Cautious Specialization of Strategy Schedules (Extended Abstract)

Filip Bártek[1,2], Karel Chvalovský[1] and Martin Suda[1]

[1]Czech Technical University in Prague – Czech Institute of Informatics, Robotics and Cybernetics. Jugoslávských partyzánů 1580/3, 160 00 Prague 6 – Dejvice, Czech Republic

[2]Czech Technical University in Prague – Faculty of Electrical Engineering. Technická 2, 166 27 Prague 6 – Dejvice, Czech Republic

Abstract

Combining theorem proving strategies into schedules is a well-known recipe for improving prover performance, as a well-chosen set of complementary strategies in practice covers many more problems within a predetermined time budget than a single strategy could. A strategy schedule can be a monolithic sequence, but may also consist of multiple schedule branches, intended to be selected based on certain problem's features and thus to *specialize* in solving the corresponding problem classes. When allowing schedule branching, there is an intuitive trade-off between covering the known (training) problems quickly, by splitting them into many classes, and the *generalization* of the obtained branching schedule, i.e., its ability to solve previously unseen (test) problems.

In this paper, we attempt to shed more light on this trade-off. Starting off with a large set of proving strategies of the automatic theorem prover Vampire evaluated on the first-order part of the TPTP library, we report on two preliminary experiments with building branching schedules while keeping track of how well they generalize to test problems. We hope to attract more attention to the exciting topic of schedule construction by this work-in-progress report.

Keywords

strategy schedules, schedule specialization, regularization, Vampire

1. Introduction

Proof search in most automatic theorem provers (ATPs) relies on parameterized heuristics. Strong configurations (*strategies*) of an ATP are, typically, specialized to certain classes of input problems – no configuration performs well on all possible problems of interest. Selecting a strong strategy for a given input problem is, in general, hard. This is namely due to the chaotic behavior of the provers [1]: Using a fixed strategy, a small change in the input problem may lead to a substantially different proof search.

Typically, if a strategy solves a problem, the solution is reached in a relatively short time [2]. In contexts where sufficient time or parallelism is available, running a portfolio of diverse strategies is a pragmatic approach to increase the power of a prover. Such portfolio, or *schedule* in case a time limit is assigned to each of the strategies, may be either hand-crafted [3] or constructed automatically [4, 5].

Further gain in performance can be achieved by *specializing* the schedule: We partition the possible input problems into classes and construct a specialized schedule for each class. We prefer the classes to be homogeneous in terms of the problems' response to strategies, so that a schedule specialized for a class needs less time to cover the same percentage of problems. Increasing the granularity of the classes increases the specialization and the associated potential performance gain.

Decision trees [6] provide a suitable architecture to capture such gradual specialization. A set of features, such as the number of atoms, the presence of certain syntactic traits, or the membership in standard logical fragments, are computed for the input problem. A binary tree is then traversed from the root to a leaf. Each internal node is labeled with a criterion that partitions the domain of one of the features into two parts. When a node is traversed, the next node is selected depending on the corresponding feature of the problem. The schedule inhabiting the final leaf is then used to attempt to solve the problem.

PAAR'24: 9th Workshop on Practical Aspects of Automated Reasoning, July 2, 2024, Nancy, France

filip.bartek@cvut.cz (F. Bártek); karel.chvalovsky@cvut.cz (K. Chvalovský); martin.suda@cvut.cz (M. Suda)

0000-0002-1822-2651 (F. Bártek); 0000-0002-0541-3889 (K. Chvalovský); 0000-0003-0989-5800 (M. Suda)

The ATP Vampire [7] won the prestigious First-Order Form theorems (FOF) division of the CADE ATP System Competition (CASC) [8] every year from 2002 to 2023 [9]. Arguably, this continued success is partially thanks to the famous "CASC mode" of the prover and the corresponding use of strong strategy schedules. The schedules were constructed by Andrei Voronkov with the help of a system called Spider [10] and arranged in a shallow decision tree for specialization.[1]

When the specialization problem classes are chosen in a way that optimizes the performance on a training set of problems, we run the risk of *overfitting*, that is increasing performance on the training problems at the cost of decreasing performance on unseen problems. *Regularization* is the process of reducing the expressive power of a machine-learned model to prevent overfitting.

