A Portfolio Solver for Answer Set Programming: Preliminary Report

M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub*, M. Schneider, and S. Ziller

Institut für Informatik, Universität Potsdam

Abstract. We propose a portfolio-based solving approach to Answer Set Programming (ASP). Our approach is homogeneous in considering several configurations of the ASP solver *clasp*. The selection among the configurations is realized via Support Vector Regression. The resulting portfolio-based solver *claspfolio* regularly outperforms *clasp*'s default configuration as well as manual tuning.

1 Introduction

Answer Set Programming (ASP; [1]) has become a prime paradigm for declarative problem solving due to its combination of an easy yet expressive modeling language with high-performance Boolean constraint solving technology. In fact, modern ASP solvers like *clasp* [2] match the performance of state-of-art satisfiability (SAT) checkers, as demonstrated during the last SAT competition in 2009. Unfortunately, there is a price to pay: despite its theoretical power [3], modern Boolean constraint solving is highly sensitive to parameter configuration. In fact, we are unaware of any true application on which *clasp* is run in its default settings. Rather, in applications, "black magic" is used to find suitable search parameters. Although this is well-known and also exploited in the SAT community, it is hardly acceptable in an ASP setting for the sake of declarativity. The most prominent approach addressing this problem in SAT is *satzilla* [4], aiming at selecting the most appropriate solver for a problem at hand.

Inspired by *satzilla*, we address the lack of declarativity in ASP solving by exploring a portfolio-based approach. To this end, we concentrate on the solver *clasp* and map a collection of instance features onto an element of a portfolio of distinct *clasp* configurations. This mapping is realized by appeal to Support Vector Regression [5]. In what follows, we describe the approach and architecture of the resulting *claspfolio* system. We further provide an empirical analysis contrasting *claspfolio*'s performance with that of *clasp*'s default setting as well as the manually tuned settings used during the 2009 ASP competition. In addition, we compare the approach of *claspfolio* with *paramils* [6], a tool for parameter optimization based on local search.

2 Architecture

Given a logic program, the goal of *claspfolio* is to automatically select a suitable configuration of the ASP solver *clasp*. In view of the huge configuration space, the attention is

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

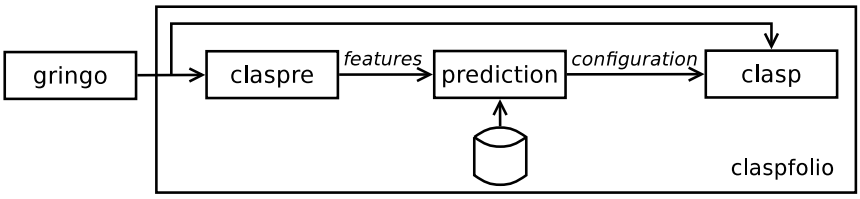

Fig. 1. Architecture of *claspfolio*

limited to some (manually) selected configurations belonging to a portfolio. Each configuration consists of certain *clasp* options, e.g., "*--heuristic=VSIDS --local-restarts*." To approximate the behavior of such a configuration, *claspfolio* applies a model-based approach predicting solving performance from particular features of the input.

As shown in Figure 1, ASP solving with *claspfolio* consists of four parts. First, the ASP grounder *gringo* [7] instantiates a logic program. Then, a light-weight version of *clasp*, called *claspre*, is used to extract features and possibly even solve a (too simple) instance. If the instance was not solved by *claspre*, the extracted features are mapped to a score for each configuration in the portfolio. Finally, *clasp* is run for solving, using the configuration with the highest score. Note that *claspre* and *clasp* use distinct copies of the input (see Figure 1) because preprocessing done by *claspre* may interfere with *clasp* options of configurations in the portfolio.

The features determined by *claspre* can be distinguished into data- and search-oriented ones. The former include 50 properties, such as number of constraints, number or variables, etc. Beyond that, *claspre* performs a limited amount of search to also collect information about solving characteristics. In this way, additional 90 search-related features are extracted, such as average backjump length, length of learned clauses, etc.

Given the features of an instance, *claspfolio* scores each configuration in the portfolio. To this end, it makes use of models generated by means of machine learning during a training phase. In the case of *claspfolio*, we applied Support Vector Regression, as implemented by the `libSVM` package [8]. Upon training, the score $s_k(i)$ of the k-th configuration on the i-th training instance is simply the runtime $t_k(i)$ in relation to the minimum runtime of all configurations in the portfolio: $s_k(i) = \frac{\min_j (t_j(i))}{t_k(i)}$.

