

**Computer-Aided Design
of High-Performance Algorithms:
Principled Procedures
for Building Better Solvers**

Holger H. Hoos

BETA Lab
Department of Computer Science
University of British Columbia
Canada

Tutorial, 2nd International Workshop on Engineering SLS Algorithms
Brussels, Belgium, September 2009

High-performance heuristic algorithms are difficult to design

- ▶ many design choices (representation / search space; neighbourhoods; search strategy; variable/value selection heuristic; restart rules; pre-processing; data structures; ...)
- ▶ best performance often achieved by combination of various heuristics (e.g., best improvement + random restart, multi-phase search, systematic search + preprocessing, iterated local search, local + systematic search)
- ▶ various heuristic components interact in complex ways
 ~> unexpected, emergent behaviour
- ▶ performance can be tricky to assess due to
 - ▶ differences in behaviour across problem instances
 - ▶ stochasticity

Therefore ...

- ▶ time-consuming design process, success often critically dependent on experience, intuition, luck
- ▶ resulting algorithms often complex, somewhat ad-hoc, not fully optimised

Real-world example:

- ▶ Application: Solving SAT-encoded software verification problems
- ▶ Given: High-performance DPLL-type SAT solver (SPEAR)
 - ▶ 26 parameters (7 categorical, 3 Boolean, 12 continuous, 4 integer-valued)
 - ▶ control variable/value ordering heuristics, clause learning, restarts, ...
- ▶ Goal: Minimize expected run-time on 'typical' SAT instances from software verification tool
- ▶ Problems:
 - default settings $\rightsquigarrow \approx 300$ seconds / run
 - good performance on a few instances may not generalise

Outline

1. Introduction
2. From traditional to computer-aided algorithm design
3. Design spaces and design patterns
4. Meta-algorithmic search and optimisation procedures
5. Selected computer-aided algorithm design procedures
6. Towards a software environment for computer-aided design of high-performance algorithms

From traditional to computer-aided algorithm design

Traditional algorithm design approach:

- ▶ iterative, manual process
- ▶ designer gradually introduces/modifies components or mechanisms
- ▶ test performance on benchmark instances
- ▶ design often starts from generic or broadly applicable problem solving method (e.g., evolutionary algorithm)

Note:

- ▶ During the design process, many decisions are made.
- ▶ Some choices take the form of parameters, others are hard-coded.
- ▶ Design decisions interact in complex ways.

Problems:

- ▶ Design process is labour-intensive.
- ▶ Design decisions often made in *ad-hoc* fashion, based on limited experimentation and intuition.
- ▶ Human designers typically over-generalise observations, explore few designs.
- ▶ Implicit assumptions of independence, monotonicity are often incorrect.
- ▶ Number of components and mechanisms tends to grow in each stage of design process.

↪ complicated designs, unfulfilled performance potential

Solution: Computer-aided Algorithm Design

- ▶ Goal: construct high-performance algorithms automatically
- ▶ Key idea: use fully formalised procedures to effectively explore large space of candidate designs

↪ genetic programming, hyper-heuristics, learning and intelligent optimisation, SLS engineering

Human designer:

- ▶ specifies (possibly large) space of candidate algorithm design
- ▶ supplies set of problem instances for performance evaluation
- ▶ specifies performance metric

Meta-algorithmic system:

- ▶ explores design space in principled manner
- ▶ evaluates candidate design
- ▶ finds high-performance designs

Advantages:

- ▶ lets human designer focus on higher-level issues
- ▶ enables better exploration of larger design spaces
- ▶ exploits complementary strengths of different approaches for solving a given problem
- ▶ uses principled, fully formalised methods for algorithm design
- ▶ can be used to customise algorithms for use in specific applications with minimal human effort

Example: SAT-based software verification

Babic, Hutter, Hu – FMCAD'07

- ▶ **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 ParamLLS, a generic algorithm configuration procedure

[Hutter, Hoos, Stützle – AAAI'07]

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)

Design spaces and design patterns

Special cases of computer-aided algorithm design:

- ▶ **parameter optimisation (for given set of instances)**

Birattari et al. 2002; Adenso-Diaz & Laguna 2006, Hutter et al. 2007;
Bartz-Beielstein 2006

- ▶ **algorithm configuration from components
(for given set of instances)**

Fukunaga 2002, Chiarandini et al. 2008, KhudaBukhsh et al. 2009

- ▶ **restart strategies**

Luby et al. 1993; Gagliolo & Schmidhuber 2007

Special cases of computer-aided algorithm design (2):

- ▶ instance-based algorithm configurators

Hutter et al. 2006

- ▶ on-line algorithm control / reactive search

Carchrae & Beck 2005; Battiti et al. 2008

- ▶ instance-based algorithm selection

Rice 1976; Leyton-Brown et al. 2003; Guerri & Milano 2004;

Xu et al. 2008

- ▶ algorithm portfolios (static and dynamic)

Huberman et al. 1997, Gomes & Selman 2001;

Gagliolo & Schmidhuber 2007

↪ meta-algorithmic design patterns, induce design spaces

Meta-algorithmic search and optimisation procedures

How to search design spaces?

