Regularization in Spider-Style Strategy Discovery and Schedule Construction

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

[1] Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague, Prague, Czech Republic
{filip.bartek,karel.chvalovsky,martin.suda}@cvut.cz

[2] Faculty of Electrical Engineering, Czech Technical University in Prague, Prague, Czech Republic

Abstract. To achieve the best performance, automatic theorem provers often rely on schedules of diverse proving strategies to be tried out (either sequentially or in parallel) on a given problem. In this paper, we report on a large-scale experiment with discovering strategies for the Vampire prover, targeting the FOF fragment of the TPTP library and constructing a schedule for it, based on the ideas of Andrei Voronkov's system Spider. We examine the process from various angles, discuss the difficulty (or ease) of obtaining a strong Vampire schedule for the CASC competition, and establish how well a schedule can be expected to generalize to unseen problems and what factors influence this property.

Keywords: Saturation-based theorem proving · Proving strategies · Strategy schedule construction · Vampire

1 Introduction

In 1997 at the CADE conference, the automatic theorem prover Gandalf [30] surprised its contenders at the CASC-14 competition [29] and won the MIX division there. One of the main innovations later identified as a key to Gandalf's success was the use of multiple theorem proving strategies executed sequentially in a time-slicing fashion [31,32]. Nowadays, it is well accepted that a single, universal strategy of an Automatic Theorem Prover (ATP) is invariably inferior, in terms of performance, to a well-chosen portfolio of complementary strategies, most of which do not even need to be complete or very strong in isolation.

Many tools have already been designed to help theorem prover developers discover new proving strategies and/or to combine them and construct proving schedules [7,9,12,16,21,24,33,34]. For example, Schäfer and Schulz employed genetic algorithms [21] for the invention of strong strategies for the E prover [23], Urban developed BliStr and used it to significantly strengthen strategies for the same prover via iterative local improvement and problem clustering [33],

Manuscript with additional appendices [1]: https://arxiv.org/abs/2403.12869.

© The Author(s) 2024
C. Benzmüller et al. (Eds.): IJCAR 2024, LNAI 14739, pp. 194–213, 2024.
https://doi.org/10.1007/978-3-031-63498-7_12

and, more recently, Holden and Korovin applied similar ideas in their HOS-ML system [7] to produce schedules for iProver [14]. The last work mentioned—as well as, e.g., MaLeS [16]—also include a component for *strategy selection*, the task of predicting, based on the input problem's features, which strategy will most likely succeed on it. (Selection is an interesting topic, which is, however, orthogonal to our work and will not be further discussed here.)

For the Vampire prover [15], schedules were for a long time constructed by Andrei Voronkov using a tool called Spider, about which little was known until recently. Its author finally revealed the architectural building blocks of Spider and the ideas behind them at the Vampire Workshop 2023, declaring Spider "a secret weapon behind Vampire's success at the CASC competitions" and "probably the most useful tool for Vampire's support and development" [34]. Acknowledging the importance of strategies for practical ATP usability, we decided to analyze this powerful technology on our own.

In this paper, we report on a large-scale experiment with discovering strategies for Vampire, based on the ideas of Spider (recalled in Sect. 2.1).[1] We target the FOF fragment of the TPTP library [28], probably the most comprehensive benchmark set available for first-order theorem proving. As detailed in Sect. 3, we discover and evaluate (on all the FOF problems) more than 1000 targeted strategies to serve as building blocks for subsequent schedule construction.

Research on proving strategies is sometimes frowned upon as mere "tuning for competitions". While we briefly pause to discuss this aspect in Sect. 4, our main interest in this work is to establish how well a schedule can be expected to generalize to unseen problems. For this purpose, we adopt the standard practice from statistics to randomly split the available problems into a train set and a test set, construct a schedule on one, and evaluate it on the other. In Sect. 6, we then identify several techniques that *regularize*, i.e., have the tendency to improve the test performance while possibly sacrificing the training one.

Optimal schedule construction under some time budget can be expressed as a mixed integer program and solved (given enough time) using a dedicated tool [7,24]. Here, we propose to instead use a simple heuristic from the related set cover problem [3], which leads to a polynomial-time greedy algorithm (Sect. 5). The algorithm maintains the important ability to assign different time limits to different strategies, is much faster than optimal solving (which may overfit to the train set in some scenarios), allows for easy experimentation with regularization techniques, and, in a certain sense made precise later, does not require committing to a single predetermined time budget.

