Date of acceptance Grade

Instructor

Algorithm portfolios in constraint solving

Juho-Kustaa Kangas

Helsinki May 31, 2013

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Juho-Kustaa Kangas

Työn nimi — Arbetets titel — Title

Algorithm portfolios in constraint solving

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	May 31, 2013	16 pages + 16 appendices

Tiivistelmä — Referat — Abstract

Although NP-hard problems are generally thought to be intractable in the worst case, in practice many instances can be solved efficiently by heuristic algorithms. Typically, different algorithms perform well on different instances and there is no single best algorithm. Indeed, while an algorithm might solve many instances in a few seconds, solving other instances of same size may instead take several weeks. One approach in such cases is to construct an algorithm portfolio that comprises multiple algorithms and attempts to select the best one for a given problem instance.

In this report we present SATzilla, a generic process for constructing an algorithm portfolio that utilizes so called empirical hardness models to predict an algorithm's running time on given instance. Such models are constructed by first identifying a set of instance features and then using machine learning techniques to exploit correlations between features and algorithm performance. While the SATzilla approach originally targets boolean satisfiability, the same techniques have been successfully applied to various other constraint problems.

ACM Computing Classification System (CCS):

Avainsanat — Nyckelord — Keywords

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

For many problems it is recognized that there is no single best algorithm. Rather, different algorithms perform well on different instances of the problem, with large differences in running times between algorithms. This phenomenon is highlighted among NP-hard constraint problems such as boolean satisfiability (SAT) [XHLB07], which are unlikely to be tractable in general yet in many cases turn out to be solvable. Specifically, many large instances can be solved with "heuristic" algorithms that solve some instances quickly but have no (reasonable) upper bounds for worst-case complexity. Due to the inherent hardness of the problems, such heuristics tend to exhibit a significant degree of variance in the running time between different instances, even if the instances are of roughly equal size.

This raises a question commonly known as the "algorithm selection problem": given a problem instance and a set of (heuristic) algorithms, which algorithm or algorithms should be run to minimize the expected running time (or some other measure of performance)? While ideally we would like to select the algorithm with the best performance on the particular instance, predetermining the performance accurately without actually running the algorithm is in general not possible. Instead, the traditional approach has been to select for each class or distribution of instances the algorithm known to have the best average-case performance on that particular class, also known as the "winner-take-all" approach. While this often achieves a good performance, it has the drawback of dismissing algorithms that perform poorly on the average but nevertheless excel on some particular instances. Conversely, the algorithm that performs best on the average might be highly inefficient on some instances of the class.

1.1 Algorithm portfolios

A more recent approach to the problem has been to use an *algorithm portfolio* that makes the decision of algorithm selection "online", based on individual properties of the problem instance, rather than a predetermined instance distribution. In a broad sense of the term, an algorithm portfolio comprises a set of multiple component algorithms and some strategy of running them to solve a given problem instance. Originally named thus in [HLH97], algorithm portfolios have received a considerable amount of interest during the last decade and have demonstrated success on a variety of NP-hard problems, such as SAT [NLBH$^+$04, XHHLB08, NMJ11, SS12],

MaxSAT [MPLMS08], constraint satisfaction (CSP) [GS01, HHN08], mixed integer programming [GS01], winner determination (WDP) [LBNS02, LBNA+03], answer set programming [GKK+11], zero-one integer programming and A.I. planning [SS12].

The key idea behind most such portfolios is the use of *empirical hardness models* [LBNS02] that produce approximate predictions of an algorithm's performance on a given problem instance, based on the algorithm's past performance on similar instances. An empirical hardness model is constructed by first identifying a set of characteristic properties – or *features* – of problem instances, that exhibit correlation with the performance of algorithms and are relatively cheap to compute. To train the model, the features are computed for a set of problem instances and each algorithm's performance on those instances is measured. A machine learning method such as classification or regression is then applied to predict an algorithm's performance on a given set of features.

