Snappy: A Simple Algorithm Portfolio
(Tool Paper)

Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, Meinolf Sellmann

IBM Watson Research Center, Yorktown Heights, NY 10598, USA
{samulowitz,creddy,ashish.sabharwal,meinolf}@us.ibm.com

Abstract. Algorithm portfolios try to combine the strength of individual algorithms to tackle a problem instance at hand with the most suitable technique. In the context of SAT the effectiveness of such approaches is often demonstrated at the SAT Competitions. In this paper we show that a competitive algorithm portfolio can be designed in an extremely simple fashion. In fact, the algorithm portfolio we present does not require any offline learning nor knowledge of any complex Machine Learning tools. We hope that the utter simplicity of our approach combined with its effectiveness will make algorithm portfolios accessible by a broader range of researchers including SAT and CSP solver developers.

1 Introduction

Algorithm portfolios [cf. 6, 8, 10] for combinatorial problems such as Boolean Satisfiability (SAT) and Constraint Satisfaction Problems (CSPs) have emerged as a highly successful approach for combining the strength and effectiveness of a variety of core solution techniques ("base solvers") that excel on various subsets of problem instances. By using Machine Learning based data-driven methods to select the most promising algorithm (or a schedule of several promising ones) based on past performance of each base solver on hundreds or even thousands of instances, such portfolios often achieve much more robust performance across a broad range of problem domains than any single base solver. In the context of SAT, a regression based portfolio called `SATzilla2009` [16] showcased the benefits of algorithm portfolios by dominating several categories at annual SAT Competitions (`www.satcompetition.org`) in the past. Over the years, researchers working on portfolio algorithms have employed a number of techniques with their own benefits, including Scheduling based approaches [11, 15], Decision Forests [18], Nearest Neighbor classification [9], and Collaborative Filtering [14].

Unfortunately, despite the existence of such techniques for several years, portfolio algorithms have not truly been adopted by the SAT and CSP research communities. The users of these portfolios are often none other than their own creators, and sometimes a few other research groups working on competing portfolios. In other words, portfolios have not been embraced by the community as a generic tool. This is in sharp contrast with base solvers such as `Minisat` [13], `Glucose` [2], `CryptoMinisat` [12] and `Lingeling` [3], just to name a few, which are

widely used by a large subset of the SAT community on a regular basis. Similarly, CSP solvers such as `Gecode` [5] and `Choco` [4] are commonly used.

Our hope is that the availability of an easy to use portfolio tool would encourage SAT and CSP researchers to explore new avenues. E.g., if there was a new restart strategy for `Glucose` that made it work better on some instances but worse on others, a simple 2-solver portfolio could be employed to try to achieve the best of both worlds.

The motivation for this work is the belief that a key reason for the lack of common adoption of portfolio solvers is usability. For example, the "portfolio builder" or trainer of some existing portfolios such as `3S` is not publicly available due to proprietary reasons, while that of others such as `SATzilla2012` requires a license to a relatively new version of MATLAB as well as enough familiarity with the offline training aspect of the code to be able to effectively adapt it to every new benchmark. Further, training can be expensive: it can take a few minutes to hours for `SATzilla2009` and `3S`, and much more for `SATzilla2012` due to quadratic scaling in the number of base solvers [1]. The resulting effort and time investment often deters most researchers from benefiting from powerful portfolio technology.

The goal of this work is to develop an algorithm portfolio that (a) is simple to use and experiment with, (b) is data-driven and can thus exploit years of experience with various base solvers, (c) has *no offline training phase* or "portfolio builder", and (d) can improve its own performance through *online learning*.

To this end, we propose a training-less algorithm portfolio called `snappy` (Simple Neighborhood-based Algorithm Portfolio in PYthon). Packaged as a single file written in Python (`www.python.org`), it relies on a relatively simple prediction mechanism based on base solver performances on nearest neighbors. The availability of an extensive set of Python libraries allows the user to easily experiment with various aspects of interest such as distance measures, neighborhood size, weighting, cost function, feature reduction, etc. By skipping the traditional training phase altogether, `snappy` opens up the possibility of learning and improving itself on-the-fly when run on a number of test instances.

