INFORMS TutORials in Operations Research

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Meta-algorithms

Meinolf Sellmann

To cite this entry: Meinolf Sellmann. Meta-algorithms. *In* INFORMS TutORials in Operations Research. Published online: 26 Oct 2015; 47-55.
https://doi.org/10.1287/educ.2015.0138

TUTORIALS IN
OPERATIONS RESEARCH
INFORMS 2015

 | ISBN 978-0-9843378-8-0
http://dx.doi.org/10.1287/educ.2015.0138

Meta-algorithms

From Algorithm Tuning and Configuration to Algorithm Portfolios

Meinolf Sellmann
IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, meinolf@us.ibm.com

Abstract Efficiency and accuracy are of primary concern when developing analytics solutions in operations research. Typically, there is more than one algorithmic approach, and none dominates the others on all practically important problem instances. Moreover, algorithms usually have implicit or explicit parameters that often greatly affect performance. *Meta-algorithmics* is the subject that is concerned with the development of effective automatic tools that tune algorithm parameters and, at runtime, choose the approach that is best suited for the given input. In this tutorial, we summarize the core lessons learned when devising such meta-algorithmic tools.

Keywords meta-algorithms; algorithm portfolios; algorithm tuning; algorithm configuration; tools

1. Introduction

Operations research (OR) develops advanced predictive and prescriptive analytical methods. Practically all OR systems have parameters that are either implicit (i.e., constants in the code itself) or explicit (i.e., exposed to the user who can thus choose a particular instance from a whole family of algorithms). The particular choice of parameter settings often greatly affects system performance, such as prediction accuracy or the time to find and prove an optimal solution to a prescription problem. Further complicating the matter, rarely one parameter setting or any one algorithmic approach works best for all inputs. The characteristics of program parameters have deep consequences:

- Manually setting parameters requires the user to understand the underlying algorithm itself, thus necessitating some form of educational material such as a deep user's manual. Moreover, any parameter whose meaning the system developer finds too hard to communicate remains implicit and will at best be tuned on inputs that will always only imperfectly reflect the end user's input distribution.
- Even when users have a good understanding of the algorithmic approach, choosing good parameters for specific input distributions is usually very tedious and remains difficult to perform well by hand. Evidence shows that developers frequently do a poor job even when tuning their very own algorithms.
- Comparing parameterized algorithms in practice is difficult as all approaches need to be tuned equally well first. Even statistically significant differences between implementations may only reflect different levels in tuning rather than allowing an assessment of the true potential of the underlying algorithmic approaches.
- When developing OR systems, the choice of which algorithmic system components or which parameterizations thereof are used is often based on some few, nonrepresentative inputs and therefore frequently suboptimal. OR systems could potentially work much more efficiently if

parameterizations and even entire system components were chosen automatically after the complete system is implemented, potentially even at runtime.

Meta-algorithms have been developed to configure systems automatically. They tune algorithms based on representative input instances, either statically, where the same parameterization is always used at runtime, or dynamically, where the parameterization or even the algorithm itself is chosen depending on the given input. These tools enable highly effective automatic customization of analytics systems with respect to the true user distributions of inputs. They level the playing field when comparing algorithmic approaches in practice. Perhaps most importantly, they change the way we develop superior systems: the tools presented in this tutorial shift the focus away from robust approaches with default parameters that work reasonably well across the board to highly parameterized algorithmic components that may only work well in specific situations, but here, with stellar performance.

In this tutorial, we summarize some key lessons learned when configuring algorithms automatically. We do *not* attempt to give a comprehensive literature review of all relevant approaches, nor do we try to achieve any theoretical guarantees on the quality of automatic algorithm configuration. Instead, we will review the current state of the art and the insights that have led to the most practically effective algorithm configurators and algorithm portfolios to date.

2. Input-Oblivious Algorithm Configuration

Assume we have access to a set of training inputs that accurately represents the kind of inputs the algorithm we aim to tune (our *target algorithm*) will likely face at runtime. We can then try to find a parameterization that works well for the given training set and use this one parameterization at runtime, without considering the concrete input the algorithm has to process. The tuning is thus static; at runtime, we will use the same parameterization for all inputs. A typical scenario where such input-oblivious algorithm configuration is conducted is when a developer tries to find good *default parameters* for a target algorithm. We state this formally as follows.

