Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection

Lin Xu and **Holger H. Hoos** and **Kevin Leyton-Brown**
Department of Computer Science, University of British Columbia
2366 Main Mall, Vancouver, B.C., Canada, V6T 1Z4
{xulin730, hoos, kevinlb}@cs.ubc.ca

Abstract

The AI community has achieved great success in designing high-performance algorithms for hard combinatorial problems, given both considerable domain knowledge and considerable effort by human experts. Two influential methods aim to automate this process: automated algorithm configuration and portfolio-based algorithm selection. The former has the advantage of requiring virtually no domain knowledge, but produces only a single solver; the latter exploits per-instance variation, but requires a set of relatively uncorrelated candidate solvers. Here, we introduce `Hydra`, a novel technique for combining these two methods, thereby realizing the benefits of both. `Hydra` automatically builds a set of solvers with complementary strengths by iteratively configuring new algorithms. It is primarily intended for use in problem domains for which an adequate set of candidate solvers does not already exist. Nevertheless, we tested `Hydra` on a widely studied domain, stochastic local search algorithms for SAT, in order to characterize its performance against a well-established and highly competitive baseline. We found that `Hydra` consistently achieved major improvements over the best existing individual algorithms, and always at least roughly matched—and indeed often exceeded—the performance of the best portfolios of these algorithms.

Introduction

Computationally hard combinatorial problems are ubiquitous in AI. While these problems are intractable in the worst case, in practice they can often be solved by sophisticated heuristic techniques. Work has now progressed to a point where good algorithms can usually be developed for a particular domain given two key ingredients: (i) domain knowledge about the structure that arises in the domain, and the algorithmic techniques that most effectively exploit such structure (e.g., codified in the research literature); (ii) the effort of human experts (e.g., introducing or modifying algorithm components; evaluating them to determine which are effective in the domain). Such algorithms have achieved many notable successes; however, their development has been extremely expensive in terms of human time.

Recently, researchers have focused on automated methods for designing algorithms for new domains, thereby reducing reliance on domain knowledge and/or human experts. Examples include algorithm synthesis (Minton 1993; Gaspero and Schaerf 2007; Monette, Deville, and Hentenryck 2009) as well as parallel portfolios and online selection (Huberman, Lukose, and Hogg 1997; Gomes and Selman 2001; Carchrae and Beck 2005; Gagliolo and Schmidhuber 2006; Streeter, Golovin, and Smith 2007). Two further techniques are particularly relevant to our work.

The first such technique is *automated algorithm configuration* (Gratch and Dejong 1992; Fukunaga 2002; Balaprakash, Birattari, and Stützle 2007; Hutter et al. 2007; 2009; Ansótegui, Sellmann, and Tierney 2009). This approach takes as input a highly parameterized algorithm, a set of benchmark instances, and a performance metric, and then optimizes the algorithm's empirical performance automatically. One prominent application of this idea is `SATenstein` (KhudaBukhsh et al. 2009), which makes use of a large space of stochastic local search (SLS) algorithms for SAT. Automatically configured algorithms from this space were shown to outperform state-of-the-art SLS algorithms on six well-known distributions of SAT instances. Overall, automated configuration is appealing because it requires no domain knowledge beyond a parameterized algorithm framework, and no human effort to target a new domain. However, it produces only a single algorithm, which is designed to achieve high performance overall, but which may perform badly on many individual instances. This drawback is particularly serious when the instance distribution is heterogeneous.

A second influential technique is portfolio-based algorithm selection (Rice 1976; Leyton-Brown et al. 2003b; Guerri and Milano 2004; Leyton-Brown, Nudelman, and Shoham 2009). Under this approach, predictive models are used to select among a portfolio of existing algorithms on a per-instance basis. This design framework has been used to build very practical algorithms, notably the `SATzilla` portfolios (Nudelman et al. 2004; Xu et al. 2008) that won 10 medals in the 2007 and 2009 SAT competitions. This approach has the advantage that it exploits per-instance variation across solvers, but the drawback that it requires relatively significant domain knowledge, including especially a set of relatively uncorrelated candidate solvers.

