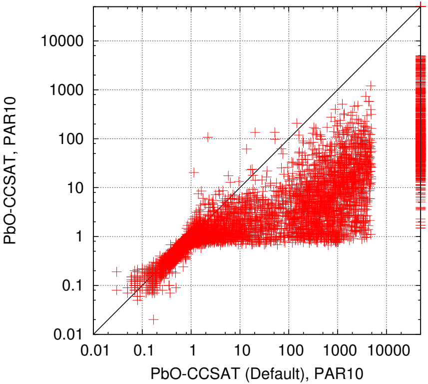
- Mixed integer programming (CPLEX):
  76 parameters, 2...52× speed-up
  Hutter, Leyton-Brown, HH (2010)

- AI Planning (LPG):
  62 parameters, 3...118× speed-up
  Vallati, Fawcett, Gerevini, HH, Saetti (2011)

- Propositional satisfiability (PbO-CCSAT):
  23 parameters, 3..230× speed-up
  Luo, HH, Cai (under review)

PbO-CCSAT on revenue-optimising spectrum repacking (FCC)

(performance on test instances not used for configuration)



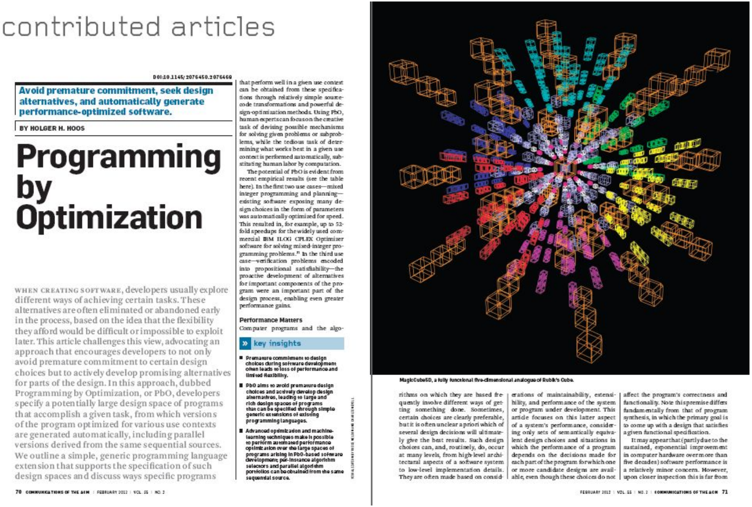
running time (PAR10) 6554 $\rightarrow$ 1979 CPU sec;

average speedup on instances solved by both configurations: 50-fold

Further success stories:

- garbage collection in Java
  Lengauer & Mössenböck (2014)
- bike sharing rebalancing
  Dell'Amico *et al.* (2016)
- Machine learning (Auto-WEKA): 768 parameters
  Thornton, Hutter, HH, Leyton-Brown (2013);
  Kotthoff, Thornton, HH, Leyton-Brown (2017)

  $\rightsquigarrow$ automated machine learning (AutoML)



DOI:10.1145/2076450.2076469

Avoid premature commitment, seek design alternatives, and automatically generate performance-optimized software.

BY HOLGER H. HOOS

Programming by Optimization

WHEN CREATING SOFTWARE, developers usually explore different ways of achieving certain tasks. These alternatives are often eliminated or abandoned early in the process, based on the idea that the flexibility they afford would be difficult or impossible to exploit later. This article challenges this view, advocating an approach that encourages developers to not only avoid premature commitment to certain design choices but to actively develop promising alternatives for parts of the design. In this approach, dubbed Programming by Optimization, or PbO, developers specify a potentially large design space of programs that accomplish a given task, from which versions of the program optimized for various use contexts are generated automatically, including parallel versions derived from the same sequential sources. We outline a simple, generic programming language extension that supports the specification of such design spaces and discuss ways specific programs that perform well in a given use context can be obtained from these specifications through relatively simple source-code transformations and powerful design-optimization methods. Using PbO, human experts can focus on the creative task of devising possible mechanisms for solving given problems or subproblems, while the tedious task of determining what works best in a given use context is performed automatically, substituting human labor by computation.