Definition 1. Assume we are given a target algorithm family $A = \{A_c \mid c \text{ configuration}\}$ associated with a set of possible inputs X and a probability distribution $P\colon X \to [0,1]$ such that (X,P) forms a probability space. For each configuration c, we consider the random variable $\text{Perf}_c\colon X \to \mathcal{R}$, which measures the expected performance (such as time efficiency or prediction accuracy) of A_c when input $x \in X$ is chosen at random according to distribution P. *Input-oblivious algorithm configuration* consists in finding a configuration c_{opt} such that the expected performance is best, or

$$c_{\text{opt}} \leftarrow \arg\max_c E(\text{Perf}_c),$$

whereby the probability distribution P is not given explicitly but imperfectly in the form of a set of randomly drawn input samples.

So far, we did not specify what constitutes a configuration. In the following, we will assume a configuration is a vector in the Cartesian product of some intervals of rational numbers (*continuous parameters*), some intervals of integer numbers (*ordinal parameters*), and some general finite sets (*categorical parameters*). Even when there are no continuous parameters, algorithm families are frequently extremely large, typically exceeding 10^{50} different configurations. Moreover, given the infinite number of potential inputs in X, even assessing the expected performance for one given configuration is not straightforward.[1]

[1] Whereby in this tutorial we will not even consider stochastic algorithms, which adds a whole additional dimension to the problem.

2.1. Comparing Configurations

The first question that arises when comparing two or more configurations is when one is actually better than the other(s). The complication here is that we cannot evaluate the objective function directly, as the expected performance is determined by the unknown measure P. One option is to evaluate all configurations on all training inputs and to declare the one with the best average performance the winner. Obviously, this is very costly, especially when processing inputs takes a lot of time or when the training set is large. Many practical approaches therefore select a random subset of training inputs that changes during the course of the configuration process. One beautiful finding is that there is a more principled, cost-saving way.

Finding 1. There are effective methods to decide, based on a nonexhaustive number of comparisons and with confidence guarantees, that one parameterization works better than another.

Birattari et al. [4] suggest racing candidates against each other. *Racing* means that configurations are run on inputs one by one and then *ranked* based on the performances of each input. The core observation they make is this: Under the null hypothesis that n candidate configurations perform equally well, and thus all performance rankings on various inputs are equally likely, the rankings approximately follow a χ^2 distribution with $n-1$ degrees of freedom.

The shift of focus from absolute performance value to relative performance ranking thus opens the door for statistical significance testing. Birattari et al. [4] apply a Friedman test to see whether the null hypothesis can be rejected with some probability and, if so, perform a Students t-test to identify significantly underperforming configurations. These are then excluded from further consideration. That is, configurations that have been ruled out as inferior by a *principled statistical test* no longer need to be run on any additional inputs, which saves a lot of processing time.

2.2. Searching the Configuration Space

Highly effective search guidance on where to look for good configurations is key for efficient algorithm configuration. In practice, to assess the performance of a parameterization, we need to run the target algorithm on at least some training inputs. Since this takes a lot of processing time, we need to make progress toward good configurations very quickly to keep the overall processing costs within reason. Moreover, dual bounds on performance are not readily available; e.g., there is no relaxation that could tell us how well a target algorithm could perform maximally given a partial assignment to the parameters. As a result, state-of-the-art algorithm configurators perform a primal search in the parameter space with a strong focus on making rapid improvements.

Finding 2. The high cost of evaluating configurations can make elaborate search procedures worthwhile.

We need to relearn primal search for problems where one objective function evaluation costs seconds, if not minutes, of CPU time. In most of the local search literature, e.g., we find that being able to process more candidate solutions per second often yields better results than when trading some of these evaluation cycles in for a more elaborate computation of where to search next. However, when even sophisticated search guidance takes only a small percentage of just one candidate evaluation, the situation is fundamentally different.