A model, i.e., a function mapping instance features to scores, is then generated from the feature-score pairs available for the training set. In production mode, only the features (collected by *claspre*), but not the configurations' scores, are available. Hence, the models are queried to predict the scores of all configurations in the portfolio, among which the one with the highest predicted score is selected for setting up *clasp*.

The portfolio used by *claspfolio* (version 0.8.0) contains 12 *clasp* configurations, included because of their complementary performances on the training set. The options of these configurations mainly configure the preprocessing, the decision heuristic, and the restart policy of *clasp* in different ways. This provides us with a collection of solving strategies that have turned out to be useful on a range of existing benchmarks. In fact, the hope is that some configuration is (a) well-suited for a user's application and (b) automatically selected by *claspfolio* in view of similarities to the training set.

3 Experiments

We conducted experiments on benchmark classes of the 2009 ASP competition [9].[1] All experiments were run on an Intel Xeon E5520 machine, equipped with 2.26 GHz processors and 48 GB RAM, under Linux. The considered systems are *clasp* (1.3.4) and *claspfolio* (0.8.0; based on *clasp* 1.3.4). Runtimes in seconds, per class and in total, are shown in Table 1. The first two columns give benchmark classes along with their numbers of instances (#). The subsequent columns denote particular variants of the considered systems: *clasp* default (*clasp*), *clasp* manually tuned[2] (*clasp*^{m}), *claspfolio* running a random configuration (*claspfolio*^{r}), *claspfolio* running the best configuration[3] (*claspfolio*^{b}), *claspfolio* default (*claspfolio*) as available at [7], and *claspfolio* obtained by cross validation (*claspfolio*^{v}). The runtime per benchmark instance was limited to $1,200$ seconds, and timeouts are taken as $1,200$ seconds within accumulated results. The third last and the last column ($\times$) in Table 1 provide the speedup of *claspfolio* and *claspfolio*^{v}, respectively, over *clasp*, i.e., the runtime of *clasp* divided by the one of *claspfolio* or *claspfolio*^{v}, per benchmark class (and in total in the last row).

The role of *claspfolio*^{v} is to evaluate *claspfolio* on unseen instances. We do so by using 10-fold cross validation where the set of all available instances is randomly divided into a training set and a test set, consisting of 90 and 10 percent of the inspected instances, respectively. The regression models generated on the training set are then evaluated on the (unseen) test set. By repeating this procedure ten times, every instance is once solved based on models not trained on the instance.

Comparing *clasp* with *clasp*^{m} in Table 1, manual tuning turns out to be mostly successful, and it decreases total runtime roughly by a factor of 3. On two classes, Labyrinth and WireRouting, manual tuning was however counterproductive. This can be explained by the 2009 ASP competition mode, revealing only a few of the available instances per benchmark class during a setup phase, so that the manually tuned parameters may fail on unseen instances. In fact, *claspfolio*, trained on a collection of 3096 instances from the Asparagus benchmark repository[4] and the 2009 ASP competition, turns out to be even more successful in total than *clasp*^{m}. In particular, it performs better on Labyrinth and WireRouting, where *clasp*^{m} failed to improve over *clasp*. Of course, there are also benchmark classes on which manual tuning beats *claspfolio* (most apparently, WeightDomSet), but the fact that *claspfolio* exhibits a total speedup of 3.3 over *clasp* clearly shows the potential of automatic parameter selection. Notably, the total runtime of *claspfolio* exceeds the best possible one, *claspfolio*^{b}, only by a factor of 1.45, while the expected runtime of a random configuration, *claspfolio*^{r}, is in total more than a factor of 4 greater than the one of *claspfolio*.

---

[1] Some too easy/unbalanced classes or instances, respectively, of the competition are omitted. On the other hand, we also ran additional instances for some classes. All instances used in our experiments are available at `http://www.cs.uni-potsdam.de/claspfolio`.

[2] The respective parameter settings per benchmark class are reported at `http://dtai.cs.kuleuven.be/events/ASP-competition/Teams/Potassco.shtml`.