1.2 SATzilla

In this report we present SATzilla, an automated process for constructing an algorithm portfolio that utilizes empirical hardness models [XHHLB08] for algorithm selection. Given a new problem instance, the trained portfolio predicts the running time of each algorithm based on the instance features and then runs the algorithm predicted to have the best running time. The SATzilla process was most notably used to construct a SAT portfolio by the same name, which has proven particularly successful as verified in the 2007 SAT Competition.

We present the SATzilla process in a more general fashion, using SAT-specific details as examples along the way. We begin by outlining the portfolio construction in Section 2, as well as discussing the task of selecting proper training instances, features, and algorithms. In Section 3 we show to use the selected data to construct the empirical hardness models for predicting algorithm performance. We present some results and conclude the discussion in Section 4.

2 Portfolio construction

In this section we outline the SATzilla approach to constructing an algorithm portfolio for any problem with several heuristic algorithms available. We follow closely the original description presented in [XHHLB08], with supporting remarks incorporated from [LBNS02, NLBH$^+$04, HHHLB06, XHLB07], all of which follow the same main idea of portfolio construction. We use "performance" and "running time" interchangeably, noting that the methods presented are applicable to any measure of performance. In Section 2.1 we first present an overview of the entire process. We then discuss the selection of training data, features and algorithms in 2.2, 2.3 and 2.4, respectively, leaving empirical hardness models for Section 3.

2.1 Overview

We consider the portfolio design in two parts. The *offline* phase is where the actual portfolio construction happens, before seeing any instances that we are going to solve with the portfolio. In this phase, the following steps are performed:

1. Select a set of problem instances from the target distribution.
2. Select a set of features that characterize problem instances.
3. Select a set of available algorithms for solving the problem.
4. Compute each feature for each instance.
5. Run each algorithm on each instance and measure the performance.
6. Using the data obtained in steps 4 and 5, train an empirical hardness model for each algorithm to predict its performance on a given set of features.
7. Based on the observed performance, select the final set of algorithms to use in the portfolio, as well as a set of *pre-solvers* and one *backup solver*.

In the *online* phase, we are given a new problem instance that we want solve with the portfolio, utilizing the models constructed in the offline phase. Given this new instance, the following steps are performed:

1. Run each pre-solver on the instance for a short while, sequentially. If a pre-solver solves the instance, stop, otherwise proceed.

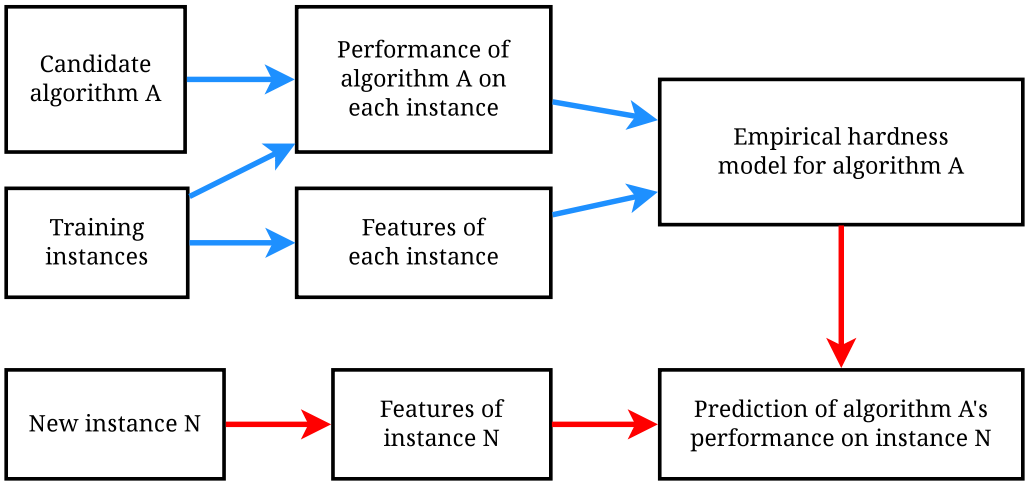

Figure 1: A flow chart of the work performed on each candidate algorithm A. Blue and red arrows represent computation in the offline and online phase, respectively.

