From Stochastic Search to Programming by Optimisation:

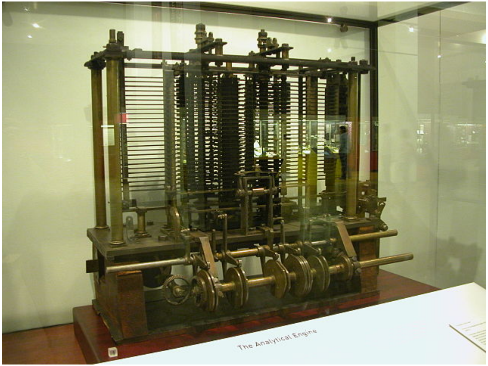
My Quest for Automating the Design of High-Performance Algorithms

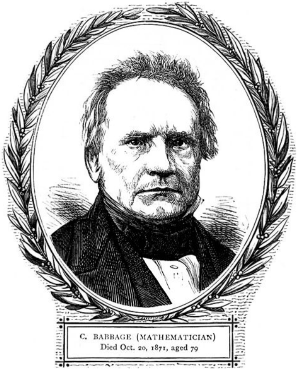
Holger H. Hoos

Department of Computer Science
University of British Columbia
Canada

TAO Seminar
Université de Paris Sud, LRI
2014/10/14

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)

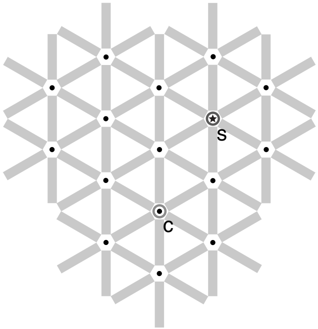
Analysis / Design

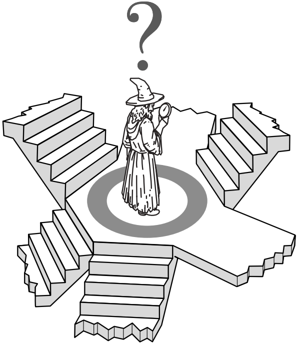
Lesson #1:

Pay attention to theory, but not too much.

1

Stochastic Local Search





Key problem: getting stuck at locally optimal candidate solutions

Remedy:

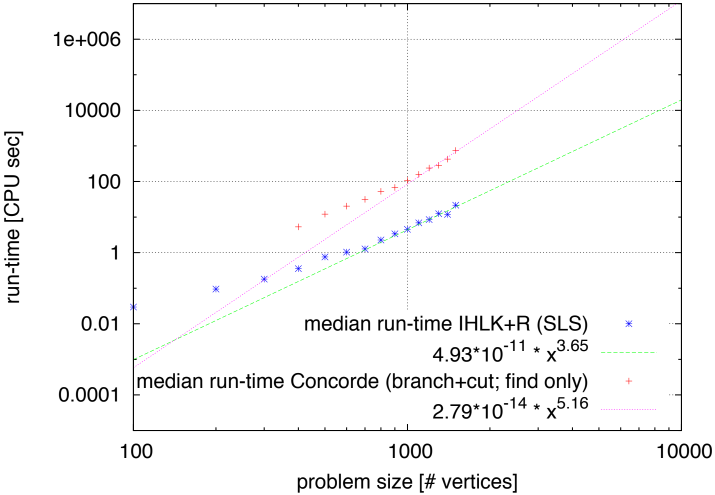
- multiple runs with random initialisation
- randomise search steps

$\leadsto$ balance heuristic guidance (given by evaluation function) and diversification features (often stochastic)

Some prominent SLS methods:

- Random Walk (of theoretical interest)
- Simulated Annealing (inspired by physical model)
- Ant Colony Optimisation (inspired by biological model)
- Iterated Local Search (very successful for TSP, ...)
- ...

SLS vs branch & cut on TSP (RUE benchmark)



Advantages of SLS:

- high performance potential
- broadly applicable, flexible
- typically easy to implement
- anytime behaviour
- easy to parallelise

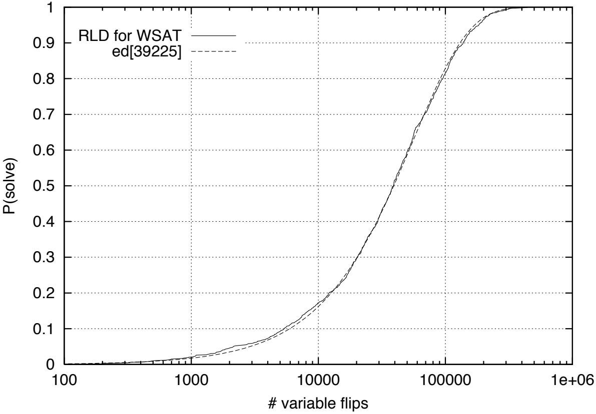
Problems for which I developed SLS algorithms:

- SAT, MAX-SAT
- TSP, QAP
- Combinatorial auction winner determination
- Linear planning
- MPE finding in Bayes nets
- RNA secondary structure design,
  DNA word design,
  protein structure prediction
- Voice separation in music

My methodological work on SLS methods:

- Max-Min Ant System (with Thomas Stützle)
- Empirical properties
- Dynamic parameter adjustment
- Stagnation criteria
- Search space analysis
- Generalised Local Search Machines

WalkSAT has exponential RTDs



$$ed[m] := 1 - 2^{-x/m}$$

Lesson #2:

Don't give up easily
– the best mountains
are hard to climb.

Lesson #3:

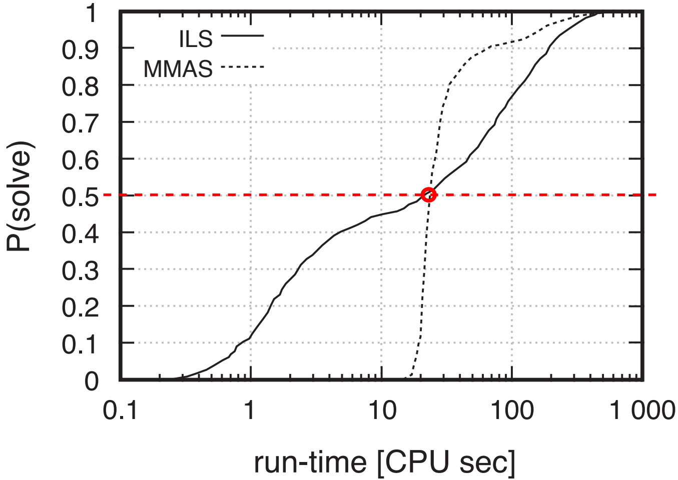
If it looks too good to be true, it typically isn't true.