The potential of PbO is evident from recent empirical results (see the table here). In the first two use cases—mixed integer programming and planning—existing software exposing many design choices in the form of parameters was automatically optimized for speed. This resulted in, for example, up to 52-fold speedups for the widely used commercial IBM ILOG CPLEX Optimizer software for solving mixed-integer programming problems.[21] In the third use case—verification problems encoded into propositional satisfiability—the proactive development of alternatives for important components of the program were an important part of the design process, enabling even greater performance gains.

Performance Matters

Computer programs and the algorithms on which they are based frequently involve different ways of getting something done. Sometimes, certain choices are clearly preferable, but it is often unclear a priori which of several design decisions will ultimately give the best results. Such design choices can, and, routinely, do, occur at many levels, from high-level architectural aspects of a software system to low-level implementation details. They are often made based on considerations of maintainability, extensibility, and performance of the system or program under development. This article focuses on this latter aspect of a system's performance, considering only sets of semantically equivalent design choices and situations in which the performance of a program depends on the decisions made for each part of the program for which one or more candidate designs are available, even though these choices do not affect the program's correctness and functionality. Note this premise differs fundamentally from that of program synthesis, in which the primary goal is to come up with a design that satisfies a given functional specification.

It may appear that (partly due to the sustained, exponential improvement in computer hardware over more than five decades) software performance is a relatively minor concern. However, upon closer inspection this is far from

» key insights

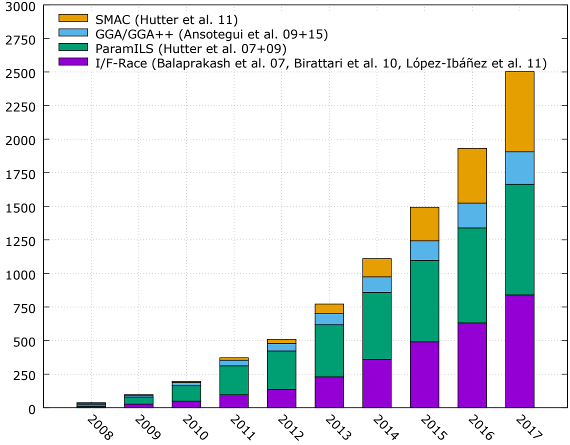
- Premature commitment to design choices during software development often leads to loss of performance and limited flexibility.
- PbO aims to avoid premature design choices and actively develop design alternatives, leading to large and rich design spaces of programs that can be specified through simple generic extensions of existing programming languages.
- Advanced optimization and machine-learning techniques make it possible to perform automated performance optimization over the large spaces of programs arising in PbO-based software development; per-instance algorithm selectors and parallel algorithm portfolios can be obtained from the same sequential source.

MagicCube5D, a fully functional five-dimensional analogue of Rubik's Cube.

Communications of the ACM, 55(2), pp. 70–80, February 2012

www.prog-by-opt.net

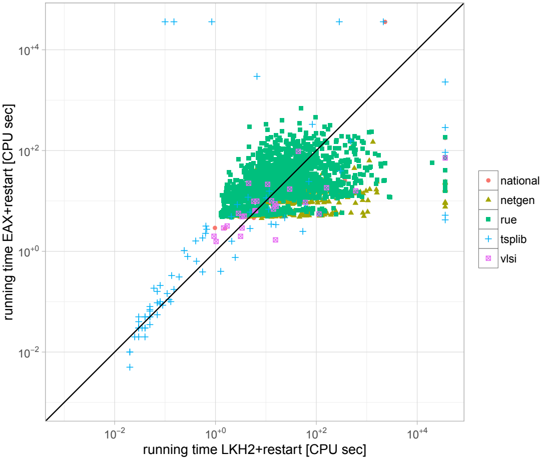
Total citations for key publications on automated algorithm configuration



(Data from Google Scholar; year *vs* total # citations up that year)