2. Compute all features of the instance. If the computation exceeds a given time limit, run the backup solver, otherwise proceed.

3. Using the empirical hardness models trained in the offline phase, predict the performance of each algorithm based on the features.

4. Run the algorithm predicted to have the best performance. If an algorithm solves the instance, stop. If an algorithm fails (e.g. runs out of memory), continue with the algorithm predicted to have the next best performance.

The work performed on each algorithm is illustrated in Figure 1.

2.2 Selecting instances

In order to build the empirical hardness models, in step 1 we need to identify a set of problem instances to train and validate our models with. Ideally, we would like to draw these instances uniformly at random from the same target distribution of instances that we aim to solve in the online phase. If the target distribution is synthetic or similar enough to such a distribution, it may be possible to generate sample instances automatically. When this is not possible, we will typically consider some available set of instances that we assume to be representative of the target

distribution. The instances are then randomly split into training data, which is used to learn the empirical hardness models, and validation data, which is used for selecting model parameters, such as the final set of algorithms used.

In case of SATzilla the target distribution was the set of benchmark instances used in the 2007 SAT competition [XHHLB08]. Therefore, instances from all previous competitions were used for constructing the models. All instances that were not solved by any of the candidate algorithms within the competition time limit of 1200 seconds were omitted from the data set.

2.3 Identifying initial features

In order to predict an algorithm's performance on a given problem instance, we need to identify a set of features that characterize instances and are correlated with algorithm performance. Furthermore, the features have to be relatively cheap to compute so that the time spent in step 2 of the online phase is – on the average – more than compensated by the time saved by making a good choice in step 4. In the SATzilla approach, only such features that can be computed for any given instance are considered. The task of selecting features can be split into two steps:

1. Identify an initial set of characteristic features that are cheap to compute.
2. Determine a subset of such features that are useful for making predictions.

In step 1 we can consider any features that we believe might be correlated with empirical hardness. As most such features are specific to the problem, in general this initial step cannot be performed automatically, and should instead be based on our understanding of the problem domain [XHHLB08]. However, distinguishing between informative and non-informative features in this phase is not crucial. Rather, a wide selection is encouraged, with the aim of picking any features that seem reasonable [NLBH$^+$04]. While such an approach is likely to include many useless features in the process, there are ways to exclude such features in step 2. Furthermore, even seemingly useless features may turn out to be informative when used in conjunction with other features.

Although features vary between domains, there are certain types of features are common between prolems. For constraint problems in particular, a useful feature identified in one problem may be directly applicable to another problem [NLBH$^+$04]. One common feature is the size of the instance, described by some natural measure

such as the number of clauses and variables in a SAT formula or the number of vertices and edges in a graph. Typically problems tend to become harder as the size increases. Thus, one might wish to fix the size in order to find out which features account for the remaining variations in the performance of algorithms [LBNS02].

In some cases it is possible to model the structure of the instance with a graph, as demonstrated with SAT [XHHLB08] (clause–variable graphs) and WDP [LBNS02]. In such cases one can consider any graph theoretic properties such as the maximum or average degree as features. Another method applicable to various optimization problems (and problems where the quality of a partial solution can be measured) is to use local search algorithms to probe the search space. A local algorithm is run for a short while and statistics extracted from the search are treated as features. Such features can include e.g. branching factors, the number of search steps to reach a local optimum or the average improvement per step.

In SATzilla, a total of 84 initial features were considered, including features related to problem size, balance, variable–clause graph, local search statistics and properties of a related linear program. Some of the features were eventually dropped due to a high computational cost. Others were dropped in step 2 due to being uninformative or highly correlated with other features. We elaborate on the process of identifying such features in Section 3.2. We also discuss the approach of combining base features into new ones in Section 3.3.

