Computer-Aided Algorithm Design: Automated Tuning, Configuration, Selection and Beyond

Holger H. Hoos

BETA Lab
Department of Computer Science
University of British Columbia
Canada
How to build better solvers for hard problems?
How to build better solvers for hard problems?

▸ construct a provably good solver
How to build better solvers for hard problems?

- construct a provably good solver
- roll up your sleeves and do the best you can
How to build better solvers for hard problems?

- construct a provably good solver

- roll up your sleeves and do the best you can
How to build better solvers for hard problems?

- construct a provably good solver
- roll up your sleeves and do the best you can
- use your little grey cells, then your little black chips
How to build better solvers for hard problems?

▪ construct a provably good solver

▪ roll up your sleeves and do the best you can

▪ use your little grey cells, then your little black chips

⇝ principled experimentation + generic techniques
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
- liberates human designers from boring, menial tasks and lets them focus on higher-level design issues
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
- liberates human designers from boring, menial tasks and lets them focus on higher-level design issues
- enables effective exploration of larger design spaces
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms

- liberates human designers from boring, menial tasks and lets them focus on higher-level design issues

- enables effective exploration of larger design spaces

- facilitates principled design of heuristic algorithms
Computer-aided Algorithm Design ...

▶ leverages computational power to construct better algorithms

▶ liberates human designers from boring, menial tasks and lets them focus on higher-level design issues

▶ enables effective exploration of larger design spaces

▶ facilitates principled design of heuristic algorithms

▶ profoundly changes how we build and use algorithms
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 (Howe et al. 1999; Fox & Long 2001; Roberts et al. 2007; Richter & Westphal 2009; Valenzano et al. 2010; ...)

▶ 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
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
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
  (Howe et al. 1999; Fox & Long 2001; Roberts et al. 2007; Richter & Westphal 2009; Valenzano et al. 2010; ...)

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
  (Howe et al. 1999; Fox & Long 2001; Roberts et al. 2007; Richter & Westphal 2009; Valenzano et al. 2010; ...)

- various heuristic components interact in complex ways
  \( \leadsto \) unexpected, emergent behaviour
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 (Howe et al. 1999; Fox & Long 2001; Roberts et al. 2007; Richter & Westphal 2009; Valenzano et al. 2010; ...)

- various heuristic components interact in complex ways $\Rightarrow$ 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
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
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 $\Rightarrow \approx 300$ seconds / run
  - good performance on some instances may not generalise
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
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 $\sim \approx 300$ seconds / run
  - good performance on some 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. Three success stories (SAT, timetabling, MIP)
6. The next step: Programming by Optimisation
From traditional to computer-aided algorithm design

Traditional algorithm design approach:

- iterative, manual process
From traditional to computer-aided algorithm design

Traditional algorithm design approach:

- iterative, manual process
- designer gradually introduces/modifies components or mechanisms
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
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.
Note:

- During the design process, many decisions are made.
- Some choices take the form of parameters, others are hard-coded.
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.
Problems:

- Design process is labour-intensive.
- Design decisions often made in *ad-hoc* fashion, based on limited experimentation and intuition.
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.
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.
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.
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.

\(\Rightarrow\) complicated designs, unfulfilled performance potential
Solution: Computer-aided Algorithm Design

- Goal: construct high-performance algorithms automatically
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
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, reactive search
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, reactive search; learning and intelligent optimisation, SLS engineering; meta-learning; program synthesis
Human designer:

- specifies (possibly large) space of candidate algorithm design
Human designer:

- specifies (possibly large) space of candidate algorithm design
- supplies set of problem instances for performance evaluation
Human designer:

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

▶ lets human designer focus on higher-level issues
▶ enables better exploration of larger design spaces
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
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
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
Hutter, Babic, HH, Hu (2007)

- **Goal:** Solve suite of SAT-encoded software verification instances as fast as possible
Example: 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)
Example: 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
Example: 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)
  \begin{itemize}
  \item[\texttt{SPEAR}] = highly parameterised heuristic algorithm
  \item[(26 parameters, \(\approx 8.3 \times 10^{17}\) configurations)]
  \end{itemize}

- 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>
</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)
**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</strong> original</td>
<td>298/302</td>
<td>787.1 CPU sec</td>
</tr>
</tbody>
</table>

