Structure at the meta-level:

Observations on the structure of design spaces of high-performance solvers for hard combinatorial problems

Holger H. Hoos

BETA Lab
Department of Computer Science
University of British Columbia
Canada

based on joint work with

Chris Fawcett, Frank Hutter, Kevin Leyton-Brown, Thomas Stützle

Acknowledgements

Collaborators:

- Domagoj Babić
- Sam Bayless
- Chris Fawcett
- Quinn Hsu
- Frank Hutter
- Erez Karpas
- Chris Nell
- Eugene Nudelman
- Steve Ramage
- Gabriele Röger
- Marius Schneider
- James Styles
- Dave Tompkins
- Mauro Vallati
- Lin Xu

- Thomas Bartz-Beielstein
  (FH Köln, Germany)
- Marco Chiarandini
  (Syddansk Universitet, Denmark)
- Alfonso Gerevini
  (Università degli Studi di Brescia, Italy)
- Malte Helmert
  (Universität Basel, Switzerland)
- Alan Hu
- Kevin Leyton-Brown
- Kevin Murphy
- Alessandro Saetti
  (Università degli Studi di Brescia, Italy)
- Thomas Stützle
  (Université Libre de Bruxelles, Belgium)

Research funding:

- NSERC, Mprime, GRAND, CFI
- IBM, Actenum Corp.

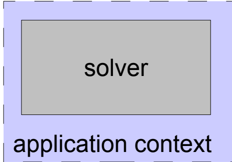
Computing resources:

- Arrow, BETA, ICICS clusters
- Compute Canada / WestGrid

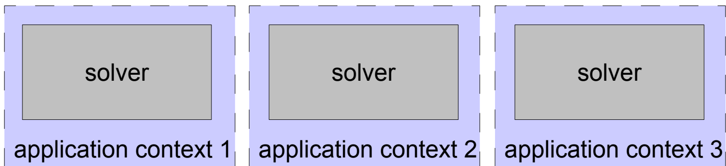
Take-home message:

- exploiting structure in problem instances permits practical solution of hard problems
  $\rightsquigarrow$ instance-level structure
- structure in space of algorithms (+ human creativity) facilitates effective construction of good solvers for hard problems
  $\rightsquigarrow$ meta-level structure
- meta-level structure may differ substantially from instance-level structure
- PbO (rich algorithm design space + automated configuration) permits (partial) automation of building effective solvers;
  efficacy depends on exploitation of meta-level structure

application context


solver
application context

application context 1
application context 2
application context 3

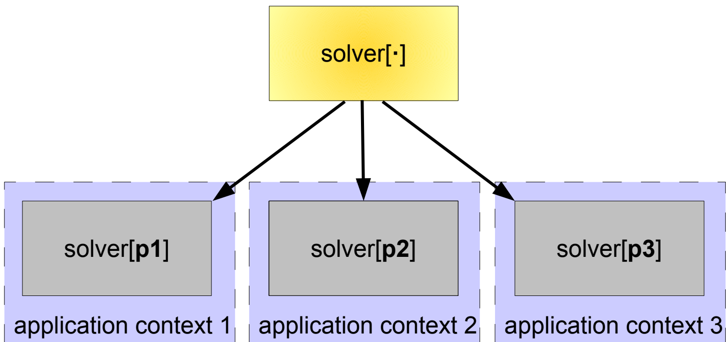

solver
solver
solver
application context 1
application context 2
application context 3

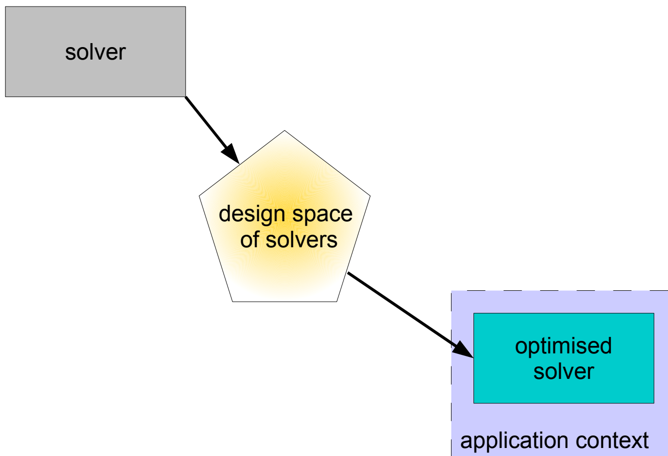
solver[·]