2.4 Selecting the final set of algorithms

In step 3 we are free to consider any available algorithms for solving the problem. If we could perfectly predict each algorithm's performance on a given instance, in theory we could use all of them as component solvers in the portfolio. However, since in practice predictions cannot be completely accurate, we can achieve a better expected performance by selecting only a subset of good algorithms in step 7. The performances of these algorithms should be relatively uncorrelated and each algorithm should achieve the best performance on at least some of the training instances. In the construction of SATzilla this subset selection is automatized by performing an exhaustive search over all possible choices: For each subset of algorithms, the process considers the corresponding portfolio, which uses the empirical hardness models trained in step 6 to select an algorithm from the subset. Each such portfolio is evaluated by measuring its performance on the validation data set, and the subset which yields the best performance is selected.

In addition to the main solvers, we select a set of pre-solvers, which are always run for a short amount of time (typically a few seconds) before computing the features. This enables the portfolio to solve easy instances quickly, without any overhead from using the empirical hardness models. A good set of pre-solvers is such that a large proportion of training instances is solved quickly by at least one solver in the set. Consequently, the set should consists of solvers that solve many instances quickly and are relatively uncorrelated with each other. For decision problems such as SAT, pre-solvers may also include local search algorithms, which may find a solution fast if it exists, but cannot solve all instances. In later versions of SATzilla, local search methods were also considered among component solvers.

Finally, among all algorithms we choose one backup solver, which is run exclusively in the case when the instance isn't solved by pre-solvers and computing the features takes too long. A sensible choice for a backup solver is the algorithm, which achieves the best average-case performance on such instances in the validation set. In case the number of such instances is too small to justify the choice, we can simply use the best component solver as a backup solver.

3 Building empirical hardness models

Once the set of training instances and the set of candidate features have been selected, the features are computed for each instance and the performance of each algorithm on those instances is measured, as described in Section 2. Given this data, the task is to build a predictive model that exploits correlations between features and the performance of algorithms. There are various machine learning techniques to do this, such as classification and regression. The specific method used in SATzilla is ridge regression, a type of linear regression that penalizes for large coefficients in the learned function.

We detail the ridge regression technique in Section 3.1. In Section 3.2 we show how to make regression stable by excluding non-informative features automatically. In Section 3.3 we describe a way to extend linear regression to accommodate non-linear correlations. In Section 3.4 we show how to include information from runs that take too long to finish and have to be terminated prematurely. Finally, in Section 3.5 we show how to combine the presented models into hierarchical hardness models that take into account differences between fundamentally different types of instances. Since a hardness model is trained separately for each algorithm, we consider a fixed

algorithm for the remainder of this section.

3.1 Regression

The basic idea in regression is to fit a function – based on the training data – that takes as its input a set of *explanatory variables* (in our case, features) and outputs a prediction of a *dependent variable* (running time). Regression is a natural choice when all variables involved are continuous or at least numerical, as is the case with running time and many features. The function is typically fitted with the objective of minimizing an error metric that measures how much predictions on the training instances differ from their actual running times. To avoid overfitting, the hypothesis space, i.e. the set of all possible functions, should be kept relatively simple.

The most common type of regression is *linear regression* in which the predicted variable is a linear function of all other variables. To define the technique formally, let n be the number of training instances and m the number of features. Let Φ be an $n \times m$ matrix where $\Phi_{i,j}$ is the value of feature j for instance i. Let $t = [t_1, \ldots, t_n]$ be a vector where t_i is a measure of the algorithm's performance on instance i. Since running times can be very large, in SATzilla each t_i is actually a logarithm of the true running time. Based on Φ and t, we fit a function f_w of form

$$f_w(\varphi) = w^T \varphi = w_1\varphi_1 + w_2\varphi_2 + \cdots + w_m\varphi_m \ ,$$

