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.
Stochastic Local Search
Key problem: getting stuck at locally optimal candidate solutions

Remedy:

- multiple runs with random initialisation
- randomise search steps

⇒ 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)

![Graph showing run-time vs problem size for median run-time IHLK+R (SLS) and median run-time Concorde (branch+cut; find only). The graph includes data points and trend lines.]

- Median run-time IHLK+R (SLS): $4.93 \times 10^{-11} \times x^{3.65}$
- Median run-time Concorde (branch+cut; find only): $2.79 \times 10^{-14} \times x^{5.16}$
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!
Lesson #5:

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

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

<table>
<thead>
<tr>
<th>instance size</th>
<th>polynomial model</th>
<th>root-exponential model</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 000</td>
<td>[2 298.22, 3 160.39]</td>
<td>[2 854.21, 3 977.55]</td>
</tr>
<tr>
<td>3 000</td>
<td>[9 430.35, 16 615.93]</td>
<td>[19 338.88, 49 132.62]</td>
</tr>
<tr>
<td>4 500</td>
<td>[38 431.20, 87 841.09]</td>
<td>[253 401.82, 734 363.20]</td>
</tr>
</tbody>
</table>

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.
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)
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 $\pi$

- **Objective:** select from $A$ the algorithm expected to solve instance $\pi$ *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

⇒ 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
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 \texttt{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

<table>
<thead>
<tr>
<th>solver</th>
<th>num. solved</th>
<th>mean run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MiniSAT 2.0</td>
<td>302/302</td>
<td>161.3 CPU sec</td>
</tr>
<tr>
<td><strong>SPEAR original</strong></td>
<td>298/302</td>
<td>787.1 CPU sec</td>
</tr>
<tr>
<td><strong>SPEAR generic. opt. config.</strong></td>
<td>302/302</td>
<td>35.9 CPU sec</td>
</tr>
<tr>
<td><strong>SPEAR specific. opt. config.</strong></td>
<td>302/302</td>
<td>1.5 CPU sec</td>
</tr>
</tbody>
</table>

- ≈ 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
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 \((\text{cf. racing})\)
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

→ standard optimisation methods are insufficient
CPLEX on Wildlife Corridor Design

\[ \sim 52.3 \times \text{speedup on average!} \]
Sequential Model-based Optimisation

- parameter response
- measured
Sequential Model-based Optimisation

- Parameter response
- Measured
- Model
Sequential Model-based Optimisation

- parameter response
- measured
- model
- predicted best
Sequential Model-based Optimisation

- parameter response
- measured
- model
Sequential Model-based Optimisation

- parameter response
- measured
- model
- predicted best
Sequential Model-based Optimisation

parameter response
measured
model
Sequential Model-based Optimisation

Diagram illustrating parameter response, measured, model, and predicted best values.
Sequential Model-based Optimisation

- parameter response
- measured
- model
Sequential Model-based Optimisation

- Parameter response
- Measured
- Model
- Predicted best

New incumbent found!
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 %)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Instances</th>
<th>#Features</th>
<th>#Classes</th>
<th>Best</th>
<th>Def.</th>
<th>TPE</th>
<th>SMAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semeion</td>
<td>1115+478</td>
<td>256</td>
<td>10</td>
<td>8.18</td>
<td>8.26</td>
<td>5.08</td>
<td></td>
</tr>
<tr>
<td>KR-vs-KP</td>
<td>2237+959</td>
<td>37</td>
<td>2</td>
<td>0.31</td>
<td>0.31</td>
<td>0.31</td>
<td></td>
</tr>
<tr>
<td>Waveform</td>
<td>3500+1500</td>
<td>40</td>
<td>3</td>
<td>14.40</td>
<td>14.23</td>
<td>14.42</td>
<td></td>
</tr>
<tr>
<td>Gisette</td>
<td>4900+2100</td>
<td>5000</td>
<td>2</td>
<td>2.81</td>
<td>3.94</td>
<td>2.24</td>
<td></td>
</tr>
<tr>
<td>MNIST Basic</td>
<td>12k+50k</td>
<td>784</td>
<td>10</td>
<td>5.19</td>
<td>5.19</td>
<td>3.64</td>
<td></td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>50k+10k</td>
<td>3072</td>
<td>10</td>
<td>64.27</td>
<td>66.01</td>
<td>61.15</td>
<td></td>
</tr>
</tbody>
</table>