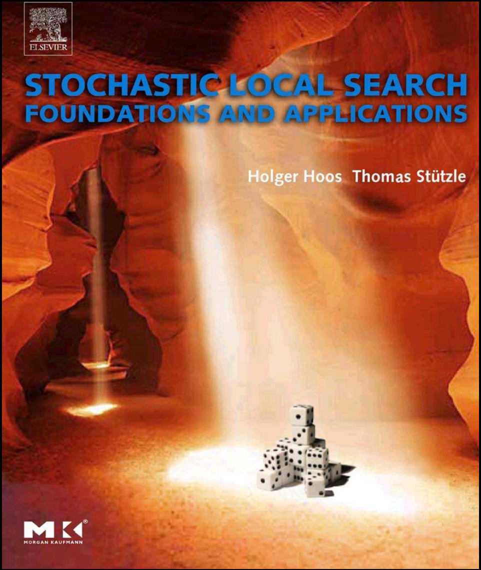
Lesson #4:

Look at the data!

Investigate unexpected behaviour!

Almost identical medians, completely different RTDs!





www.sls-book.net

Lesson #5:

It's never perfect, it's never finished,
– let it go when it's good enough.

2

Modelling the run-time behaviour of Concorde

Hoos & Stützle (EJOR 2014)

Goal:

Study empirical time complexity of solving 2D Euclidean TSP instances using state-of-the-art solver.

Consider two classes of TSP instances:

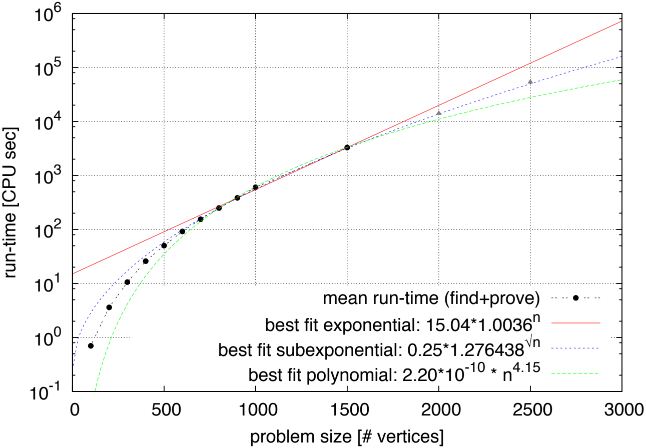
- random uniform Euclidean (RUE)
- TSPLIB (EUC 2D, CEIL 2D, ATT)

State-of-the-art exact TSP solver: Concorde

[Applegate *et al.*, 2003]

- complex heuristic branch & cut algorithm
- iteratively solves series of linear programming relaxations
- uses CLK local search procedure for initialisation

Empirical scaling of running time with input size (state-of-the-art exact TSP solver, Concorde)



RMSE (test): exp = 5820.66, poly = 3058.22, root-exp = 329.79

Statistical validation of scaling model

Compare observed median run-times for Concorde on large TSP instances against 95% bootstrap confidence intervals for predictions

instance size	exponential model	observed median run-time
2 000	[3 793.00 , 5 266.68]	3 400.82 (1000/1000)
3 000	[70 584.38 , 147 716.740]	30 024.49 (99/100)
4 500	[5 616 741.54 , 21 733 073.57]	344 131.05 (65/100)

instance size	polynomial model	root-exponential model
2 000	[2 298.22 , 3 160.39]	[2 854.21 , 3 977.55]
3 000	[9 430.35 , 16 615.93]	[19 338.88 , 49 132.62]
4 500	[38 431.20 , 87 841.09]	[253 401.82 , 734 363.20]

root exponential: $a \cdot b^{\sqrt{n}}$ with $a \in [0.115, 0.373]$, $b \in [1.2212, 1.2630]$

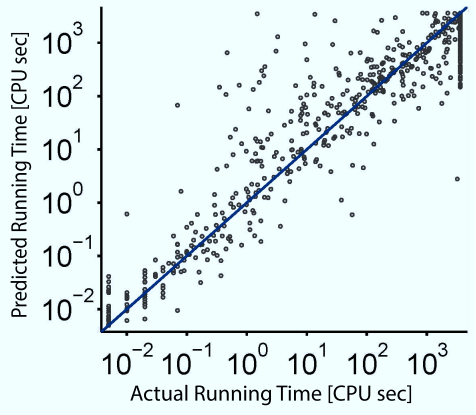
Empirical performance models

Hutter, Xu, HH, Leyton-Brown (AIJ 2014)

Goal:

Predict running time of state-of-the-art solvers for SAT, TSP, MIP on broad classes of instances, using many instance features

MiniSAT 2.0 on SAT Competition Benchmarks
Random Forest Model



Spearman correlation coefficient = 0.90

Instance features:

- Use generic and problem-specific features that correlate with performance and can be computed (relatively) cheaply:
  - number of clauses, variables, ...
  - constraint graph features
  - local & complete search probes

- Use as features statistics of distributions, *e.g.*, variation coefficient of node degree in constraint graph

- For some types of models, consider combinations of features (*e.g.*, pairwise products $\leadsto$ quadratic basis function expansion).

Lesson #6:

Talk to and work with good people.


