EPiC Series in Computing

Volume 100, 2024, Pages 360–369

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Prover9 Unleashed: Automated Configuration for Enhanced Proof Discovery*

Kristina Aleksandrova, Jan Jakubův, and Cezary Kaliszyk

University of Innsbruck, Innsbruck, Austria
{jakubuv,CezaryKaliszyk}@gmail.com

Abstract

While many of the state-of-art Automated Theorem Provers (ATP) like E and Vampire, were subject to extensive tuning of strategy schedules in the last decade, the classical ATP prover Prover9 has never been optimized in this direction. Both E and Vampire provide the user with an automatic mode to select good proof search strategies based on the properties of the input problem, while Prover9 provides by default only a relatively weak auto mode. Interestingly, Prover9 provides more varied means for proof control than its competitors. These means, however, must be manually investigated and that is possible only by experienced Prover9 users with a good understanding of how Prover9 works.

In this paper, we investigate the possibilities of automatic configuration of Prover9 for user-specified benchmark problems. We employ the automated strategy invention system Grackle to generate Prover9 strategies with both basic and advanced proof search options which require sophisticated strategy space features for Grackle. We test the strategy invention on AIM train/test problem collection and we show that Prover9 can outperform both E and Vampire on these problems. To test the generality of our approach we train and evaluate strategies also on TPTP problems, showing that Prover9 can achieve reasonable complementarity with other ATPs.

1 Introduction

For many automated reasoning problems a combination of complementary strategies is significantly better than better than a single strategy. For this reason, many provers support configurable options that can be user specified or automatically tuned. This has been done for many provers [18, 4] and led to their good performance on various benchmarks [16]. Prover9, despite its popularity among mathematicians [9] is mostly configured manually.

In this paper, we discuss the specification of the Prover9 options using the strategy invention system Grackle [4]. Apart from all the basic option [1] we include the various Prover9 specific advanced options that require adaptations to the system. We also specify multi-staged domains. Starting with a preliminary set of of basic strategies, the system derives a large number of new

*Supported by the Czech MEYS under the ERC CZ project no. LL1902 *POSTMAN* and ERC PoC grant no. 101156734 *FormalWeb3*. We are grateful to Bob Veroff for valuable comments and discussions.

strategies for the AIM dataset and for subparts of TPTP and show that Prover9 can perform significantly better than the other provers on some of these datasets.

A considerable amount of effort has been dedicated to parameter tuning in state-of-the-art theorem provers (mainly unpublished, unfortunately), aiming to discover a universal proof search strategy or a portfolio of strategies. However, Grackle's primary objective differs slightly. Instead of seeking a generic strategy or portfolio that excels across all benchmarks or competition problems, Grackle endeavors to develop a set of strategies capable of solving as many problems as possible from a benchmark provided by the user. This approach proves beneficial in scenarios where users encounter problems distinct from the competition problems typically optimized for by state-of-the-art provers.

2 Prover9 and Its Proof Search Options

Prover9 is an automated theorem prover for first-order logic with equality that implements the resolution/paramodulation calculi. It extends earlier work by McCune on Otter [13] in several ways, including stricter rules on symbols, invidivual quantifiers for each variable, as well as outputs, conclusions and weighting functions, and most notably a human-designed automatic mode that we aim to improve upon in the current paper. Prover9 relies on specific term orderings, including Knuth-Bendix Ordering (KBO), Lexicographic Path Ordering (LPO), and Recursive Path Ordering (RPO), to guide its theorem proving process. The main algorithm, often called the *given clause loop*, maintains two sets of clauses: the *set of support* with clauses waiting to be processed, and the set *usable* with already processed clauses. At each step, it selects a *given clause* from the set of support, applies all enabled inference rules combining the given clause with all other usable clauses, and marks the given clause as usable. When a new *generated* clause is found redundant, it is immediately discarded, while other clauses are *retained/kept* in the usable list.