3

LKH2+restart vs EAX+restart



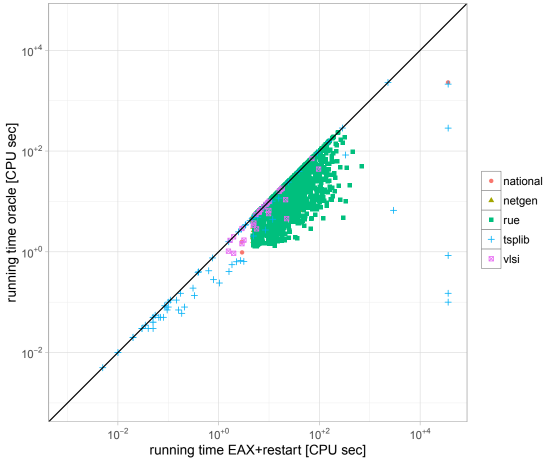
Per-instance algorithm selection (Rice 1976):

- *Given:* set S of algorithms for a problem, problem instance π
- *Objective:* select from S the algorithm expected to solve π *most efficiently*, based on (cheaply computable) *features* of π

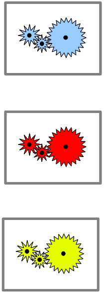
Note:

Best case performance bounded by oracle, which selects the best $s \in S$ for each π = *virtual best solver (VBS)*

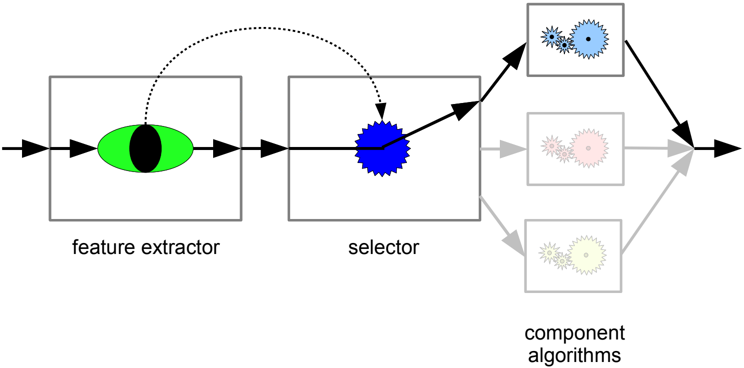
EAX+restart vs perfect selector (VBS)



Per-instance algorithm selection



Per-instance algorithm selection



Key components:

- set of (state-of-the-art) *solvers*
- set of cheaply computable, informative *features*
- efficient procedure for mapping features to solvers (*selector*)
- *training data*
- procedure for building good selector based on training data (*selector builder*)

"The overall champion in 2007 was SATzilla, which was actually not a separate SAT solver but rather a program that knew how to choose intelligently between *other* solvers an any given instance. [...]

This 'portfolio' approach, which tunes itself nicely to the characteristics of vastly different sets of clauses, has continued to dominate the international competitions ever since.

Of course portfolio solvers rely on the existence of 'real' solvers, invented independently and bug-free, which shine with respect to particular classes of problems. And of course the winner of the competition may not be the best actual system for practical use."

Donald Knuth, The Art of Computer Programming,
Vol. 4, Fascicle 6 (Satisfiability), p. 132*f.*

Methods for per-instance selection:

- **classification-based:** predict the best solver, *e.g.*, using ...
  - decision trees
    (Guerri & Milano 2004)
  - case-based reasoning
    (Gebruers *et al.* 2004)
  - (weighted) k-nearest neighbours
    (Malitsky *et al.* 2011; Kadioglu *et al.* 2011)
  - pairwise cost-sensitive decision forests + voting
    (Xu, Hutter, HH, Leyton-Brown 2012)
- **regression-based:** predict running time for each solver, select the one predicted to be fastest
  (Leyton-Brown *et al.* 2003; Xu, Hutter, HH, Leyton-Brown 2007–9)

Per-instance selection for the TSP

Kotthoff, Kerschke, HH, Trautmann (2016);
Kerschke, Bossek, Kotthoff, HH, Trautmann (2017)

- use 5 high-performance inexact TSP solvers
- consider large benchmark collection (TSPLIB, VLSI, RUE, ...)
- build per-instance selectors using range of feature sets, feature selection + machine learning methods
- assess using cross-validation

Results (PAR10):

- single best solver (EAX+restart): 36.30 CPU sec
- regression-based selector (based on SVM): 16.75 CPU sec
- oracle: 10.73 CPU sec

Excellent results for many other problems:

- SAT $\rightsquigarrow$ SAT competitions
  (*e.g.*, Xu, Hutter, HH, Leyton-Brown 2008, 2012)
- AI planning
  (*e.g.*, Helmert, Röger, Karpas 2011)
- Container pre-marshalling
  (*e.g.*, Tierney & Malitsky 2015)
- ...