Frank Hutter
UBC

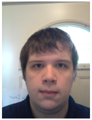

Lin Xu
UBC


Chris Fawcett
UBC


Chris Thornton
UBC


Nima Aghaeepour
UBC


Marius Schneider
U. Potsdam


James Styles
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


Martin Müller
U. of Alberta

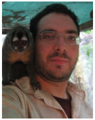

Marco Chiarandini
U. Southern Denmark


Alfonso Gerevini
U. di Brescia


Alessandro Saetti
U. di Brescia


Mauro Vallati
U. di Brescia


Matle Helmert
U. Freiburg


Erez Karpas
Technion

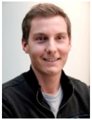

Gabriele Röger
U. Freiburg


Jendrik Seipp
U. Freiburg


Thomas Barz-Beielstein
FH Köln


Eyan Brinkman
BC Cancer Agency


Richard Scheuerman
Craig Venter Institute


Raphael Gottardo
Hutchinson Cancer
Research Center


Greg Finak
Hutchinson Cancer
Research Center


Tim Mosmann
U. of Rochester


Bernd Bischl
TU Dortmund


Heike Trautmann
U. Münster

Lesson #7:

Do something bold and crazy
(every once in a while).

Poly-time prediction of satisfiability

Hutter, Xu, HH, Leyton-Brown (CP 2007)

- **Crazy idea:** Use machine learning techniques to build a poly-time satisfiability predictor
- Sparse Multinomial Logistic Regression (SMLR) on 84 polytime-computable instance features per instance
- **Surprising result:** 73–96% correct predictions on a wide range of SAT benchmark sets!

  (Predictor used in SATzilla, a state-of-the-art, portfolio-based SAT solver developed by Xu, Hutter, HH, Leyton-Brown)

3

Algorithm selection

Rice (1976)

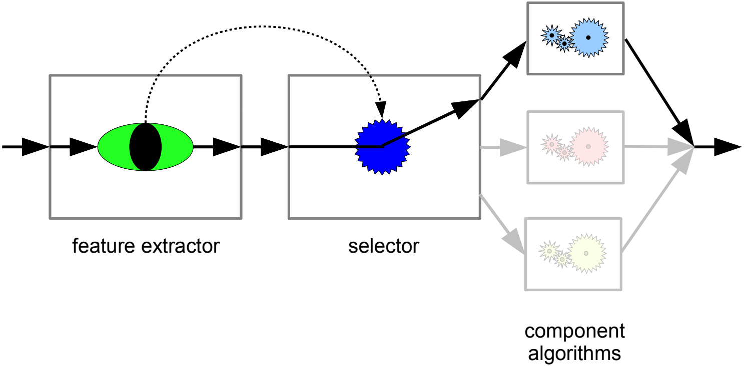
Observation: Different (types of) problem instances are best solved using different algorithms

Idea: Select algorithm to be applied in a given situation from a set of candidates

Per-instance algorithm selection problem:

- *Given:* set A of algorithms for a problem, problem instance π
- *Objective:* select from A the algorithm expected to solve instance π *most efficiently*

Per-instance algorithm selection



Key components:

- set of state-of-the-art *solvers*
  with weakly correlated performance
- 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*)

SATzilla 2011–12

Xu, Hutter, HH, Leyton-Brown (SAT 2012)

- uses cost-based decision forests to select solver based on features
- one predictive model for each pair of solvers (which is better?)
- majority voting (over pairwise predictions) to select solver to be run

$\rightsquigarrow$ 1st prizes in 2 of the 3 main tracks, 2nd in the 3rd main track, 1st in the sequential portfolio track of the 2012 SAT Challenge

4

SAT-based software verification

Hutter, Babic, HH, Hu (2007)

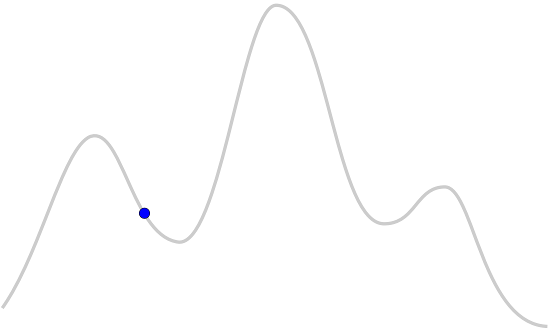
- **Goal:** Solve suite of SAT-encoded software verification instances as fast as possible
- new DPLL-style SAT solver SPEAR (by Domagoj Babic)
  = highly parameterised heuristic algorithm
  (26 parameters, $\approx 8.3 \times 10^{17}$ configurations)
- manual configuration by algorithm designer
- automated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

SPEAR: Empirical results 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

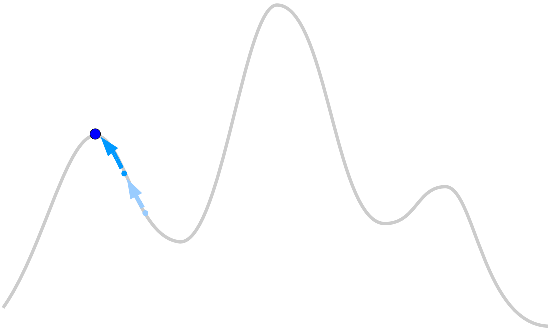
- $\approx$ 500-fold speedup through use automated algorithm configuration procedure (ParamILS)
- new state of the art
  (winner of 2007 SMT Competition, QF_BV category)