In our previous work [4], we studied the regularization of monolithic schedule construction, i.e., the case where the algorithm's output is a single sequence with no specialization to problem features. We discovered and evaluated more than a thousand Vampire strategies targeting the first-order fragment of the TPTP library [11], which are now available as an independent data set for experimentation [12].

In this work, we use the same data set to start shedding some light on regularization during schedule specialization. Namely, after brief preliminaries (Sect. 2), we first discuss (and measure) how a single branching criterion (i.e., a single problem feature) can affect the generalization of the simplest branching schedule with just one level of branching (Sect. 3). We complement this view by an experiment with gradient boosted trees (Sect. 4), which explores the opposite extreme case where every problem can — based on its features — be, in principle, assigned its unique schedule. Our paper also briefly overviews related work (Sect. 5) and offers several concluding remarks (Sect. 6).

We are far from any definite answer on how specialization in schedule construction should be best done. Yet we hope to spark more interest in this exciting topic by this work-in-progress report.

2. Preliminaries

2.1. Training and evaluation data

In our previous work [4], we generated a collection of 1096 strong and mutually complementary strategies for the ATP Vampire. We evaluated the performance of each of these strategies on a set of 7866 first-order logic (FOL) problems from TPTP [11] v8.2.0. The results of these evaluations constitute a dataset suitable for experimenting with schedule construction and specialization [12].

We denote the set of strategies as S ($|S| = 1096$), the set of problems as P ($|P| = 7866$), and the *evaluation matrix* as $E_p^s : S \times P \to \mathbb{N} \cup \{\infty\}$. The value E_p^s is the time at which strategy s solves problem p, or ∞ in case strategy s failed to solve problem p within the time limit.

The time is measured in whole CPU *megainstructions (Mi)*, where $1\,\mathrm{Mi} = 2^{20}$ instructions reported by the tool `perf` and $2000\,\mathrm{Mi}$ take approximately 1 second of wallclock time on our hardware.

2.2. Vampire's CASC-mode problem features

For our schedule specialization experiments, we use problem features already available in Vampire and used there by the CASC mode. This feature set consists mainly of various syntactic counters (the number of: goal clauses, axiom clauses, Horn clauses, equational clauses, etc.), most prominent of which is the number of atoms (roughly corresponding to problem size); problem class according to TPTP classification (NEQ, HEQ, …, EPR, UEQ); and a number of bitfields checking for properties such as "has $X = Y$", "has functional definitions", or "has inequality resolvable with deletion".

These features are not included in the mentioned dataset [12], but can be obtained easily by running a modified version of Vampire[2] in "profile mode" via `--mode profile <problem_name>`.

[1] E.g., https://github.com/vprover/vampire/blob/c7564c1d65020771079f29787ca2d5d7743f5d6a/CASC/Schedules.cpp#L5676.
[2] https://github.com/vprover/vampire/tree/ourProfile

2.3. Strategy schedules

A strategy *schedule* is a mapping $t_s : S \to \mathbb{N}$. The value t_s is the time limit assigned to strategy s. A schedule may be used to attack an arbitrary problem with Vampire: The strategies are run sequentially with the assigned time limits, each attempting to solve the problem, until a solution is found or every strategy has depleted its allocated time. In the rest of this work, instead of running the prover, we use the evaluation matrix to simulate the evaluation of schedules, restricting our scope to problems in P.

Similarly to strategies, schedules are typically evaluated in a time-constrained setting. Schedule t_s satisfies (time) *budget* $T \in \mathbb{N}$ if and only if $\sum_{s \in S} t_s \leq T$.

In our previous work [4], we proposed a greedy algorithm that constructs a strategy schedule from an evaluation matrix and a budget. The algorithm allows finding a relatively strong (with respect to the evaluation matrix) schedule satisfying the budget quickly.

A *strategy schedule recommender* produces a schedule for an arbitrary input problem. To evaluate a schedule recommender on a set of problems, we estimate, using the evaluation matrix, the proportion of problems solved by the respective schedules produced by the recommender.

2.4. Overfitting and regularization

Extreme schedule specialization is discouraged in CASC [13]:

> All techniques used must be general purpose, and expected to extend usefully to new unseen problems. [...] If machine learning procedures are used to tune a system, the learning must ensure that sufficient generalization is obtained so that no there is no specialization to individual problems.