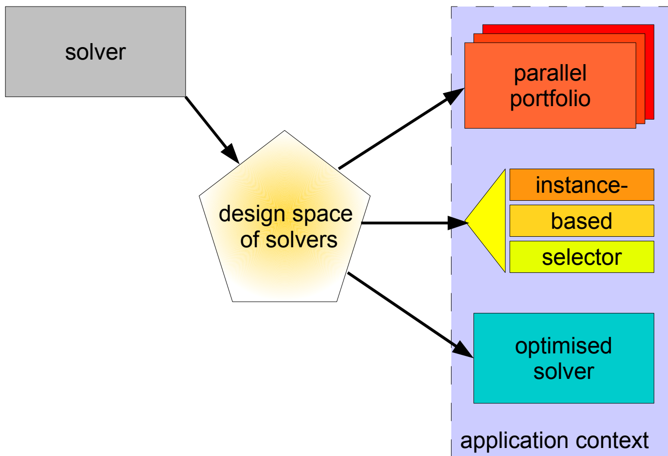
application context 1

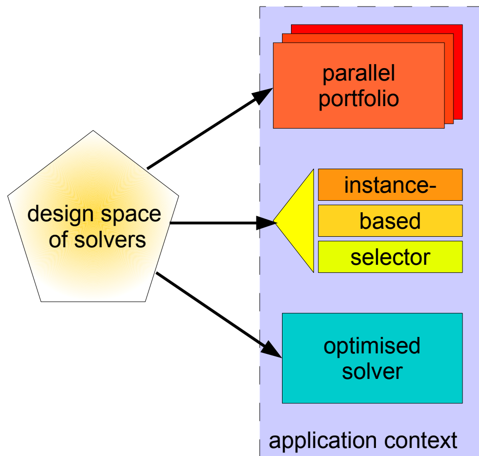
application context 2

application context 3


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


solver
design space
of solvers
optimised
solver
application context


solver
design space
of solvers
parallel
portfolio
instance-
based
selector
optimised
solver
application context


parallel
portfolio
design space
of solvers
instance-
based
selector
optimised
solver
application context

Programming by Optimisation (PbO)

- 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)

contributed articles

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.

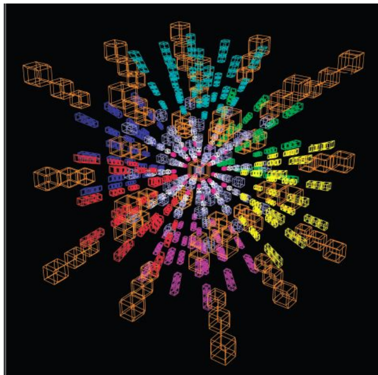
Performance Matters

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

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.

Success in optimising speed:

Application, Design choices	Speedup	PbO level
SAT-based software verification (SPEAR), 41 Hutter, Babić, HH, Hu (2007)	4.5–500 ×	2–3
AI Planning (LPG), 62 Vallati, Fawcett, Gerevini, HH, Saetti (2011)	3–118 ×	1
Mixed integer programming (CPLEX), 76 Hutter, HH, 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, HH (2009)

Machine learning / Classification, 803 design choices, PbO level 0–1
⇝ outperforms specialised model selection & hyper-parameter optimisation methods from machine learning
Thornton, Hutter, HH, Leyton-Brown (2012)

Outline

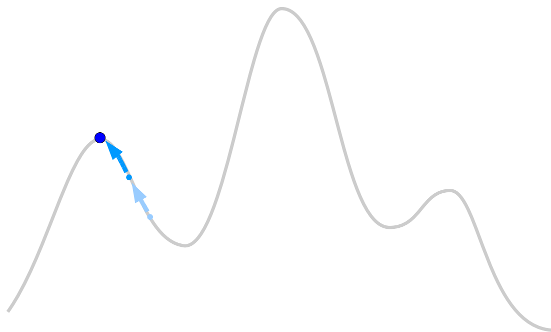
1. Introduction
2. Design spaces & design optimisation
3. Which choices matter? Global perspectives
4. Which choices matter? A local perspective
5. Speculation and open questions