- ▶ use powerful heuristic search and optimisation procedures, combined with significant amounts of computing power
- ▶ use machine learning methods (classification, regression), combined with significant amount of training data

Some examples:

- ▶ parameter tuning:
 - ▶ numerical optimisation techniques
e.g., CMA-ES (Hansen & Ostermeier 2001)
 - ▶ model-based optimisation methods
e.g., SPO (Bartz-Beielstein 2006),
SPO⁺ (Hutter et al. 2009)
- ▶ algorithm configuration:
 - ▶ genetic programming
e.g., CLASS (Fukunaga 2002)
 - ▶ racing procedures
e.g., F-Race (Birattari et al. 2002)
 - ▶ advanced stochastic local search procedures
e.g., ParamILS (Hutter et al. 2007)

More examples:

- ▶ instance-based algorithm selection
 - ▶ classification approaches (e.g., Guerri & Milano 2004)
 - ▶ regression approaches (e.g., Leyton-Brown et al. 2003, Xu et al. 2008)
- ▶ dynamic algorithm portfolios (time allocators)
 - ▶ bandit solvers (e.g., Gagliolo & Schmidhuber 2007)
 - ▶ evolutionary algorithms (e.g., Harick & Lobo 1999)

Many open questions:

- ▶ Which procedure for which type of design space?
- ▶ How to deal with hybrid design patterns?
- ▶ How to best deal with censored, sparse data?

Selected computer-aided algorithm design procedures:

- ▶ F-Race / Iterative F-Race

Birattari, Stützle, Paquete, Varrentrapp (2002);

Balaprakash, Birattari, Stützle (2007)

- ▶ ParamILS

Hutter, Hoos, Stützle (2007); Hutter, Hoos, Leyton-Brown, Stützle (2009)

- ▶ SPO / SPO⁺

Bartz-Beielstein (2006); Hutter, Hoos, Leyton-Brown (2009)

- ▶ SATzilla

Xu, Hutter, Hoos, Leyton-Brown (2008)

Automated algorithm selection/configuration using F-Race

Birattari, Stützle, Paquete, Varrentrapp (2002); Balaprakash, Birattari, Stützle (2007)

Key idea:

- ▶ *Given*: set S of algorithms for a problem, set of problem instances Π
- ▶ *Select* from S the algorithm expected to solve instances from Π *most efficiently* on average

F-Race (Birattari, Stützle, Paquete, Varrentrapp 2002)

- ▶ inspired by methods for model selection methods in machine learning

(Maron & Moore 1994; Moore & Lee 1994)

- ▶ sequentially evaluate algorithms/configuration, in each iteration, perform one new run per algorithm/configuration
- ▶ eliminate poorly performing algorithms/configurations as soon as sufficient evidence is gathered against them
- ▶ use Friedman test to detect poorly performing algorithms/configurations

Iterative F-Race (Balaprakash, Birattari, Stützle 2007)

Problem: When using F-Race for algorithm configuration, number of initial configurations considered is severely limited.

Solution:

- ▶ perform *multiple iterations of F-Race on limited set of configurations*
- ▶ sample candidate configurations based on *probabilistic model* (independent normal distributions centred on surviving configurations)
- ▶ gradually reduce variance over iterations (*volume reduction*)

↪ good results for

- MAX-MIN Ant System for the TSP (6 parameters)
- simulated annealing for stochastic vehicle routing (4 parameters)
- estimation-based local search for PTSP (3 parameters)

Automated algorithm configuration using ParamILS

Hutter, Hoos, Stützle (2007); Hutter, Hoos, Leyton-Brown, Stützle (2009)

Key idea:

- ▶ *Given*: parameterised algorithm A for a problem, set of problem instances Π
- ▶ *Select* parameter values of A to solve instances from Π *most efficiently based on search in configuration space*

Goal: Apply to algorithms with

- ▶ many parameters, relatively few instances.
- ▶ categorical parameters

ParamILS (Hutter, Hoos, Stützle 2007)

- ▶ initialisation: pick *best* of default + R random configurations
- ▶ iterated local search in configuration space
- ▶ 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 'new' configurations