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)
Algorithm Scheduling

algorithms
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)
  \[ \sim \text{expensive; limited to few solvers, cutoff times} \]

- based on optimisation procedure
  - using integer programming (IP) techniques
    \[ 3S - \text{Kadioglu et al. (2011)} \]
  - using answer-set-programming (ASP) formulation + solver
    \[ \text{aspeed} - \text{HH, Kaminski, Schaub, Schneider (2012)} \]
Empirical result:

Performance of pure scheduling can be surprisingly 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.

\[ \text{running time on instance } \pi = (\# \text{ solvers}) \times (\text{running time of best component solver on } \pi) \]

Examples:

- **ManySAT**
  Hamadi, Jabbour, Sais (2009); Guo, Hamadi, Jabbour, Sais (2010)

- **Plingeling**
  Biere (2010–11)

- **ppfolio**
  Roussel (2011)

\[ \text{excellent performance (see 2009, 2011 SAT competitions)} \]
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 parallelisation*).

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)

<table>
<thead>
<tr>
<th>solver</th>
<th>PAR1</th>
<th>PAR10</th>
<th>#timeouts</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManySAT (1.1)</td>
<td>1887</td>
<td>16003</td>
<td>213/679</td>
</tr>
<tr>
<td>ManySAT (2.0)</td>
<td>1998</td>
<td>17373</td>
<td>232/679</td>
</tr>
<tr>
<td>Plingeling (276)</td>
<td>1850</td>
<td>15437</td>
<td>205/679</td>
</tr>
<tr>
<td>Plingeling (587)</td>
<td>1684</td>
<td>13812</td>
<td>183/679</td>
</tr>
<tr>
<td>Greedy-MT4 (Lingeling)</td>
<td>1717</td>
<td>13712</td>
<td>181/679</td>
</tr>
<tr>
<td>ppfolio</td>
<td>1646</td>
<td>13310</td>
<td>176/679</td>
</tr>
<tr>
<td>CryptoMiniSat</td>
<td>1600</td>
<td>12271</td>
<td>161/679</td>
</tr>
<tr>
<td>VBS over all of the above</td>
<td>1282</td>
<td>10296</td>
<td>136/679</td>
</tr>
</tbody>
</table>
Programming by Optimisation (PbO)

HH (2010 – present)

Key idea:

- program $\mapsto$ (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:

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

... 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

PbO-<L> source(s) → PbO-<L> weaver → parametric <L> source(s) → PbO design optimiser → instantiated <L> source(s)

design space description

benchmark inputs

use context

deployed executable

Holger Hoos: From Stochastic Search to Programming by Optimisation 56
solver

design space of solvers

optimised solver

application context
Parallel portfolio

Instance-based selector

Optimised solver

Design space of solvers

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.
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 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 PO), 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 for a given use context can be obtained from these specifications through relatively simple automated transformation rules and powerful design optimization methods. Using PO, business processes become the central focus of driving possible improvements for solving given problems and 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 PO is evidenced by recent empirical results (see the table here). In this finite state space—mixed integer programming and planning—existing software requiring many design choices in the form of parameters and heuristically specified for speed. This resulted in, for example, up to 50-fold speedup for the widely used commercial B2B Optimizer software for analyzing mixed integer programming problems. In the third use case—optimal design models for distributed software systems components of the program, new important part of the design process, enabling new, greater performance gains.

Performance Metrics

Compute graphs and the algorithm:

**Top Insights**

- Trade-offs in design choices during software development
- Architecture level optimization of programs
- Automatic generation of parallel programs

Advantages of Optimization: Design choices can be systematically explored and the best one chosen, which can significantly reduce the complexity of software development.

The PO approach allows developers to explore a large design space and select the most promising solutions automatically, leading to more efficient and effective software development processes.
Problems I currently work on

SAT
MIP
TSP
ASP
SMT
Planning
Timetabling
supervised ML
cluster editing
Current research directions/projects

- PbO best practices
- PbO software development support
- parallel model-based algorithm configuration
- multi-objective configuration
- parallel portfolios from parametric sources
- selectors/schedules from parametric sources
- configuring algorithm selection/scheduling systems
- configuration for scaling performance
- per-instance algorithm configuration
- Auto-ML
- algorithm selection + configuration for MIP
- algorithm selection for TSP
- selection, configuration, performance prediction for planning
- new SAT / SMT solvers
- scaling analysis
Overall research goal:

Take computation to the next level,
by combining machine learning and optimisation,
human ingenuity and computational power
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!