Once a state-of-the-art portfolio exists for a domain—such as `SATzilla` for various SAT distributions—how should new research aim to improve upon it? One approach is to build new stand-alone algorithms either by hand or using automatic configuration, with the goal of replacing the portfolio. This approach has the drawback that it reinvents the wheel: the new algorithm must perform well on all the instances for which the portfolio is already effective, and must also make additional progress. Alternatively, we might try to build a new algorithm to *complement* the portfolio, which has been dubbed "boosting as a metaphor for algorithm design" (Leyton-Brown et al. 2003a). The boosting algorithm in machine learning builds an ensemble of classifiers by focusing on problems that are handled poorly by the existing ensemble. The proposal is to approach algorithm design analogously, focusing on problems for which the existing portfolio performs poorly. In particular, the suggestion is to use sampling (with replacement) to generate a new benchmark distribution that will be harder for an existing portfolio, and for new algorithms to attempt to minimize average runtime on this benchmark. Indeed, such a method was shown to be very effective for inducing new, hard distributions. While we agree with the core idea of aiming explicitly to build algorithms that will complement a portfolio, we have come to disagree with its concrete realization as described most thoroughly by Leyton-Brown, Nudelman, and Shoham (2009), realizing that average performance on a new benchmark distribution is not always an adequate proxy for the extent to which a new algorithm would complement a portfolio.

We note that a region of the original distribution that is exceedingly hard for all candidate algorithms can dominate the new distribution, leading to stagnation. A further problem is illustrated in the following, more complex example (due to Frank Hutter). Consider a uniform distribution over instance types A, B, and C. The current portfolio solves C instances in 0.01 seconds, and A and B instances in 20 seconds each. The new distribution S thus emphasizes instance types A and B. There are three kinds of algorithms. X algorithms solve A instances in $0.1 \pm \varepsilon$ seconds and B instances in $100 \pm \varepsilon$ seconds each, where ε is a number between 0 and 0.01; the actual runtime varies randomly within this range across given algorithm–instance pairs. Y algorithms solve B instances in $0.1 \pm \varepsilon$ seconds and A instances in $100 \pm \varepsilon$ seconds each. Z algorithms solve both A and B instances in $25 \pm \varepsilon$ seconds each. All three algorithm types solve C instances in $10 \pm \varepsilon$ seconds each. The best average performance on S will be achieved by some Z algorithm, which we thus add to the portfolio. However, observe that this new Z algorithm is dominated by the current portfolio. Thus our new distribution S' will be the same as S. The process thus stagnates (endless algorithms of type Z exist), and we never add any X or Y algorithm to the portfolio, although adding any pair of these would lead to improved overall performance.

In this paper we introduce `Hydra`, a new method for automatically designing algorithms to complement a portfolio. This name is inspired by the Lernaean Hydra, a mythological, multi-headed beast that grew new heads for those cut off during its struggle with Greek hero Heracles. `Hydra`, given only a highly parameterized algorithm and a set of instance features, automatically generates a set of configurations that form an effective portfolio. It thus does not require any domain knowledge in the form of existing algorithms or algorithm components that are expected to work well, and can be applied to any problem. `Hydra` is an anytime algorithm: it begins by identifying a single configuration with the best overall performance, and then iteratively adds algorithms to the portfolio. It is also able to drop previously added algorithms when they are no longer helpful.

`Hydra` differs in key respects from stochastic offline programming (SOP), another recent method that performs instance-specific selection from an automatically generated set of search algorithms (Malitsky and Sellmann 2009). Firstly, SOP assumes that each of these algorithms has a particular structure, iteratively (i) sampling from a distribution over heuristics and (ii) using the sampled heuristic for one search step. In contrast, `Hydra` works with any parameterized algorithm. Secondly, SOP clusters the instances based on features and then builds one algorithm for each cluster, while `Hydra` considers all instances at each iteration and uses features solely for portfolio-based selection. Finally, SOP builds its set of algorithms using a custom optimization method, while `Hydra` can make use of any algorithm configuration procedure and portfolio building technique.

We also note that CPHYDRA (O'Mahony et al. 2008) has a name similar to our `Hydra`. CPHYDRA is a portfolio-based solver for constraint programming problems that uses case-based reasoning to determine a sequential schedule, which specifies how long to run each solver from a portfolio. It can be viewed as an approach for algorithm selection in a particular domain. In contrast, `Hydra` is a general method for combining portfolio-based algorithm selection with automated algorithm configuration.