Design optimisation

Simplest case: Configuration / tuning

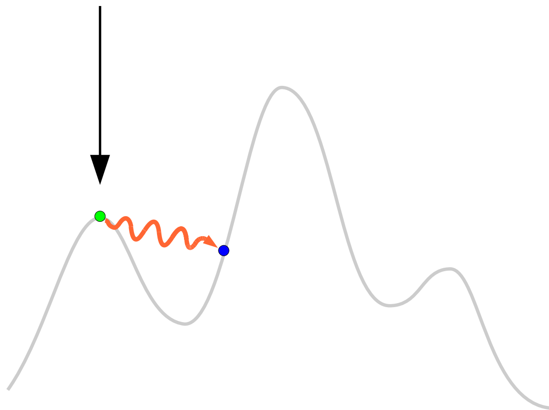
- Standard optimisation techniques
  (*e.g.*, CMA-ES – Hansen & Ostermeier 01; MADS – Audet & Orban 06)
- Advanced sampling methods
  (*e.g.*, REVAC, REVAC++ – Nannen & Eiben 06–09)
- Racing
  (*e.g.*, F-Race – Birattari, Stützle, Paquete, Varrentrapp 02;
  Iterative F-Race – Balaprakash, Birattari, Stützle 07)
- Model-free search
  (*e.g.*, ParamILS – Hutter, HH, Stützle 07;
  Hutter, HH, Leyton-Brown, Stützle 09)
- Sequential model-based optimisation
  (*e.g.*, SPO – Bartz-Beielstein 06; SMAC – Hutter, HH, Leyton-Brown 11–12)

Iterated Local Search



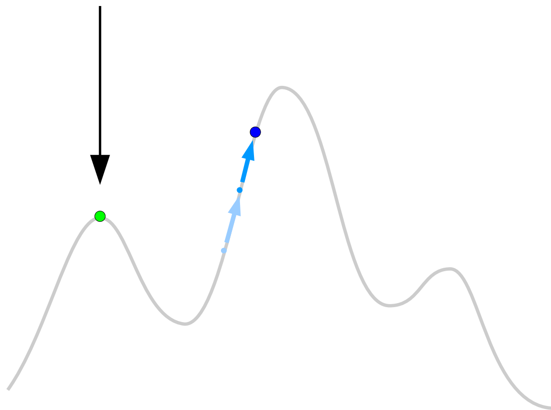
Local Search

Iterated Local Search

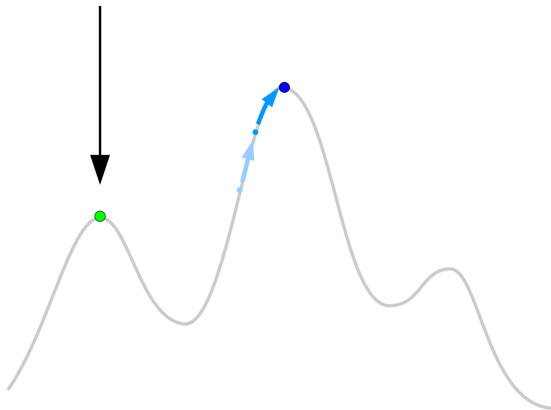


Perturbation

Iterated Local Search

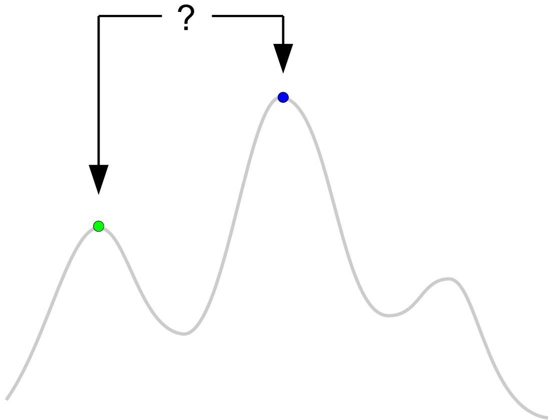


Iterated Local Search



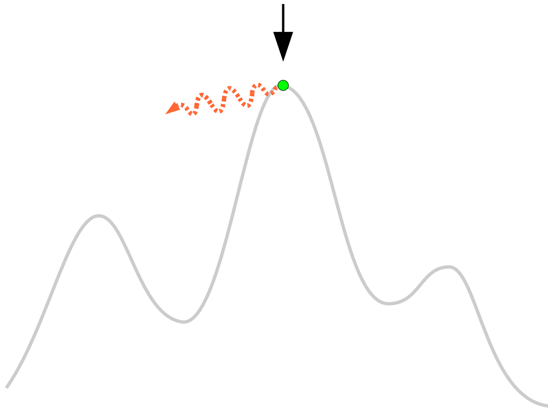
Local Search

Iterated Local Search



Selection (using Acceptance Criterion)