In machine learning (ML), it is common for a trained system to perform better on the data it was trained on than on previously unseen data. The system is said to *overfit* to the training data. In most applications, we are ultimately interested in optimizing the performance on unseen data, so overfitting is undesirable. A system that does not overfit is said to *generalize* (to unseen data).

To evaluate generalization of our ML training procedures, we repeatedly randomly split the set of problems P into a training set ($80\,\%$) and a test set ($20\,\%$), train the system on the training set, and evaluate the trained system on the test set. The final score is the mean of the per-split test scores (in our case, schedule recommender success rates). This way, we estimate the performance on unseen data.

Moreover, before training we exclude strategies that have their "witness problem" in the test set. Roughly speaking, this prunes the set of strategies only to those that could have been discovered if a given test set was fixed beforehand for the strategy discovery phase. See [4, Section 6] for more details.

3. Single branching point

Creating a schedule can be seen as an act of "covering" the training problems with the help of the known solutions of the previously evaluated strategies. It is clear that if many strategies need to be involved with long running times in order to eventually cover every training problem, the running time of the whole schedule will need to be high.[3] Schedule specialization comes to rescue to reduce this running time, as it allows the covering process to focus, within each branch, on a different subset of the training problems. In a nutshell, on each branch, there is less covering work to do and during evaluation, only the "right" branch is always used.

However, from the perspective of generalization, i.e., the effort of ensuring that our specializing schedule performs well also on problems unseen in training, splitting the work into branches may be detrimental. Indeed, if many training problems end up in their respective subset for mostly a random

[3] Here we conceptually depart from the position where there is a fixed budget and we are trying to solve as many problems as possible within that budget. This is because we aim, in this section, to present an analysis which answers a question for all budgets at once. We can do it thanks to the anytime nature of the mentioned greedy algorithm we employ here [4].

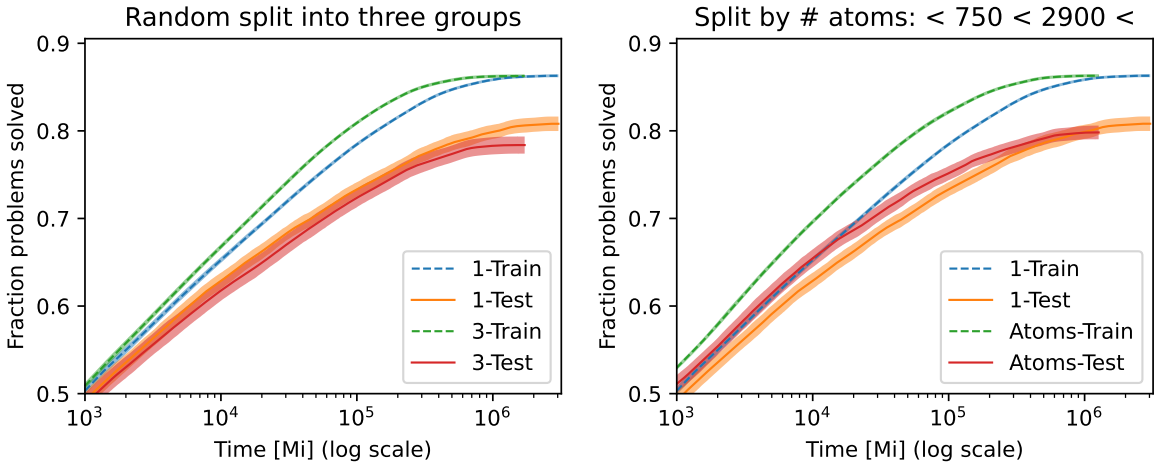

Figure 1: Monolithic ("1") vs single branching schedule performance. Problems split into three random subsets on the left ("3") and into small-medium-large subsets based on the number of atoms ("Atoms") on the right. Showing averages over 100 random $80 : 20$ train/test split runs. Shading marks the zone of one standard deviation around the averages (and got four times larger for the Test runs compared to the Train runs just from the effect of normalizing the averages from the unequal train/test set sizes to the reported fractions).

reason, the chances of a similar testing problem (that could be solved by the same strategy) being classified into the same subset and thus subjected to the same schedule branch are low.