Iterated Local Search



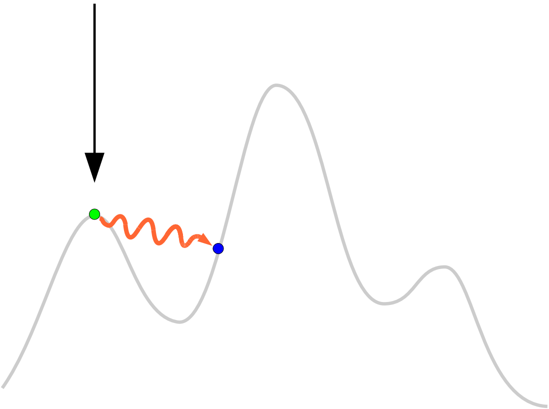
Initialisation

Iterated Local Search



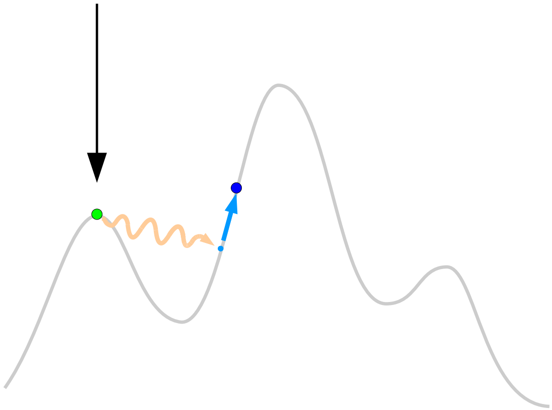
Local Search

Iterated Local Search



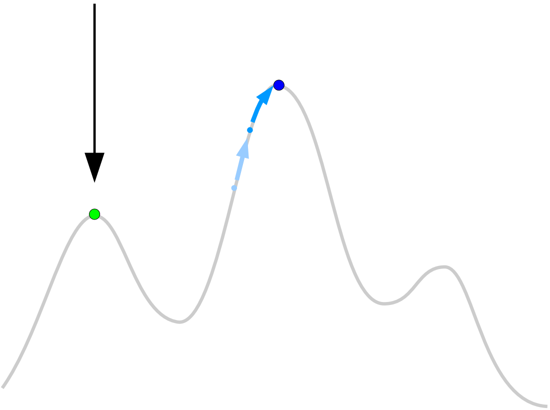
Perturbation

Iterated Local Search



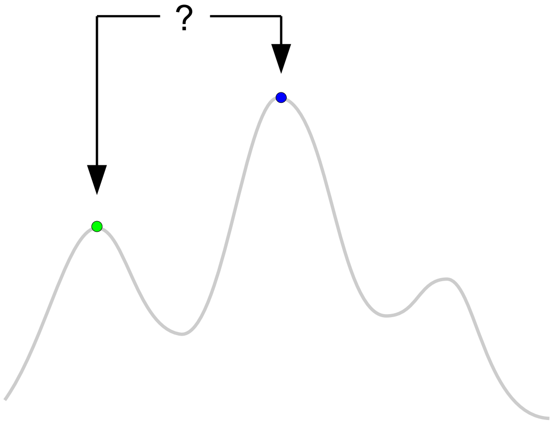
Local Search

Iterated Local Search



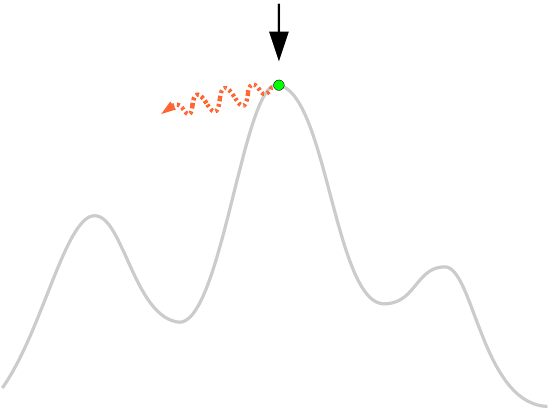
Local Search

Iterated Local Search



Selection (using Acceptance Criterion)

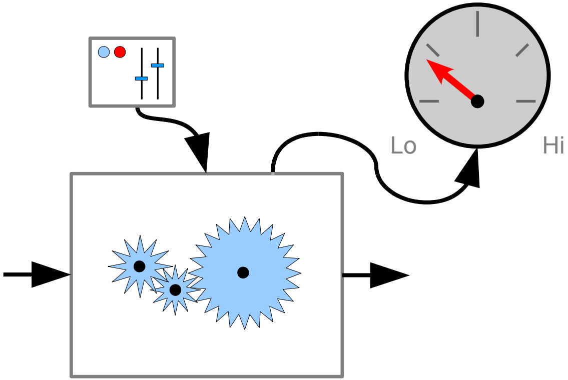
Iterated Local Search

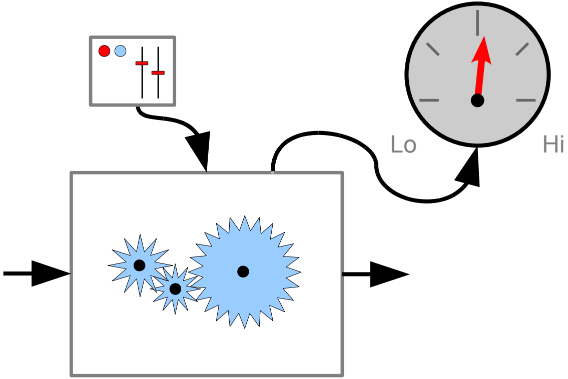


Perturbation

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 (*cf.* racing)


Lo
Hi


Lo
Hi

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)

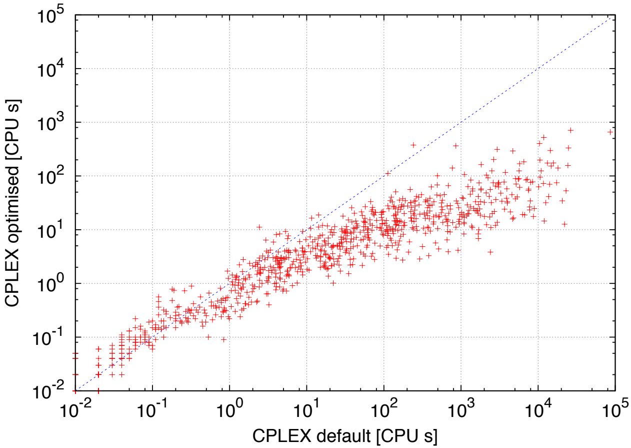
Want: $c^* \in \arg\min_{c \in C} m(A[c], I)$

Algorithm configuration is challenging:

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

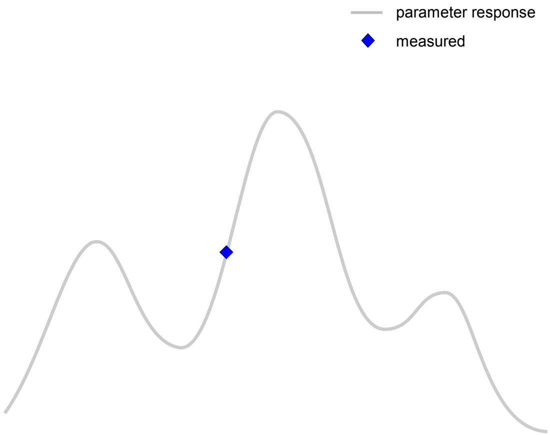
$\rightsquigarrow$ standard optimisation methods are insufficient