where $\varphi = [\varphi_1, \ldots, \varphi_m]$ is an input vector of feature values, and $w = [w_1, \ldots, w_m]$ is a vector of free parameters. The parameters are fitted to minimize the error metric, which in case of linear regression is typically

$$\sum_{i=1}^{n} (f([\Phi_{i,1}, \ldots, \Phi_{i,m}]) - t_i)^2 \ ,$$

that is, the sum of squared differences between predicted and true running times. This is an appealing choice as it favors models that mispredict all instances equally [LBNS02]. Furthermore, it can be minimized simply by setting $w = (\Phi^T\Phi)^{-1}\Phi^T t$, which means the offline computation is dominated by the roughly cubic time required by matrix inversion [NLBH+04].

A drawback with the squared errors metric is that it penalizes differences superlinearly and may thus be unsuitable if there are many outliers [NLBH+04]. In SATzilla this problem is addressed by using *ridge regression*, a variant of linear regression with a slightly modified error metric that penalizes large coefficient in w and thus increases

numerical stability. Ridge regression is achieved by setting $w = (\delta I + \Phi^T \Phi)^{-1} \Phi^T t$, where I is the identity matrix and δ is a small penalty constant. In SATzilla the value $\delta = 10^{-3}$ was found to work well. Note that when $\delta = 0$, this is just "normal" linear regression. The computational complexity is also essentially the same.

Other techniques such as lasso regression, SVN regression and Gaussian process regression have also been considered in the development of SATzilla. However, all of these are computationally more expensive while not offering substantially better results than ridge regression [LBNS02, NLBH+04, HHHLB06, XHHLB08].

3.2 Feature selection

As mentioned in Section 2.3, in general we wish to discard useless features before applying the regression techniques presented above. The main reason for this is the fact that linear regression tends to become unstable when several features are highly correlated with each other or have no correlation with algorithm performance. On the other hand, eliminating useless features reduces the amount of computation required in the online phase and also makes the resulting model easier to interpret. Specifically, we may be interested in discovering features useful to the model by comparing the corresponding coefficients in the linear function. This becomes difficult in the presence of correlated features, as two correlated but uninformative features could have arbitrarily large coefficients with opposite signs, thus appearing to be important while having no actual contribution [NLBH+04].

Whereas the initial feature selection required specific domain expertise, the task of identifying redundant features can be done automatically with statistical means. In theory, the best subset of features could be selected by using exhaustive search with cross-validation, in the same fashion as done with algorithms in Section 2.4. Unfortunately, this becomes infeasible if the number of features is very large, especially after applying expansion techniques discussed in Section 3.3. In practice, subset selection is performed with heuristic methods, which may produce good sets even if they fail to find the optimal choice.

The specific technique employed by SATzilla is *forward selection*, which starts with an empty feature set and then greedily adds features, always picking the feature that minimizes the resulting cross-validation error. A dualistic approach known as *backward elimination* instead starts with the complete set of features, removing one feature in each step with the same objective. A third common technique is *sequential*

replacement, which extends forward selection by adding the possibility of replacing a feature with another in each step. In previous work [NLBH+04], a combination of all three has been considered, along with shrinkage techniques specific to linear regression.

3.3 Basis function expansion

An obvious limitation of linear regression is that, as such, it cannot properly fit non-linear correlations. While the technique can be generalized to fit e.g. polynomials of higher degrees, an easier and more general method is to introduce new features that are functions of the original ones. Specifically, if we wish to consider dependencies expressed by function g, we can add $g(x)$ for each feature x, thus allowing such correlations to be "noticed" by the regular linear regression. This technique is generally known as *basis function expansion* and, in theory, allows the model to take into account correlations expressed by any arbitrarily complex functions.