In summary, we make the following main contributions:

- We outline a pragmatic approach to schedule construction that uses a greedy algorithm (Sect. 5), contrasting it with optimal schedules in terms of the quality of the schedules and the computational resources required for their construction (Sect. 6.2). In particular, our findings demonstrate a relative efficacy of the greedy approach for datasets similar to our own.

[1] Not claiming any credit for these, potential errors in the explanation are ours alone.

- Leveraging the adaptability of the greedy algorithm, we introduce a range of regularization techniques aimed at improving the robustness of the schedules in unseen data (Sect. 6). To the best of our knowledge, this represents the first systematic exploration into regularization of strategy schedules.
- The strategy discovery and evaluation is a computationally expensive process, which in our case took more than twenty days on 120 CPU cores. At the same time, there are further interesting questions concerning Vampire's strategies than we could answer in this work. To facilitate research on this paper's topic, we made the corresponding data set available online [2].

2 Preliminaries

The behavior of Vampire is controlled by approximately one hundred *options*. These options configure the preprocessing and clausification steps, control the saturation algorithm, clause and literal selection heuristics, determine the choice of generating inferences as well as redundancy elimination and simplification rules, and more. Most of these options range over the Boolean or a small finite domain, a few are numeric (integer or float), and several represent ratios.

Every option has a *default* value, which is typically the most universally useful one. Some option settings make Vampire incomplete. This is automatically recognized, so that when the prover finitely saturates the input without discovering a contradiction, it will report "unknown" (rather than "satisfiable").

A *strategy* is determined by specifying the values of all options. A *schedule* is a sequence $(s_i, t_i)_{i=1}^{n}$ of strategies s_i together with assigned time limits t_i, intended to be executed in the prescribed order. We stress that in this work we do not consider schedules that would branch depending on problem features.

2.1 Spider-Style Strategy Discovery and Schedule Construction

We are given a set of problems P and a prover with its space of available strategies $\mathbb{S}$. *Strategy discovery* and *schedule construction* are two separate phases. We work under the premise that the larger and more diverse a set of strategies we first collect, the better for later constructing a good schedule.

Strategy discovery consists of three stages: random probing, strategy optimization, and evaluation, which can be repeated as long as progress is made.

Random Probing. We start strategy discovery with an empty pool of strategies $S = \emptyset$. A straightforward way to make sure that a new strategy substantially contributes to the current pool S is to always try to solve a problem not yet solved (or *covered*) by any strategy collected so far. We repeatedly pick such a problem and try to solve it using a *randomly sampled* strategy out of the totality of all available strategies $\mathbb{S}$. The sampling distribution may be adapted to prefer option values that were successful in the past (cf. Sect. 3.3). This stage is computationally demanding, but can be massively parallelized.

Strategy Optimization. Each newly discovered strategy s, solving an as-of-yet uncovered problem p, will get *optimized* to be as fast as possible at solving p. One explores the strategy neighborhood by iterating over the options (possibly in several rounds), varying option values, and committing to changes that lead to a (local) improvement in terms of solution time or, as a tie-breaker, to a default option value where time differences seem negligible. We evaluate the impact of this stage in Sect. 3.4.

Strategy Evaluation. In the final stage of the discovery process, having obtained an optimized version s' of s, we evaluate s' on all our problems P. (This is another computationally expensive, but parallelizable step.) Thus, we enrich our pool and update our statistics about covered problems. Note that every strategy s' we obtain this way is associated with the problem $p_{s'}$ for which it was originally discovered. We will call this problem the *witness problem* of s'.

Schedule Construction can be tried as soon as a sufficiently rich (cf. Sect. 3) pool of strategies is collected. Since we, for every collected strategy, know how it behaves on each problem, we can pose schedule construction as an optimization task to be solved, e.g., by a (mixed) integer programming (MIP) solver.