CPLEX on Wildlife Corridor Design

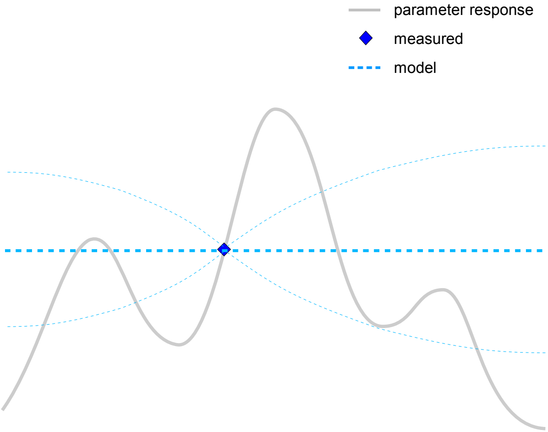


$\rightsquigarrow$ 52.3 $\times$ speedup on average!

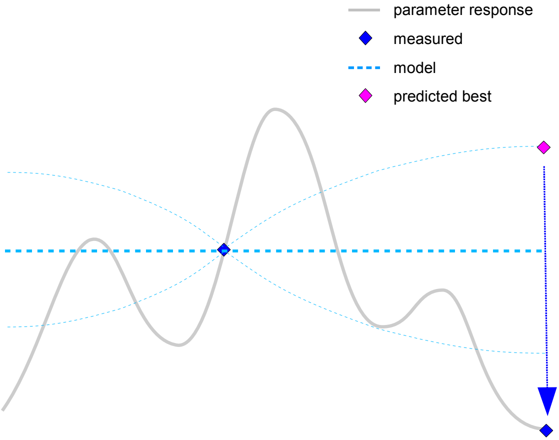
Sequential Model-based Optimisation



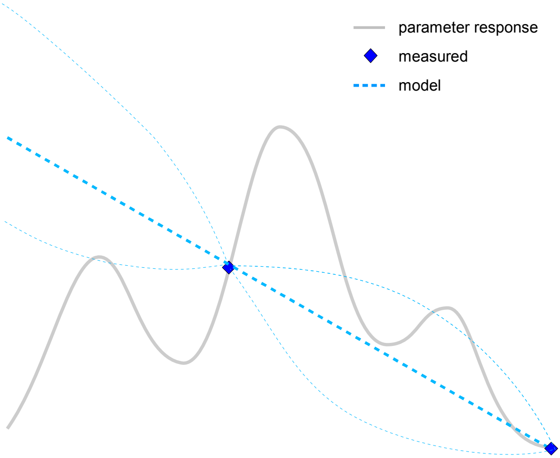
Sequential Model-based Optimisation



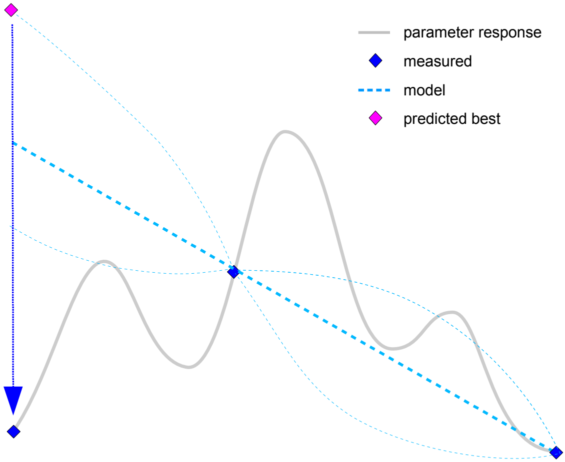
Sequential Model-based Optimisation



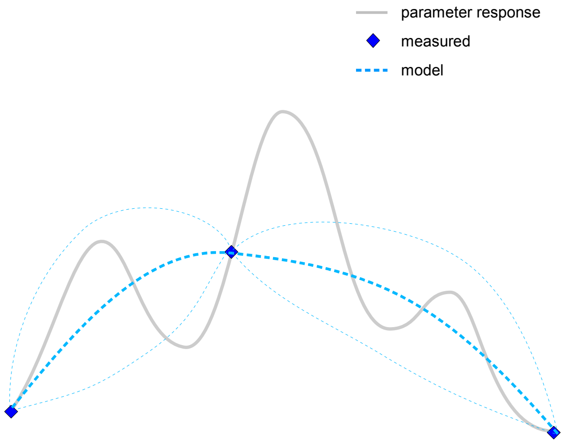
Sequential Model-based Optimisation



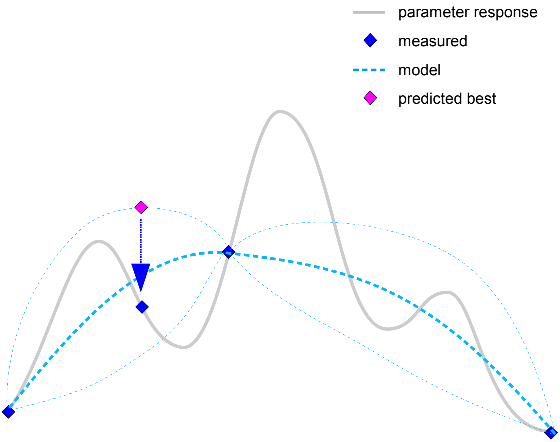
Sequential Model-based Optimisation



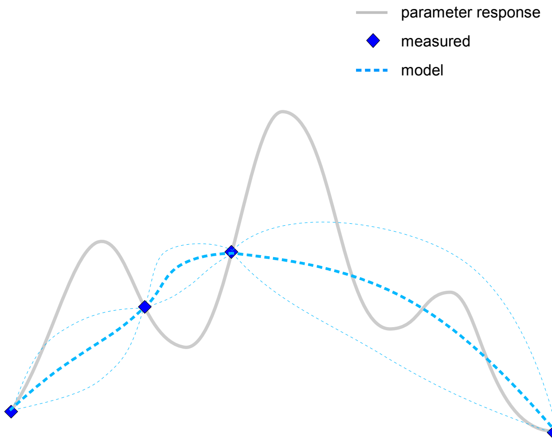
Sequential Model-based Optimisation



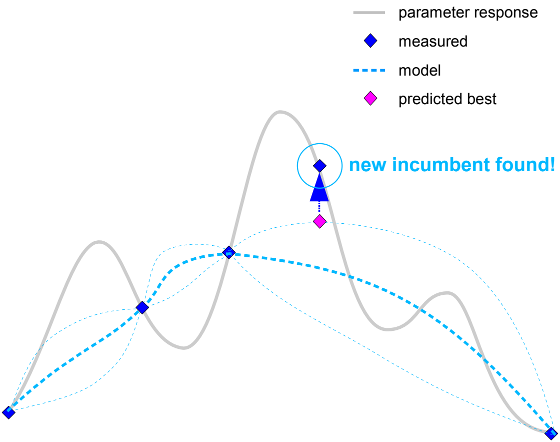
Sequential Model-based Optimisation



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
- impose time-limit for algorithm based on performance observed so far *(adaptive capping)*
- initialisation with single configuration (algorithm default or randomly chosen)

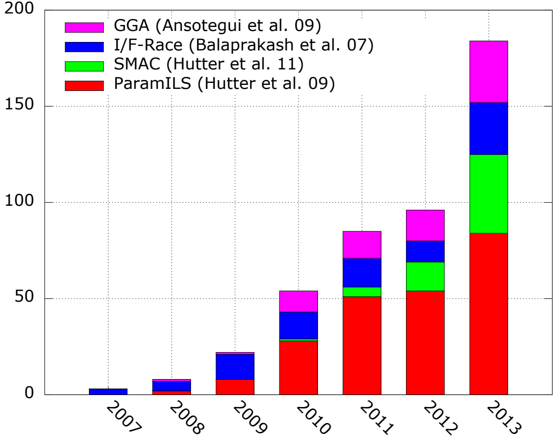
Results for combined selection & configuration of classification algorithms in WEKA (mean error rate in %)

					Auto-WEKA	
Dataset	#Instances	#Features	#Classes	Best Def.	TPE	SMAC
`Semeion`	1115+478	256	10	8.18	8.26	**5.08**
`KR-vs-KP`	2237+959	37	2	0.31	0.54	**0.31**
`Waveform`	3500+1500	40	3	14.40	**14.23**	14.42
`Gisette`	4900+2100	5000	2	2.81	3.94	**2.24**
`MNIST Basic`	12k+50k	784	10	5.19	12.28	**3.64**
`CIFAR-10`	50k+10k	3072	10	64.27	66.01	**61.15**

Auto-WEKA better than full grid search in 15/21 cases

Further details: Thornton, Hutter, HH, Leyton-Brown (KDD 2013)

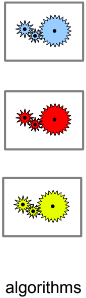
Citations to key publications on algorithm configuration



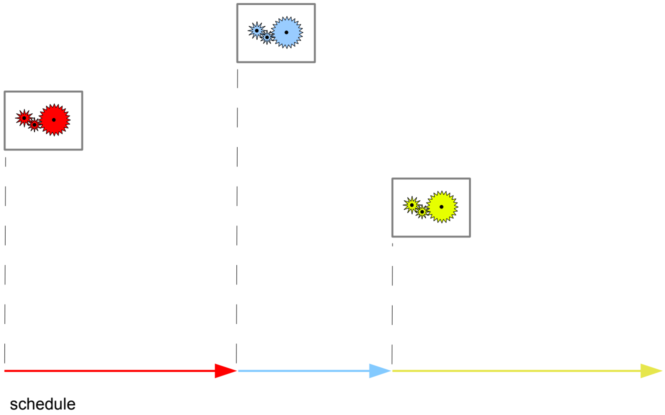
(Data from Google Scholar)

5

Algorithm Scheduling



Algorithm Scheduling