The basis functions need not be limited to mappings from one feature to another. Indeed, SATzilla employs a special case known as *quadratic expansion*, which adds all pairwise products of original features, thus allowing the model to fit quadratic correlations. One problem with such an expansion is that the number of features can quickly explode, thus making the regression task intractable. Furthermore, since the expansion effectively broadens the hypothesis space, it can also lead to overfitting [NLBH+04]. However, as most new features are likely to be either non-informative or correlated with other features, the problem can be addressed by using the techniques discussed in Section 3.2. In SATzilla, forward selection is applied both before and after performing quadratic expansion, thus greatly reducing the final set of features used in regression.

3.4 Dealing with terminated runs

While the high variance in running times of algorithms is the primary motivation for constructing portfolios, the same fact unfortunately makes the necessary task of measuring running times costly. This problem is particularly evident with NP-hard problems such as SAT, where an algorithm could run for weeks without yielding a solution. Spending this much time just to find out how long the algorithm runs is naturally not feasible in practice. Instead, a common solution is to set up a fixed cutoff time and terminate any algorithm exceeding it prematurely.

This raises the question of how the information that a run was terminated should be used when training the empirical hardness models. One solution that has been considered in experiments is to simply ignore terminated runs entirely. Intuitively, this approach seems illogical, since having to terminate a run suggests that the instance is potentially very hard for the algorithm. Naturally, we would like to use this information to avoid running the algorithm on similar instances, just like we use runs that terminate close to the cutoff time. Another proposed method is to treat the cutoff time as the true running time of the algorithm. While this approach takes the hardness of the instance into account, it's nevertheless unsatisfactory as it systematically underestimates the hardness of terminated instances, potentially resulting in overoptimistic predictions [XHHLB07].

A more theoretically sound method is to construct a model that treats the cutoff time as a lower bound on the true running times of terminated runs, thus making as much use of the information available as possible. In statistics this is commonly known as *censoring* the true value. The approach of using censored samples has been studied e.g. in survival statistics, and was apparently first applied to SAT in [GS06b]. In the SATzilla approach, censored samples are included using a method presented in [SH79]. This technique first trains an empirical hardness model while treating the cutoff times as if they were the true running times on the terminated instances. Then, the following two steps are repeated.

1. The trained model is used to predict the running times of the terminated instances.

2. A new empirical hardness model is trained, this time treating the predictions made in the previous step as the true running times of the terminated instances (for other instances the actual running times are still used).

Depending on the machine learning method used in model construction, the predictions in the first step will likely differ from the cutoff times. The two steps are repeated until the predictions converge, i.e. do not significantly change between two iterations. Empirically, this method has proven to clearly outperform both ignoring the terminated runs and using the cutoffs times as true values. The theoretical explanation is that, unlike the two other approaches, the iterative method does not introduce bias into the model.

3.5 Hierarchical hardness models

It has been observed that empirical hardness models can achieve a better accuracy in predictions when restricted to some fundamental "class" of instances. In SAT, specifically, there is a natural division into satisfiable and unsatisfiable instances. If a given instance is known to be satisfiable, it makes sense to predict its hardness using a model that has been trained on safisfiable instances only, and vice versa. In practice, of course, we want to predict the hardness of instances even if we do not have any knowledge of their classes a priori.

This motivates the use of *hierarchical hardness models* as employed by SATzilla [XHLB07, XHHLB08]. Let Z be the set of instance classes. Offline, we first train for each $z \in Z$ a so called *conditional model* M_z. Each M_z is trained using the same methods as presented above, but with instances of class z only. We then train a classifier to predict the class of a given instance, based on the same features as used by the conditional models to predict running times. Specifically, the classifier returns a vector $p = [p_1, \ldots, p_{|Z|}]$, where each p_z is the probability that the instance belongs to class z. In SATzilla, Sparse Multinomial Logistic Regression is used for classification, though any method which returns probabilities would do as well.

