SATzilla: An Algorithm Portfolio for SAT*

Eugene Nudelman   Alex Devkar   Yoav Shoham
Department of Computer Science, Stanford University

Kevin Leyton-Brown   Holger Hoos
Department of Computer Science, University of British Columbia

1 Introduction

Inspired by the success of recent work in the constraint programming community on typical-case complexity, in [3] we developed a new methodology for using machine learning to study empirical hardness of hard problems on realistic distributions. In [2] we demonstrated that this new approach can be used to construct practical algorithm portfolios. In brief, the fact that algorithms for solving $\mathcal{NP}$-hard problems are often relatively uncorrelated means that it is possible for a portfolio to outperform all of its constituent algorithms. However, such uncorrelation is a knife that cuts both ways: a portfolio that makes bad choices among its constituent algorithms will often have much *worse* performance than any of its constituent algorithms.

Our methodology can be outlined as follows:

Offline, as part of algorithm development:

1. Identify a target distribution of problem instances.
2. Select a set of algorithms having relatively uncorrelated runtimes on this distribution.
3. Using domain knowledge, identify features that characterize problem instances.
4. Compute features and determine algorithm running times.
5. Use regression to construct models of algorithms' runtimes.

Online, given an instance:

1. Compute feature values.
2. Predict each algorithm's running time using learned runtime models.
3. Run the algorithm predicted to be fastest.

*See [5] for a complete discussion of `SATzilla`

2 SATzilla

`SATzilla` is a portfolio of SAT solvers built according to the methodology described above. It includes the following solvers: `2clseq`, `Limmat`, `JeruSat`, `OKsolver`, `Relsat`, `Sato`, `Satz-rand`, `zChaff`, `eqSatz`, `Satzoo`, `kcnfs`, and `BerkMin`.

We began by assembling a broad library of about 5000 SAT instances, which we gathered from various public websites. We identified 83 features that could be computed quickly and that we felt might be useful for predicting runtime. We computed these features for our set of SAT instances, dropped some features that were highly correlated, and were left with 56 distinct features. In order to keep feature values to sensible ranges, as appropriate we normalized features by the total number of clauses or number of variables. We also computed runtimes for each algorithm on each of our SAT instances. Given our features and runtime data, we had a well-defined supervised learning problem. We built models using ridge regression, a machine learning technique that finds a linear model (a hyperplane in feature space) that minimizes a combination of root mean squared error and a penalty term for large coefficients. To yield better models, we ignored all instances that were solved by all algorithms, by no algorithms, or as a side-effect of feature computation.

Upon execution, `SATzilla` begins by running a UBCSAT [6] implementation of `WalkSat` for 30 seconds. In our experience, this step helps to filter out easy satisfiable instances. Next, `SATzilla` runs the `Hypre`[1] preprocessor, which uses hyper-resolution to reason about binary clauses. This step is often able to dramatically shorten the formula, often resulting in search problems that are easier for DPLL-style solvers. Perhaps more importantly, the simplification "cleans up" instances, allowing the subsequent analysis of their structure to better reflect the problem's combinatorial "core." Third, SATzilla computes its 56 features. Sometimes, a feature can actually solve the problem; if this occurs, execution stops. Some features can also take an inordinate amount of time, particularly with very large inputs. To prevent feature computation from consuming all of our allotted time, certain features run only until a timeout is reached, at which point SATzilla gives up on computing the given feature. Fourth, SATzilla evaluates a regression model of each algorithm in order to compute a prediction of that algorithm's running time. If some of the features have timed out, a different model is used, which does not involve the missing feature and which was trained only on instances where the same feature timed out. Finally, SATzilla runs the algorithm with the best predicted runtime until the instance is solved or the allotted time is used up.

3 Features

Space restrictions prevent us from going into great detail about all elements of SATzilla. We choose to use our remaining space to give an overview of the features used by SATzilla.

The features can be roughly categorized into 9 groups. The first one captures problem size, measured in the number of clauses, variables, and their ratio. The next three groups correspond to 3 different constraint graphs associated with each SAT instances. Variable-Clause Graph is a bipartite graph representing which variables participate in which clause. Variable Graph has nodes representing variables, and an edge between any variables that occur in a clause together. Conflict Graph (CG) has nodes representing clauses, and an edge between two clauses whenever they share a negated literal. For all graphs we compute various node degree statistics. For CG we also compute statistics of clustering coefficients, defined, for each node, as the number of edges among its neighbors divided by $k(k-1)/2$, where k is the number of neighbors. The fifth group measures the balance of an instance in several different respects: the number of unary, binary, and ternary clauses, statistics of the amount of positive versus negative occurrences of variables within clauses and per variable. The sixth group measures the proximity of the instance to a Horn formula by computing the fraction of clauses that are Horn, and statistics over variables occurring in a Horn clause. These groups are motivated by known heuristics and tractable subclasses. The seventh group of features is obtained by solving a linear programming relaxation of an integer program representing the current SAT instance (a feature that can sometimes solve the SAT instance). Often, for integer programs, proximity of the LP relaxation solution to an integral solution is anticorrelated with hardness. We compute statistics of the integer slacks, as well as the actual objective value and fraction of variables set to an integer. The eighth group tries to estimate the hardness of the search space for a DPLL-type solver. For that we run DPLL procedure to a small depth and measure the number of unit propagations done at various depths. We also estimate the size of the search space [4] by randomly setting variables and then doing unit propagation until a contradiction is found. Our final group of features uses two local search algorithms, GSAT and SAPS [6]. We run both algorithms many times, each time continuing the search trajectory until a plateau cannot be escaped within a given number of steps. We then average various statistics collected during each run. It is interesting to note that local-search probing features are important to the models for most of SATzilla's algorithms, even though all of these solvers are DPLL-based.

References

[1] F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT*, 2003.

[2] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. Boosting as a metaphor for algorithm design. In *CP*, 2003.

[3] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, 2002.

[4] L. Lobjois and M. Lemaître. Branch and bound algorithm selection by performance prediction. In *AAAI*, 1998.

[5] E. Nudelman, A. Devkar, Y. Shoham, and K. Leyton-Brown. Understanding random SAT: Beyond the clauses-to-variables ratio. Under review.

[6] D. Tompkins and H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *SAT*, 2004.