Prover9 offers numerous options that can be modified as part of the strategy included with each problem in the LADR encodings (Prover9 does not use TPTP [16], so for TPTP problems that we evaluate on, a LADR translation is performed in advance). The most important parameters are the search limits, including parameters which impose a limit on the size of the sos list (`sos_limit`), stops the search after n given clauses have been used (`max_given`), stops when more than n clauses have been retained (`max_kept`), stops when about n megabytes of memory have been used (`max_megs`). Each named parameter can assume values ranging from -1 to the maximum integer, where -1 indicates an absence of limits. During each iteration of the primary loop, Prover9 dynamically selects a specified clause from the SOS list, relocates it to the usable list, and deduces inferences from it and other clauses within the usable list. The process of selecting the given clause involves six distinct components which refers to the clause with the lowest ID (the oldest clause) (`age_part`), pertains to the (oldest) clause with the lowest weight (`weight_part`), corresponds to the (oldest) lightest false clause (`false_part`), relates to the (oldest) lightest true clause (`true_part`), represents a (pseudo-) random clause (`random_part`), denotes the lightest clause that aligns with a given hint (`hints_part`). The distinction between false and true clauses is established through a set of interpretations. By default, negative clauses are considered false, while non-negative clauses are deemed true. As before, all named parameters have a permissible range from -1 to the maximum integer. Prover9 provides multiple methods for term comparison, and the effectiveness of these methods can significantly influence the outcome of problem-solving. While the default settings are suitable for many scenarios, tackling challenging problems often necessitates fine-tuning the term ordering. The key decision, made through the parameter named order, involves selecting the type of ordering: LPO, RPO,

or KBO. In addition, the option (`eq_defs`) causes changes to the term ordering. Prover9's weighting function maps clauses to integers, and it is used primarily for two purposes: selecting the given clause, and discarding inferred clauses (with the parameter max weight). There are also many inference rules including binary resolution rules and options, hyper and Unified Requirements (UR) resolution rules, paramodulation rules, and so on that we will optimize.

3 Grackle: System for Targeted Portfolio Invention

Grackle[1] [4] is designed to automate the creation of a portfolio of solver strategies tailored to user-provided benchmark problems, aiming to maximize the problem-solving effectiveness. By inputting a set of benchmark problems, Grackle autonomously invents a diverse set of solver strategies to tackle as many of these problems as possible. It currently integrates various solvers, such as ATP solvers E [15], Vampire [10], Lash [3], and SMT solvers Bitwuzla [14] and cvc5 [2]. Adding support for additional solvers is straightforward: users just need to specify solver strategy parameters and implement a basic wrapper for solver execution. This paper introduces an extension of Grackle to accommodate the ATP solver Prover9 [12] (see Section 4), and assesses its performance on various first-order benchmarks (Section 5).

Grackle, building upon its predecessors BliStr [18], BliStrTune [8, 7] and EmpireTune [6], inherits the core strategy invention algorithm from BliStr. While its predecessors are hardwired for specific solvers (E, Vampire), Grackle expands the algorithm's applicability to an arbitrary solver. The BliStr/Grackle approach involves a dual phase process: a *strategy evaluation* followed by a *strategy invention* phase. In the evaluation phase, all provided strategies are assessed on a comprehensive set of benchmark problems, typically with a higher resource limit T. This evaluation categorizes problems based on individual strategy performance, yielding sets of problems P_S where each strategy S performs best. Subsequently, the best-performing strategy S undergoes specialization on its corresponding problem set P_S during the strategy invention phase. This entails launching an external parameter tuning software, such as ParamILS [5] or SMAC3 [11], on problems P_S, using strategy S as the starting point. Furthermore, a reduced resource limit t compared to the evaluation phase (T) guides the tuning process towards enhancing performance on P_S. Previous studies [18, 7, 6, 4] confirm the core notion that enhancing performance on P_S leads to improvements on other unsolved problems. The invented strategy is then integrated into the portfolio, initiating a fresh evaluation phase. For a comprehensive understanding of Grackle, readers are directed to its detailed exposition [4].

In order to describe the solver strategy space, Grackle employs the same approach as ParamILS. The strategy space is described by a set of available parameters, their potential values, and the default value for each parameter. Thus, a single strategy is represented by a set of parameter/value pairs. The responsibility of the solver wrapper lies in translating the strategy representation into the actual solver input. Typically, the parameters directly correspond with solver command line options, though advanced transformations are feasible. We describe several embeddings of advanced Prover9 options within a Grackle strategy space in Section 4. Furthermore, we extend Grackle with *staged strategy invention*, where a vast strategy space is subdivided into multiple smaller spaces and tuned separately. This approach resembles hierarchical tuning in BliStrTune and EmpireTune.