Online, given an instance with a feature vector φ, we first use the classifier to obtain the probabilities p based on φ. Then, based on φ and p, we must somehow combine predictions given by the conditional models. One possibility would be to simply use the model M_z for which the probability p_z is highest. However, this approach doesn't take into account the accuracy of models and thus ignores the potentially high cost of making a bad choice. Indeed, in SAT it has been observed that, while selecting a good conditional model may significantly improve predictions, making a wrong choice can have an even greater negative impact [XHLB07]. Furthermore, even if we could perfectly guess the class of a given instance, the best conditional model for that instance (i.e. the one that selects the best algorithm) isn't necessarily the one trained on the same class.

A more theoretically justified approach is to treat the hierarchical hardness model as a probabilistic model, in which the feature vector φ and the classifier probabilities p are observed variables and the running time τ is an unknown variable. Under this interpretation, we can get the expected running time by averaging over predictions of all conditional models, weighting each model by how likely it is to yield the best predictions. To this end, let $\hat{z}$ be an additional variable that denotes (the class of) the best conditional model for a given instance. Our belief in the value of $\hat{z}$ depends

exclusively on φ and p. Thus, the expected running time conditional on the observed variables is

$$\mathbf{E}[\tau \mid \varphi, p] = \sum_{z \in Z} P(\hat{z} = z \mid \varphi, p) \cdot w_z^T \varphi$$

where w_z is the coefficient vector for the linear function of model M_z.

The only problem with evaluating this formula is that we do not know the weighting functions $P(\hat{z} = z \mid \varphi, p)$. However, just like we learned the prediction functions in Section 3.1, we can the learn the weighting functions by considering a suitable hypothesis space and fitting free parameters to minimize some error metric between the predicted and true running times of training instances. In SATzilla the weighting functions are so called softmax functions commonly used in similar cases, and the error metric is the sum of squared differences. We omit further discussion of these choices.

An early version of SATzilla employs only two conditional models, one for satisfiable and another unsatisfiable instances. A later version uses six models, each trained or satisfiable or unsatisfiable instances of three different target distributions, which has turned out to improve predictions greatly. We remark that the approach presented here is (at least theoretically) applicable to machine learning techniques other than linear regression and performance measures other than expected running time.

4 Discussion

4.1 Results

The performance of SATzilla was measured in the 2007 SAT Competition, where it competed in all of the four different categories: industrial instances, handmade instances, randomly generated instances, and one category comprising all instances in the other three categories. Each category was further divided into satisfiable, unsatisfiable, and all instances.

In each category, SATzilla was able to clearly outperform its component solvers, achieving a better average running time than any component solver would have achieved alone. The greatest improvement over component solvers was achieved for random and all instances, where the heterogenous instance distribution emphasized differences between solvers and thus made good algorithm selection more significant. For random instances the performance was very close to a hypothetical oracle that

always selects the best algorithm, with an option of not running pre-solvers and with with no overhead from feature computation. Such an oracle provides a lower bound on the performance that SATzilla can hope to achieve.

Classification into satisfiable and unsatisfiable instances also turned out to be quite accurate, with 92%, 70%, 94% and 78% of instances from the respective categories (industrial, handmade, random, all) classified correctly. In a closer analysis it was noted that an unsatisfiable instance was far more commonly misclassified as satisfiable than vice versa, most clearly in the the handmade category.

4.2 Conclusion

Algorithm portfolios comprise a broad range of techniques for utilizing multiple algorithms to achieve a better average performance than any single algorithm. Such approaches can be useful when there is no single best algorithm for a problem and the performance of different algorithms is relatively uncorrelated. Algorithm portfolios are particularly appealing for NP-hard problems where the choice of an algorithm can have a huge impact on the time required to solve an instance.

Constructing a successful portfolio depends crucially on the ability to estimate each algorithm's (relative or absolute) performance on a given instance. This estimation should be relatively cheap so that the overhead is adequately compensated by making good choices on which algorithms to run. The SATzilla approach presented uses machine learning to train empirical hardness models, which predict an algorithm's performance based on hardness-related features of the problem instance. Aside from offering a solution to the algorithm selection problem, such models are also useful for gaining understanding of a problem's hardness. This is particularly useful for NP-hard constraint problems, which can be otherwise very difficult to analyze.

