Portfolio-based Algorithm Selection for SAT

Holger H. Hoos

BETA Lab
Department of Computer Science
University of British Columbia
Canada

based on joint work with
Lin Xu, Frank Hutter and Kevin Leyton-Brown

ECAI 2012 CoCoMile Workshop
Montpellier, France, 2012/08/27
What is SAT?

*Given:* A formula $F$ in propositional logic

*Objective:* Determine whether $F$ is satisfiable (i.e., can be made true by assigning truth values to the variables in $F$)

Why SAT?

- *Drosophila* problem for computing science (and beyond)
  - prototypical NP-hard problem
  - prominent in various areas of CS and beyond
  - important applications
  - conceptual simplicity aids solver design / development

- hotbed for the kind of research discussed here
What fuels progress in SAT solving?

- Insights into SAT (theory)
- Creativity of algorithm designers
  - heuristics
- Advanced debugging techniques
  - fuzz testing, delta-debugging
- Principled experimentation
  - statistical techniques, experimentation platforms
- SAT competitions
### SAT 2011 competition

<table>
<thead>
<tr>
<th>Organizing committee</th>
<th>Matti Järvisalo, Daniel Le Berre and Olivier Roussel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Judges</td>
<td>Uwe Egly, Alexander Nadol, Ashish Sabharwal and Moshe Vardi</td>
</tr>
<tr>
<td>Benchmarks</td>
<td>whole selection tar of b2 4 files, 1.7 GB</td>
</tr>
<tr>
<td>Solvers</td>
<td>static binaries / dynamic libraries / source code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAT+UNSAT</th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
<th>Crafted</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>glucose</td>
<td>glueminisat</td>
<td>lingeling</td>
<td>ppololio seq</td>
<td>3S</td>
</tr>
<tr>
<td>UNSAT</td>
<td>glucose</td>
<td>glueminisat</td>
<td>lingeling</td>
<td>ppololio seq</td>
<td>3S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAT+UNSAT</th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
<th>Crafted</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>ppololio seq</td>
<td>ppololio seq</td>
<td>ppololio seq</td>
<td>ppololio seq</td>
<td>3S</td>
</tr>
<tr>
<td>UNSAT</td>
<td>ppololio seq</td>
<td>ppololio seq</td>
<td>ppololio seq</td>
<td>ppololio seq</td>
<td>3S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Best Minisat Hack</th>
<th>CnMinisat hack</th>
</tr>
</thead>
</table>

In 2010, we have a new SAT Race!

### SAT 2009 competition

<table>
<thead>
<tr>
<th>Organizing committee</th>
<th>Daniel Le Berre, Olivier Roussel and Laurent Simon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Judges</td>
<td>Andreas Goerdt, Ines Lynce and Aaron Stump</td>
</tr>
<tr>
<td>Benchmarks</td>
<td>random (7z, 46MB), crafted (7z, 171MB), industrial (7z, 385MB)</td>
</tr>
<tr>
<td>Solvers</td>
<td>binaries (7z, 33MB), sources (7z, 25MB)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAT+UNSAT</th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
<th>Crafted</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>preocast</td>
<td>glucose</td>
<td>lpsat</td>
<td>clasp</td>
<td>SATzilla2009_c</td>
</tr>
<tr>
<td>UNSAT</td>
<td>glucose</td>
<td>preocast</td>
<td>lpsat</td>
<td>clasp</td>
<td>SATzilla2009_c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel solver application</th>
<th>ManySAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel solver random</td>
<td>gNovelly2+</td>
</tr>
<tr>
<td>Best Minisat Hack</td>
<td>Minisat 09z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAT+UNSAT</th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
<th>Crafted</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT</td>
<td>IUT EMB SAT</td>
<td>SATzilla2009_c</td>
<td>SATzilla2009_c</td>
<td>March hi</td>
<td></td>
</tr>
<tr>
<td>UNSAT</td>
<td>IUT EMB SAT</td>
<td>SATzilla2009_c</td>
<td>SATzilla2009_c</td>
<td>March hi</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Special prizes</th>
<th>Bronze</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Minisat Hack</td>
<td>gNovelly2+</td>
</tr>
</tbody>
</table>

2009: 5 of 27 medals  
2011: 28 of 54 medals
## 2012 SAT Challenge

<table>
<thead>
<tr>
<th>Rank</th>
<th>Application</th>
<th>Hard Combinatorial</th>
<th>Random (SAT only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SATzilla2012 App</td>
<td>SATzilla2012 COMB</td>
<td>CCASat</td>
</tr>
<tr>
<td>2</td>
<td>SATzilla2012 ALL</td>
<td>SATzilla2012 ALL</td>
<td>SATzilla2012 RAND</td>
</tr>
<tr>
<td>3</td>
<td>Industrial SAT Solver</td>
<td>ppfolio2012</td>
<td>SATzilla2012 ALL</td>
</tr>
<tr>
<td>4</td>
<td>lingeling (2011)</td>
<td>interactSAT_c</td>
<td>Sparrow (2011)</td>
</tr>
<tr>
<td>5</td>
<td>interactSAT</td>
<td>pfolioUZK</td>
<td>EagleUP (2011)</td>
</tr>
<tr>
<td>6</td>
<td>glucose</td>
<td>aspeed-crafted</td>
<td>sattime2012</td>
</tr>
<tr>
<td>7</td>
<td>SINN</td>
<td>Clasp-crafted</td>
<td>ppfolio2012</td>
</tr>
</tbody>
</table>