In this experiment, we set out to demonstrate this phenomenon at its extreme. Our method of covering is the greedy algorithm presented in previous work [4] and we run it repetitively over random $80 : 20$ train/test splits of our problem set P. We compare the average performance of a monolithic (non-branching) schedule, a schedule branching into three groups based on the number of problem's atoms, and a schedule branching into three random equally-sized groups. For the branching on atoms, we split the problems into *small* (number of atoms < 750), *large* (number of atoms > 2900) and *medium* (the rest). The cutoff points were chosen *not* to split the problems into equally-sized subsets, but by aiming for a split for which each subset can be fully covered within approximately the same budget.

The results are presented in cactus plot form in Fig. 1. The results of the monolithic schedule building (1-Train/1-Test) are the same in both the left and the right plot (and roughly correspond to Fig. 7 in Appendix E of our previous work [4]). The monolithic schedule requires approximately 2 million Mi to cover all the training problems and the test performance converges to around $93\,\%$ of the training one. The train performance of both the random 3-split (3-Train) and the atom split (Atom-Train) are at all times higher than that of the monolithic schedule until they (necessarily) converge to the same value and stabilize. (The atom split tends to finish its covering job a bit earlier than the random split.) However, the pictures for the test performance of the branching schedules differ. While splitting on atoms improves over monolithic even in test performance (up until around the 1 million Mi mark), and for the lower budgets even improves over the monolithic's train performance, the test performance of the random 3-split (3-Test) is at all times worse.

It is perhaps not surprising that random features are useless from the perspective of branching schedule generalization, however, we find the "backwards version" of this perspective to be quite revealing, namely the realization that even reasonable-sounding problem features may differ in how useful (useless, or even detrimental) they could be in this regard. Schedule construction is sometimes organized by *first* fixing the problem groups based on features and only *then* looking for strategies in each group separately (one can save computational resources by not having to evaluate each strategy on every training problem; evaluating just within the relevant group would be enough). However, if such initial split is only based on educated guessing of what problems reasonably belong together, opportunities for better (or even the best possible) generalization can be missed.

Ideally, we would like to come up with generalization-conducive splits of the problems beforehand,

by just studying our evaluation matrix and asking a question such as "What could be a nice, easy-to-delineate subset of problems for which certain strategies work well together and complement each other, while elsewhere they are mostly useless?" We remark that such splits might be defined by other properties than those expressible by our current fixed feature set. Discovering such new properties that would still be easy to compute (and thus easy to understand) would be an interesting instance of the explainable AI endeavour. At this moment, however, we do not have a good way of doing this and leave this research direction for future work.

4. Gradient boosting

Unlike in the previous section, where a hand-crafted split based on the number of atoms occurring in the problem is discussed, we want to produce finer splits of the problems based on Vampire's CASC-mode problem features here. In other words, we want to learn more complicated decision trees, or as in our case ensembles of small decision trees. Hence we want to train a model that for a given problem p produces a schedule $t_1, \ldots, t_{|S|}$, where $\sum_{s \in S} t_s \leq T$, such that there is a strategy $s \in S$ that solves $p \in P$ in time at most t_s.

Although we want to obtain schedules, we may characterize them by probability distributions. A discrete probability mass function $\mathbf{p}\colon S \to [0, 1]$ defines a schedule by setting $t_s = T \cdot \mathbf{p}_s$. Hence we can use standard approaches for multi-class classification, where usually we get a probability distribution over possible classes and select the class as the one with the highest estimated probability. Here, however, we take the estimated probability distribution and compute a schedule based on it.

This of course immediately leads to the question of what is the true probability distribution we want to estimate. One possible approach is to assign, for example, probability 1 to the best strategy for the problem. However, we likely need a more fine-grained approach as we are interested in generalization. First, we are happy to split the budget among more strategies, as long as at least one of them solves the problem. Second, we do not require that the problem be solved by the best available strategy.

Hence instead of having, for each (training) problem, a fixed true distribution that we want to learn, we may produce a "true" distribution iteratively during the training based on our previous estimates and change it as necessary. For example, we have a training problem p and an estimated probability $\mathbf{p}_1, \ldots, \mathbf{p}_{|S|}$ for it. We may want to produce a "true" distribution for p by assigning $\mathbf{p}_i = 0$ if the strategy i does not solve the problem in the given budget T, and hence increase $\mathbf{p}_j$ if the strategy j solves the problem. In other words, we want to split the budget only among strategies that can possibly solve the problem. Moreover, we can boost the strategy that is closest to solving the problem; the strategy requiring the smallest increase in the estimated budget to solve the problem. And there are many other ways how to get a new "true" distribution. It is worth noticing that we may obtain this "true" distribution in different ways: one way for those training examples that our model predicts correctly (estimated schedule solves the problem) and another for those that are predicted incorrectly.