Grackle/ParamILS also allows the specification of *parameter conditions* to describe parameters that are effective only when a *master* parameter has a specific value. For instance, the parameter `ur_nucleus_limit` specifies the maximum number of literals for UR-resolution. Set-

[1] https://github.com/ai4reason/grackle

ting the value for this parameter will only have effect when UR-resolution is activated by the master parameter `ur_resolution`. This allows a more effective strategy space search.

4 Grackle Strategy Space for Prover9

Unlike the other solvers supported by Grackle thus far, Prover9 does not accept the proof search strategy as command-line options. Instead, the proof search options are provided in a text file passed to the solver. This enables greater flexibility, and indeed, Prover9 supports more sophisticated constructs to guide the proof search than many other ATP provers. Some of these are described below and addition details can be found in the Prover9 manual.[2]

Default space. Prover9 supports more than 100 basic options, which are either boolean, integer-valued, or categorical. Boolean option flags are written in a Prolog-like syntax, for example, "`set(back_subsume).`" or "`clear(factor).`" to set the corresponding flag to true or false, respectively. The integer-valued options are written like "`assign(max_weight, 100).`" to assign the selected value. The same syntax is used for categorical options, for example, "`assign(order, kbo).`". We select 61 parameters, out of which 27 are boolean, and construct a basic Prover9 parameter space denoted as *default* below. For integer-valued parameters, we manually sample the available domain depending on whether the parameter sets a weight, limit, or otherwise. The space covers approximately 10^{47} different strategies, and further details can be investigated in the source code.[3]

Given clause selection options control how the given clause is selected from the set of support. The default space already contains parameters for clause selection (`age_part`, ...), but they can be overridden by a construct of the following form.

```
list(given_selection).
   part(queue1, high, age   ,  weight < 500 & -horn  ) = 50.
   part(queue2, high, weight,  depth < 15            ) = 50.
   part(queue3, low , weight,  -false                ) = 5.
   part(rest  , low , random,  all                   ) = 1.
end_of_list.
```

This defines four queues for clauses that alternate given clause selection according to the ratios specified after "`=`". Each `part` creates one queue where clauses that satisfy the condition in the fourth argument are sorted by the criterion given by the third argument (`age`, `weight`, or `random`). The first argument is a text identifier, and the second is the queue priority (`high` or `low`). The low-priority queues are used only when the high-priority queues are empty. The conditions can be built from atomic properties (boolean or numeric) using logical connectives (`&`, `|`, and `-` for negation). See the Prover9 manual for more details.[4]

To represent this construction in the Grackle space we fix the maximal number of queues (n) as well as the maximal number of subparts in each condition (m), and we introduce a parameter set H_i for the i-th high-priority `part` (where $0 < i \leq n$ and $0 < j \leq m$), namely,

1. H_i^{order} with the domain $\{\texttt{age}, \texttt{weight}, \texttt{random}\}$ to select the sorting criterion,

[2] https://www.cs.unm.edu/~mccune/prover9/manual-examples.html
[3] https://github.com/ai4reason/grackle/blob/v0.3/grackle/trainer/prover9/default.py
[4] https://www.cs.unm.edu/~mccune/prover9/manual/2009-11A/advanced.html

2. $H_{i,j}^{cond}$ to select the atomic property of the j-th part in the condition,
3. $H_{i,j}^{connect}$ for the logical connective between condition parts j and $j+1$ (`and`, `or`),
4. $H_{i,j}^{val}$ to select the numeric value to compare the j-th condition value with,
5. $H_{i,j}^{neg}$ to specify whether the j-th part of the condition is negated (`yes` or `no`), and
6. H_{i}^{ratio} to describe the queue selection ratio.

