

Chapter 2

Related Work

Automatic algorithm configuration is a quickly evolving field that aims to overcome the limitations and difficulties associated with manual parameter tuning. Many techniques have been attempted to address this problem, including meta-heuristics, evolutionary computation, local search, etc. Yet despite the variability in the approaches, this flood of proposed work mainly ranges between four ideas: algorithm construction, instance-oblivious tuning, instance-specific regression, and adaptive methods. The four sections of this chapter discuss the major works for each of these respective ideas and the final section summarizes the chapter.

2.1 Algorithm Construction

Algorithm construction focuses on automatically creating a solver from an assortment of building blocks. These approaches define the structure of the desired solver, declaring how the available algorithms and decisions need to be made. A machine learning technique then evaluates different configurations of the solver, trying to find the one that performs best on a collection of training instances.

The MULTI-TAC system [80] is an example of this approach applied to the constraint satisfaction problem (CSP). The backtracking solver is defined as a sequence of rules that determine which branching variable and value selection heuristics to use under what circumstances, as well as how to perform forward checking. Using a beam search to find the best set of rules, the system starts with an empty configuration. The rules or routines are then added one at a time. A small Lisp program corresponding to these rules is created and run on the training instances. The solver that properly completes the most instances proceeds to the next iteration. The strength of this approach is the ability to represent all existing solvers while automatically finding changes that can lead to improved performance. The algorithm, however, suffers from the search techniques used to

find the best configurations. Since the CSP solver is greedily built one rule or routine at a time, certain solutions can remain unobserved. Furthermore, as the number of possible routines and rules grows or the underlying schematic becomes more complicated, the number of possible configurations becomes too large for the described methodology.

Another approach from this category is the CLASS system, developed by Fukunaga [35]. This system is based on the observation that many of the existing local search (LS) algorithms used for SAT are seemingly composed of the same building blocks with only minor deviations. The Novelty solver [78], for example, is based on the earlier GWSAT solver [102], except instead of randomly selecting a variable in a broken clause, it chooses the one with the highest net gain. Minor changes like these have continuously improved LS solvers for over a decade. The CLASS system tries to automate this reconfiguration and fine tune the process by developing a concise language that can express any existing LS solver. A genetic algorithm then creates solvers that conform to this language. To avoid overly complex solvers, all cases having more than two nested conditionals are automatically collapsed by replacing the problematic sub-tree with a random function of depth 1. The resulting solvers were shown to be competitive with the best existing solvers. The one issue with this approach, however, is that developing such a grammar for other algorithms or problem types can be difficult, if not impossible.

As another example, in [86] Oltean proposed constructing a solver that uses a genetic algorithm (GA) automatically. In this case, the desired solver is modeled as a sequence of the selection, combination and mutation operations of a GA. For a given problem type and collection of training instances, the objective is to find the sequence of these operations that results in the solver requiring the fewest iterations to train. To find this optimal sequence of operations, Oltean proposes using a linear genetic program. The resulting algorithms were shown to outperform the standard implementations of genetic algorithms for a variety of tasks. However, while this approach can be applied to a variety of problem types, it ultimately suffers from requiring a long time to train. To evaluate an iteration of the potential solvers, each GA needs to be run 500 times on all the training instances to determine the best solver in the population accurately. This is fine for rapidly evaluated instances, but once each instance requires more than a couple of seconds to evaluate, the approach becomes too time-consuming.

Algorithm construction has also been applied to create a composite sorting algorithm used by a compiler [71]. The authors observed that there is no single sorting strategy that works perfectly on all possible input instances, with different strategies yielding improved performance on different instances. With this observation, a tree-based encoding was used for a solver that iteratively partitioned the elements of an instance until reaching a single element in the leaf node, and then sorted the elements as the leaves were merged. The primitives defined how the data is partitioned and under what conditions the sorting algorithm should change its approach. For example, the partitioning algorithm employed would depend on the amount of data that needs to be sorted. To make their method instance-specific, the authors use two features encoded as a six-bit string. For training, all instances are

split according to the encodings and each encoding is trained separately. To evaluate the instance, the encoding of the test instance is computed and the algorithm of the nearest and closest match is used for evaluation. This approach has been shown to be better than all existing algorithms at the time, providing a factor two speedup. The issue with the approach, however, is that it only uses two highly disaggregated features to identify the instance and that during training it tries to split the data into all possible settings. This becomes intractable as the number of features grows.