Our aim is not to create the best portfolio approach to date but to provide a tool that is effective and has a low barrier to being adopted by a wide range of algorithmic researchers. We provide empirical evidence that the simple, training-less approach embodied by `snappy` can be competitive with the current state of the art in portfolios for SAT. This is not to say that more sophisticated approaches to portfolios do not have value. For example, the SAT/UNSAT predictor in `SATzilla` is a powerful tool in itself, and so is the mixed integer programming based algorithm scheduler of `3S`. Nonetheless, the sophisticated approaches also have a higher barrier to entry, which we hope to overcome through the simple yet competitive approach of `snappy`.

2 Background

We assume familiarity with the basics of SAT and CSP solvers, and briefly discuss algorithm portfolios in this context. The main concept behind any solver

portfolio is to utilize data (collected offline) on the performance of various base solvers on a relatively large set of "training" instances in order to predict, given a new "test" instance, which single base solver or schedule of base solvers is most cost-effective for solving the test instance. The cost of interest is often the runtime, but may be other quantities such as solution quality in case of an optimization problem. The various portfolios referred to earlier differ in *how* they go about using training data to make this prediction. One aspect common to them is the abstract representation of instances in terms of a (small) set of "features", such as instance size, constraint density, etc.

Once the training data is collected, state-of-the-art portfolios such as `3S` and `SATzilla` (winners of the past two SAT Competitions) go through an extensive offline training and internal cross-validation phase in order to learn the parameters of their prediction model. This phase typically requires understanding the portfolio builder well and can often be costly [1]. In the interest of space, we refer the reader to Xu et al. [16, 17] and Kadioglu et al. [9] for details.

3 Portfolio Tool Description

Our portfolio, `snappy`, is packaged as one Python file available at:
`http://researcher.watson.ibm.com/researcher/files/us-samulowitz/snappy.zip`
Its basic usage is reported by running `python snappy.py -h`.

The tool can be run in two modes, analysis (`ana`) and execution (`exe`). In both modes, like all portfolio solvers, it expects "training data" which consists of a `.times` file specifying, in a header-less comma-separated format, the runtime of each base solver (columns) on a number of instances (rows), and a `.features` file specifying the features (columns) of each instance. Additionally, the desired timeout is specified for performance evaluation purposes. Options in both modes include what neighborhood size to consider, what penalty to use for performance evaluation when hitting the timeout, what distance measure to use (e.g., Euclidean or Minkowski), and whether to use distance-based weighting.

In the analysis (`ana`) mode, the tool expects a set of test instances and evaluates the performance of the portfolio on this test dataset in relation to the single best base solver (SBS) or the virtual best solver (VBS). The intent is that users can use the analysis mode to experiment with and tune the parameters of the portfolio for their benchmark of interest. One may optionally specify a given pre-schedule of base solvers, which is taken into account in all performance evaluations. Enabling the online "learning" mode makes `snappy` alter its behavior based on the runtimes it has observed on test instances.

In the execution (`exe`) mode, the tool expects a comma-separated list of features. Alternatively, the user may specify the name of a feature extraction tool and a test instance such that executing the tool will produce a comma-separated list of features of the instance.

The main components of `snappy` are:

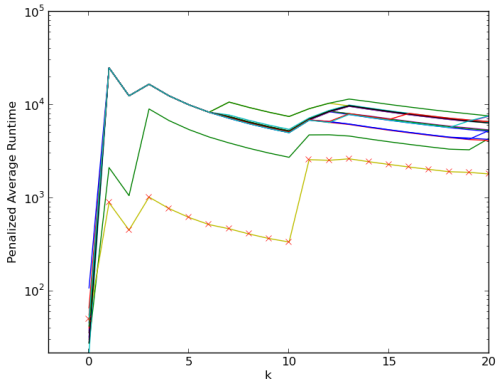

Fig. 1. A sample of the PAR score of various algorithms for varying k

Data Normalization: All feature values are cut off at a fixed maximum/minimum value and features are scaled to the same unit (e.g., [0..1]). This is motivated by the fact that we use distance to select nearest neighbors.

Feature Space and Distance Measures: There exist various ways of computing the distance between feature vectors besides the Euclidean distance, such as standardized Euclidean, Minkowski, and Canberra, which are available through the `scipy.spatial.distance` Python package. More sophisticated measures such as the Mahalanobis distance impose requirements on the underlying data (e.g., being able to take the inverse of the Covariance matrix). To enable such measures we also support a PCA based Eigenvector representation of the data where features with low variance are eliminated. This allows the approach to stay somewhat more robust against uninformative and potentially misleading features.