When $H_{i,j}^{cond}$ is set to a boolean property, then $H_{i,j}^{val}$ is ignored. When $H_{i,j}^{cond}$ is set to a numeric property, then the corresponding condition is $H_{i,j}^{cond} < H_{i,j}^{val}$ iff $H_{i,j}^{neg}$ is false (`no`), and $H_{i,j}^{cond} \geq H_{i,j}^{val}$ otherwise. $H_{i,j}^{cond}$ can be additionally set to `none` when $j > 1$ to end the condition without using all available parts. When H_{i}^{ratio} is 0, then the queue is disabled and not displayed at all independently of the other values. For the i-th queue H_i to be enabled and displayed, all queues H_k with $k < i$ must be enabled. Hence, when $H_1^{ratio} = 0$, then there is no high-priority queue at all. These dependencies are expressed using ParamILS parameter conditions (see Section 3). For example, `queue1` above is described as follows.

$$H_1^{order} = \texttt{age} \quad H_{1,1}^{cond} = \texttt{weight} \quad H_{1,1}^{val} = 500 \quad H_{1,1}^{neg} = \texttt{no} \quad H_{1,1}^{connect} = \texttt{and}$$
$$H_1^{ratio} = 50 \quad H_{1,2}^{cond} = \texttt{horn} \quad H_{1,2}^{val} = N/A \quad H_{1,2}^{neg} = \texttt{yes} \quad H_{1,3}^{cond} = \texttt{none}$$

Similarly, we introduce parameters L_i for low-priority queues. Additionally, we introduce a low-priority queue that contains `all` generated clauses, just like the queue `rest` above, for the sake of completeness. This queue is added when at least one queue is enabled.

Actions are used to change the search strategy during the proof search. For example:

```
generated=5000 -> assign(max_weight, 42).
```

changes the value of the parameter `max_weight` to 42 when the number of generated clauses exceeds 5,000. In addition to the `generated` above, an action can be triggered by the number of `given` (processed) clauses, by the number of retained clauses (`kept`), or by the proof search `level` in the case of the breadth-first search. Only a limited subset of parameters can be changed during the proof search.[5] Similarly to the clause selection, we fix the maximal number of actions (n), and we embed Prover9 actions to change integer-valued options in the Grackle strategy space using several parameter sets A_i ($0 < i \leq n$). Namely, we use $A_i^{counter}$ to select the trigger counter, A_i^{cond} to select the threshold value, A_i^{action} to select the parameter to be changed, and A_i^{value} to select the new value. The counter $A_i^{counter}$ can be additionally set to `none` to disable the action at all, and again, all previous actions A_k for $k < i$ must be enabled for the action A_i to be displayed. Similarly, we define actions F_i that change boolean flags.

Keep and delete rules are sets of conditions to describe clauses to keep or delete during the proof search. When a clause satisfies one of the delete conditions, it is deleted immediately after it is generated, unless it satisfies one of the keep conditions. Clauses not matching any of the conditions are always kept. The syntax for the conditions is the same as in the case of conditions for the given clause selection. Hence, once again, we fix the number of rules in each list of conditions (n), and we fix the number of subparts for each condition (m). We define parameter sets K_i and D_i for keep and delete conditions, respectively. Just like for the clause selection, we use parameters $K_{i,j}^{cond}$, $K_{i,j}^{connect}$, $K_{i,j}^{val}$, $K_{i,j}^{neg}$. The only difference is that $K_{i,j}^{cond}$ can be set to `none` even for $j = 1$ to disable the rule altogether. Similarly for $D_{i,j}$.

[5] Only 9 parameters are currently supported in Prover9 even though the manual lists more.

space	*solved*			*single*	*strategies*			*inventions*	
	total	*new*	*unique*	*best*	*new*	*needed*	*advanced*	*total*	*failed*
default	552	+238	6	**504**	79	9	0	113	34
full	**599**	**+285**	**51**	494	101	14	49	136	35
2-*staged*	507	+193	1	414	75	16	69	84	9
4-*staged*	495	+181	2	449	52	8	52	52	0

Table 1: Grackle strategy invention statistics on AIM train problems.