2.2 Instance-Oblivious Tuning

Given a collection of sample instances, instance-oblivious tuning attempts to find the parameters resulting in the best average performance of a solver on all the training data. There are three types of solver parameters. First, parameters can be categorical, controlling decisions such as what restart strategy to use or which branching heuristic to employ. Alternatively, parameters can be ordinal, controlling decisions about the size of the neighborhood for a local search or the size of the tabu list. Finally, parameters can be continuous, defining an algorithm's learning rate or the probability of making a random decision. Due to these differences, the tuning algorithms used to set the parameters can vary wildly. For example, the values of a categorical parameter have little relation to each other, making it impossible to use regression techniques. Similarly, continuous parameters have much larger domains than ordinal parameters. Here we discuss a few of the proposed methods for tuning parameters.

One example of instance-oblivious tuning focuses on setting continuous parameters. Coy et al. [29] suggested that by computing a good parameter set for a few instances, averaging all the parameters will result in parameters that would work well in the general case. Given a training set, this approach first selects a small diverse set of problem instances. The diversity of the set is determined by a few handpicked criteria specific to the problem type being solved. Then analyzing each of these problems separately, the algorithm tests all possible extreme settings of the parameters. After computing the performance at these points, a response surface is fitted, and greedy descent is used to find a locally optimal parameter set for the current problem instance. The parameter sets computed for each instance are finally averaged to return a single parameter set expected to work well on all instances. This technique was empirically shown to improve solvers for set covering and vehicle routing. The approach, however, suffers when more parameters need to be set or if these parameters are not continuous.

For a small set of possible parameter configurations, F-Race [20] employs a racing mechanism. During training, all potential algorithms are raced against each other, whereby a statistical test eliminates inferior algorithms before the remaining algorithms are run on the next training instance. But the problem with this is that F-Race prefers small parameter spaces, as larger ones would require a lot of testing in the primary runs. Careful attention must also be given to how and when certain

parameterizations are deemed pruneable, as this greedy selection is likely to end with a suboptimal configuration.

Alternatively, the CALIBRA system, proposed in [4], starts with a factorial design of the parameters. Once these initial parameter sets have been run and evaluated, an intensifying local search routine starts from a promising design, whereby the range of the parameters is limited according to the results of the initial factorial design experiments.

For derivative-free optimization of continuous variables, [11] introduced a mesh adaptive direct search (MADS) algorithm. In this approach, the parameter search space is partitioned into grids, and the corner points of each grid are evaluated for the best performance. The grids associated with the current lower bound are then further divided and the process is repeated until no improvement can be achieved. One of the additional interesting caveats to the proposed method was to use only short-running instances in the training set to speed up the tuning. It was observed that the parameters found for the easy instances tended to generalize to the harder ones, thus leading to significant improvements over classical configurations.

As another example, a highly parameterized solver like SATenstein [62] was developed, where all the choices guiding the stochastic local search SAT solver were left open as parameters. SATenstein can therefore be configured into any of the existing solvers as well as some completely new configurations. Among the methods used to tune such a solver is ParamILS.

In 2007, ParamILS [56] was first introduced as a generic parameter tuner, able to configure arbitrary algorithms with very large numbers of parameters. The approach conducts focused iterated local search, whereby starting with a random assignment of all the parameters, a local search with a one-exchange neighborhood is performed. The local search continues until a local optimum is encountered, at which point the search is repeated from a new starting point. To avoid randomly searching the configuration space, at each iteration the local search gathers statistics on which parameters are important for finding improved settings, and focuses on assigning them first. This blackbox parameter tuner has been shown to be successful with a variety of solvers, including Cplex [59], SATenstein [62], and SAPS [57], but suffers due to not being very robust, and depending on the parameters being discretized.

As an alternative to ParamILS, in 2009 the gender-based genetic algorithm [5] (GGA) was introduced. This black box tuner conducts a population-based local search to find the best parameter configuration. This approach presented a novel technique of introducing competitive and non-competitive genders to balance exploitation and exploration of the parameter space. Therefore, at each generation, half of the population competes on a collection of training instances. The subset of parameter settings that yield the best overall performance are then mated with the non-competitive population, with the children removing the worst-performing individuals from the competitive population. This approach was shown to be remarkably successful in tuning existing solvers, often outperforming ParamILS.