Questions:

1. How to determine that sequence?
2. How much performance can be obtained from solver scheduling only?

Methods for algorithm scheduling methods:

- exhaustive search (as done SATzilla)
  $\rightsquigarrow$ expensive; limited to few solvers, cutoff times

- based on optimisation procedure
  - using integer programming (IP) techniques
    3S – Kadioglu *et al.* (2011)
  - using answer-set-programming (ASP) formulation + solver
    aspeed – HH, Kaminski, Schaub, Schneider (2012)

Empirical result:

Performance of pure scheduling can be suprisingly close to that of combined scheduling + selection (full SATzilla).

HH, Kaminski, Schaub, Schneider (2012);

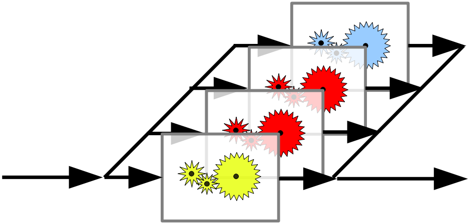
Xu, Hutter, HH, Leyton-Brown (in preparation)

Notes:

- the ASP solver *clasp* used by *aspeed* is powered by a (state-of-the-art) SAT solver core
- pure algorithm scheduling (*e.g.*, *aspeed*) does not require instance features
- sequential schedules can be parallelised easily and effectively

  HH, Kaminski, Schaub, Schneider (2012)

Parallel Algorithm Portfolios



Application to decision problems (like SAT, SMT):

Concurrently run given component solvers until the first of them solves the instance.

$\leadsto$ running time on instance π =
(# solvers) $\times$ (running time of best component solver on π)

Examples:

- ManySAT
  Hamadi, Jabbour, Sais (2009); Guo, Hamadi, Jabbour, Sais (2010)
- Plingeling
  Biere (2010–11)
- ppfolio
  Roussel (2011)

$\leadsto$ excellent performance (see 2009, 2011 SAT competitions)

Constructing portfolios from a single parametric solver

HH, Leyton-Brown, Schaub, Schneider (under review)

Key idea: Take single parametric solver, find configurations that make an effective parallel portfolio.