`Hydra` offers the greatest potential benefit in domains where only one highly parameterized algorithm is competitive (e.g., certain distributions of mixed-integer programming problems), and the least potential benefit in domains where a wide variety of strong, uncorrelated solvers already exist. Nevertheless, we chose to evaluate `Hydra` on SAT—possibly the most extreme example of the latter category—effectively building a `SATzilla` of `SATensteins`. We did so for several reasons. Most of all, to demonstrate the usefulness of the approach, we considered it important to work on a problem for which the state of the art is known to be very strong. SLS-based SAT algorithms have been the subject of a large and sustained research effort over the past two decades, and the success of `SATzilla` demonstrates that existing SAT algorithms can be combined together to form very strong portfolios. The bar is thus set extremely high in this domain. Further, studying SLS for SAT also offered several pragmatic benefits: a wide variety of datasets exist and are agreed to be interesting; effective instance-based features are available; and `SATenstein` is a suitable configuration target. Finally, because SAT is an important problem, even small improvements are significant.

In our experiments, `Hydra` consistently achieved significant improvements over the best existing individual algorithms designed both by human experts and automatic configuration methods. More importantly, it always at least roughly matched—and indeed often exceeded—the performance of the best portfolio of such algorithms. We believe that `Hydra`'s performance on SAT is suggestive of its promise in other domains, in which there is no strong incumbent portfolio against which we could compare.

Hydra

The key idea behind Hydra is that a new candidate algorithm should be preferred exactly to the extent that it improves upon the performance of a (slightly idealized) portfolio. Hydra is thus implemented by changing the performance measure given to the algorithm configuration. A candidate algorithm is scored with its actual performance in cases where it is better than the existing portfolio, but with *the portfolio's performance* in cases where it is worse. Thus an algorithm is not penalized for bad performance on instances for which it should not be selected anyway, but is only rewarded to the extent that it outperforms the portfolio. The examples given earlier would be handled properly by this approach: the presence of intractable instances does not lead us to ignore performance gains elsewhere, and X and Y algorithms would be chosen in the first two iterations.

As shown in pseudocode, Hydra takes five inputs: a parameterized solver s, a set of training problem instances I, an algorithm configuration procedure AC with a performance metric m to be optimized, and a procedure PB for building portfolio-based algorithm selectors.

In its first iteration, Hydra uses configurator AC to produce a configuration of s, dubbed s_1, that is optimized on training set I according to performance metric m. Solver s_1 is then run on all instances of I in order to collect data that can eventually be input to PB; runs performed during the earlier configuration process can be cached and reused as appropriate. We define portfolio P_1 as the portfolio that always selects s_1, and solver set S_1 as $\{s_1\}$.

Then, in each subsequent iteration $k \geq 2$, Hydra defines a modified performance metric m_k as the better of the performance of the solver being assessed and the performance of the current portfolio, both measured according to performance metric m. The configurator AC is run to find a configuration s_k of s that optimizes m_k on the entire training set I. As before, the resulting solver is evaluated on the entire set I and then added to the solver set S. We then use PB to construct a new portfolio P_k from the given set of solvers. In each iteration of Hydra, the size of the candidate solver set S_k grows by one; however, PB may drop solvers that do not contribute to the performance of portfolio P_k (this is done, e.g., in SATzilla (Xu et al. 2008)). Slightly modifying the second example we gave earlier, if Z algorithms have slightly *better* performance on A and B instances than the current portfolio, some Z algorithm *will* be chosen in the first iteration. However, X and Y algorithms are chosen in the next two iterations, at which point the Z algorithm will be dropped, because it is dominated by the pair of X and Y algorithms.

Hydra can be terminated using various criteria, such as a user-specified bound on the number of iterations and/or a total computation-time budget.

The algorithm configuration procedure AC used within Hydra must be able to deal efficiently with configurations having equal performance on some or all instances, because such configurations can be expected to be encountered frequently. (For example, all configurations dominated by portfolio P_{k-1} will have equal performance under performance metric m_k.) It is also possible to exploit m_k for computational gain when optimizing runtime (as we do in our experimental study below). Specifically, a run of s on some instance $i \in I$ can be terminated during configuration once its runtime reaches portfolio P_{k-1}'s runtime on i. (See analogous discussion of capping in algorithm configuration by Hutter et al. (2009).)

```
**Procedure** Hydra($s$, $I$, $AC$, $m$, $PB$)
Input: Parametric solver $s$; Instance set $I$;
       Algorithm configurator $AC$;
       Performance metric $m$;
       Portfolio builder $PB$
Output: Portfolio $P$
$k := 1$; $m_1 = m$ ;
obtain a solver $s_1$ by running configurator $AC$ on parametric
solver $s$ and instance set $I$ with performance metric $m_1$;
measure performance of $s_1$ on all instances in $I$, using
performance metric $m$;
let $P_1$ by a portfolio that always selects $s_1$;
let $S_1 := \{s_1\}$;
while termination condition not satisfied do
    $k := k + 1$;
    define performance metric $m_k$ as the better of the
    performance of the solver being assessed and the
    performance of portfolio $P_{k-1}$, both measured using
    performance metric $m$;
    obtain a new solver $s_k$ by running configurator $AC$ on
    parametric solver $s$ and instance set $I$ with performance
    metric $m_k$;
    measure performance of $s_k$ on all instances in $I$, using
    performance metric $m$;
    $S_k = S_{k-1} \cup \{s_k\}$;
    obtain new portfolio $P_k$ by running portfolio builder $PB$
    on $S$;
return $P$
```