Recently, a sequential model-based algorithm configuration (SMAC) [55] was introduced, in 2010. This approach proposes generating a model over the solver's

parameters to predict the likely performance. This model can be anything from a random forest to marginal predictors and is used to identify aspects of the parameter space, such as what parameters are the most important. Possible configurations are then generated according to this model and compete against the current incumbent. The best configuration continues onto the next iteration. While this approach has been shown to work on some problems, it ultimately depends on the accuracy of the model used to capture the interrelations of the parameters.

2.3 Instance-Specific Regression

One of the main drawbacks of instance-oblivious tuning is ignoring the specific instances, striving instead for the best average case performance. However, works like [82, 112], and many others have observed that not all instances yield to the same approaches. This observation supports the “no free lunch” theorem [122], which states that no single algorithm can be expected to perform optimally over all instances. Instead, in order to gain improvements in performance for one set of instances, it will have to sacrifice performance on another set. The typical instance-specific tuning algorithm computes a set of features for the training instances and uses regression to fit a model that will determine the solver’s strategy.

Algorithm portfolios are a prominent example of this methodology. Given a new instance, the approach forecasts the runtime of each solver and runs the one with the best predicted performance. SATzilla [126] is an example of this approach as applied to SAT. In this case the algorithm uses ridge regression to forecast the log of the run times. Interestingly, for the instances that timeout during training, the authors suggest using the predicted times as the observed truth, a technique they show to be surprisingly effective. In addition, SATzilla uses feedforward selection over the features it uses to classify a SAT instance. It was found that certain features are more effective at predicting the runtimes of randomly generated instances as opposed to industrial instances, and vice-versa. Overall, since its initial introduction in 2007, SATzilla has won medals at the 2007 and 2009 SAT Competitions [1].

In algorithm selection the solver does not necessarily have to stick to the same algorithm once it is chosen. For example, [40] proposed running in parallel (or interleaved on a single processor) multiple stochastic solvers that tackle the same problem. These “algorithm portfolios” were shown to work much more robustly than any of the individual stochastic solvers. This insight has since led to the technique of randomization with restarts, which is commonly used in all state-of-the-art complete SAT solvers. Algorithm selection can also be done dynamically. As was shown in [36], instead of choosing the single best solver from a portfolio, all the solvers are run in parallel. However, rather than allotting equal time to everything, each solver is biased, depending on how quickly the algorithm thinks it will complete. Therefore, a larger time share is given to the algorithm that is assumed to be the first to finish. The advantage of this technique is that it is less susceptible to an early error in the performance prediction.

In [88], a self-tuning approach is presented that chooses parameters based on the input instance for the local search SAT solver WalkSAT. This approach computes an estimate of the invariant ratio of a provided SAT instance, and uses this value to set the noise of the WalkSAT solver, or how frequently a random decision is made. This was shown to be effective on four DIMACS benchmarks, but failed for those problems where the invariant ratio did not relate to the optimal noise parameter.

In another approach, [53, 54] tackle solvers with continuous and ordinal (but not categorical) parameters. Here, Bayesian linear regression is used to learn a mapping from features and parameters into a prediction of runtime. Based on this mapping for given instance features, a parameter set that minimizes predicted runtime is searched for. The approach in [53] led to a twofold speedup for the local search SAT solver SAPS [57].

An alternative example expands on the ideas introduced in SATzilla by presenting Hydra [124]. Instead of using a set of existing solvers, this approach uses a single highly parameterized solver. Given a collection of training instances, a set of different configurations is produced to act as the algorithm portfolio. Instances that are not performing well under the current portfolio are then identified and used as the training set for a new parameter configuration that is to be added to the portfolio. Alternatively, if a configuration is found not to be useful any longer, it is removed from the portfolio. A key ingredient to making this type of system work is the provided performance metric, which uses a candidate's actual performance when it is best and the overall portfolio's performance otherwise. This way, a candidate configuration is not penalized for aggressively tuning for a small subset of instances. Instead, it is rewarded for finding the best configurations and thus improving overall performance.

An alternative to regression-based approaches for instance-specific tuning, CPHydra [87] attempts to schedule solvers to maximize the probability of solving an instance within the allotted time. Given a set of training instances and a set of available solvers, CPHydra collects information on the performance of every solver on every instance. When a new instance needs to be solved, its features are computed and the k nearest neighbors are selected from the training set. The problem then is set as a constraint program that tries to find the sequence and duration in which to invoke the solvers so as to yield the highest probability of solving the instance. The effectiveness of the approach was demonstrated when CPHydra won the CSP Solver Competition in 2008, but also showed the difficulties of the approach since the dynamic scheduling program only used three solvers and a neighborhood of 10 instances.