4.1. Model

Gradient boosting for decision trees, see, e.g., [14], fits nicely into this picture. Loosely speaking, it combines weak models that iteratively improve their predictions by correcting the previous mistakes. In particular, we learn decision trees that split our problems into different subsets based on the static Vampire's features. Or, more precisely, in each iteration, we learn $|S|$ regression trees (multi-class problem); one per strategy. When we want to estimate the probability distribution after i iterations, we combine the output of all i regression trees learned for each strategy, and get the probability distribution by the softmax function. In the next iteration, we fix the "true" distribution based on the newly estimated distribution and iterate this process until we get a satisfactory model or reach the limit on the number of iterations.

Table 1
We compare the success rate of the split into small-medium-large subsets based on the number of atoms ("Atoms", see Sect. 3) and the schedule produced by gradient boosting.

Budget	Atoms		Gradient boosting	
	Train	Test	Train	Test
1000	0.5297	0.5113	0.6196	0.5660
10000	0.6965	0.6547	0.7650	0.6998
100000	0.8219	0.7519	0.8358	0.7807

4.2. Initial experiments

We use LightGBM [15], a framework for gradient boosting decision trees, with custom objective functions. We get probability estimates from our models by the softmax function and use the cross-entropy between predicted probabilities and "true" probabilities as the loss function.

We report some preliminary results, as we have not performed much of hyper-parameter tuning.[4] We use the learning rate 0.05, the maximal number of bins that feature values are bucketed in is 8, the number of boosting iterations is 500, and, most importantly, we restrict the number of leaves to 2; hence we use decision stumps as weak learners.

Our custom objective functions create "true" probabilities $\mathbf{p}'_1, \ldots, \mathbf{p}'_{|S|}$ from the estimated probabilities $\mathbf{p}_1, \ldots, \mathbf{p}_{|S|}$ for a problem p. We have tried various combinations of the following procedures:

(a) $\mathbf{p}'_i = 0$ if the strategy i does not solve p in the time budget T, otherwise $\mathbf{p}'_j = \mathbf{p}_j$,

(b) $\mathbf{p}'_i = 1$ if the strategy i is the best available strategy[5] and $\mathbf{p}'_j = 0$ for $i \neq j$,

(c) boost[6] the best available strategy,

and then L1-normalize $\mathbf{p}'_1, \ldots, \mathbf{p}'_{|S|}$. It is even possible to combine different ways how to create "true" probabilities based on whether the predicted schedule solves the training problem or not. Surprisingly, (a) works quite well even though it can get stuck in a local minima. Hence, for simplicity, all the subsequent results use this objective function. It takes roughly $10\,\mathrm{s}$, using one core, to train one iteration of the model with our non-optimized custom objective function written in Python.

We train our models using 10 random 80:20 train/test splits over the dataset described in Section 2.1. In Table 1, we report the success rate. The best iteration is selected based on the number of solved problems on the training set. For shorter budgets, the gradient boosting model significantly outperforms the simple split into small-medium-large subsets based on the number of atoms in the problem. As expected, the gap is narrowing as the budget is increasing.

Interestingly, when we inspect the produced schedules, the majority of problems (both on the train and test sets) are solved by multiple strategies in the schedule. Moreover, it is unlikely that a problem is solved by the overall best available strategy for the problem, supporting our idea behind the careful creation of "true" probabilities.

As the "true" probability that we aim to estimate evolves, it is possible that the majority of trees learned do not offset incorrect estimates from previous iterations but rather compensate for the changing "true" probability. Note that we often add the same tree (with different leaf values) multiple times.

These initial experiments show that it is possible to automatically train a model that both leverages the available feature space and keeps a reasonable level of generalization.

[4] The performance of LightGBM is sensitive to choices of hyper-parameters. Moreover, our custom objective functions make these choices even more significant.