Some example applications:

- ▶ SLS for 2D/3D HP protein structure prediction (5 parameters)
Thachuk, Shmygelska, Hoos (2007)
- ▶ DPLL for SAT-encoded software verification (26 parameters)
Hutter, Babic, Hoos, Hu (2007)
- ▶ CPLEX for mixed integer programming (63 parameters)
Hutter, Hoos, Leyton-Brown, Stützle (2009)
- ▶ University timetabling (7+11 parameters)
Chiarandini, Fawcett, Hoos (2008); [Fawcett, Chiarandini, Hoos \(2009\)](#)

↪ substantial improvements in state of the art for solving these (and other) problems

Model-based parameter tuning using Sequential Parameter Optimisation

Bartz-Beielstein (2006); Hutter, Hoos, Leyton-Brown (2009)

Key idea:

- ▶ *Given*: parameterised algorithm A for a problem, set of problem instances Π
- ▶ *Select* parameter values of A to solve instances from Π *most efficiently* based on *predictive performance model*

Sequential Parameter Optimisation (SPO):

Bartz-Beielstein (2006)

- ▶ perform runs for selected configurations (initial design) and fit (noise-free Gaussian process) model
- ▶ iteratively select promising configuration C , run A using C and update model
- ▶ initial design: *Latin Hypercube Design (LHD)*
- ▶ use *expected improvement criterion* to select promising configurations
- ▶ *intensification mechanism*:
 - ▶ gradually increase number of runs for each configuration;
 - ▶ ensure that incumbent always has same number of runs as 'new' configurations

Latest variant: SPO⁺ (Hutter, Hoos, Leyton-Brown 2009)

- ▶ model/predict *log-transformed* performance data
- ▶ modified intensification mechanism ensures that sufficiently many runs are performed before changing incumbent

Example applications:

- ▶ CMA-ES (state-of-the-art continuous optimisation procedure)
- ▶ SAPS (high-performance SAT algorithm)

↪ substantial performance improvements over default configurations

(Ongoing work on better handling of tuning over multiple instances.)

Instance-specific algorithm selection based on run-time predictions

Leyton-Brown, Nudelman, Shoham (2002); Xu, Hutter, Hoos, Leyton-Brown (2008)

Key idea: (Rice 1976)

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

Here:

- ▶ problem instance \rightsquigarrow vector of cheaply computable features
- ▶ features \rightsquigarrow performance prediction for given set of solvers
- ▶ run solver with best predicted performance

Instance features:

- ▶ Use generic and problem-specific features that correlate with performance and can be computed cheaply.
- ▶ Examples (for SAT):
 - ▶ 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
- ▶ Consider pairwise products of features (quadratic basis function expansion).

Run-time prediction:

- ▶ Collect feature and performance data on (large & diverse) set of training instances.
- ▶ Use feature selection to avoid problems due to correlated / uninformative features.
- ▶ Use ridge regression on training data to build predictive model.

Application example: SATzilla

Xu, Hutter, Hoos, Leyton-Brown (2008)

- ▶ Start from state-of-the-art complete (DPLL) and incomplete (local search) SAT solvers.
- ▶ Use pre-solvers to solve 'easy' instances quickly.
- ▶ Build run-time predictors for various types of instances, use classifier to select best predictor based on instance features.
- ▶ Predict time required for feature computation; if that time is too long, use back-up solver.

↪ prizes in 5 of the 9 main categories of the 2009 SAT Solver Competition (3 gold, 2 silver medals)

Towards a software environment for computer-aided design of high-performance algorithms

How to best support use of computer-aided algorithm design methods?

- ▶ develop powerful procedures for *searching* large design spaces
- ▶ develop useful abstractions for *specifying* design spaces
- ▶ develop best practices for computer-aided algorithm design
- ▶ provide comfortable software environments for computer-aided algorithm design

HAL: High-performance Algorithm Lab

Nell, Fawcett, Hoos, Leyton-Brown (work in progress)

- ▶ support *algorithm design* and *empirical analysis*
- ▶ support wide range of design patterns, procedures
- ▶ support effective utilisation of parallel computation
- ▶ support multiple platforms
(Linux, Windows, MacOS, Chrome)
- ▶ web-based UI, component-based architecture
- ▶ open source, easy to use & expand

Computer-aided Algorithm Design ...

- ▶ leverages computational power to construct better algorithms
- ▶ liberates human designers from boring, menial tasks and let them focus on higher-level design issues
- ▶ enables effective exploration of larger design spaces
- ▶ facilitates principled design of heuristic algorithms
- ▶ revolutionises the way we build and use algorithms