2/3 first, 3/3 second, 3/3 third places
Meta-algorithmic techniques

- algorithms that operate upon other algorithms (SAT solvers)

≠ meta-heuristics

- *here*: algorithms whose inputs include one or more SAT solvers
  - configurators *(e.g., ParamILS, GGA, SMAC)*
  - selectors *(e.g., SATzilla, 3S)*
  - schedulers *(e.g., aspeed; also: 3S, SATzilla)*
  - *(parallel)* portfolios *(e.g., ManySAT, ppfolio)*
  - run-time predictors
  - experimentation platforms *(e.g., EDACC, HAL)*
Why are meta-algorithmic techniques important?

- no one knows how to best solve SAT (or any other NP-hard problem)
  - no single dominant solver

- state-of-the-art performance often achieved by combinations of various heuristic choices (e.g., pre-processing; variable/value selection heuristic; restart rules; data structures; ...)
  - complex interactions, unexpected behaviour

- performance can be tricky to assess due to
  - differences in behaviour across problem instances
  - stochasticity
  - potential for suboptimal choices in solver development, applications
Why are meta-algorithmic techniques important?

- human intuitions can be misleading, abilities are limited
  → substantial benefit from augmentation with computational techniques

- use of fully specified procedures (rather than intuition / ad hoc choices) can improve reproducibility, facilitate scientific analysis / understanding
Outline

1. Introduction
2. Portfolio-based Algorithm Selection
3. SATzilla
4. Beyond Algorithm Selection for SAT
5. Conclusions & Outlook
Instance-based algorithm selection (Rice 1976):

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

Note:

Best case performance bounded by oracle, which selects the best $s \in S$ for each $\pi = \textit{virtual best solver (VBS)}$
Instance-based algorithm selection
Instance-based algorithm selection
Instance-based algorithm selection
Instance-based algorithm selection

feature extractor  selector  component algorithms
Instance-based algorithm selection

feature extractor

selector

component algorithms
Key components:

- set of (state-of-the-art) solvers
- 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)
Methods for instance-based selection:

- **classification-based**: predict the best solver, using:
  - decision trees
    - (Guerri & Milano 2004)
  - case-based reasoning
    - (Gebruers, Guerri, Hnich, Milano 2004; OMahony, Hebrard, Holland, Nugent, OSullivan 2008)
  - (weighted) \(k\)-nearest neighbours
    - (Malitsky, Sabharwal, Samulowitz, Sellmann 2011; Kadioglu, Malitsky, Sabharwal, Samulowitz, Sellmann 2011)
  - pairwise cost-sensitive decision forests + voting
    - (Xu, Hutter, HH, Leyton-Brown 2012)

- **regression-based**: predict running time for each solver, select the one predicted to be fastest
  - (Leyton-Brown, Nudelman, Shoham 2003; Xu, Hutter, HH, Leyton-Brown 2007–9)
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

- Consider combinations of features (e.g., pairwise products \(\sim\) quadratic basis function expansion).
SATzilla 2007–9 (Xu, Hutter, HH, Leyton-Brown):

- use state-of-the-art complete (DPLL/CDCL) and incomplete (local search) SAT solvers
- extract (up to) 84 polytime-computable instance features
- use ridge regression on selected features to predict solver run-times from instance features (one model per solver)
- run solver with best predicted performance
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 (or error occurs), use back-up solver
- use method by Schmee & Hahn (1979) to deal with censored run-time data

\[\Rightarrow\] prizes in 5 of the 9 main categories of the 2009 SAT Solver Competition (3 gold, 2 silver medals)
The problem with standard classification approaches

**Crucial assumption:** solvers behave similarly on instances with similar features

*But do they really?*

- uninformative features
- correlated features
- feature normalisation (tricky!)
- cost of misclassification
SATzilla 2011–12 (Xu, Hutter, HH, Leyton-Brown 2012):

- uses cost-based decision forests to directly 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
## 2011 SAT Competition Data (Inst. Solved)

<table>
<thead>
<tr>
<th>Solver</th>
<th>Application</th>
<th>Crafted</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle (VBS)</td>
<td>84.7%</td>
<td>76.3%</td>
<td>82.2%</td>
</tr>
<tr>
<td>SATzilla 2011</td>
<td>75.3%</td>
<td>66.0%</td>
<td>80.8%</td>
</tr>
<tr>
<td>SATzilla 2009</td>
<td>70.3%</td>
<td>63.0%</td>
<td>80.3%</td>
</tr>
<tr>
<td>Gold medalist (SBS)</td>
<td>71.7%</td>
<td>54.3%</td>
<td>68.0%</td>
</tr>
</tbody>
</table>