Most recently, a new version of SATzilla was entered into the 2012 SAT Competition [127]. Foregoing the original regression-based methodology, this solver trained a tree classifier for predicting the preferred choice for each pair of solvers in the portfolio. Therefore when a new instance had to be addressed, the solver that was chosen most frequently was the one that got evaluated. In practice this worked very well, with the new version of SATzilla winning gold in each of the three categories. Yet this approach is also restricted to a very small portfolio of

solvers, as each addition to the portfolio requires exponentially many new classifiers to be trained.

2.4 Adaptive Methods

All of the works presented so far were trained offline before being applied to a set of test instances. Alternative approaches exist that try to adapt to the problem they are solving in an online fashion. In this scenario, as a solver attempts to solve the given instance, it learns information about the underlying structure of the problem space, trying to exploit this information in order to boost performance.

Algorithm selection is closely related to the algorithm configuration scenario that is tuning one categorical variable. For example, in [73] a sampling technique selects one of several different branching variable selection heuristics in a branch-and-bound approach. A similar approach was later presented in [64], but instead of choosing a single heuristic, this approach tries to learn the best branching heuristic to use at each node of a complete tree search.

An example of this technique is STAGE [21], an adaptive local search solver. While searching for a local optima, STAGE learned an evaluation function to predict the performance of a local search algorithm. At each restart, the solver would predict which local search algorithm was likely to find an improving solution. This evaluation function was therefore used to bias the trajectory of the future search. The technique was empirically shown to improve the performance of local search solvers on a variety of large optimization problems.

Impact-based search strategies for constraint programming (CP) [96] are another example of a successful adaptive approach. In this work, the algorithm would keep track of the domain reduction of each variable after the assignment of a variable. Assuming that we want to reduce the domains of the variables quickly and thus shrink the search space, this information about the impact of each variable guides the variable selection heuristic. The empirical results were so successful that this technique is now standard for Ilog CP Solver, and used by many other prominent solvers, like MiniSAT [33].

In 1994, an adaptive technique was proposed for tabu search [13]. By observing the average size of the encountered cycles, and how often the search returned to a previous state, this algorithm dynamically modified the size of its tabu list.

Another interesting result for transferring learned information between restarts was presented in Disco–Novo–GoGo [101]. In this case, the proposed algorithm uses a value-ordering heuristic while performing a complete tree search with restarts. Before a restart takes place, the algorithm observes the last tried assignment and changes the value ordering heuristic to prefer the currently assigned value. In this way, the search is more likely to explore a new and more promising portion of the search space after the restart. When applied to constraint programming and satisfiability problems, orders of magnitude performance gains were observed.

2.5 Chapter Summary

In this chapter, related work for automatic algorithm configuration was discussed. The first approach of automatic algorithm construction focused on how solving strategies and heuristics can be automatically combined to result in a functional solver by defining the solver's structure. Alternatively, given that a solver is created where all the controlling parameters are left open to the user, the instance-oblivious methodology finds the parameter settings that result in the best average-case performance. When a solver needs to be created to perform differently depending on the problem instance, instance-specific regression is often employed to find an association between the features of the instance and the desired parameter settings. Finally, to avoid extensive offline training on a set of representative instances, adaptive methods that adapt to the problem dynamically are also heavily researched.

All these techniques have been shown empirically to provide significant improvements in the quality of the tuned solver. Each approach, however, also has a few general drawbacks. Algorithm construction depends heavily on the development of an accurate model of the desired solver; however, for many cases, a single model that can encompass all possibilities is not available. Instance-oblivious tuning assumes that all problem instances can be solved optimally by the same algorithm, an assumption that has been frequently shown impossible in practice. Instance-specific regression, on the other hand, depends on accurately fitting a model from the features to a parameter, which is intractable and requires a lot of training data when the features and parameters have non-linear interactions. Later developments with trees alleviate the issues presented by regression, but existing approaches don't tend to scale well with the size of the portfolio. Adaptive methods require a high overhead since they need to spend time exploring and learning about the problem instance while attempting to solve it. The remainder of this book focuses on how instance-oblivious tuning can be extended to create a modular and configurable framework that is instance-specific.

<http://www.springer.com/978-3-319-11229-9>

Instance-Specific Algorithm Configuration

Malitsky, Y.

2014, IX, 134 p. 13 illus., 11 illus. in color., Hardcover

ISBN: 978-3-319-11229-9