SATzilla2012: Improved Algorithm Selection Based on Cost-sensitive Classification Models

Lin Xu, Frank Hutter, Jonathan Shen, Holger H. Hoos and Kevin Leyton-Brown
Computer Science Dept., University of British Columbia
Vancouver, BC, Canada

***Abstract*—`SATzilla2012` is an improved version of `SATzilla` that performs algorithm selection based on cost-sensitive classification models. It was trained on all three SAT competitions and three SAT Races since 2006 and utilized 31 publicly-available SAT solvers.**

I. Introduction

Empirical studies often observe that the performance of different algorithms across problem instance distributions can be quite uncorrelated. When this occurs, there is an incentive to investigate the use of portfolio-based approaches that draw on the strengths of multiple algorithms. `SATzilla` is such a portfolio-based approach for SAT; it was first deployed in the 2003 and 2004 SAT competitions [1], and later versions won a number of prizes in the 2007 and 2009 SAT competitions [2], [3], [4], including gold medals in the random, crafted and application categories in 2009.

Different from previous versions of `SATzilla`, which utilized *empirical hardness models* [5], [6] for estimating each candidate algorithm's performance on a given SAT instance, `SATzilla2012` is based on cost-sensitive classification models [7]. We also introduced a new procedure that generates a stand-alone `SATzilla` executable based on models learned within Matlab. Finally, we used new component algorithms and training instances.

Overall, `SATzilla2012` makes use of the same methodology as described in [8].

Offline, as part of algorithm development:

1) Identify a target distribution of problem instances.
2) Select a set of candidate solvers that are known or expected to perform well on at least a subset of the instances in the target distribution.
3) Use domain knowledge to identify features that characterize problem instances. To be usable effectively for automated algorithm selection, these features must be related to instance hardness and relatively cheap to compute.
4) On a training set of problem instances, compute these features and run each solver to determine its running times. We use the term *performance score* to refer to the quantity we aim to optimize.
5) Automatically determine the best-scoring combination of pre-solvers and their corresponding performance score. Pre-solvers will later be run for a short amount of time before features are computed (Step 2 below), in order to ensure good performance on very easy instances and to allow the predictive models to focus exclusively on harder instances.
6) Using a validation data set, determine which solver achieves the best performance for all instances that are not solved by the pre-solvers and on which the feature computation times out. We refer to this solver as the *backup solver*.
7) **New:** Construct a classification model (decision forest, DF) for predicting whether the cost of computing feature is too expensive, given the number of variables and clauses in an instance.
8) **New:** Construct a cost-sensitive classification model (DF) for every pair of solvers in the portfolio, predicting which solver performs better on a given instance based on instance features.
9) Automatically choose the best-scoring subset of solvers to use in the final portfolio.

Then, online, to solve a given problem instance, the following steps are performed:

1) Predict whether the feature computation time is above 90 CPU seconds. If the feature computation is too costly, run the backup solver identified in Step 6 above; otherwise continue with the following steps.
2) Run the presolvers in the predetermined order for up to their predetermined fixed cutoff times.
3) Compute feature values. If feature computation cannot be completed due to an error, select the backup solver identified in Step 6 above; otherwise continue with the following steps.
4) For every pair of solvers, predict which solver performs better using the DF trained in Step 8 above, and cast a vote for it.
5) Run the solver that received the highest number of votes. If a solver fails to complete its run (e.g., it crashes), run the solver with the second-highest number of votes. If that solver also fails, run the backup solver.

II. New Improvements in SATzilla2012

`SATzilla2012` implements a number of improvements over `SATzilla2009`.

New algorithm selector. Our new selection procedure uses an explicit cost-sensitive loss function—punishing misclassifications in direct proportion to their impact on portfolio performance—without predicting runtime. We introduced this approach in [9], [8]. To the best of our knowledge, this is the first time this approach is applied to algorithm selection: all other existing classification approaches use a simple 0–1 loss function that penalizes all misclassifications equally, whereas previous versions of `SATzilla` used regression-based runtime predictions. We construct cost-sensitive DFs as collections of 99 cost-sensitive decision trees [7], following standard random forest methodology [10].