Iterated Local Search



Perturbation

ParamILS

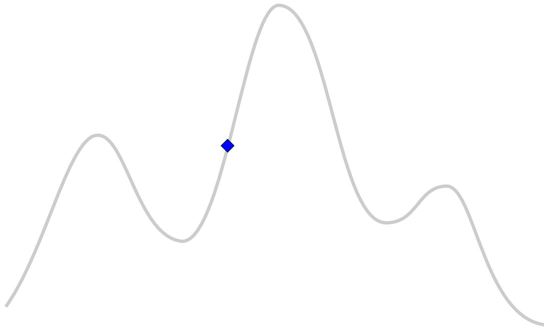
- iterated local search in configuration space
- initialisation: pick *best* of default + R random configurations
- subsidiary local search: iterative first improvement, change one parameter in each step
- perturbation: change s randomly chosen parameters
- acceptance criterion: always select *better* configuration
- number of runs per configuration increases over time; ensure that incumbent always has same number of runs as challengers

Sequential Model-based Optimisation

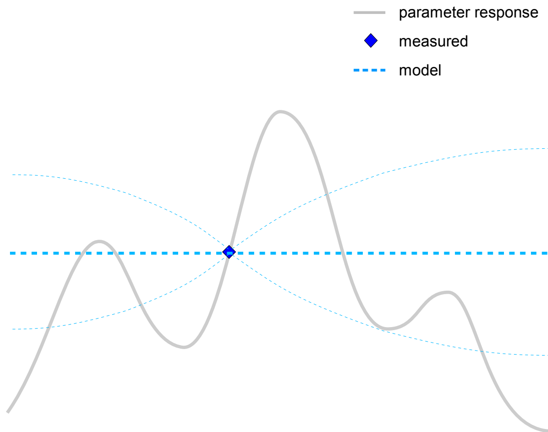
e.g., Jones (1998), Bartz-Beielstein (2006)

- **Key idea:**
  use predictive performance model (response surface model) to find good configurations
- perform runs for selected configurations (initial design) and fit model (*e.g.*, noise-free Gaussian process model)
- iteratively select promising configuration, perform run and update model