Sequential model-based optimization for general algorithm configuration (SMAC) (Hutter et al. [8]) is a state-of-the-art input-oblivious algorithm configurator. It works by training a machine learning model (a so-called *surrogate model*) that predicts where high-quality parameterizations may be found. In particular, SMAC first evaluates some candidate

configurations by running the target algorithm. It then uses the observed performances to train a random forest (Breiman [5]) that predicts target performance given a set of parameters. This model is then used to find a good next candidate configuration, on which the target algorithm is evaluated. The result of this evaluation is then used to retrain the random forest, and so on, until the set configuration time is exhausted.

Note how much effort SMAC invests in the decision of which configuration to run next. In each iteration it (re)trains a machine learning model and then runs a nested local search on this model to select a new parameterization for the target algorithm. The latter is necessary because even unconstrained optimization over a random forest is NP-hard itself. In SMAC, the objective of the nested optimization is to find a configuration that, on one hand, has high potential based on the random forest's assessment and, on the other hand, will improve the accuracy of the machine learning model itself. That is, SMAC searches for a parameterization that scores high on predicted performance as well as uncertainty associated with that prediction.

The sequential approach followed by SMAC makes it hard to incorporate racing ideas or to exploit multicore parallelism easily. For these reasons, among others, genetic algorithms (Doerner et al. [6]), which usually have a hard time competing against sequential stochastic local search approaches when objective function evaluations are cheap, have been shown to work extremely well for automatic algorithm configuration.

Finding 3. Population-based methods offer potential for racing and parallelization.

GGA (Ansotegui et al. [1]) stands for gender-based genetic algorithm and is arguably the most successful algorithm configurator to date. For example, a MaxSAT solver tuned by GGA won nine different problem categories in the 2013 and 2014 Max-SAT evaluations. GGA evolves a population of candidate configurations, but since evaluating the performance on the target algorithm is so costly, the population size is kept very small—usually no more than 100 individuals. Moreover, the number of generations is kept low as well, usually between 50 and 100. The small population size means diversity is limited, and that problem is aggravated by the fact that the evolutionary process needs to make very aggressive progress to arrive at a good configuration after so few generations.

To deal with this issue, GGA splits the population into two genders: one is competitive, the other noncompetitive. New individuals are assigned to either one of those genders with equal probability. In each generation, the competitive individuals compete for the right of mating by "racing" against each other. Conceptually, the methods from Birattari et al. [4] could be used here, but the implementations of GGA actually just sample a small subset of training instances in each generation and evaluate the competitive individuals on all of them. When the performance measure to be optimized is target algorithm runtime, once the top competitive configurations have finished, all configuration evaluations can be stopped. This saves a lot of time as the best—i.e., fastest—configurations determine the time it takes to evaluate the objective.

The competitive winners are then each mated to multiple noncompetitive individuals, whereby the latter are chosen such that each noncompetitive individual creates the same number of offspring during its lifetime. An aging process is used to keep the population size stable.

Interestingly, this setup allows very aggressive optimization. In GGA, usually only one in eight competitive individuals wins the right of mating. At the same time, the noncompetitive individuals serve as a pool of diversity so that premature convergence in heavily suboptimal local optima is not observed despite the aggressive selection method.

Following the successful use of surrogate models within SMAC, the latest version of GGA now also uses a surrogate model that is specifically designed to forecast high-performance regions. This model is used during the mating: rather than conducting a randomized crossover to generate offspring, the next generation is engineered to maximize performance as forecasted by the surrogate model (Ansotegui et al. [2]).

3. Input-Specific Algorithm Configuration

Input-oblivious configuration works best when the training inputs are rather homogeneous and representative of the inputs the target algorithm will have to process at runtime. In general, however, no one parameterization will dominate all others on all relevant inputs. In this case, what is truly needed is a configurator that will dynamically prescribe a configuration *depending on the given input*. We define this formally below.

Definition 2. We are given a set of configurations C and a target algorithm family $A = \{A_c \mid c \in C\}$ associated with a set of possible inputs X and a probability distribution $P\colon X \to [0,1]$ such that (X,P) forms a probability space. For each mapping from inputs to configurations $\mu\colon X \to C$, we consider the random variable $\text{Perf}_\mu\colon X \to \mathcal{R}$, which measures the expected performance (such as time efficiency or prediction accuracy) of $A_{\mu(x)}$ when input $x \in X$ is chosen at random from the distribution P. *Input-specific algorithm configuration* consists in finding a mapping $\mu_{\text{opt}}\colon X \to C$ such that

$$\mu_{\text{opt}} = \arg\max_{\mu} E(\text{Perf}_\mu).$$

As before, the probability distribution P is not given explicitly; we only have some information on the input distribution by having been given a number of samples drawn from X according to P.