We note that Hydra need not be started from an empty set of algorithms, or only consider one parameterized algorithm. For example, it is straightforward to initialize S with existing state-of-the-art algorithms before running Hydra, or to optimize across multiple parameterized algorithms.

Experimental Setup

We chose inputs for Hydra to facilitate comparisons with past work, setting s, I, AC, and m as in KhudaBukhsh et al. (2009), and taking PB from Xu et al. (2008). Inputs s, I and m define the application context in which Hydra is run. In contrast, AC and PB are generic components; we chose these "off the shelf" and made no attempt to modify them to achieve domain-specific performance improvements. We do not expect that an end user would have to vary them either.

Parametric Solver: SATenstein-LS

As our parametric solver s, we chose SATenstein-LS, a generalized, highly parameterized stochastic local search (SLS) framework (KhudaBukhsh et al. 2009). It includes components taken from or inspired by a wide range of state-of-the-art SLS algorithms. SATenstein-LS has 41 parameters that control the selection and behavior of its components, leading to a total configuration space of size 4.82×10^{12}. Automatically-identified configurations of SATenstein-LS have been shown to achieve better performance than existing state-of-the-art SLS algorithms on several well-known instance distributions.

Instances

We investigated the effectiveness of Hydra on four distributions, drawing on well-known families of SAT instances. Because no state-of-the-art SLS algorithms are able to prove unsatisfiability, we considered only satisfiable instances. We identified these by running all complete algorithms that won a SAT competition category between 2002 and 2007 for one hour. First, the BM data set is constructed from 500 instances taken from each of the six distributions used by KhudaBukhsh et al. (2009) (QCP, SWGCP, FACT, CBMC, R3FIX, and HGEN), split evenly into training and test sets. Second, the INDU data set is a mixture of 500 instances from each of the CBMC and FACT distributions, again split evenly into training and test sets. Third and fourth, the HAND and RAND data sets include all satisfiable instances from the *Random* and *Handmade* categories of the SAT Competitions held between 2002 and 2007; we split the data 1141:571 and 342:171 into training and test sets, respectively.

Algorithm Configurator: FocusedILS

As our algorithm configurator AC, we chose the FocusedILS procedure from the ParamILS framework, version 2.3 (Hutter et al. 2009). This is the only existing method able to deal with extremely large configuration spaces such as SATenstein-LS's, and indeed was the method used to identify the high-performing SATenstein-LS configurations mentioned previously. FocusedILS compares a new configuration with an incumbent by running on instances one at a time, and rejects the new configuration as soon as it yields weakly worse overall performance on the set of instances than the incumbent. Because we expect many ties in Hydra's modified performance measures m_k, particularly in later iterations, we changed this mechanism so that new configurations are rejected only once they yield strictly worse overall performance. We also modified FocusedILS to cap all runs at the corresponding runtime for the portfolio P_{k-1}, as discussed previously.

Performance Metric: PAR

As our performance metric we would have liked to use mean runtime. However, the mean is not defined when long runs are capped during configuration, which is necessary in practice. We thus followed KhudaBukhsh et al. (2009), capping runs at 5 seconds and setting our performance metric m to be Penalized Average Runtime–10 (PAR-10); PAR-k of a set of r runs is the mean over the r runtimes, where capped runs are counted as having taken k times the captime. We performed 10 independent FocusedILS runs on training data with different instance orderings and with a one-day time bound.[1] We kept the parameter configuration that yielded the best PAR score on training data.

[1] SATenstein-LS's parameter space is divided into two disjoint parts; we performed half of our FocusedILS runs on each.