New `SATzilla` executable. Our `SATzilla` version used in [8] was based on classification models built in Matlab, and its execution required the installation of the free Matlab runtime environment (MRE). In order to avoid the need for installing MRE, we now converted our Matlab-built models to Java and provide Java code to make predictions using them. Thus, running `SATzilla2012` now only requires the scripting language Ruby (which is used for running the `SATzilla` pipeline).

New component algorithms and training instances. We updated the component solvers used in `SATzilla2009` with the 31 newest publicly-available SAT solvers. These include 28 solvers from [8], the two versions of `Spear` optimized for software and hardware verification in [11], and `MXC 0.99` [12] (the list of solvers can also be found in the execution script of `SATzilla2012`).

Our training set is based on a collection of SAT instances that includes all instances from all three SAT competitions and three SAT Races since 2006: 1362 instances for Random SAT, 767 instances for Crafted SAT+UNSAT, and 1167 instances for Application SAT+UNSAT. We droppped instances that could not be solved by any of our 31 solvers within 900 CPU seconds. For training a general version of `SATzilla2012` that works well across categories, we used 1614 instances: 538 randomly sampled instances from each of Crafted, Application, and Random (SAT+UNSAT).

III. Running SATzilla2012

We submit a package containing one main executable for `SATzilla2012` that can be customized for each of the four categories in the 2012 SAT challenge by an input parameter. The callstring for `SATzilla2012` is:
`ruby SATzilla12.rb <type> <cnf file>`, where `<type>` should be chosen as `INDU` for target category Application SAT+UNSAT, `HAND` for Application SAT+UNSAT, `RAND` for Random SAT, and `ALL` for the Special Track for Sequential Portfolio Solvers. In order to run properly, a 64-bit machine with Ruby and JAVA is required. The subdirectory `bin` should contain all binaries for `SATzilla`'s component solvers and its feature computation; subdirectory `models` should contain all models for algorithm selection and predicting the cost of feature computation. We note that `SATzilla2012` has an optional 3rd input parameter `<seed>` that will be forwarded to any randomized component solver it runs; by default, that seed is set to `1234`.

References

[1] E. Nudelman, A. Devkar, Y. Shoham, K. Leyton-Brown, and H. Hoos, “SATzilla: An algorithm portfolio for SAT,” 2004.

[2] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Satzilla2007: a new & improved algorithm portfolio for SAT,” 2004, solver description, SAT competition 2007.

[3] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, “SATzilla2009: an Automatic Algorithm Portfolio for SAT,” 2009, solver description, SAT competition 2009.

[4] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “SATzilla: portfolio-based algorithm selection for SAT,” *Journal of Artificial Intelligence Research*, vol. 32, pp. 565–606, June 2008.

[5] K. Leyton-Brown, E. Nudelman, and Y. Shoham, “Learning the empirical hardness of optimization problems: The case of combinatorial auctions,” in *Proc. of CP-02*, 2002, pp. 556–572.

[6] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham, “Understanding random SAT: Beyond the clauses-to-variables ratio,” in *Proc. of CP-04*, 2004, pp. 438–452.

[7] K. M. Ting, “An instance-weighting method to induce cost-sensitive trees,” *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 3, pp. 659–665, 2002.

[8] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, “Evaluating component solver contributions to algorithm selectors,” in *Proc. of SAT 2012*, 2012, under review.

[9] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, “Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming,” in *RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion at the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011.

[10] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[11] F. Hutter, D. Babić, H. H. Hoos, and A. J. Hu, “Boosting Verification by Automatic Tuning of Decision Procedures,” in *Proc. of FMCAD'07*, 2007, pp. 27–34.

[12] D. R. Bregman, “The SAT solver MXC, version 0.99,” 2009, solver description, SAT competition 2009.