There is a fine line between "learning" algorithms and input-specific algorithm configuration. *Learning* algorithms are those that adjust their behavior dynamically during runtime, such as, e.g., tabu search algorithms that dynamically adjust the length of the tabu list (Battiti and Tecchiolli [3]). Input-specific algorithm configuration, on the other hand, only considers the initial configuration the target algorithm is started with—and which the algorithm may well adjust dynamically during runtime. The initial configuration is chosen based on a static analysis of the input itself. This is needed to enable the development of general-purpose tools for input-specific algorithm configuration, which must treat the target algorithm as a black box.

The first empirical question to consider is the trade-off between improvement potential and practical risk when choosing parameterizations based on the given input. Obviously, if configurations could be chosen perfectly, the input-specific choice would always be at least as good as the best input-oblivious default configuration. In practice, however, there is a risk that dynamically chosen parameterizations may not work as robustly and, in fact, lead to lower performance than the default, which works well for a broad spectrum of inputs. However, for the current generation of input-specific algorithm configurators, we have significant empirical evidence of the following.

Finding 4. Input-specific configuration offers substantial additional potential for improving algorithm efficiency while the risk of prescribing brittle configurations can be managed well in practice.

In this section we review how this is achieved. At first, it is tempting to try to train a statistical model that predicts algorithm performance based on features of the input as well as the configuration that is chosen. When starting the target algorithm at runtime, the model would then be consulted to select a new configuration that has good predicted performance. Some early works in this space took this route (see, e.g., Hutter and Hamadi [7]). This contrasts with the following finding.

Finding 5. Parameters need to work well together. Interpolating high-quality configurations rarely results in competitive parameterizations.

Consider what the statistical model in the above setting is required to accomplish: it shall, for the entire family of algorithms, predict performance based on some fixed set of feature values that are derived from the given inputs. In practice, any such model will be imperfect and prescribe formerly unseen and untested configurations based on some form of interpolation of parameterizations that worked well on other inputs. Such interpolation does, however, not preserve the intricate composition of parameter values that, more than anything else, need to work well *together*. Consequently, the currently most successful input-specific configurators do not prescribe configurations at runtime that have not proven their competitiveness during training. That is, state-of-the-art input-specific algorithm configurators only consider some few parameterizations at runtime. Formally, they produce mappings μ_{opt} that have a very small image.

To arrive at such a mapping, there are really two tasks that need to be carried out. The first regards the production of some configurations that together will form the image of μ_{opt}. We call these *image configurations*. The second is to devise and train a mechanism that chooses one of these configurations at runtime based on the given input. We devote the entire next section to the second task, which is also known as building *algorithm portfolios*, and we focus on the first task for the remainder of this section.

Presently, there are two main methods for producing image configurations, which are somewhat orthogonal to one another. Both build on the work on input-oblivious algorithm configuration.

The first is Hydra (Xu et al. [13]), which works sequentially. It begins by producing a configuration for all training inputs. Then it searches for a second configuration that improves performance on some training inputs. That is, a new configuration is searched that improves performance where possible while relying on the first found configuration to handle the ones the new configuration cannot improve. In each additional iteration, Hydra searches for a new configuration that will improve performance on some inputs while all inputs where no improvement is achieved are considered to incur the cost of the current portfolio of configurations. All these configurations are built using an instance-oblivious algorithm configurator. The only difference in subsequent iterations is the per-instance cost model used, which is defined by the performance of the current portfolio of configurations.

The second approach is instance-specific algorithm configuration (ISAC) (Kadioglu et al. [9]), which works by clustering inputs based on some *features* that are defined based on the problem the target algorithm tackles. ISAC uses an instance-oblivious algorithm configurator to produce one high-performance configuration for each cluster. Note that the features above are needed to select the configuration in the portfolio stage anyway, so the need for features characterizing inputs is not an additional requirement. However, clustering means that there are several technicalities that need to be taken care of, most of which we will only mention here without going into details.