Portfolio Builder: SATzilla

As our portfolio builder PB we used the SATzilla framework (Xu et al. 2008). In brief, SATzilla works as follows. Let S denote a set of algorithms, let I denote a set of training instances, let D represent performance data for each algorithm from S run on each instance in I, and let F represent features computed for each instance in I. First, up to two solvers from S are selected as "presolvers" using on a local-search-based subset selection technique, with a fixed time budget taken from $\{0s, 2s, 5s, 10s\}$. We use forward selection to reduce the set of features, perform a quadratic basis function expansion, and then again perform forward selection; call the resulting feature set F'. For each solver in S, we then use a variant of ridge regression to predict D given F'; that is, we build a mapping from the features in F' to a real value predicting performance. The ridge regression variant deals correctly with censored data and exploits hierarchical hardness models, as described by Xu et al. (2008). At runtime, the portfolio first runs the presolver(s) sequentially. If the instance remains unsolved it then computes the features, evaluates the performance predictor for each algorithm, and runs the algorithm predicted to perform best.

Because performance prediction is imperfect, portfolio performance can improve if a solver is excluded from S. We use solver subset selection to determine the set S that achieves the best performance on the training data, building the portfolio as described above for each candidate set S.

We computed the same set of features as Xu et al. (2009). For BM and INDU, we only used 40 very efficently computable features (with average feature computation time of 0.04 seconds in both cases) since initial, exploratory experiments showed us that Hydra could achieve performance on the order of seconds on these data sets. For the same reason, we also reduced the time allowed for subset selection on these distributions by a factor of 10, allowing time budgets taken from $\{0s, 0.2s, 0.5s, 1s\}$. For RAND and HAND, we used all features except the most expensive ones (LP-based and clause-graph-based features); the average feature computation times were 4.2 seconds and 4.9 seconds, respectively.

Challengers

As previously explained, one reason that we studied SLS for SAT is that a wide variety of strong solvers exist for this domain. In particular, we identified 17 such algorithms, which we dub "challengers." Following KhudaBukhsh et al. (2009), we included all 7 SLS algorithms that won a medal in any of the SAT Competitions between 2002 and 2007, and also 5 additional prominent high-performance algorithms. We also included the 6 SATenstein-LS configurations introduced by KhudaBukhsh et al. (2009). While in some sense this set a high bar for Hydra (it had to compete against strong configurations of its own parametric solver) we included these configurations because they were shown to outperform the previous state of the art.

Experimental Environment

We collected training data and performed ParamILS runs on two different compute clusters. The first had 55 dual 3.2GHz Intel Xeon machines with 2MB cache and 2GB RAM, running OpenSuSE Linux 11.1; the second had 384 dual 3.0GHz Intel Xeon E5450 quad-core machines with 16GB of RAM running Red Hat Linux 4.1.2. Although the use of different machines added noise to the runtime observations in our training data, it had to be done to leverage additional computational resources. To ensure that our results were meaningful, we gathered all test data using only the first cluster; all results reported in this paper were collected using this data, and the data was used for no other purpose. Runtime was always measured as CPU time.

Results

To establish a baseline for our empirical evaluation, we first ran all 17 challenger algorithms on each of our test sets. The best-performing challengers are identified in the third column of Table 1, and their PAR-10 scores are shown in the fourth column. We also report the percentages of instances that each algorithm solved.

Next, we used `SATzilla` to automatically construct portfolios, first from the 11 manually crafted challenger algorithms, and then from the full set of 17 challengers that also included the 6 `SATenstein-LS` solvers. As can be seen from column 6 of Table 1, the latter portfolios perform much better than the best individual challenger, and the same holds for the former, more limited portfolios (column 5) as compared to the best of their 11 handcrafted component solvers. As one would expect, the performance gain was particularly marked for instance set `BM`, which is highly heterogeneous. In all cases, the inclusion of the 6 `SATenstein-LS` solvers, which were derived by automatic configuration on the six instance distributions considered by KhudaBukhsh et al. (2009), led to improved performance. While this was expected for `BM` and `INDU`, which are combinations of the instance distributions for which the 6 `SATenstein-LS` solvers were built, we were more surprised to observe the same qualitative effect for `RAND` and `HAND`.

In column 7, Table 1 shows the performance of the single `SATenstein-LS` configuration that was obtained in the initial phase of `Hydra`. Comparing these results to those the portfolio obtained after 7 iterations (column 8), we see that `Hydra` is indeed able to automatically configure solvers to work well as components of a portfolio. Furthermore, in all cases the `Hydra` portfolio outperformed the portfolio of 11 challengers. The `Hydra` portfolio outperformed the portfolio of 17 challengers in `RAND` and `HAND`, and effectively tied with it in `BM` and `INDU`. Note that these latter distributions are those for which `SATenstein-LS` solvers were specifically built; indeed, we found that the 17-challenger portfolios relied very heavily on these solvers. Furthermore, we note that KhudaBukhsh et al. (2009) devoted about 240 CPU days to the construction of the 6 `SATenstein-LS` solvers, while the construction of the entire `Hydra`[D,7] portfolio required only about 70 CPU days.