▶ 500-fold speedup through use of automated algorithm configuration procedure (ParamILS)

▶ New state of the art (winner of 2007 SMT Competition, QF BV category)
**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>Spear original</td>
<td>298/302</td>
<td>787.1 CPU sec</td>
</tr>
<tr>
<td>Spear generic. opt. config.</td>
<td>302/302</td>
<td>35.9 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)
**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>SPEAR original</td>
<td>298/302</td>
<td>787.1 CPU sec</td>
</tr>
<tr>
<td>SPEAR generic. opt. config.</td>
<td>302/302</td>
<td>35.9 CPU sec</td>
</tr>
<tr>
<td>SPEAR specific. opt. config.</td>
<td>302/302</td>
<td>1.5 CPU sec</td>
</tr>
</tbody>
</table>
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>SPEAR original</td>
<td>298/302</td>
<td>787.1 CPU sec</td>
</tr>
<tr>
<td>SPEAR generic. opt. config.</td>
<td>302/302</td>
<td>35.9 CPU sec</td>
</tr>
<tr>
<td>SPEAR specific. opt. config.</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)
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. (2007–9), Ansótegui et al. (2009); Bartz-Beielstein (2006)
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. (2007–9), Ansótegui et al. (2009); Bartz-Beielstein (2006)

- algorithm configuration from components
  (for given set of instances)
  Fukunaga (2002), Chiarandini et al. (2008), KhudaBukhsh et al. (2009)
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. (2007–9), Ansótegui et al. (2009); 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);
  Streeter et al. (2007)
Special cases of computer-aided algorithm design (2):

- instance-based algorithm configurators
  Hutter et al. (2006); Malitsky & Sellmann (2009)
Special cases of computer-aided algorithm design (2):

- **instance-based algorithm configurators**
  Hutter *et al.* (2006); Malitsky & Sellmann (2009)

- **on-line algorithm control / reactive search**
  Carchrae & Beck (2005); Battiti *et al.* (2008)
Special cases of computer-aided algorithm design (2):

- **instance-based algorithm configurators**
  Hutter *et al.* (2006); Malitsky & Sellmann (2009)

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

- **instance-based algorithm configurators**
  Hutter *et al.* (2006); Malitsky & Sellmann (2009)

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

- **instance-based algorithm configurators**
  Hutter *et al.* (2006); Malitsky & Sellmann (2009)

- **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
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)

- 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)
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$^+$, TB-SPO *(Hutter *et al.* 2009–10)*
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$^+$, TB-SPO (Hutter et al. 2009–10)

- **algorithm configuration:**
  - genetic programming
    - *e.g.*, CLASS (Fukunaga 2002)
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$^+$, TB-SPO (Hutter *et al.* 2009–10)

- **algorithm configuration:**
  - genetic programming
    - *e.g.*, CLASS (Fukunaga 2002)
  - racing procedures
    - *e.g.*, F-Race (Birattari *et al.* 2002)
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^+, TB-SPO (Hutter et al. 2009–10)

▪ 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.*, Harik & Lobo 1999)
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)

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)
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., Harik & Lobo 1999)
Many open questions:

- Which procedure for which type of design space?
Many open questions:

- Which procedure for which type of design space?
- How to deal with hybrid design patterns?
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?
How good are current methods for computer-aided algorithm design?
Three success stories

How good are current methods for computer-aided algorithm design?

“The proof is in the pudding”:

- Propositional Satisfiability
- Course Timetabling
- Mixed Integer Programming
Three success stories

How good are current methods for computer-aided algorithm design?

“The proof is in the pudding”:

- Propositional Satisfiability
- Course Timetabling
- Mixed Integer Programming

Further successes:

- protein structure prediction (Thachuk et al. 2007)
- SAT (KhudaBukhsh et al. 2009; Xu et al. – to appear; Tompkins & HH – to appear)
- TSP (Styles & HH – in preparation)
SATzilla: Portfolio-based algorithm selection for SAT

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

Key idea: Instance-based Algorithm Selection (Rice 1976)

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

SATzilla in a nutshell:

- CNF formula $\Rightarrow$ 84 polytime-computable instance features
- features $\Rightarrow$ performance prediction for set of SAT solvers
- run solver with best predicted performance
SATzilla: Portfolio-based algorithm selection for SAT
Xu, Hutter, HH, Leyton-Brown (2008)

Key idea: Instance-based Algorithm Selection (Rice 1976)

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

SATzilla in a nutshell:

- CNF formula \( \sim \) 84 polytime-computable instance features
SATzilla: Portfolio-based algorithm selection for SAT

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

Key idea: Instance-based Algorithm Selection (Rice 1976)

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

SATzilla in a nutshell:

- CNF formula $\leadsto$ 84 polytime-computable instance features
- features $\leadsto$ performance prediction for set of SAT solvers
SATzilla: Portfolio-based algorithm selection for SAT
Xu, Hutter, HH, Leyton-Brown (2008)

Key idea: Instance-based Algorithm Selection (Rice 1976)

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

SATzilla in a nutshell:

- CNF formula $\leadsto$ 84 polytime-computable instance features
- features $\leadsto$ performance prediction for set of SAT solvers
- run solver with best predicted performance
Under the hood:

- Use state-of-the-art complete (DPLL) and incomplete (local search) SAT solvers.
Under the hood:

- Use state-of-the-art complete (DPLL) and incomplete (local search) SAT solvers.
- Use ridge regression on selected features to predict solver run-times from instance features.
Under the hood:

- Use state-of-the-art complete (DPLL) and incomplete (local search) SAT solvers.
- Use ridge regression on selected features to predict solver run-times from instance features.
- Use method by Schmee & Hahn (1979) to deal with censored run-time data.
Some bells and whistles:

▶ Use pre-solvers to solve ‘easy’ instances quickly.
Some bells and whistles:

- 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.

prizes in 5 of the 9 main categories of the 2009 SAT Solver Competition (3 gold, 2 silver medals)
Some bells and whistles:

- 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.
Some bells and whistles:

- 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)
Post-Enrolment Course Timetabling
Chiarandini, Fawcett, HH (2008); Fawcett, HH, Chiarandini (in preparation)

Post-Enrolment Course Timetabling:

- students enroll in courses
- courses are assigned to rooms and time slots, subject to *hard constraints*
- preferences are represented by *soft constraints*
Post-Enrolment Course Timetabling

Chiarandini, Fawcett, HH (2008); Fawcett, HH, Chiarandini (in preparation)

Post-Enrolment Course Timetabling:

▶ students enroll in courses
▶ courses are assigned to rooms and time slots, subject to *hard constraints*
▶ preferences are represented by *soft constraints*

Our solver:

▶ modular multiphase stochastic local search algorithm
▶ hard constraint solver: finds feasible course schedules
▶ soft constraint solver: optimise schedule (maintaining feasibility)
Our first solver:

- developed over ca. 1 month
- starting point: Chiarandini et al. (2003)
- *soft constraint solver* unchanged
- automatically configured *hard constraint solver*
Our first solver:

- developed over ca. 1 month
- starting point: Chiarandini et al. (2003)
- *soft constraint solver* unchanged
- automatically configured *hard constraint solver*

Design space for hard constraint solver:

- parameterised combination of constructive search, tabu search, diversification strategy
- 7 parameters, 50 400 configurations
Our first solver:

- developed over ca. 1 month
- starting point: Chiarandini et al. (2003)
- soft constraint solver unchanged
- automatically configured hard constraint solver

Design space for hard constraint solver:

- parameterised combination of constructive search, tabu search, diversification strategy
- 7 parameters, 50,400 configurations

Automated configuration process:

- configurator: FocusedILS 2.3 (Hutter et al. 2009)
- performance objective: solution quality after 300 CPU sec
2nd International Timetabling Competition (ITC), Track 2

<table>
<thead>
<tr>
<th>Distance To Feasibility</th>
<th>Aggregate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cambazard et al.</td>
<td></td>
</tr>
<tr>
<td>Atsuta et al.</td>
<td></td>
</tr>
<tr>
<td>Our Solver 2008</td>
<td></td>
</tr>
<tr>
<td>Nothegger et al.</td>
<td></td>
</tr>
<tr>
<td>Muller</td>
<td></td>
</tr>
</tbody>
</table>

Holger Hoos: Computer-aided algorithm design
Our latest solver:

- developed over ca. 6 months
- starting point: our previous solver
- automatically configured hard & soft constraint solvers
Our latest solver:

- developed over ca. 6 months
- starting point: our previous solver
- automatically configured *hard & soft constraint* solvers

Design space for soft constraint solver:

- highly parameterised simulated annealing algorithm
- 11 parameters, $2.7 \times 10^9$ configurations
Our latest solver:

- developed over ca. 6 months
- starting point: our previous solver
- automatically configured hard & soft constraint solvers

Design space for soft constraint solver:

- highly parameterised simulated annealing algorithm
- 11 parameters, $2.7 \times 10^9$ configurations

Automated configuration process:

- configurator: FocusedILS 2.4 (new version, multiple stages)
- multiple performance objectives
  (final stage: solution quality after 600 CPU sec)
2-way race against ITC Track 2 winner

Our Solver wins beats ITC winner on 20 out of 24 competition instances

Application to university-wide exam scheduling at UBC (≈ 1650 exams, 28,000 students)
2-way race against ITC Track 2 winner

- our solver wins beats ITC winner on 20 out of 24 competition instances
- application to university-wide exam scheduling at UBC
  (≈ 1650 exams, 28,000 students)
Mixed Integer Programming (MIP)
Hutter, HH, Leyton-Brown, Stützle (2009); Hutter, HH, Leyton-Brown (2010)

- MIP is widely used for modelling optimisation problems
- MIP solvers play an important role for solving broad range of real-world problems

CPLEX:
- prominent and widely used commercial MIP solver
- exact solver, based on sophisticated branch & cut algorithm and numerous heuristics
- 159 parameters, 81 directly control search process
“A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models.”

[CPLEX 12.1 user manual, p. 478]
“A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models.”

[CPLEX 12.1 user manual, p. 478]

Automatically Configuring CPLEX:

▶ starting point: factory default settings
▶ 63 parameters (some with ‘AUTO’ settings)
▶ $1.38 \times 10^{37}$ configurations
“A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models.”

[CPLEX 12.1 user manual, p. 478]

Automatically Configuring CPLEX:

- starting point: factory default settings
- 63 parameters (some with ‘AUTO’ settings)
- $1.38 \times 10^{37}$ configurations
- configurator: FocusedILS 2.3 (Hutter et al. 2009)
- performance objective: minimal mean run-time
- configuration time: $10 \times 2$ CPU days
“A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models.”

[CPLEX 12.1 user manual, p. 478]

Automatically Configuring CPLEX:

- starting point: factory default settings
- 63 parameters (some with ‘AUTO’ settings)
- $1.38 \times 10^{37}$ configurations
- configurator: FocusedILS 2.3 (Hutter et al. 2009)
- performance objective: minimal mean run-time
- configuration time: $10 \times 2$ CPU days
### CPLEX on various MIPS benchmarks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BCOL/Conic.sch</td>
<td>5.37</td>
<td>2.35 (2.4 ± 0.29)</td>
<td>2.2</td>
</tr>
<tr>
<td>BCOL/CLS</td>
<td>712</td>
<td>23.4 (327 ± 860)</td>
<td>30.4</td>
</tr>
<tr>
<td>BCOL/MIK</td>
<td>64.8</td>
<td>1.19 (301 ± 948)</td>
<td>54.4</td>
</tr>
<tr>
<td>CATS/Regions200</td>
<td>72</td>
<td>10.5 (11.4 ± 0.9)</td>
<td>6.8</td>
</tr>
<tr>
<td>RNA-QP</td>
<td>969</td>
<td>525 (827 ± 306)</td>
<td>1.8</td>
</tr>
</tbody>
</table>

(Timed-out runs are counted as $10 \times$ cutoff time.)
## CPLEX on various MIPS benchmarks

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BCOL/Conic.sch</td>
<td>5.37</td>
<td>2.35 (2.4 ± 0.29)</td>
<td>2.2</td>
</tr>
<tr>
<td>BCOL/CLS</td>
<td>712</td>
<td>23.4 (327 ± 860)</td>
<td>30.4</td>
</tr>
<tr>
<td>BCOL/MIK</td>
<td>64.8</td>
<td>1.19 (301 ± 948)</td>
<td>54.4</td>
</tr>
<tr>
<td>CATS/Regions200</td>
<td>72</td>
<td>10.5 (11.4 ± 0.9)</td>
<td>6.8</td>
</tr>
<tr>
<td>RNA-QP</td>
<td>969</td>
<td>525 (827 ± 306)</td>
<td>1.8</td>
</tr>
</tbody>
</table>