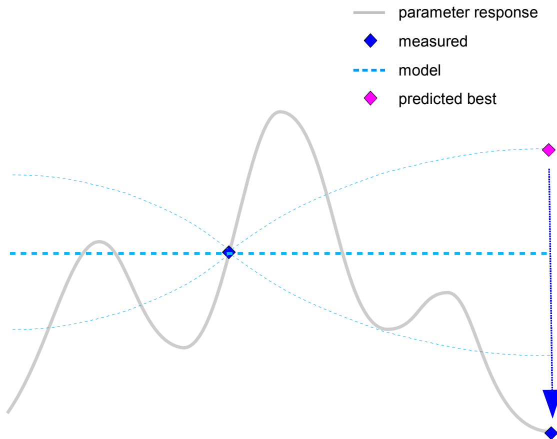
Sequential Model-based Optimisation



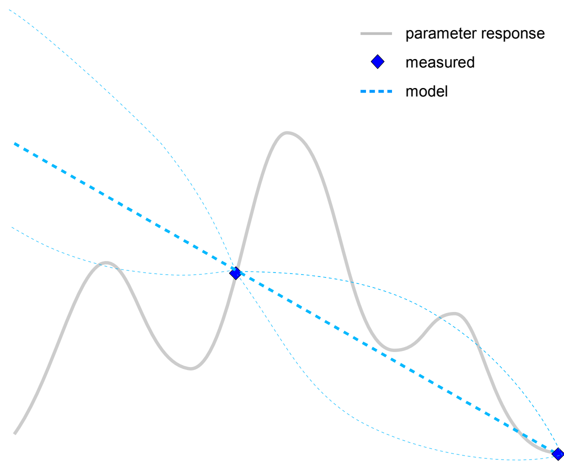
Sequential Model-based Optimisation



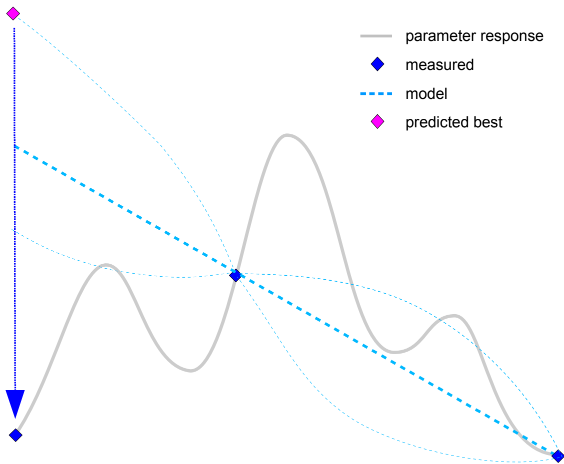
Sequential Model-based Optimisation



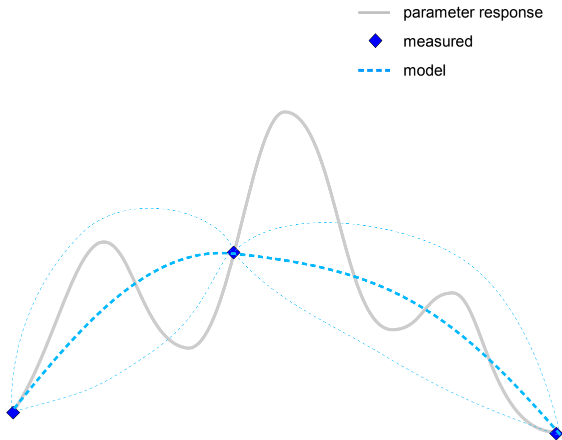
Sequential Model-based Optimisation



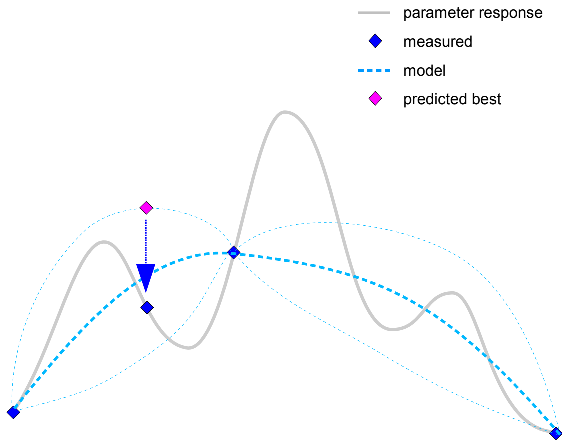
Sequential Model-based Optimisation



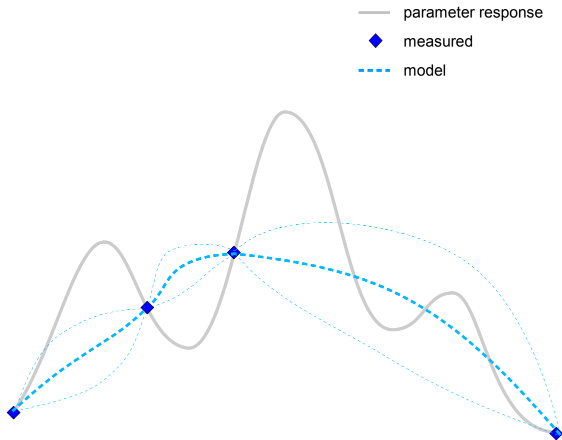
Sequential Model-based Optimisation



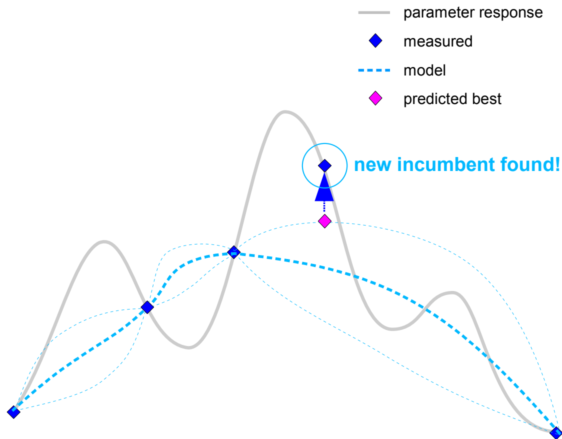
Sequential Model-based Optimisation



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)

Which choices matter? Global perspectives

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

Forward selection based on empirical performance models

Hutter, HH, Leyton-Brown (2013)

Key idea:

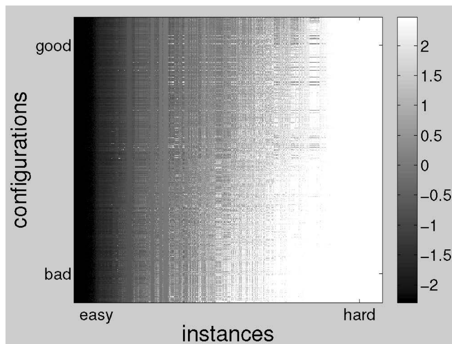
- build regression models of algorithm performance as a function of input parameters (= design choices)