In recent years there have also been other approaches to building SAT portfolios that are competitive and much more simple than the SATzilla approach. One such portfolio [NMJ11] uses k-means to select an algorithm that has performed well on close neighbors of the given instance. Other techniques involve using meta-level classifiers to assess the accurracy of predictions, as well as sharing knowledge between sequential solvers [MSSS13]. Such techniques have previously been used in parallel solvers and, as some of them enhance the behavior of individual solvers, they can result in portfolios that outperform the hypothetical oracle.

References

GKK+11 Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M. T. and Ziller, S., A portfolio solver for answer set programming: Preliminary report. *LPNMR*, 2011, pages 352–357.

GS01 Gomes, C. P. and Selman, B., Algorithm portfolios. *Artificial Intelligence*, 126,1-2(2001), pages 43–62.

GS06a Gagliolo, M. and Schmidhuber, J., Dynamic algorithm portfolios. *ISAIM*, 2006.

GS06b Gagliolo, M. and Schmidhuber, J., Impact of censored sampling on the performance of restart strategies. *CP*, 2006, pages 167–181.

HHHLB06 Hutter, F., Hamadi, Y., Hoos, H. H. and Leyton-Brown, K., Performance prediction and automated tuning of randomized and parametric algorithms. *Proceedings of the 12th international conference on Principles and Practice of Constraint Programming*, CP'06, Berlin, Heidelberg, 2006, Springer-Verlag, pages 213–228.

HHN08 Hebrard, E., Holl, A. and Nugent, C., Using case-based reasoning in an algorithm portfolio for constraint solving. *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

HLH97 Huberman, B. A., Lukose, R. M. and Hogg, T., An economics approach to hard computational problems. *Science*, 27, pages 51–53.

LBNA+03 Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J. and Shoham, Y., A portfolio approach to algorithm selection. *IJCAI*, 2003, pages 1542–1542.

LBNS02 Leyton-Brown, K., Nudelman, E. and Shoham, Y., Learning the empirical hardness of optimization problems: The case of combinatorial auctions. *CP*, 2002, pages 556–572.

MPLMS08 Matos, P. J., Planes, J., Letombe, F. and Marques-Silva, J., A MAX-SAT algorithm portfolio. *ECAI*, 2008, pages 911–912.

MSSS13 Malitsky, Y., Sabharwal, A., Samulowitz, H. and Sellmann, M., Boosting sequential solver portfolios: Knowledge sharing and accuracy prediction. *LION*, 2013.

NLBH+04 Nudelman, E., Leyton-Brown, K., Hoos, H. H., Devkar, A. and Shoham, Y., Understanding random SAT: Beyond the clauses-to-variables ratio. *CP*, 2004, pages 438–452.

NMJ11 Nikolic, M., Maric, F. and Janicic, P., Simple algorithm portfolio for SAT. *CoRR*, abs/1107.0268.

SH79 Schmee, J. and Hahn, G. J., A Simple Method for Regression Analysis with Censored Data. *Technometrics*, 21,4(1979), pages 417–432.

SS12 Streeter, M. J. and Smith, S. F., New techniques for algorithm portfolio design. *CoRR*, abs/1206.3286.

XHHLB07 Xu, L., Hutter, F., Hoos, H. H. and Leyton-Brown, K., The design and analysis of an algorithm portfolio for SAT. *CP*, 2007, pages 712–727.

XHHLB08 Xu, L., Hutter, F., Hoos, H. H. and Leyton-Brown, K., SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, 32, pages 565–606.

XHLB07 Xu, L., Hoos, H. H. and Leyton-Brown, K., Hierarchical hardness models for SAT. *CP*, 2007, pages 696–711.