(Timed-out runs are counted as $10 \times$ cutoff time.)
CPLEX on BCOL/CLS

![Graph showing run-time comparison between default and optimised runs. The x-axis represents the default run-time [CPU s], and the y-axis represents the optimised run-time [CPU s]. The data points are plotted on a logarithmic scale.]
CPLEX on BCOL/Conic.sch
Latest results: Gurobi on BCOL/MIK

Configuration time: $10 \times 2$ CPU days
Latest results: Gurobi on BCOL/MIK

Configuration time: $10 \times 2$ CPU days
Latest results: Ipsoolve on CA-WDP

Configuration time: $10 \times 2$ CPU days
Latest results: lpsolve on CA-WDP

Configuration time: 10 × 2 CPU days
How to use computer-aided algorithm design?
How to use computer-aided algorithm design?

application context
How to use computer-aided algorithm design?

application context
+

design space
How to use computer-aided algorithm design?

application context
+

design space
+

optimisation procedure
How to use computer-aided algorithm design?

application context
  +
  design space
  +
optimisation procedure
  +
compute power
How to use computer-aided algorithm design?

application context +

  design space +

  optimisation procedure +

  compute power =

  success
The next step: Programming by Optimisation

How to *easily* use computer-aided algorithm design?

Need effective support for...
▶ specification of rich design spaces
▶ automated design (and analysis) process

Holger Hoos: Computer-aided algorithm design
The next step: Programming by Optimisation

How to *easily* use computer-aided algorithm design?

Need effective support for ...

- specification of rich design spaces
The next step:
Programming by Optimisation

How to *easily* use computer-aided algorithm design?

Need effective support for …

- specification of rich design spaces
- automated design (and analysis) process
HAL: High-performance Algorithm Lab
Nell, Fawcett, HH, Leyton-Brown (under review)

- support *algorithm design* and *empirical analysis*
HAL: High-performance Algorithm Lab
Nell, Fawcett, HH, Leyton-Brown (under review)

- support *algorithm design* and *empirical analysis*
- support wide range of design patterns, procedures
HAL: High-performance Algorithm Lab
Nell, Fawcett, HH, Leyton-Brown (under review)

- support algorithm design and empirical analysis
- support wide range of design patterns, procedures
- support effective utilisation of parallel computation
HAL: High-performance Algorithm Lab
Nell, Fawcett, HH, Leyton-Brown (under review)

- support *algorithm design* and *empirical analysis*
- support wide range of design patterns, procedures
- support effective utilisation of parallel computation
- support multiple platforms
  (Linux, MacOS; *later*: Windows, Chrome OS?)
HAL: High-performance Algorithm Lab
Nell, Fawcett, HH, Leyton-Brown (under review)

- support *algorithm design* and *empirical analysis*
- support wide range of design patterns, procedures
- support effective utilisation of parallel computation
- support multiple platforms
  (Linux, MacOS; *later*: Windows, Chrome OS?)
- web-based UI, component-based architecture
HAL: High-performance Algorithm Lab
Nell, Fawcett, HH, Leyton-Brown (under review)

- support *algorithm design* and *empirical analysis*
- support wide range of design patterns, procedures
- support effective utilisation of parallel computation
- support multiple platforms
  (Linux, MacOS; *later*: Windows, Chrome OS?)
- web-based UI, component-based architecture
- open source, easy to use & expand
HAL 1.0

New Tasks

Evaluate algorithm performance
Analyse performance of an algorithm on an instance set.

Compare algorithm performance
Compare the performance of two algorithms on an instance set.

Configure algorithm
Optimize parameter settings to maximize algorithm performance on an instance set.