First, clustering requires some form of distance metric. Importantly, this metric must not be disturbed by features operating on vastly different scales. Therefore, feature normalization is needed. Second, features that do not actually help distinguish between inputs that can be solved efficiently using the same configuration add noise to the metric and might need to be removed before clustering. Finally, clustering is essentially a two-objective optimization problem. Consequently, an additional criterion is needed to pick one representative from the Pareto frontier. Regarding the two objectives, on one hand, we try to have few clusters so that the configurations found are not too brittle; they must prove themselves on a larger sets of inputs. On the other hand, we want clusters to have low diameter so that instances within each cluster are very much alike, hopefully allowing one parameterization to efficiently handle all inputs in the same cluster. ISAC uses g-means clustering, which recursively splits clusters in two until all clusters pass a statistical test for normal distribution around each cluster center. To avoid generating brittle parameterizations that are overtuned for too few inputs, ISAC eventually merges clusters below a certain size with nearby clusters.

Note that both approaches, Hydra and ISAC, could be combined by starting out with a portfolio of configurations that were produced for each cluster according to ISAC, and then to continue adding configurations based on the Hydra methodology.

4. Algorithm Portfolios

The vexing problem of finding high-performance parameterizations in a super-large—if not infinite—family of algorithms now is reduced to selecting the best configuration from a small set of candidates. We devote a separate section to this final step in input-specific algorithm configuration because it is an important problem in its own right. When tackling hard predictive and prescriptive analytics problems, there is usually more than one algorithmic approach available to process a given input. In general, no one approach dominates all others on all inputs. Algorithm portfolios are built to harness the complementary strengths of different algorithms and/or algorithm configurations, which are potentially brittle and may be competitive only in very special niches, and to assemble them into one high-performance robust system that excels broadly.

In machine learning terms, we are facing a classification problem: label an input, as characterized by a set of features, with one algorithm in the portfolio. To learn, we can use the training inputs, compute their features, and label the respective feature vector with the algorithm that exhibits the best performance on the respective input. The latter is practically feasible because of the limited number of image configurations or algorithms in the portfolio. Based on this approach, we could use any classification algorithm, from k-nearest neighbor to decision trees and forests, to support vector machines. But we find the following.

Finding 6. Not all misclassifications are equally costly. Classification must be cost sensitive.

The problem of making good predictions actually becomes simpler when taking into account that the second- and third-best choices may often still work with very good performance. That is, taking actual costs of misclassifications into account can liberate us from having to correctly predict only the top performer. The focus on actual performances rather than considering only the best algorithm for each input thus led the community to regression models that would predict algorithm performances based on the features characterizing the input (Leyton-Brown et al. [11]).

Clearly, it would certainly be sufficient for algorithm selection if we were able to accurately forecast algorithm performance based on input features: we would simply compute the forecast for all algorithms in the portfolio and choose the best one. However, although sufficient, such a forecast is not necessary to select good algorithms in cost-sensitive ways. Experience shows that not taking the detour via predicting performance results in superior algorithm selection.

Finding 7. Forecasting performance is an unnecessary detour.

3S (Kadioglu et al. [10]) is an example of a cost-sensitive classification approach for algorithm selection that does not predict performance. Instead, it bases its choice on a cost analysis on the neighbors in the training set nearest to the input, as represented by feature vectors. As with the clustering approach in ISAC, k-nearest neighbor needs a lot of care to work well, especially when some features are not really indicative of algorithm performance.

Finding 8. Learning must be robust and of low bias.

Easier to use is the approach in Xu et al. [14], which suggests using a set of random forests, one for each pair of algorithms, that conducts cost-sensitive binary classification. When aggregating the binary choices, cost information is lost, however. One of the current best portfolio builders performs cost-sensitive multiclassification directly, based on an approach that is able to handle poor features gracefully. The approach is called cost-sensitive hierarchical

clustering (CSHC) and can be viewed as a technique for training random forests where each tree is built with the objective of enabling robust, cost-sensitive multiclassification (Malitsky et al. [12]). Namely, each branching decision, which bipartitions the training set at the parent node, is based on a score that reflects how well the training inputs that fall into the same partition can agree to be solved by the same algorithm. CSHC was used to build highly efficient algorithm portfolios that outperformed all other entrants at several SAT and Max-SAT competitions.