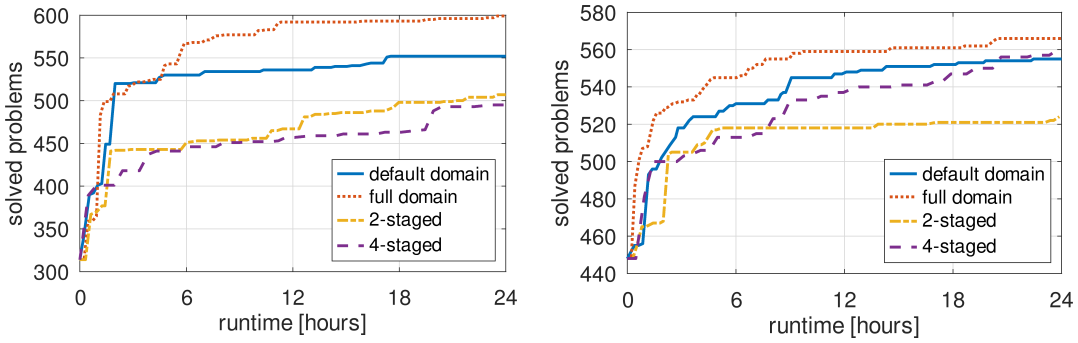

Figure 1: Problems solved by Grackle over time on AIM (left) and TPTP/NUM (right).

Staged strategy invention The full strategy space, which is later used in the experiments, encompasses two low- and high-priority queues, one flag-changing action, three value-changing actions, and keep/delete rules with two conditions each, where all conditions have only two parts. This space covers approximately 10^{110} different Prover9 strategies. Since the space might be quite big, we additionally implement a *staged* strategy invention, where the parameters are split into several independent parts and explored sequentially. We always start with the default space with advanced features disabled. The best parameters found are then fixed, and advanced features are explored next. The next stage can either explore all advanced features at once, which we call a 2-*phased* strategy invention, or the three kinds of advanced features can be explored one by one, which gives rise to a 4-*phased* process. This is similar to the hierarchical tuning in BliStrTune/EmpireTune.

5 Experiments with Grackle and Prover9

Experiments on AIM. We evaluate[6] Grackle strategy invention for Prover9 on the AIM benchmark used in the CASC 2016 ATP competition [17], consisting of 1020 training and 200 evaluation problems from a large theorem proving project in loop theory [9]. We launch 4 Grackle runs with strategy spaces described in Section 4: (1) with the *default* space, (2) with the *full* space including default and advanced features, (3) 2-*staged* strategy invention consequently tuning the default and advanced features, and finally (4) the 4-*staged* invention, tuning the advanced features one by one. All four runs use the time limit of $T = 10$ seconds for each prover run in the evaluation phase, $t = 5$ seconds in the invention phase, and the invention

[6]On two AMD EPYC 7513 32-Core processors @ 3680 MHz, with 2 GB memory limit per single prover run.

prover	*adds*	*adds%*	*sum*	*alone*
Prover9	+91	-	91	**91**
Vampire	+0	+0%	91	50
E	+0	+0%	91	36

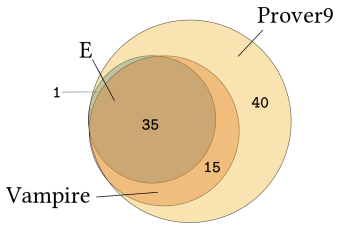

Figure 2: Results on AIM test problems. Greedy cover (left) and Venn diagrams (right).

phase is launched for $S = 5$ minutes. In the case of staged invention, the time limit S is the limit for a single stage. All runs start with the same 5 initial strategies that can solve together 314 training problems (with a time limit of T). We allow Grackle to run for 24 hours, and each run uses 16 CPU cores.

The results of Grackle invention are in Table 1. The column *total* shows the total number of problems solved by all invented strategies, *new* shows problems unsolved by the initial strategies, and *unique* states the number of problems not solved by other runs. The column *single best* shows the performance of the best invented strategy. Columns *strategies* describe how many *new* strategies were invented, how many are *needed* to cover the *total* problems from each run, and how many of the strategies use *advanced* Prover9 features. Columns *inventions* show how many strategies were invented in *total*, and how many times the invention *failed*, which happens when the invented strategy is already known. We can see that the *full* space performs the best, solving 285 new problems. On the other hand, the best strategy comes out from the run with the *default* space. The staged runs invented fewer strategies since the invention takes longer ($2S$ and $4S$). All the runs, except the *default*, have the freedom to choose whether to use advanced features or not. Roughly half of the strategies in the *full* space run use advanced features. The 4-staged run seems to enforce advanced features, but it does not lead to better overall performance in this experiment. The staged runs also minimize *failed* inventions. The graphical representation of the Grackle invention is in Figure 1 (left). The graph shows the number of solved problems over time during the Grackle runs.