$\rightsquigarrow$ ASlib (Bischl *et al.* 2016)

Combining configuration and selection:

- performance complementarity between different configurations of given solver

  $\rightsquigarrow$ selection over automatically determined configurations

  (*e.g.*, Xu, Hutter, HH, Leyton-Brown 2011

- many design choices in selector construction, different selectors perform best in different applications

  $\rightsquigarrow$ AutoFolio = automatic configuration of algorithm selectors

  (Lindauer, HH, Hutter, Schaub 2015)

Recap

1. The machine learning revolution
2. Which parameter settings? (algorithm configuration)
3. Which solver? (algorithm selection)

Wait a second ...

Faster: ✓

Better?Better: ✓

AutoML; Hutter, HH, Leyton-Brown (2010); Pagnozzi & Stützle (2018); ...

Cheaper?Cheaper: ✓

running time, manual performance optimjsation = money

Recap

1. The machine learning revolution
2. Which parameter settings? (algorithm configuration)
3. Which solver? (algorithm selection)
4. Where the road goes ...

Making automated solver construction accessible

- *problem:* automated solver design methods (AS,AC,...)
  require substantial expertise, experience to use
  $\rightsquigarrow$ full potential currently not exploited

- *idea:* develop abstractions & platform
  to facilitate use, development
  $\rightsquigarrow$ substantially lowered barrier to entry

- *proof-of-concept:* algorithm selection for SAT
  $\rightsquigarrow$ Sparkle platform, Sparkle SAT Challenge

[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

When we develop or assess algorithms, we often think of that process like a competition, a race. (And sometimes, as in SAT, we even have prominent competitions.)

In a competition, all the glory goes to the winner; Even the 2nd and 3rd place pale in comparison.

[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

Instead, we want to give a single gold medal, but cut it into pieces, recognising how much every competitor contributes to the state of the art in solving a given class of problem instances.

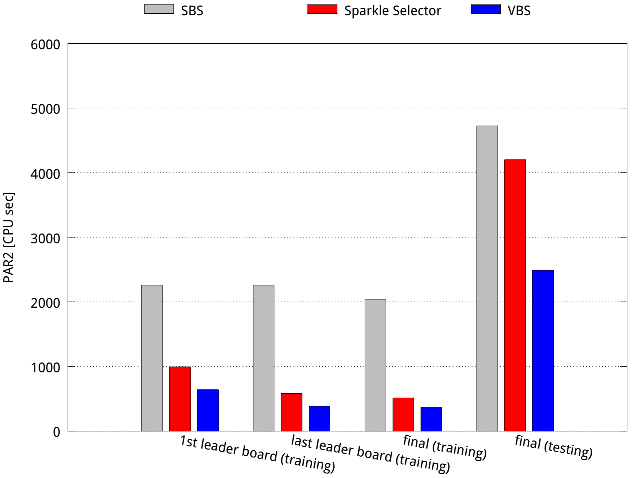

1
2
3


XXXI Olimpíada Rio 2016

Sparkle SAT Challenge 2018

- part of FLoC Olympic Games, coordinated with 2018 SAT Competition
- launched March 2018, leader board phase 5–15 April, final results @ FLoC 2018 (July)
- 23 solvers submitted (19 open-source, 4 closed-source, *hors concours*)
- details: http://ada.liacs.nl/events/sparkle-sat-18