Active Tasks

<table>
<thead>
<tr>
<th>Status</th>
<th>ID</th>
<th>Name</th>
<th>Start Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>queued</td>
<td>5</td>
<td>Compare GGA/PILS SPEAR</td>
<td>N/A</td>
<td>0 s</td>
</tr>
<tr>
<td>99%</td>
<td>3</td>
<td>ParamILS SPEAR SWV</td>
<td>2010-04-02 17:25:05.0</td>
<td>258122.70 s</td>
</tr>
<tr>
<td>97%</td>
<td>4</td>
<td>GGA SPEAR SWV</td>
<td>2010-04-02 17:31:31.0</td>
<td>253904.81 s</td>
</tr>
<tr>
<td>8%</td>
<td>6</td>
<td>Compare GGA/PILS SATenstein</td>
<td>2010-04-05 08:47:26.0</td>
<td>1322.16 s</td>
</tr>
</tbody>
</table>

Completed Tasks

<table>
<thead>
<tr>
<th>Status</th>
<th>ID</th>
<th>Name</th>
<th>Start Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>done</td>
<td>1</td>
<td>ParamILS SATenstein QCP</td>
<td>2010-04-02 15:07:35.0</td>
<td>188920.02 s</td>
</tr>
<tr>
<td>done</td>
<td>2</td>
<td>GGA SATenstein QCP</td>
<td>2010-04-02 15:08:41.0</td>
<td>181342.54 s</td>
</tr>
</tbody>
</table>
HAL 1.0

New Algorithm Configuration Task

Target Algorithm
Choose a target algorithm to configure

Configuration Space
Choose the Configuration Space for the target Algorithm

Problem Instances
Choose an instance set to use for training

Configurator
Choose a configurator to run

Execution Environment
Choose an execution environment to use

Task Name: pILS SPEAR-SWV
Run
HAL 1.0

New Tasks

Evaluate algorithm performance
Analyse performance of an algorithm on an instance set.

Compare algorithm performance
Compare the performance of two algorithms on an instance set.

Configure algorithm
Optimize parameter settings to maximize algorithm performance on an instance set.

Active Tasks

<table>
<thead>
<tr>
<th>Status</th>
<th>ID</th>
<th>Name</th>
<th>Start Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>queued</td>
<td>5</td>
<td>Compare GGA/ILS SPEAR</td>
<td>N/A</td>
<td>0 s</td>
</tr>
<tr>
<td>99%</td>
<td>3</td>
<td>ParamILS SPEAR SWV</td>
<td>2010-04-02 17:25:05.0 258122.70 s</td>
<td>KILL</td>
</tr>
<tr>
<td>97%</td>
<td>4</td>
<td>GGA SPEAR SWV</td>
<td>2010-04-02 17:31:31.0 253904.81 s</td>
<td>KILL</td>
</tr>
<tr>
<td>8%</td>
<td>6</td>
<td>Compare GGA/ILS SATenstein</td>
<td>2010-04-05 08:47:26.0 1322.16 s</td>
<td>KILL</td>
</tr>
</tbody>
</table>

Completed Tasks

<table>
<thead>
<tr>
<th>Status</th>
<th>ID</th>
<th>Name</th>
<th>Start Time</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>done</td>
<td>1</td>
<td>ParamILS SATenstein QCP</td>
<td>2010-04-02 15:07:35.0 188920.02 s</td>
<td></td>
</tr>
<tr>
<td>done</td>
<td>2</td>
<td>GGA SATenstein QCP</td>
<td>2010-04-02 15:08:41.0 181342.54 s</td>
<td></td>
</tr>
</tbody>
</table>
Wilcoxon signed-rank test: p=2.63E-15
Wilcoxon winner: SPEAR-ParamILS-Best
Spearman correlation test: p=2.22E-208

SPEAR-ParamILS-Best: 0.017 0.104 0.922 1.670 3.171 1.712 3.216
SPEAR-GGA-Best: 0.019 0.135 0.926 2.073 3.888 1.747 3.052
Programming by Optimisation (PbO)

HH (work in progress)
Programming by Optimisation (PbO)

HH (work in progress)

Key idea:

▶ avoid premature, uninformed, possibly detrimental design choices

▶ encourage developers to parameterise, provide functionally equivalent alternatives
Programming by Optimisation (PbO)

HH (work in progress)

Key idea:

▶ avoid premature, uninformed, possibly detrimental design choices

▶ encourage developers to parameterise, provide functionally equivalent alternatives