5. Conclusions

State-of-the-art portfolio builders work with amazing accuracy. When compared with a perfect oracle that picks the best algorithm for each input, many research papers report 92% or better test performance, which is remarkable. Both algorithm configurators and portfolio builders have also undergone extensive independent testing when the systems they support participated in numerous international competitions in various fields such as satisfiability, maximum satisfiability, quantified Boolean formulae, and constraint programming. Note that most of these winning entrants are based on older solvers. By winning these competitions, it was shown that, just by combining complementary algorithmic approaches and tuning algorithms, we can make up for one year—in some cases even two years—of core solver development.

Note that this performance is achieved by rigorous application of machine learning's best practices—in particular, a strict separation of training and test sets for evaluating the effectiveness of the meta-algorithmic techniques developed. Reproducibility is a core ingredient of the scientific method; the hypothesis formed based on experiments on one set of instances must *always* be tested on another. Meta-algorithmic research shows clearly that this also holds for the development and comparison of predictive and prescriptive algorithms—which raises concerns regarding the common use of benchmark libraries in the OR literature.

In summary, meta-algorithmics research has provided a set of tools for boosting analytics performance. They enable a fairer experimental comparison of algorithmic approaches. They liberate the developer from tedious tuning exercises. Most importantly, they hold the promise of making algorithmic ideas valuable even if they only work well in niche scenarios. By automatically choosing the right approach and configuration at runtime, a collection of highly specialized approaches is transformed into an overall robust system that works well across a wide range of inputs, with unprecedented performance. Meta-algorithmics thus drives a paradigm shift in the way we develop cutting-edge analytics software.

References

[1] C. Ansótegui, M. Sellmann, and K. Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. I. P. Gent, ed. *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, Vol. 5732, Springer, Berlin, 142–157, 2009.

[2] C. Ansótegui, Y. Malitsky, H. Samulowitz, M. Sellmann, and K. Tierney. Model-based genetic algorithms for algorithm configuration. Paper presented at the 24th International Joint on Artificial Intelligence, Buenos Aires, Argentina, July 30, 2015.

[3] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing* 6(2):126–140, 1994.

[4] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. W. B. Langdon, E. Cantú-Paz, K. E. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, et al. *Proceedings of the Genetic and Evolutionary Computation Conference*, Morgan Kaufmann, San Francisco, 11–18, 2002.

[5] L. Breiman. Random forests. *Journal of Machine Learning* 45(1):5–32, 2001.

[6] K. F. Doerner, M. Gendreau, P. Greistorfer, W. Gutjahr, R. F. Hartl, and M. Reimann, eds. *Metaheuristics: Progress in Complex Systems Optimization*. Springer, New York, 2007.

[7] F. Hutter and Y. Hamadi. Parameter adjustment based on performance prediction: Towards an instance-aware problem solver. Technical Report MSR-TR-2005-125, Microsoft Research, Cambridge, UK, 2005.

[8] F. Hutter, H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. C. A. C. Coello, ed. *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, Vol. 6683. Springer, Berlin, 507–523, 2011.

[9] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC—Instance-specific algorithm configuration. *Proceedings of the 19th European Conference on Artificial Intelligence, Lisbon, Portugal*, 751–756, 2010.

[10] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm selection and scheduling. J. Lee, ed. *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, Vol. 6876. Springer, Berlin, 454–469, 2011.

[11] K. Leyton-Brown, E. Nudelman, and Y. Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM* 56(4):Article 22, 2009.

[12] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm portfolios based on cost-sensitive hierarchical clustering. F. Rossi, ed. *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. AAAI Press, Palo Alto, CA, 608–614, 2013.

[13] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, AAAI Press, Palo Alto, CA, 210–216, 2010.

[14] L. Xu, F. Hutter, J. Shen, H. H. Hoos, and K. Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz, eds. *Proceedings of SAT Competition: Solver and Benchmark Descriptions*, University of Helsinki, Helsinki, Finland, 57–58, 2012.