Aggregation Schemes and Solver Selection: Since we do not fix a single neighborhood size k a priori but choose it dynamically (see e.g., [7]), we employ an aggregation scheme based on the performance of each base algorithm for each possible $k \leq k_{max}$. Essentially we have a $k_{max} \times$ #Algorithms matrix and each entry (k, A) contains a performance measure, namely the average penalized runtime, of algorithm A on the first k neighbors of the test instance. A sample visualization of such a matrix is shown in Figure 1 where we plot the penalized average runtime (PAR) per $k \in [0..20]$ and algorithm (1 to 18) including the (unknown) performance of each algorithm on the test instance ($k = 0$). The line corresponding to the selected algorithm is marked with crosses.

Based on this data one can deploy various aggregation schemes to select a solver. One simple scheme, which is the "default" setting and underlies all experiments reported in this paper, is one that selects the base algorithm that has the minimum PAR score for some $k \in [k_{min}, k_{max}]$:

$$\arg \min_{A \in \text{Algorithms}} \min_{k \in [k_{min}, k_{max}]} \text{PAR}(A, k \text{ nearest neighbors of the test instance})$$

One can also consider various other aggregation schemes such as distance-based weighted voting. Some of these are also available in `snappy`.

Online-Learning: Since we do not perform any offline learning, it gives us the liberty to easily incorporate knowledge that becomes available as the portfolio is run. To this end we consider the following ways of adding knowledge incrementally. First, every time a test instance is considered we add its feature vector to the current set of training instances and re-normalize the entire data again before selecting the next algorithm. We noticed that re-normalizing has a positive impact on performance. Second, after we select an algorithm for a given test instance, we add the actual runtime information for the selected algorithm on this instance to our data set. This means that, over time, the k-neighborhood of different algorithms might differ as some algorithms will be selected more often than others. This simple addition also seemed to have a positive effect.

4 Empirical Demonstration

We demonstrate the effectiveness of `snappy` by comparing it with two state-of-the-art algorithm portfolios for SAT that won in the last two competitions: `3S` [9] and `SATzilla2012` [18]. Our comparison does not rely on generating any new runtime or feature data, or running any base solver, but is based entirely on benchmarks used previously by the respective portfolio designers to showcase the merits of `3S` and `SATzilla`, resp. Being competitive on these benchmarks thus speaks to the strength of our simple approach.

Our Python based tool is expected to work well across multiple platforms. All experiments reported here were conducted on a Linux machine with Python 2.6.6 installed with the following packages: Numpy 1.6.2, Scipy 0.11, and Matplotlib 1.2. All performance numbers of `snappy` are based on one, fixed, "default" setting of various parameters. We note that the performance of `snappy` varies with different settings of the command-line parameters and we expect the users of this tool to experiment with parameter settings that work best in their domain. Finally, all comparisons are performed on exactly the same datasets (in particular identical solver base) the results for `3S` and `SATzilla` were obtained on.

We begin with a comparison with `3S` using no scheduler[1] in Table 1. The first benchmark is the one with challenging training/test splits used in the original paper on `3S` [9]. The other four benchmarks are based on an updated set of solvers and instances used to train the `ISS` solver (a information-sharing extension of `3S`) for SAT Challenge 2012. These four benchmarks are divided into two categories based on the original 48 `SATzilla` features ("f1") and a new set of more efficiently computed 32 features ("f2") used by `ISS`. Within each category, there are cross-validation splits ("10-fold") and a competition-style split ("comp").

While we trained and evaluated `3S` on all of these benchmarks, we were unsuccessful in doing so with `SATzilla2012` as this appeared to require significant

[1] When `3S` and `snappy` use the same schedule, the relative difference in performance remains similar to what is reported in Table 1.