(SATzilla assessed by 10-fold cross-validation)
# 2012 SAT Challenge Results

<table>
<thead>
<tr>
<th>Rank</th>
<th>Application</th>
<th>Hard Combinatorial</th>
<th>Random (SAT only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SATzilla2012 App</td>
<td>SATzilla2012 COMB</td>
<td>CCASat</td>
</tr>
<tr>
<td>2</td>
<td>SATzilla2012 ALL</td>
<td>SATzilla2012 ALL</td>
<td>SATzilla2012 RAND</td>
</tr>
<tr>
<td>3</td>
<td>Industrial SAT Solver</td>
<td>ppfolio2012</td>
<td>SATzilla2012 ALL</td>
</tr>
<tr>
<td>4</td>
<td>lingeling (2011)</td>
<td>interactSAT_c</td>
<td>Sparrow (2011)</td>
</tr>
<tr>
<td>5</td>
<td>interactSAT</td>
<td>pfolioUZK</td>
<td>EagleUP (2011)</td>
</tr>
<tr>
<td>6</td>
<td>glucose</td>
<td>aspeed-crafted</td>
<td>sattime2012</td>
</tr>
<tr>
<td>7</td>
<td>SINN</td>
<td>Clasp-crafted</td>
<td>ppfolio2012</td>
</tr>
</tbody>
</table>

SATzilla: 2/3 first, 3/3 second, 2/3 third places
2012 SAT Challenge: Single best solver vs SATzilla
Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection
Xu, HH, Leyton-Brown (2010)

Note:

- SATzilla builds algorithm selector based on given set of SAT solvers
  
  *but*: success entirely depends on quality of given solvers

- Automated algorithm configuration produces solvers that work well on average on a given set of SAT instances
  (e.g., SATenstein – KhudaBukhsh, Xu, HH, Leyton-Brown 2009)
  
  *but*: may have to settle for compromises for broad, heterogenous sets

Idea: Combine the two approaches \( \sim \) portfolio-based selection from set of automatically constructed solvers
Configuration + Selection = Hydra

parametric algorithm
Configuration + Selection = Hydra
Configuration + Selection = Hydra

feature extractor

selector

parametric algorithm
(multiple configurations)
Configuration + Selection = Hydra
Configuration + Selection = Hydra

Holger Hoos: Portfolio-based Algorithm Selection for SAT
Simple combination:

1. build solvers for various types of instances using automated algorithm configuration
2. construct portfolio-based selector from these

Drawback: Requires suitably defined sets of instances

Better solution:

iteratively build & add solvers that improve performance of given portfolio

Ḥydra

Note: Builds portfolios solely using

- generic, highly configurable solver (e.g., SATenstein)
- instance features (as used in SATzilla)
Results on mixture of 6 well-known benchmark sets
Results on mixture of 6 well-known benchmark sets
Note:

- Hydra can use arbitrary algorithm configurators, selector builders

- different approaches are possible: e.g., ISAC, based on feature-based instance clustering, distance-based selection (Kadioglu, Malitsky, Sellmann, Tierney 2010)
The next step: Programming by Optimisation (PbO)

Key idea:

- specify large, rich design spaces of solver for given problem
  - avoid premature, uninformed, possibly detrimental design choices
  - active development of promising alternatives for design components
- automatically make choices to obtain solver optimised for given use context
solver

design space of solvers
solver

design space of solvers

application context
Holger Hoos: Portfolio-based Algorithm Selection for SAT
Holger Hoos: Portfolio-based Algorithm Selection for SAT
design space of solvers

parallel portfolio

instance-based selector

optimised solver

application context
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 (<em>SPEAR</em>), 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
\[\sim\] new state of the art; UBC exam scheduling
Fawcett, Chiarandini, HH (2009)
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 autocode transformations and a powerful design optimization engine using PBO.

The potential of PBO is evident from recent empirical results (see the table). In the forefront is source code optimization, including automatic parallelization of sequential programs. Several experiments have demonstrated the feasibility of this approach and validated its effectiveness. In particular, PBO can be used to automatically parallelize sequential programs and improve their performance significantly. The key insight is that by allowing developers to specify a large design space of programs, they can explore different ways of achieving a given task, from which the most efficient and scalable versions are automatically selected. This approach has potential applications in various domains, including scientific computing, data analysis, and even user interface design.

Performance Metrics

- Key insights
- Template: overview of design choices and tools for generating and analyzing performance of parallel programs
- Applications: examples of PBO in action
- Challenges: overcoming challenges in implementing and using PBO
Conclusions & Outlook

- State of the art in SAT solving:
  portfolio-based algorithm selectors;
  powered by machine learning, component solvers, features

- Further progress likely:
  component solvers, features, insights
Portfolio-based algorithm selectors and other meta-algorithmic techniques will become dominant in most (all?) areas of AI.

Programming by Optimisation (or a closely related approach) will greatly facilitate this development.

Of interest to CoCoMile / ML:
Auto-WEKA: Automated Selection and Hyper-Parameter Optimization of Classification Algorithms (Thornton, Hutter, HH, Leyton-Brown 2012)
http://arxiv.org/abs/1208.3719

Driven by SAT, SMT, PbO and advances in automated testing / debugging, program synthesis from higher-level designs will become practical and widely used.