Overall, recall that the success of the challenger-based portfolios depends critically upon the availability of domain knowledge in the form of very strong solvers (some handcrafted, such as 11 of the challengers, and some constructed automatically based on clearly-delineated instance distributions, such as the 6 `SATenstein-LS` solvers). In contrast, `Hydra` always achieved equivalent or significantly better performance *without* relying on such domain knowledge.

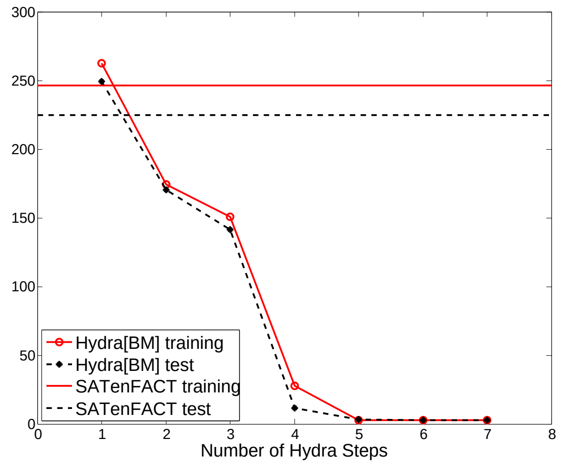
Figure 1 shows the PAR-10 performance improvements achieved in each `Hydra` iteration, considering both training and test data for `BM` and `INDU`. (The plots for `HAND` and `RAND` are not shown here, but closely resemble that for `BM`.). In all cases, test performance closely resembled training performance. `Hydra`'s test performance improved monotonically from one iteration to the next. Furthermore, on `BM`, `HAND` and `RAND`, `Hydra` achieved better performance than the best challenger after at most two iterations. On `INDU`, `Hydra` took five iterations to outperform the best challenger, `SATenstein-LS[CBMC]`. While this might seem surprising considering that the latter is a configuration of `SATenstein-LS`, it is explained by the fact that each `Hydra` iteration was allowed much less CPU time than KhudaBukhsh et al. (2009) allocated for the construction of `SATenstein-LS[CBMC]`.

Figure 2 compares the test-set performance of `Hydra`[D,1] and `Hydra`[D,7] for `BM` and `INDU`. (The plots for `HAND` and `RAND` are not shown here, but resemble the `BM` plot.) Note that `Hydra`[D,7] is substantially stronger than `Hydra`[D,1], particularly on hard instances. The fact that `Hydra`[D,1] sometimes outperforms `Hydra`[D,7] is due to the facts that the feature-based selection does not always choose the best solver from the given portfolio, and that the algorithms are randomized.

Table 2 shows, over each of the 7 iterations, the fraction of training instances solved by each `Hydra` portfolio component. Obviously, a total of k solvers are available in each stage k. Note that solver subset selection does lead `Hydra` to exclude solvers from the portfolio; this happens, e.g., on `RAND`, where the third solver was dropped in iteration 7. Another interesting effect can be observed in iteration 3 on `INDU`, where the second solver was effectively replaced by the third, whose instance share is marginally higher. Had we allowed the algorithm configurator to run longer in iteration 2, it would eventually have found this latter solver. The fact that it was found in the subsequent iteration illustrates `Hydra`'s ability to recover from insufficient allocation of runtime to the algorithm configurator. A similar phenomenon occurred in iterations 6 and 7 on `INDU`. The solver found in iteration 6 turned out not to be useful at all, and was therefore dropped immediately; in the next round of algorithm configuration a useful solver was found. (However, we see in Figure 1 that the overall benefit derived from using this latter solver turned out to be quite small.) Finally, we note that for all four distributions, the `Hydra`[D,7] portfolios consisted of at least 5 solvers, each of which were executed on between 6.8 and 41.8% of the instances. This indicates that the individual solvers constructed by `Hydra` indeed worked well on sizeable subsets of our instance sets, without the explicit use of problem-dependent knowledge (such as instance features) for partitioning these sets.