[3] Note that results of *claspfolio*^{r} and *claspfolio*^{b} are calculated a posteriori per benchmark instance, using the average or smallest, respectively, runtime of all *clasp* variants in the portfolio.

[4] Available at `http://asparagus.cs.uni-potsdam.de`.

Benchmark Class	#	*clasp*	*clasp*^{m}	*claspfolio*^{r}	*claspfolio*^{b}	*claspfolio*	×	*claspfolio*^{v}	×
15Puzzle	37	510	281	438	111	208	2.4	254	2.0
BlockedNQueens	65	412	374	765	139	264	1.5	410	1.0
ConnectDomSet	21	1,428	54	1,236	30	53	26.9	649	2.2
GraphColouring	23	17,404	5,844	15,304	5,746	5,867	2.9	5,867	2.9
GraphPartitioning	13	135	66	791	57	69	1.9	97	1.4
Hanoi	29	458	130	499	35	175	2.6	233	2.0
Labyrinth	29	1,249	1,728	3.949	112	785	1.5	2,537	0.5
MazeGeneration	28	3,652	569	4,086	558	581	6.2	567	6.4
SchurNumbers	29	726	726	1,193	41	399	1.8	957	0.7
Sokoban	29	18	19	34	12	57	0.3	54	0.3
Solitaire	22	2,494	631	3,569	73	317	7.8	1,610	1.5
WeightDomSet	29	3,572	248	10,091	5	1,147	3.1	5,441	0.6
WireRouting	23	1,223	2,103	1,409	43	144	8.4	289	4.2
Total	377	33,281	12,773	43,364	6,962	10,066	3.3	18,965	1.8

Table 1. Runtimes in seconds and speedups on benchmark classes of the 2009 ASP competition

Comparing *claspfolio*, trained on all available instances, with *claspfolio*^{v}, where training and test sets are disjoint, we see that applying *claspfolio*$^{(v)}$ to unseen instances yields lower prediction quality. If the training set represents the dependencies between features and runtime rather loosely, the regression models hardly generalize to unseen instances, which obstructs a good parameter selection. But even in this case, *claspfolio*^{v} is almost twice as fast as *clasp*, which shows that the trained models are still helpful.

In Table 2, we compare *claspfolio* with *paramils*, an automatic configuration tool based on iterated local search (*FocusedILS*) through the configuration space. Given that *paramils* uses a model-free approach, it can only generalize between homogeneous problem classes regarding the best configuration. In contrast, *claspfolio* is utterly applicable to heterogeneous classes in view of its regression models. To reflect this discrepancy, the column *paramils*^{c} shows the runtimes of the best configurations of *clasp* determined by *paramils* independently for each problem class, while the best configuration found over all problem classes is displayed in column *paramils*^{a}. In both cases, we ran four (randomized) copies of *paramils* for 24 hours with a timeout of 600 seconds per run on an instance, as suggested in [6], and then selected the best configuration found. Also note that, in view of only 377 instances evaluated overall, we did not split instances into a training and a test set, i.e., *paramils* was used to automatically analyze *clasp* configurations rather than predicting their performances.

As it could be expected, the configurations found by *paramils*^{c} are much faster than the global one of *paramils*^{a}. On some problem classes, e.g., WeightDomSet, *paramils*^{c} found configurations that rendered the classes almost trivial to solve. On such classes, the configurations of *paramils*^{c} also yield much better performances than the ones of *claspfolio* and *clasp*^{m}. However, on problem classes including very hard instances, like GraphColouring and Solitaire, the configurations determined by *paramils* were less successful. This can be explained by long runs on instances, so that fewer configurations could be explored by local search within the allotted 24 hours.

Benchmark Class	#	*paramils*^{c}	*paramils*^{a}	*claspfolio*	*claspfolio*^{v}	*clasp*	*clasp*^{m}
15Puzzle	37	104	322	208	254	510	281
BlockedNQueens	65	212	352	264	410	412	374
ConnectDomSet	21	28	686	53	649	1,428	54
GraphColouring	23	7,596	10,865	5,867	5,867	17,404	5,844
GraphPartitioning	13	39	86	69	97	135	66
Hanoi	29	35	147	175	233	458	130
Labyrinth	29	462	3,080	785	2,537	1,249	1,728
MazeGeneration	28	700	2,610	581	567	3,652	569
SchurNumbers	29	278	871	399	957	726	726
Sokoban	29	11	18	57	54	18	19
Solitaire	22	2,374	4,357	317	1,610	2,494	631
WeightDomSet	29	8	2,649	1,147	5,441	3,572	248
WireRouting	23	87	535	144	289	1,223	2,103
Total	377	11,934	26,578	10,066	18,965	33,281	12,773