▶ automatically make choices to obtain algorithm / software / system that performs well in a given application context
Programming by Optimisation (PbO)

HH (work in progress)

Key idea:

▶ avoid premature, uninformed, possibly detrimental design choices

▶ encourage developers to parameterise, provide functionally equivalent alternatives

⇝ generic programming language extension

▶ automatically make choices to obtain algorithm / software / system that performs well in a given application context
Programming by Optimisation (PbO)
HH (work in progress)

Key idea:

▷ avoid premature, uninformed, possibly detrimental design choices

▷ encourage developers to parameterise, provide functionally equivalent alternatives
  ↝ generic programming language extension

▷ automatically make choices to obtain algorithm / software / system that performs well in a given application context
  ↝ HAL + compute power
planner
planner

design space of planners
planner

design space of planners

application context
Holger Hoos: Computer-aided algorithm design
design space of planners

planner

parallel portfolio

instance-based selector

optimised planner

application context
Holger Hoos: Computer-aided algorithm design
Computationally too expensive?

Recent example: Hydra SAT solver (Xu et al. – to appear)

- automated construction of the solver (using ParamILS, SATzilla):
  $\approx 70$ CPU days
Computationally too expensive?

Recent example: Hydra SAT solver \((Xu\ et\ al.\ –\ to\ appear)\)

- automated construction of the solver (using ParamILS, SATzilla):
  \(\approx 70\ \text{CPU days}\)

- wall-clock time on 10 CPU cluster:
  \(\approx 7\ \text{CPU days}\)
Computationally too expensive?

Recent example: Hydra SAT solver (Xu et al. – to appear)

- automated construction of the solver (using ParamILS, SATzilla):
  \( \approx 70 \) CPU days

- wall-clock time on 10 CPU cluster:
  \( \approx 7 \) CPU days

- cost on Amazon Elastic Compute Cloud (EC2):
  20.16 USD
Computationally too expensive?

Recent example: Hydra SAT solver \((Xu \ et \ al. \ – \ to \ appear)\)

- automated construction of the solver (using ParamILS, SATzilla):
  \(\approx 70 \) CPU days

- wall-clock time on 10 CPU cluster:
  \(\approx 7 \) CPU days

- cost on Amazon Elastic Compute Cloud (EC2):
  20.16 USD

- 20.16 USD pays for ...
  - 0:38 hours of average software engineer
Computationally too expensive?

Recent example: Hydra SAT solver (Xu et al. – to appear)

- automated construction of the solver (using ParamILS, SATzilla):
  \( \approx 70 \) CPU days

- wall-clock time on 10 CPU cluster:
  \( \approx 7 \) CPU days

- cost on Amazon Elastic Compute Cloud (EC2):
  20.16 USD

- 20.16 USD pays for ...
  - 0:38 hours of average software engineer
  - 2:45 hours at minimum wage
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
- liberates human designers from boring, menial tasks and lets them focus on higher-level design issues
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
- liberates human designers from boring, menial tasks and lets them focus on higher-level design issues
- enables effective exploration of larger design spaces
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
- liberates human designers from boring, menial tasks and lets them focus on higher-level design issues
- enables effective exploration of larger design spaces
- facilitates principled design of heuristic algorithms
Computer-aided Algorithm Design ...

- leverages computational power to construct better algorithms
- liberates human designers from boring, menial tasks and lets them focus on higher-level design issues
- enables effective exploration of larger design spaces
- facilitates principled design of heuristic algorithms
- profoundly changes how we build and use algorithms
Acknowledgements

Collaborators:

- Domagoj Babic
- Alex Devkar
- Chris Fawcett
- Frank Hutter
- Chris Nell
- Eugene Nudelman
- Alena Shmygelska
- Chris Thachuk
- James Styles
- Lin Xu
- Thomas Bartz-Beielstein (FH Köln)
- Marco Chiarandini (University of Southern Denmark)
- Alan Hu
- Kevin Leyton-Brown
- Kevin Murphy
- Yoav Shoham (Stanford University)
- Thomas Stützte (Université Libre de Bruxelles)

Research funding:

- NSERC, MITACS, CFI
- IBM, Actenum Corp.

Computing resources:

- Arrow, BETA, ICICS clusters
- WestGrid