Dataset	Metric	Best Challenger	Chall. Perf	Portf. 11-Chall.	Portf. 17-Chall.	Hydra[D,1]	Hydra[D,7]
BM	PAR Score	SATenstein-LS[FACT]	224.53	54.04	3.06	249.44	3.06
	Solved (%)		96.4	99.3	100	96.0	100
INDU	PAR Score	SATenstein-LS[CBMC]	11.89	135.84	7.74	33.49	7.77
	Solved (%)		100	98.1	100	100	100
RAND	PAR Score	gNovelty$^+$	1128.63	897.37	813.72	1166.66	631.35
	Solved (%)		81.6	85.5	86.9	80.8	89.8
HAND	PAR Score	adaptG^{2}WSAT$_+$	2960.39	2670.22	2597.71	2915.22	2495.06
	Solved (%)		50.9	55.8	56.9	51.7	58.7

Table 1: *Performance comparison between Hydra, SATenstein-LS, challengers, and portfolios based on 11 (without 6 SATenstein-LS solvers) and 17 (with 6 SATenstein-LS solvers) challengers. All results are based on 3 runs per algorithm and instance; an algorithm solves an instance if its median runtime on that instance is below the given cutoff time.*



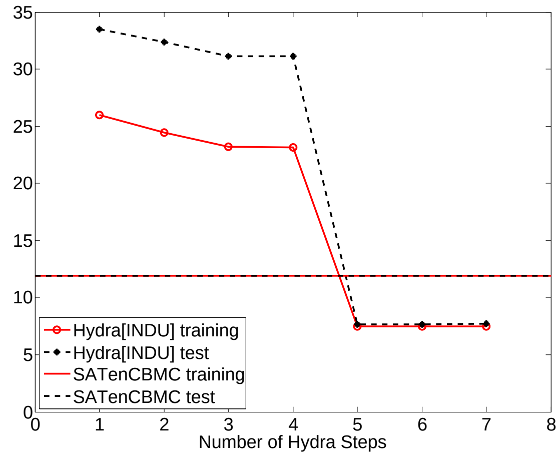


Figure 1: *Hydra's performance progress after each iteration, for BM (left) and INDU (right). Performance is shown in terms of PAR-10 score; the vertical lines represent the best challenger's performance for each data set.*

Conclusions

In this work we introduced Hydra, a new automatic algorithm design approach that combines portfolio-based algorithm selection with automatic algorithm configuration. We applied Hydra to SAT, a particularly well-studied and challenging problem domain, producing high-performance portfolios based only on a single highly parameterized SLS algorithm, SATenstein-LS. Our experimental results on widely-studied SAT instances showed that Hydra significantly outperformed 17 state-of-the-art SLS algorithms. Hydra reached, and in two of four cases exceeded, the performance of portfolios that used all 17 challengers as candidate solvers, 6 of which had been configured automatically using domain knowledge about specific types of SAT instances. At the same time, the total CPU time used by Hydra to reach this performance level for each distribution was less than a third of that used for configuring the 6 automatically-configured challengers.

One obvious direction for future work is to use Hydra to build portfolios of complete SAT solvers. We also intend to apply Hydra to mixed integer programming problems, for which there are very few strong solvers. Finally, we are interested in studying versions of Hydra that leverage different algorithm configuration and portfolio building methods.

References

Ansótegui, C.; Sellmann, M.; and Tierney, K. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proc. CP*, 142–157.

Balaprakash, P.; Birattari, M.; and Stützle, T. 2007. Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In *Hybrid Metaheuristics*, 108–122.

Carchrae, T., and Beck, J. C. 2005. Applying machine learning to low knowledge control of optimization algorithms. *Computational Intelligence* 21(4):373–387.

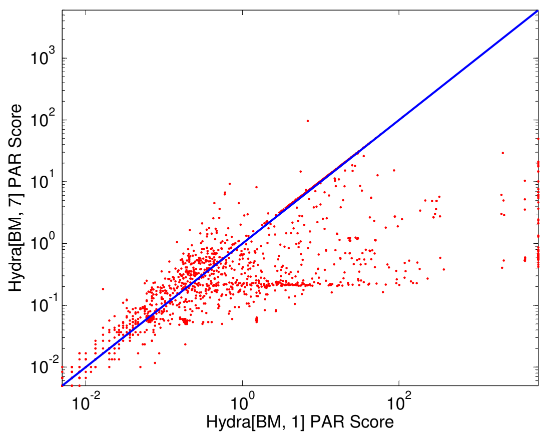
Fukunaga, A. S. 2002. Automated discovery of composite sat variable-selection heuristics. In *Proc. AAAI*, 641–648.

Gagliolo, M., and Schmidhuber, J. 2006. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence* 47(3-4):295–328.