  $\rightsquigarrow$ empirical performance models (EPMs)

- consider only subset of parameters S, ignore all others
- starting with $S = \emptyset$, iteratively add parameters one at a time
- in each iteration, greedily add parameter resuling in max. improvement in accuracy of regression model

EPMs work:

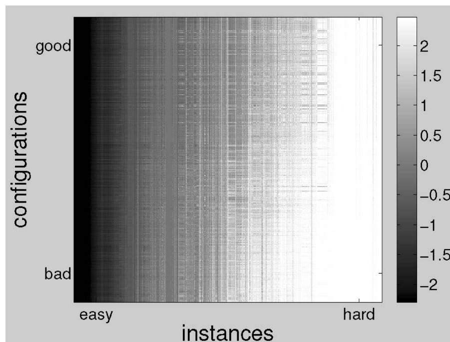
Hutter, HH, Leyton-Brown (to appear in AIJ)



SPEAR on SAT-encoded IBM software verification problems
true running times [$\log_{10}$ CPU sec]

EPMs work:

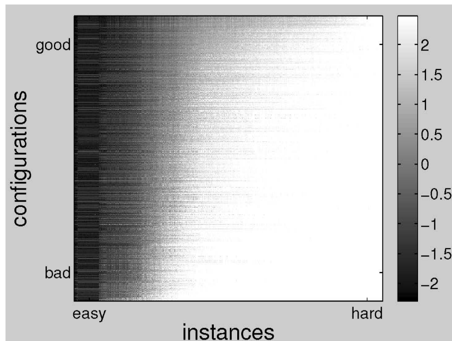
Hutter, HH, Leyton-Brown (to appear in AIJ)



SPEAR on SAT-encoded IBM software verification problems
predicted running times [$\log_{10}$ CPU sec]

EPMs work:

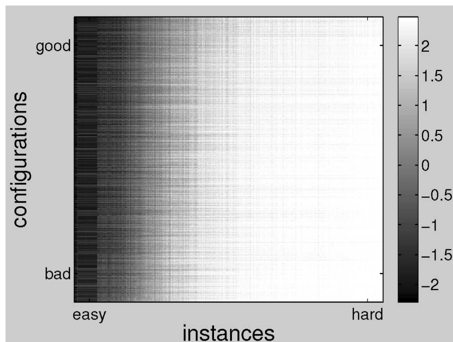
Hutter, HH, Leyton-Brown (to appear in AIJ)



CPLEX 12.1 on MIP problems from computational sustainability
true running times [$\log_{10}$ CPU sec]

EPMs work:

Hutter, HH, Leyton-Brown (to appear in AIJ)

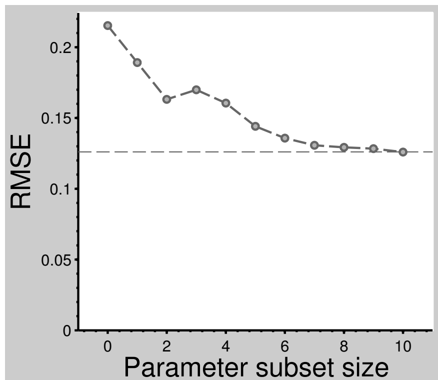


CPLEX 12.1 on MIP problems from computational sustainability
predicted running times [$\log_{10}$ CPU sec]

Empirical study:

- high-performance solvers for SAT, MIP, TSP
  (23–76 parameters),
  well-known sets of benchmark data (real-world structure)
- random forest models for performance prediction,
  trained on 1000 randomly sampled configurations per solver

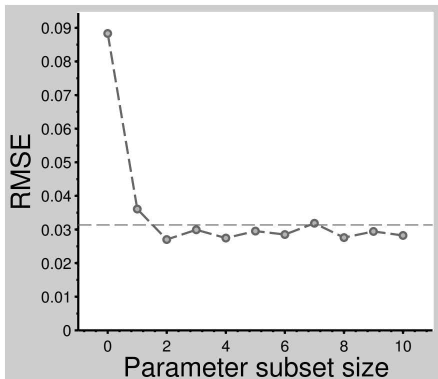
Good prediction accuracies for few parameters:



LK-H on TSPLIB

(RMSE for $\log_{10}$ running times in CPU sec)

Good prediction accuracies for few parameters:

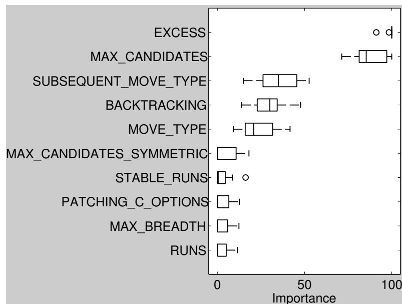


SPEAR on SAT-encoded IBM software verification problems

(RMSE for $\log_{10}$ running times in CPU sec)

How important is each parameter?