We conducted additional runs with different settings ($S \in \{15, 30\}$), resulting in a collection of 1,101 Prover9 strategies. Grackle has already evaluated all strategies with a time limit of T, enabling us to construct a greedy cover sequence comprising the 10 best strategies and evaluate it on the test problems. The greedy cover is constructed by starting with the strongest strategy, removing the problems it solves, and iterating this process as long as there is a strategy that solves some problems. To mitigate overfitting, we split the training problems in half, construct the greedy cover on one half, and evaluate it on the other. This process is iterated several times to find the least overfitting portfolio of 10 strategies, which are then evaluated on the test problems with a time limit of 30 seconds. Its performance can be compared with that of state-of-the-art theorem provers E (autoschedule mode) and Vampire (casc mode), both launched for 300 ($=10 * 30$) seconds. The results are presented in Figure 2 as a greedy cover sequence (left). The column *adds* shows how many new problems are contributed by each strategy to the portfolio, and the column *sum* counts the performance of the portfolio up to that point. The column *adds%* shows the number in the column *adds* as a percentage of the current portfolio performance (the value of *sum* from the previous line). Finally, the column *alone* shows the individual performance of each solver. Notably, Prover9 solved a remarkable 91 problems, including all problems solved by the other solvers. The graphical representation of the solved problems in the form of proportional Venn diagrams is displayed in Figure 2 (right).

prover	*adds*	*adds%*	*sum*	*alone*
Prover9	+618	-	618	**618**
Vampire	+150	+24.43%	769	611
E	+1	+0.13%	770	541

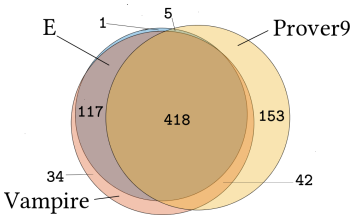

Figure 3: Results on TPTP/NUM problems. Greedy cover (left) and Venn diagrams (right).

The numbers correspond to the count of problems corresponding to each section.

Experiments on TPTP. Next, we evaluate Grackle strategy invention on TPTP problems [16]. As an initial assessment, we launch selected Grackle strategies, as well as Vampire and E, to discover that while the overall performance of Prover9 cannot compare to that of the state-of-the-art solvers, Prover9 still provides valuable contributions. TPTP problems are divided into categories, and we discover that the most significant contribution is in the category NUM, which contains problems from Number Theory. Following the main idea behind Grackle, which aims to enhance performance in areas where one performs best, we conduct several Grackle runs on 1,094 NUM problems, using the same settings as for AIM problems.

While the initial strategies solve 448 problems, the final portfolio of 23 strategies solves 618 (with 10 seconds per strategy/problem). The problems solved over time during the Grackle runs, with the same settings as for AIM, are depicted in Figure 1 (right). It is noteworthy that the *full* domain again performs the best, but the 4-*staged* approach slightly outperforms the *default* domain. After the Grackle runs, we compare the performance of the final portfolio with Vampire and E, both evaluated with a time limit of 300 seconds. This comparison with Vampire and E can be found in Figure 3, in the same format as for AIM in Figure 2. We observe that the Prover9 portfolio even slightly outperforms Vampire, and that our strategies solve significantly different problems than the other two solvers, resulting in 770 problems solved by joint efforts. Note that while problems solved by E are basically a subset of those solved by Vampire, Prover9 excels on different problems.

Different prover versions. In the above experiments, we used E version 2.6 and Vampire version 4.5.1. Since newer versions exist, namely E version 3.0.3 and Vampire version 4.8, we also evaluate these on the data. Interestingly, Vampire 4.8 shows significant improvement on AIM, solving 79 problems instead of 50, but it exhibits a notable decline on TPTP/NUM problems, where it solves 553 instead of 611. Conversely, the new version of E performs worse on both benchmarks, solving only 18 instead of 36 on AIM, and only 523 instead of 541 on TPTP/NUM. Vampire 4.8 solves 4 AIM problems not solved by Prover9/Grackle (which solves 91). The performance decrease is probably caused by optimization of solver strategy schedules for different problems.