Note: This allows to automatically obtain parallel solvers from sequential sources *(automatic parallisation)*

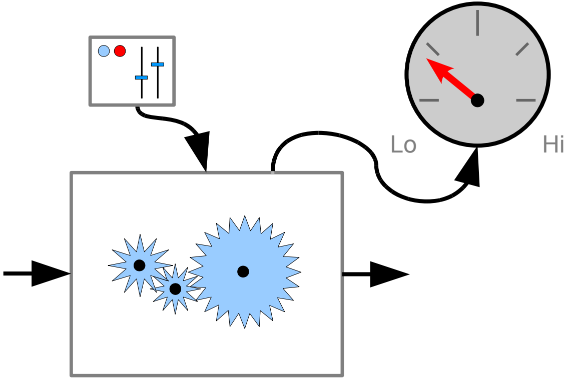
Methods for constructing such portfolios:

- global optimisation:
  simultaneous configuration of all component solvers
- greedy construction:
  add + configure one component at a time

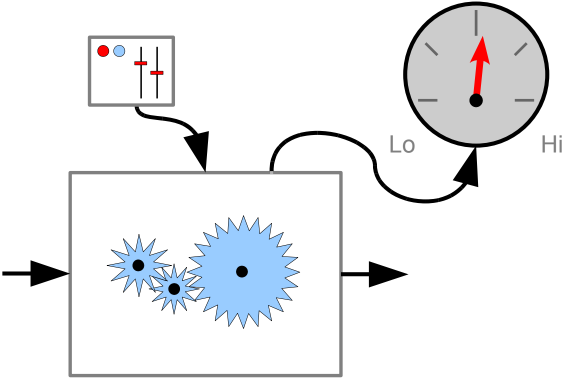
Preliminary results on competition application instances (4 components)

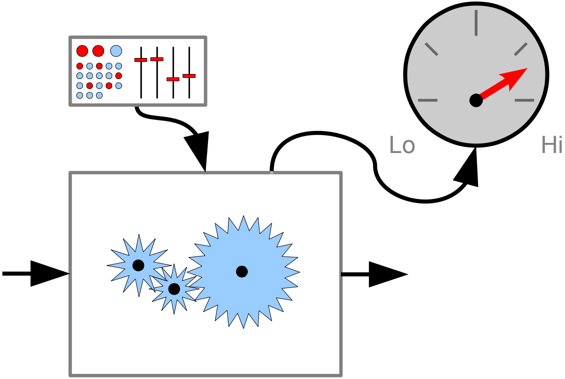
solver	PAR1	PAR10	#timeouts
ManySAT(1.1)	1887	16 003	213/679
ManySAT(2.0)	1998	17 373	232/679
Plingeling (276)	1850	15 437	205/679
Plingeling (587)	1684	13 812	183/679
Greedy-MT4(*Lingeling*)	1717	13 712	181/679
ppfolio	1646	13 310	176/679
CryptoMiniSat	1600	12 271	161/679
VBS over all of the above	1282	10 296	136/679

6



Lo
Hi


Lo
Hi


Lo
Hi

Programming by Optimisation (PbO)

HH (2010 – present)

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)

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:

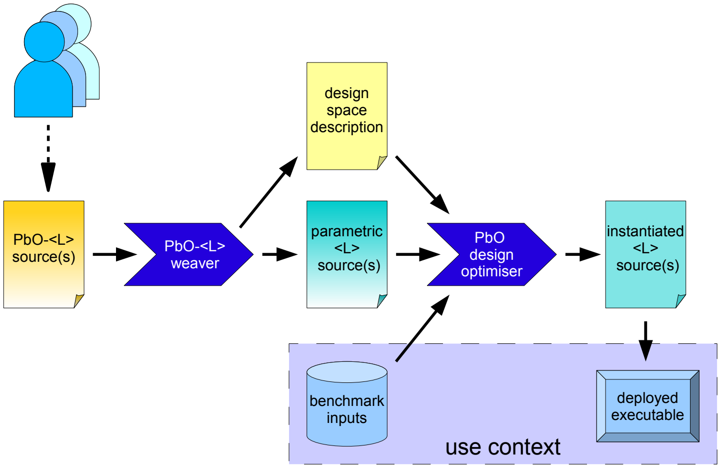
University timetabling, 18 design choices, PbO level 2–3
⇝ new state of the art; UBC exam scheduling
Fawcett, Chiarandini, HH (2009)

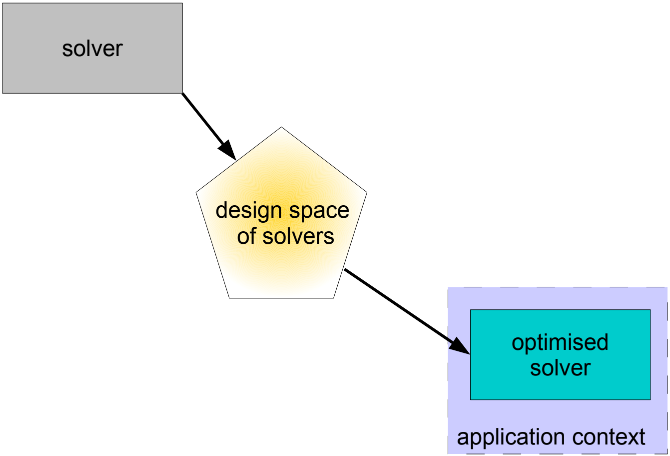
Machine learning / Classification, 786 design choices, PbO level 0–1
⇝ outperforms specialised model selection & hyper-parameter optimisation methods from machine learning
Thornton, Hutter, HH, Leyton-Brown (2012–13)

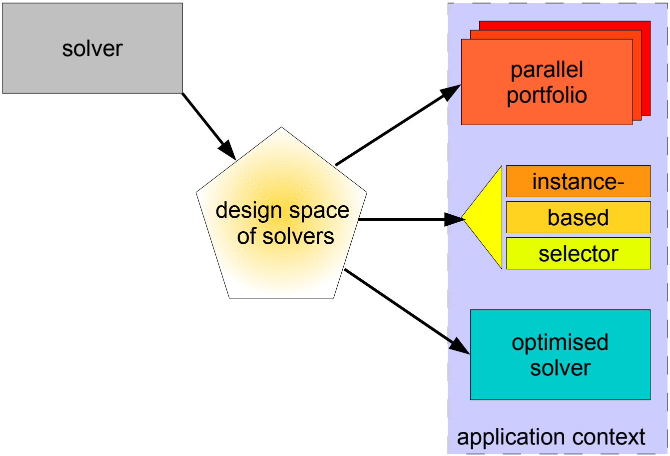
Further successful applications:

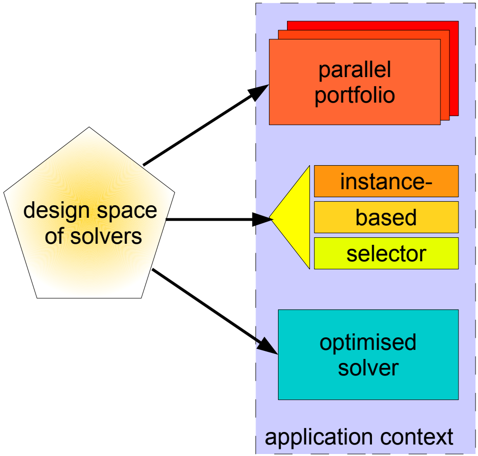
- macro learning in planning (Alhossaini & Beck 2012)
- garbage collection in Java (Lengauer & Mössenböck 2014)
- kidney exchange (Dickerson *et al.* 2012)

Software development in the PbO paradigm




solver
design space
of solvers
optimised
solver
application context


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

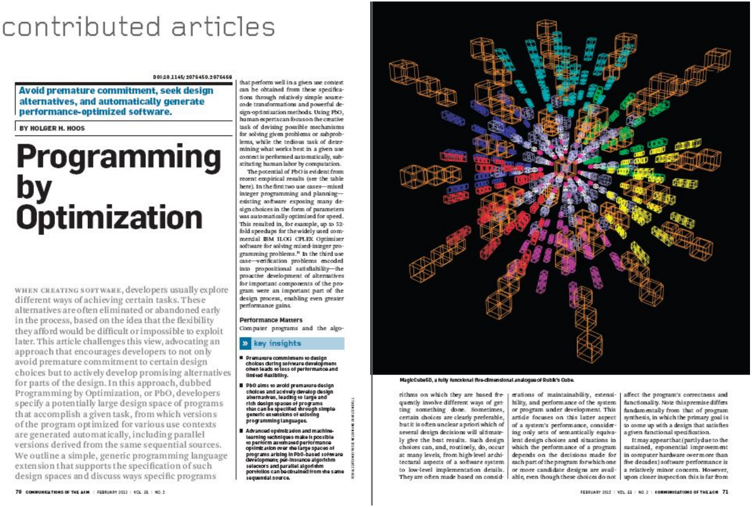

parallel portfolio
design space of solvers
instance-
based
selector
optimised solver
application context

Lesson #8:

Focus on big ideas,
but don't forget
to take care of small details.

Lesson #9:

Don't search for a big idea
– it will come to you, eventually.



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

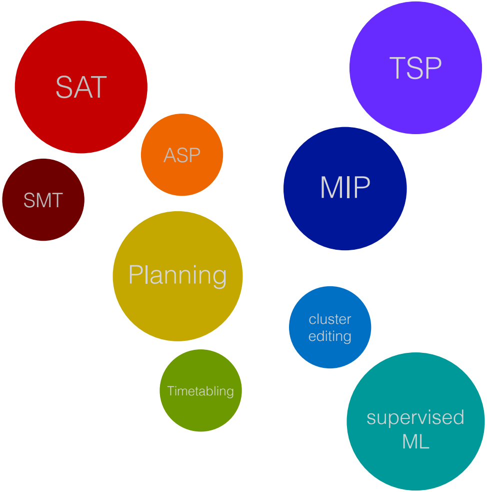
- Premature commitment to design choices during software development often leads to loss of performance and limited flexibility.
- PbO aims to avoid premature design choices and actively develop design alternatives, leading to large and rich design spaces of programs that can be specified through simple 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.

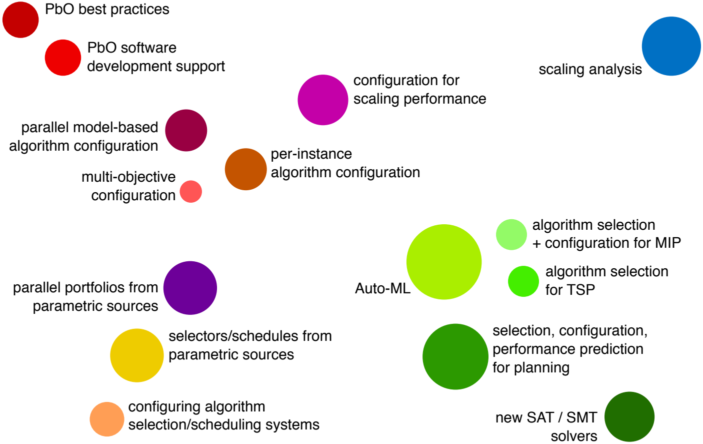
Communications of the ACM, 55(2), pp. 70–80, February 2012

www.prog-by-opt.net

Problems I currently work on



Current research directions/projects

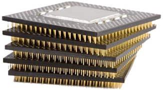


Overall research goal:

Take computation to the next level,

by combining machine learning and optimisation,

human ingenuity and computational power

 + 

Holger H. Hoos

Empirical Algorithmics

Cambridge University Press
(nearing completion)

Lesson #10:

Find your passion and stick with it!

Caminante, no hay camino,
se hace camino al andar.

Traveller, there is no path,
paths are made by walking.

Antonio Machado (1912)

Lessons learnt:

1. Pay attention to theory, but not too much.
2. Don't give up easily – the best mountains are hard to climb.
3. If it looks too good to be true, it typically isn't true.
4. Look at the data! Investigate unexpected behaviour!
5. It's never perfect, it's never finished
   – let it go when it's good enough.
6. Talk to and work with good people.
7. Do something bold and crazy (every once in a while).
8. Focus on big ideas, but don't forget
   to take care of small details.
9. Don't search for a big idea – it will come to you, eventually.
10. Find your passion and stick with it!