Improvement over time, including hors-concours solvers



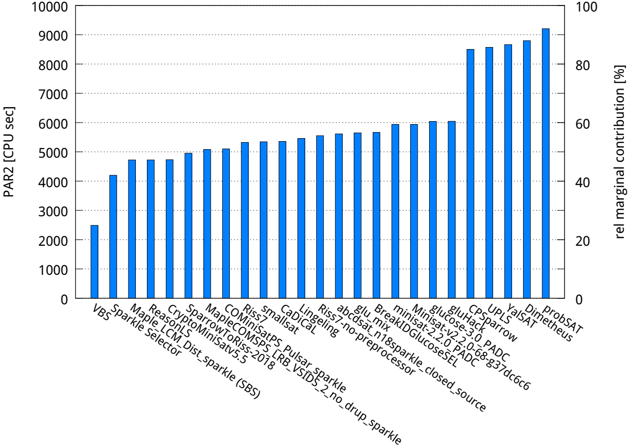
[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

Notice how the best stand-alone performance only improves right after the leaderboard phase, when some competitors who have held back their solvers finally submit them.

Notice also how the selector and the VBS improve throughout the leaderboard phase and beyond.

Test data was the same as in the SAT competition, and quite different from training data.
Still, the selector performs well.

Stand-alone and relative marginal contribution on testing set, with hors-concours solvers

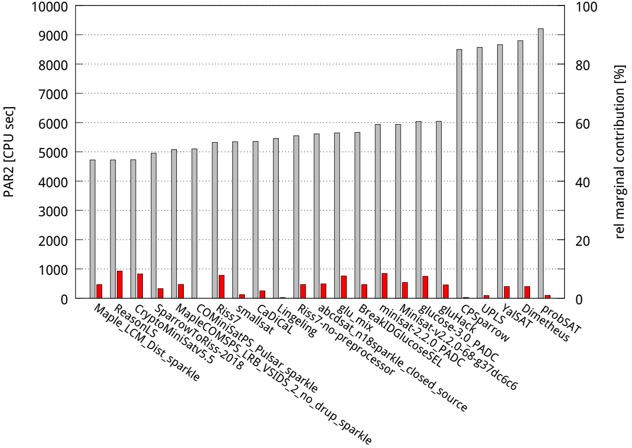


[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

The best solvers are very close w.r.t.
stand-alone performance
(typical for SAT competition).

The selector and VBS are much better.

Stand-alone and relative marginal contribution on testing set, with hors-concours solvers



[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

Notice how the best stand-alone solver
does not make the biggest contribution to the selector.
Some very low-ranked solvers contribute very similarly.

Advantages over traditional competition:

- makes it easier to gain recognition for specialised techniques
- better reflects and makes accessible state of the art
- provides incentive to improve true state of the art

Further use of Sparkle:

- continuous solver evaluation (as community service)
- specialised application contexts (reduced solver sets)
- experimentation platform for algorithm selection, configuration, programming by optimisation (PbO)

Take-home message:

- AI revolution: explicit $\rightsquigarrow$ automated programming
- Machine learning $= \subset$ automated performance optimisation
  $\subset$ automated algorithm design

  $\Rightarrow$ great potential for OR!
- Meta-algorithmic techniques (configurators, selectors, ...):
  powerful, useful, readily available;
  key to better MIP, TSP, CP, SAT, SMT, ..., ML, AI
- Next: Make those easily accessible + broadly usable
  $\rightsquigarrow$ Sparkle (community effort)

How to solve it:

1. Understand the problem.
2. Devise a plan (translate).
3. Carry out the plan (solve).
4. Look back (check and reinterpret).

[This slide was not used during the presentation; it was added to make it easier to follow the slide deck.]

There might be the potential to automate more of Polya's approach.

How to solve it:

1. Understand the problem.
2. Devise a plan (translate).
3. Carry out the plan (solve).
4. Look back (check and reinterpret).

How to solve it:

1. Understand the problem.
2. Devise a plan (translate).
3. Carry out the plan (solve).
4. Look back (check and reinterpret).

Bigger picture:

- Part of broader effort: Automation of AI (AutoAI) (prominent special case: AutoML)

  $\rightsquigarrow$ advancement + democratisation of AI

- Sparkle, PbO use key AI techniques + (lots of) computation to leverage human ingenuity + intuition

- Human-centred AI:
  AI that augments, not replaces, human intelligence

  $\Rightarrow$ Auto-OR ?!!