Gaspero, L. D., and Schaerf, A. 2007. EasySyn++: A tool for automatic synthesis of stochastic local search algorithms. In *Proc. SLS*, 177–181.

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

Gratch, J., and Dejong, G. 1992. COMPOSER: A proba-



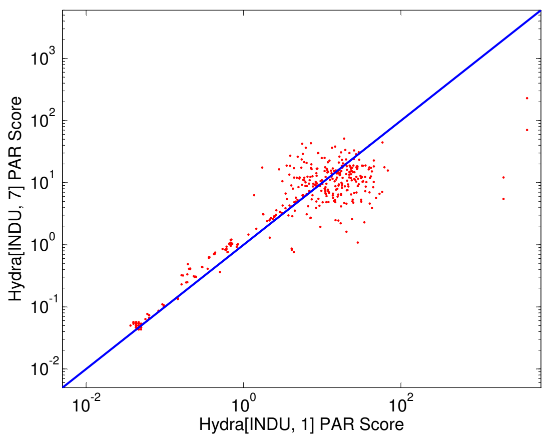


Figure 2: *Performance comparison between* `Hydra`*[D,7] and* `Hydra`*[D,1] on the test sets, for* `BM` *(left) and* `INDU` *(right). Performance is shown in terms of PAR-10 score.*

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
P_1	100	0	0	0	0	0	0
P_2	45.1	54.9	0	0	0	0	0
P_3	27.4	44.4	28.2	0	0	0	0
P_4	18.1	31.0	21.6	29.4	0	0	0
P_5	13.9	25.9	19.8	26.1	14.3	0	0
P_6	12.5	22.9	16.8	23.2	13.2	11.5	0
P_7	12.5	23.9	0	22.8	13.2	13.2	14.4

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
P_1	100	0	0	0	0	0	0
P_2	50.0	50.0	0	0	0	0	0
P_3	49.0	0	51.0	0	0	0	0
P_4	47.8	0	42.8	9.4	0	0	0
P_5	35.8	0	42.6	9.4	12.2	0	0
P_6	35.8	0	42.6	9.4	12.2	0	0
P_7	31.2	0	41.8	9.2	11.0	0	6.8

Table 2: *The percentage of instances for each solver chosen by algorithm selection at each iteration for* `RAND` *(left) and* `INDU` *(right).* P_k *and* s_k *are respectively the portfolio and algorithm instantiation obtained in iteration* k.

bilistic solution to the utility problem in speed-up learning. In *Proc. AAAI*, 235–240.

Guerri, A., and Milano, M. 2004. Learning techniques for automatic algorithm portfolio selection. In *Proc. ECAI*, 475–479.

Huberman, B.; Lukose, R.; and Hogg, T. 1997. An economics approach to hard computational problems. *Science* 265:51–54.

Hutter, F.; Babić, D.; Hoos, H. H.; and Hu, A. J. 2007. Boosting verification by automatic tuning of decision procedures. In *Proc. FMCAD*, 27–34.

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: An automatic algorithm configuration framework. *JAIR* 36:267–306.

KhudaBukhsh, A. R.; Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2009. SATenstein: Automatically building local search SAT solvers from components. In *Proc. IJCAI*, 517–524.

Leyton-Brown, K.; Nudelman, E.; Andrew, G.; McFadden, J.; and Shoham, Y. 2003a. Boosting as a metaphor for algorithm design. In *Proc. CP*, 899–903.

Leyton-Brown, K.; Nudelman, E.; Andrew, G.; McFadden, J.; and Shoham, Y. 2003b. A portfolio approach to algorithm selection. In *Proc. IJCAI*, 1542–1543.

Leyton-Brown, K.; Nudelman, E.; and Shoham, Y. 2009. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM* 56(4):1–52.

Malitsky, Y., and Sellmann, M. 2009. Stochastic offline programming. In *Proc. CP*, 784–791.

Minton, S. 1993. An analytic learning system for specializing heuristics. In *Proc. IJCAI*, 922–929.

Monette, J.; Deville, Y.; and van Hentenryck, P. 2009. Aeon: Synthesizing scheduling algorithms from high-level models. In *INFORMS Computing Society Conference*, 43–59.

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

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

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

Streeter, M.; Golovin, D.; and Smith, S. F. 2007. Combining multiple heuristics online. In *Proc. AAAI*, 1197–1203.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: portfolio-based algorithm selection for SAT. *JAIR* 32:565–606.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2009. SATzilla2009: An automatic algorithm portfolio for SAT. Solver description, 2009 SAT Competition.