Table 1. `3S` vs. `snappy` (with default settings): average percentage of instances solved and average PAR-1 score in seconds

Benchmark				`3S`		`snappy`	
Name	#Alg	#Feat.	Timeout	%	PAR-1	%	PAR-1
SAT-2011-splits	37	48	5000	91.23	772.8	94.52	512.5
SAT-2012-10fold-f1	72	48	2000	96.59	174.3	96.48	161.8
SAT-2012-comp-f1	72	48	2000	83.05	556.4	83.77	560.5
SAT-2012-10fold-f2	72	32	2000	97.23	146.1	96.17	167.5
SAT-2012-comp-f2	72	32	2000	85.42	499.1	85.42	526.3

Table 2. `SATzilla` vs. `snappy` (with default settings): average percentage of instances solved and average PAR-1 score in seconds

Benchmark				`SATzilla`		`snappy`	
Name	#Alg	#Feat.	Timeout	%	PAR-1	%	PAR-1
Industrial	18	125	5000	75.3	1685	72.6	1789
Crafted	15	125	5000	66.0	2096	63.3	2198
Random	9	125	5000	80.8	1172	80.3	1221

familiarity with the underlying MATLAB code and a need to adapt the code for every benchmark so as to give it a fair chance of success. Instead we compare the performance of `snappy` directly with the results reported by Xu et al. [17]. The corresponding benchmark is available at the `SATzilla` webpage[2] and is comprised of three categories: application, crafted, random. The results are shown in Table 2. It is worth noting that this version of `SATzilla` employs a scheduler before selecting a long-running algorithm. While this data set comprises 125 features we only consider the first 48 features in `snappy`.

As the performance numbers in both tables demonstrate, `snappy`, despite its simplicity and ease of use, is quite competitive with the state of the art.

5 Conclusion

We presented a new algorithm portfolio approach that we hope will be easy to adopt by a broad range of researches, including those designing the base solvers that underlie any such portfolio. Our tool, `snappy`, does not only provide a strong baseline, but can also be easily extended by its users. For instance, if portfolio performance becomes an issue[3] one could use a priori k-means clustering to reduce the number of instances one needs to consider—which can be done by adding just few lines of code using the `scipy` library. Similarly, if one wants to automatically tune the high level parameters (e.g., distance measure), one can also add cross-validation using one line of Python. We believe this kind of experimentation flexibility can be immensely valuable from a research perspective.

[2] `http://www.cs.ubc.ca/labs/beta/Projects/SATzilla`. We thank Lin Xu for providing the cross-validation splits used in prior work [17] on this dataset.

[3] On the largest dataset used in this paper with about $5,000$ instances, it takes around 1,5 seconds to select an algorithm.

References

[1] R. Amadini, M. Gabbrielli, and J. Mauro. An empirical evaluation of portfolios approaches for solving csps. In *CPAIOR-2013*, 2013.

[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solver. In *IJCAI-09*, July 2009.

[3] A. Biere. Lingeling and friends at the sat competition 2011. Technical report, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[4] L. Cnrs. choco: an open source java constraint programming library. *White Paper 14th International Conference on Principles and Practice of Constraint Programming CPAI08 Competition*, pp. 7—14, 2008. URL `http://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf`.

[5] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[6] C. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence Journal*, 126(1-2):43–62, 2001.

[7] M. S. Huda, K. M. R. Alam, K. Mutsuddi, M. K. S. Rahman, and C. M. Rahman. A dynamic k-nearest neighbor algorithm for pattern analysis problem. In *3rd International Conference on Electrical & Computer Engineering*, 2004.

[8] J.R.Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[9] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm selection and scheduling. *CP*, 2011.

[10] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham. A portfolio approach to algorithm selection. *Proc. of the 15th Int. Joint Conference on Artificial Intelligence (IJCAI)*, page 1542001543, 2003.

[11] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[12] M. Soos. CryptoMiniSat 3.1, 2013. *http://www.msoos.org/cryptominisat2*.

[13] N. Sorensson and N. Een. MiniSAT 2.2.0, 2010. *http://minisat.se*.

[14] D. Stern, H. Samulowitz, R. Herbrich, T. Graepel, L. Pulina, and A. Tacchella. Collaborative expert portfolio management. *AAAI*, 2010.

[15] M. Streeter and S. Smith. Using decision procedures efficiently for optimization. *ICAPS*, pp. 312–319, 2007.

[16] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *JAIR*, 32(1):565–606, 2008.

[17] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In *SAT-2012*, pp. 228–241, 2012.

[18] L. Xu, F. Hutter, J. Shen, H. Hoos, and K. Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. solver description, 2012. SAT Challenge 2012.