6 Conclusions and Future Work

We have integrated support for Prover9 into the automated strategy invention system Grackle and assessed its capabilities across two distinct benchmark problem sets. The findings reveal that Prover9's performance can be significantly enhanced through our fully automated strategy

invention process. By comparison, Prover9 in its default *auto* mode can tackle 41 AIM problems and 512 on TPTP/NUM, whereas our strategies solve 91 and 619 within the same timeframe. Surprisingly, Prover9 can even outperform state-of-the-art provers, at least on the problem domains explored in this paper. The best invented strategies are available for download.[7]

For the AIM problems, the dataset is split in train/test parts. Since TPTP has been experimented with for many years in CASC, we did not perform such a split for NUM, hence, it is anticipated that Grackle-invented strategies will be specialized for NUM problems. However, this is not a concern in the primary scenario intended for Grackle, where users aim to solve as many problems as possible from a provided benchmark set. The better performance of Grackle-invented strategies on AIM than on TPTP is most likely to be explained by the fact that AIM problems are not included in the TPTP library, for which both E's and Vampire's schedules are optimized. Furthermore, it is noteworthy that we are comparing Grackle-invented portfolios with state-of-the-art portfolios in E and Vampire, which are supposed to be universally well-performing and are the results of previous intensive tuning (unfortunately unpublished). In theory, it is possible to use Grackle to invent AIM-specific or TPTP-specific strategies for E and Vampire as well, but this is left for future research.

Out of curiosity, we evaluated all strategies invented on AIM train problems on AIM test problems as well (with 30 s limit). Surprisingly, we discovered that the top 10 strategies collectively solved an impressive 111 AIM test problems. From the experiments with different solver versions, it is evident that strategy schedules of other provers are also subject to similar specialization/overfitting. This raises the question of whether having a single or several universal strategy schedules is the only possibility, or whether it is feasible to automatically configure the prover on the fly. This question is left for future research.

References

[1] Kristina Aleksandrova. Strategy invention for Prover9. Univesity of Innsbruck Thesis, 2023.

[2] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS (1)*, volume 13243. Springer, 2022.

[3] Chad E. Brown and Cezary Kaliszyk. Lash 1.0 (system description). In *IJCAR*, volume 13385 of *Lecture Notes in Computer Science*, pages 350–358. Springer, 2022.

[4] Jan Hůla, Jan Jakubův, Mikoláš Janota, and Lukáš Kubej. Targeted configuration of an SMT solver. In *CICM*, volume 13467 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2022.

[5] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an automatic algorithm configuration framework. *J. Artificial Intelligence Research*, 36:267–306, October 2009.

[6] Jan Jakubův, Martin Suda, and Josef Urban. Automated invention of strategies and term orderings for Vampire. In *GCAI*, volume 50 of *EPiC Series in Computing*, pages 121–133. EasyChair, 2017.

[7] Jan Jakubův and Josef Urban. BliStrTune: hierarchical invention of theorem proving strategies. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 43–52. ACM, 2017.

[8] Jan Jakubův and Josef Urban. Hierarchical invention of theorem proving strategies. *AI Commun.*, 31(3):237–250, 2018.

[7] https://github.com/ai4reason/grackle/blob/v0.3/examples/prover9-strategies.tar.gz

[9] Michael K. Kinyon, Robert Veroff, and Petr Vojtechovský. Loops with abelian inner mapping groups: An application of automated deduction. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *LNCS*, pages 151–164. Springer, 2013.

[10] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[11] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization, 2021.

[12] William McCune. Prover9 and Mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005–2010.

[13] W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.

[14] Aina Niemetz and Mathias Preiner. Bitwuzla at the SMT-COMP 2020. *CoRR*, abs/2006.01621, 2020.

[15] Stephan Schulz. System description: E 1.8. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[16] Geoff Sutcliffe. The TPTP world - infrastructure for automated reasoning. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *LNCS*, pages 1–12. Springer, 2010.

[17] Geoff Sutcliffe. The 6th IJCAR automated theorem proving system competition - CASC-J6. *AI Commun.*, 26(2):211–223, 2013.

[18] Josef Urban. BliStr: The Blind Strategymaker. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, volume 36 of *EPiC Series in Computing*, pages 312–319. EasyChair, 2015.