Table 2. Comparison with *paramils* on benchmark classes of the 2009 ASP competition

Comparing *claspfolio* and *paramils*^{c}, *claspfolio* performs better in total, yet worse on ten of the thirteen classes. One reason is that *claspfolio* is based on a small set of configurations, whereas *paramils* considers a much larger configuration space (about 10^{12} configurations). In addition, *paramils*^{c} determined a suitable configuration individually for each class, while *claspfolio* applies the same configurations and models to all problem classes. In fact, we note that *claspfolio*^{v} performs better than *paramils*^{a}. From this, we conclude that the problem classes are heterogeneous, so that it is unlikely to find a single configuration well-suited for all classes. Thus, *claspfolio* appears to be a reasonable approach for configuring *clasp* for sets of heterogeneous instances.

4 Discussion

In this preliminary report, we described a simple yet effective way to counterbalance the sensitivity of ASP solvers to parameter configuration. As a result, ASP solving regains a substantial degree of declarativity insofar as users may concentrate on problem posing rather than parameter tuning. The resulting portfolio-based solver *claspfolio* largely improves on the default configuration of the underlying ASP solver *clasp*. Moreover, our approach outperforms a manual one conducted by experts.

Although our approach is inspired by *satzilla*, *claspfolio* differs in several ways. Apart from the different areas of application, SAT vs. ASP, *satzilla*'s learning and selection engine relies on Ridge Regression, while ours uses Support Vector Regression. Interestingly, *satzilla* incorporates a SAT/UNSAT likelihood prediction further boosting its performance. Our first experiments in this direction did not have a similar effect, and it remains future work to investigate the reasons for this.

Our experiments emphasize that search for an optimal configuration, e.g., via *paramils* using local search, on one (homogeneous) problem class is more effective than *claspfolio*. But the search time of *paramils* for each problem class makes *claspfolio* more efficient on a set of (heterogeneous) problem classes. In fact, predicting a good configuration with *claspfolio* is almost instantaneous, once the regression models

are trained. A recent approach to learn domain-specific decision heuristics [10] requires modifying a solver in order to learn and apply the heuristics.

It is interesting future work to investigate automatic portfolio generation. New configurations, to add to a portfolio, could be found with *paramils*. First attempts are done with *hydra* [11]. Further related work includes [12–16], whose discussion is however beyond the scope of this paper. Another goal of future work includes the investigation and selection of the extracted features to predict more precisely the runtime. Usually, feature selection decreases the prediction error of machine learning algorithms. In view of this, the potential of *claspfolio* is not yet fully harnessed in its current version.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2. We are grateful to Holger H. Hoos and Frank Hutter for fruitful discussions on the subject of this paper.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University (2003)
2. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In IJCAI'07, AAAI Press (2007) 386–392
3. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning SAT solvers with restarts. In CP'09, Springer (2009) 654–668
4. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR **32** (2008) 565–606
5. Basak, D., Pal, S., Patranabis, D.: Support vector regression. NIP **11**(10) (2007) 203–224
6. Hutter, F., Hoos, H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. JAIR **36** (2009) 267–306
7. `http://potassco.sourceforge.net`
8. `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
9. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In LPNMR'09, Springer (2009) 637–654
10. Balduccini, M.: Learning domain-specific heuristics for answer set solvers. In ICLP'10 Tech. Comm. (2010) 14–23
11. Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: Automatically configuring algorithms for portfolio-based selection. In AAAI'10, AAAI Press (2010) 210–216
12. Gagliolo, M., Schmidhuber, J.: Learning dynamic algorithm portfolios. AMAI **47**(3–4) (2006) 295–328
13. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In AAAI'07, AAAI Press (2007) 255–260
14. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In AICS'08 (2008)
15. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified Boolean formulas. Constraints **14**(1) (2009) 80–116
16. Arbelaez, A., Hamadi, Y., Sebag, M.: Continuous search in constraint programming. In ICTAI'10, IEEE Press (2010) 53–60