[5] There are many possible definitions here. For example, we have experimented with selecting the strategy that has the maximal difference between the time assigned to the strategy and the time necessary to solve the problem by the strategy.

[6] We tried two variants. First, in combination with (a), we split "saved time" not among all strategies that solve the problem, but we only boost the best available strategy. Second, we increase the allocation to the best available strategy just enough for it to solve the problem.

5. Related work

Strategy schedule specialization is closely related to *strategy selection* (also referred to as "heuristic selection"), a special case of algorithm selection [16, 17]. Automatic mode in the E prover [18] uses hand-crafted problem classes to select a strategy for the input problem [19]. Strategy selection for E has also been approached by machine learning [20]. Given an input problem, MaLeS [19] predicts solving time for all strategies in a portfolio and selects the strategy that minimizes the predicted time. Similarly, Grackle [21] selects a strategy using a trained LightGBM model that predicts a ranking of strategies in a pre-computed portfolio.

Strategy schedules have been constructed manually in Gandalf [3] and automatically in E-SETHEO [22], Spider for Vampire [10], CPhydra [23], BliStrTune [24], HOS-ML [5], and in our previous work [4]. Given an input problem, HOS-ML [5] selects a strategy schedule automatically. The problem's static features are used to identify a problem class, for which a schedule was optimized during the training. CPhydra [23] identifies 10 training problems that are the most similar to the input problem and optimizes a schedule for these problems using a pre-computed evaluation matrix.

6. Conclusion

Working with a dataset of 1096 Vampire strategies, which were evaluated on the first-order part of TPTP, and a selection of syntactic problem features, which Vampire computes efficiently for its input problem at startup, we presented two experiments with strategy schedule construction and schedule specialization. In the first, we noticed that the selection of the splitting features not only influences how fast a specialized schedule can cover the training problems, but, more interestingly, how well it can generalize to previously unseen problems. In the second, we demonstrated that a good performance can be achieved by using gradient boosted decision trees. The second approach is interesting in that it departs from the concept of problem covering and instead develops a schedule in the form of a probability distribution over the available strategies. This requires the use of a custom target distribution which develops during the learning process.

There are many avenues for future research. Our next target for experimentation is the grow-and-prune technique for decision tree construction and regularization [6] and its adaptation to cautious schedule specialization. However, we would also be more than happy for others to join us in experimenting with our strategy data [12] to see which method could eventually lead to the most useful approach.

Acknowledgments

The work on this paper was supported by the Czech Science Foundation grant 24-12759S and the project RICAIP no. 857306 under the EU-H2020 programme.

References

[1] M. Suda, Vampire getting noisy: Will random bits help conquer chaos? (system description), in: J. Blanchette, L. Kovács, D. Pattinson (Eds.), Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings, volume 13385 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 659–667. URL: https://doi.org/10.1007/978-3-031-10769-6_38. doi:10.1007/978-3-031-10769-6_38.

[2] G. Reger, Boldly going where no prover has gone before, in: M. Suda, S. Winkler (Eds.), Proceedings of the Second International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, ARCADE@CADE 2019, Natal, Brazil, August 26, 2019, volume 311 of *EPTCS*, 2019, pp. 37–41. URL: https://doi.org/10.4204/EPTCS.311.6. doi:10.4204/EPTCS.311.6.

[3] T. Tammet, Towards efficient subsumption, in: C. Kirchner, H. Kirchner (Eds.), Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings, volume 1421 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 427–441. URL: https://doi.org/10.1007/BFb0054276. doi:10.1007/BFb0054276.

[4] F. Bártek, K. Chvalovský, M. Suda, Regularization in Spider-style strategy discovery and schedule construction, 2024. URL: https://arxiv.org/abs/2403.12869. doi:10.48550/arXiv.2403.12869. arXiv:2403.12869.

[5] E. K. Holden, K. Korovin, Heterogeneous heuristic optimisation and scheduling for first-order theorem proving, in: F. Kamareddine, C. S. Coen (Eds.), Intelligent Computer Mathematics - 14th International Conference, CICM 2021, Timisoara, Romania, July 26-31, 2021, Proceedings, volume 12833 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 107–123. URL: https://doi.org/10.1007/978-3-030-81097-9_8. doi:10.1007/978-3-030-81097-9_8.

[6] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, Classification and Regression Trees, Wadsworth, 1984.