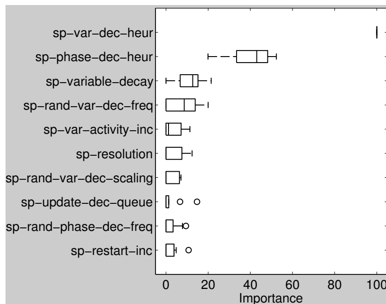
Cost of omission
$=$ impact on model accuracy from excluding single parameters



LK-H on TSPLIB
(normalised to 100 for most important parameter)

How important is each parameter?

Cost of omission
$=$ impact on model accuracy from excluding single parameters



SPEAR on SAT-encoded IBM software verification problems
(normalised to 100 for most important parameter)

Functional ANOVA based on empirical performance models

Hutter, HH, Leyton-Brown (in preparation)

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:

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

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.

Which choices matter? A local perspective

Note: We are mostly interested in good configurations
(obtained from design optimisation)

Questions:

- Which differences between two configurations matter (how much)?
- How close to the default can good performance be obtained?
- How sensitive is an optimised configuration to parameter changes?

Answers may . . .

- guide algorithm development
- facilitate configuration
- improve performance of default configurations
- improve robustness of optimised configurations

Ablation analysis

Fawcett, HH (under review)

Key idea:

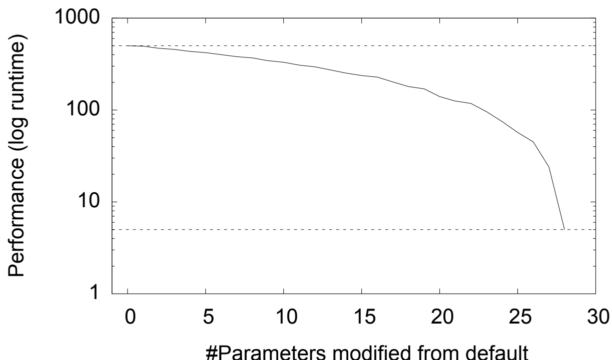
- given two configurations, A and B, change one parameter at a time to get from A to B

  $\rightsquigarrow$ ablation path

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

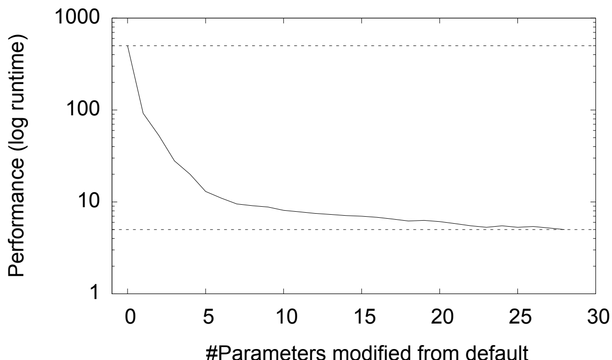
- for computational efficiency, use racing (F-race)
  for evaluating parameters considered in each step

Prototypical ablation results:



all parameters are equally important
(note: log scale!)

Prototypical ablation results:

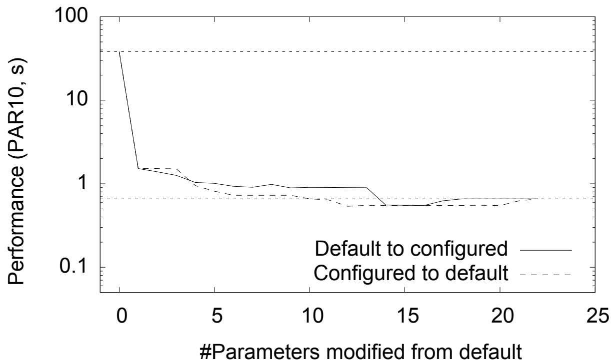


few parameters are most important

Empirical study:

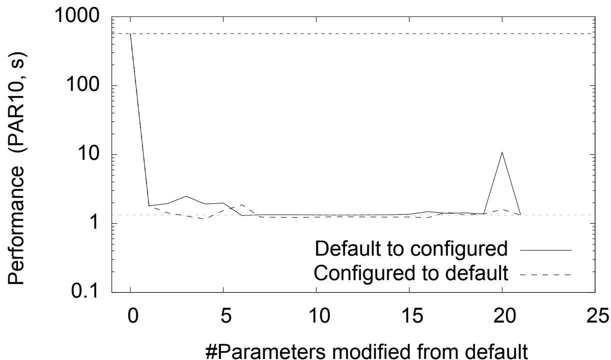
- high-performance solvers for SAT, MIP, AI Planning (26–76 parameters), well-known sets of benchmark data (real-world structure)
- optimised configurations obtained from ParamILS (minimisation of penalised average running time; 10 runs per scenario, 48 CPU hours each)

Ablation between default and optimised configurations:



LPG on Depots planning domain

Ablation between default and optimised configurations:



SPEAR on SAT-encoded IBM software verification problems

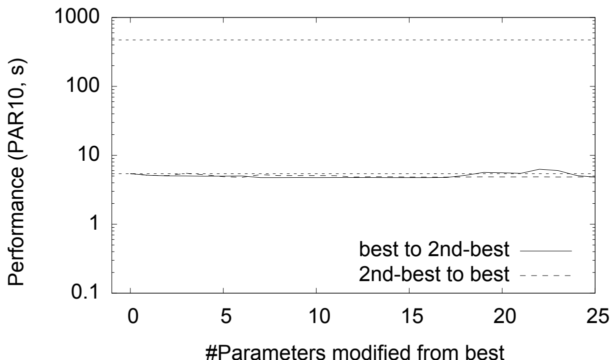
Which parameters are important?

SPEAR on SAT-encoded IBM software verification instances:

- `sp-var-dec-heur` (99.92% of overall performance gain!)
- `sp-rand-var-dec-scaling`
- `sp-res-cutoff-cls`
- `sp-first-restart`

LPG (AI Planning):
Importance of parameters varies between planning domains

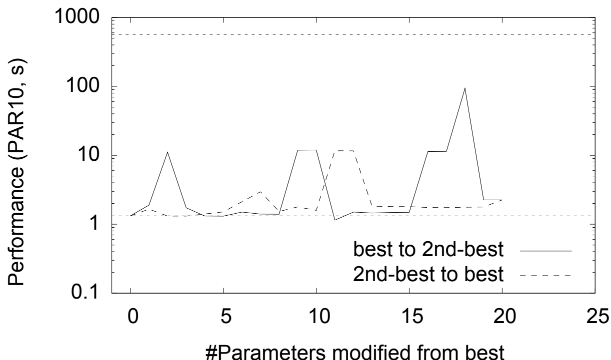
Ablation between optimised configurations:



CPLEX 12.1 on MIP problems from computational sustainability

$\leadsto$ large plateau of good configurations

Ablation between optimised configurations:



SPEAR on SAT-encoded IBM software verification problems

$\rightsquigarrow$ possibility of barriers between good configurations

Speculation and open questions

Optimisation at the meta-level

- candidate solutions are engineering designs
- evaluation is (very) noisy (problem instances)
- evaluation is expensive
- cost of evaluation often depends on quality of candidate solution (*e.g.*, for minimisation of running time)

$\rightsquigarrow$ different methods, different types of structure, different ways to exploit structure

Some hypotheses

HH (2012), Fawcett & HH (under review), Hutter, HH, Leyton-Brown (2013)

- parameters interact, but not too much
- individual parameter responses tend to be well-behaved (uni-modal)
- (few) key parameters need to have certain settings, depending on use context (*vz* backbones)

$\rightsquigarrow$ large, shallow basins around optimised configurations

- for highly parametric algorithms, there are barrier-free paths between optimised configurations
  (*vz* neutral paths in landscapes of RNA secondary structures)

Open questions

- Applicability of / insights from standard landscape analysis techniques?
  (auto-correlation, fitness distance analysis, ...)
- Insights that can be exloited for better design optimisers (configurators)?
- Principles that can guide algorithm developers using PbO to more effectively optimisable designs?

Take-home message:

- exploiting structure in problem instances permits practical solution of hard problems
  $\rightsquigarrow$ instance-level structure
- structure in space of algorithms (+ human creativity) facilitates effective construction of good solvers for hard problems
  $\rightsquigarrow$ meta-level structure
- meta-level structure may differ substantially from instance-level structure
- PbO (rich algorithm design space + automated configuration) permits (partial) automation of building effective solvers;
  efficacy depends on exploitation of meta-level structure