[7] L. Kovács, A. Voronkov, First-order theorem proving and Vampire, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, volume 8044 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 1–35. URL: https://doi.org/10.1007/978-3-642-39799-8_1. doi:10.1007/978-3-642-39799-8_1.

[8] G. Sutcliffe, The CADE ATP System Competition - CASC, AI Magazine 37 (2016) 99–101.

[9] G. Sutcliffe, The CADE ATP system competition, 2024. URL: https://tptp.org/CASC/, [Online; accessed 26-April-2024].

[10] A. Voronkov, Spider: Learning in the sea of options, 2023. URL: https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833, unpublished. Paper accepted at Vampire23: The 7th Vampire Workshop.

[11] G. Sutcliffe, The TPTP problem library and associated infrastructure, Journal of Automated Reasoning 59 (2017). doi:10.1007/s10817-017-9407-7.

[12] F. Bártek, M. Suda, Vampire strategy performance measurements, 2024. URL: https://doi.org/10.5281/zenodo.10814478. doi:10.5281/zenodo.10814478.

[13] G. Sutcliffe, CASC design and organization, 2024. URL: https://tptp.org/CASC/J12/Design.html, [Online; accessed 15-April-2024].

[14] J. H. Friedman, Greedy function approximation: A gradient boosting machine., The Annals of Statistics 29 (2001) 1189–1232. URL: https://doi.org/10.1214/aos/1013203451. doi:10.1214/aos/1013203451.

[15] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, T.-Y. Liu, LightGBM: A highly efficient gradient boosting decision tree, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, volume 30, Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[16] J. R. Rice, The algorithm selection problem, Adv. Comput. 15 (1976) 65–118. URL: https://doi.org/10.1016/S0065-2458(08)60520-3. doi:10.1016/S0065-2458(08)60520-3.

[17] P. Kerschke, H. H. Hoos, F. Neumann, H. Trautmann, Automated algorithm selection: Survey and perspectives, Evol. Comput. 27 (2019) 3–45. URL: https://doi.org/10.1162/evco_a_00242. doi:10.1162/EVCO_A_00242.

[18] S. Schulz, S. Cruanes, P. Vukmirovic, Faster, higher, stronger: E 2.3, in: P. Fontaine (Ed.), Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings, volume 11716 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 495–507. URL: https://doi.org/10.1007/978-3-030-29436-6_29. doi:10.1007/978-3-030-29436-6_29.

[19] D. Kühlwein, J. Urban, MaLeS: A framework for automatic tuning of automated theorem provers, J. Autom. Reason. 55 (2015) 91–116. URL: https://doi.org/10.1007/s10817-015-9329-1. doi:10.1007/s10817-015-9329-1.

[20] J. P. Bridge, S. B. Holden, L. C. Paulson, Machine learning for first-order theorem proving - learning to select a good heuristic, J. Autom. Reason. 53 (2014) 141–172. URL: https://doi.org/10.1007/s10817-014-9301-5. doi:10.1007/S10817-014-9301-5.

[21] J. Hůla, J. Jakubův, M. Janota, L. Kubej, Targeted configuration of an SMT solver, in: K. Buzzard, T. Kutsia (Eds.), Intelligent Computer Mathematics - 15th International Conference, CICM 2022, Tbilisi, Georgia, September 19-23, 2022, Proceedings, volume 13467 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 256–271. URL: https://doi.org/10.1007/978-3-031-16681-5_18. doi:10.1007/978-3-031-16681-5_18.

[22] S. Schulz, E-SETHEO, http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/e-setheo.html, 2024. [Online; accessed 26-April-2024].

[23] D. Bridge, E. O'Mahony, B. O'Sullivan, Case-based reasoning for autonomous constraint solving, in: Y. Hamadi, É. Monfroy, F. Saubion (Eds.), Autonomous Search, Springer, 2012, pp. 73–95. URL: https://doi.org/10.1007/978-3-642-21434-9_4. doi:10.1007/978-3-642-21434-9_4.

[24] J. Jakubův, J. Urban, BliStrTune: hierarchical invention of theorem proving strategies, in: Y. Bertot, V. Vafeiadis (Eds.), Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017, ACM, 2017, pp. 43–52. URL: https://doi.org/10.1145/3018610.3018619. doi:10.1145/3018610.3018619.