In more detail: We seek to allocate time slices $t_s > 0$ to some of the strategies $s \in S$ to cover as many problems as possible while remaining in sum below a given time budget T [7,24]. Alternatively, we may try to cover all the problems known to be solvable in as little total time as possible.[2] In this paper, we describe an alternative schedule construction method based on a greedy heuristic, with a polynomial running time guarantee and other favorable properties (Sect. 5).

2.2 CPU Instructions as a Measure of Time

We will measure computation time in terms of the number of user instructions executed (as available on Linux systems through the `perf` tool). This is, in our experience, more precise and more stable (on architectures with many cores and many concurrently running processes) than measuring real time.[3]

In fact, we report *megainstructions (Mi)*, where 1 Mi = 2^{20} instructions reported by `perf`. On contemporary hardware, 2000 Mi will typically get used up in a bit less than a second and 256 000 Mi in around 2 min of CPU time. We also set 1 Mi as the granularity for the time limits in our schedules.

3 Strategy Discovery Experiment

Following the recipe outlined in Sect. 2.1, we set out to collect a pool of Vampire (version 4.8) strategies covering the first-order form (FOF) fragment of the

[2] Strictly speaking, these only give us a set of strategy-time pairs (as opposed to a sequence). However, the strategies can be ordered heuristically afterward.

[3] For a more thorough motivation, see Appendix A of [26].

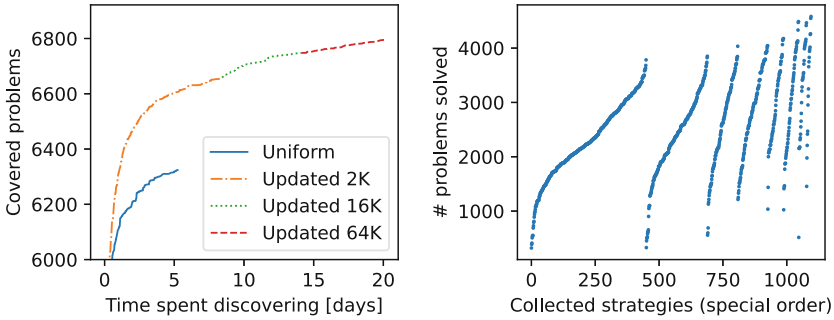

Fig. 1. Strategy discovery. *Left:* problem coverage growth in time (uniform strategy sampling distribution vs. an updated one). *Right:* collected strategies ordered by limit (2000, 4000, . . . , 256 000 Mi) and, secondarily, by how many problems can each solve.

TPTP library [28] version 8.2.0. We focused only on proving, so left out all the problems known to be satisfiable, which left us with a set P of 7866 problems. Parallelizing the process where possible, we strived to fully utilize 120 cores (AMD EPYC 7513, 3.6 GHz) of our server equipped with 500 GB RAM.

We let the process run for a total of 20.1 days, in the end covering 6796 problems, as plotted in Fig. 1 (left). The effect of diminishing returns is clearly visible; however, we cannot claim we have exhausted all the possibilities. In the last day alone, 8 strategies were added and 9 new problems were solved.

The rest of Fig. 1 is gradually explained in the following as we cover some important details regarding the strategy discovery process.

3.1 Initial Strategy and Varying Instruction Limits

We seeded the pool of strategies by first evaluating Vampire's default strategy for the maximum considered time limit of 256 000 Mi, solving 4264 problems out of the total 7866.

To save computation time, we did not probe or evaluate all subsequent strategies for this maximum limit. Instead, to exponentially prefer low limits to high ones, we made use of the Luby sequence[4] [18] known for its utility in the restart strategies of modern SAT solvers. Our own use of the sequence was as follows.

The lowest limit was initially set to 2000 Mi and, multiplying the Luby sequence members by this number, we got the progression 2000, 2000, 4000, 2000, 2000, 4000, 8000, . . . as the prescribed limits for subsequent probe iterations. This sequence reaches 256 000 Mi for the first time in 255 steps. At that point, we stopped following the Luby sequence and instead started from the beginning (to avoid eventually reaching limits higher than 256 000 Mi).

After four such cycles, the lowest, that is 2000 Mi, limit probes stopped producing new solutions (a sampling timeout of 1 h per iteration was imposed).

[4] https://oeis.org/A182105.

Here, after almost 8.5 d, the "updated 2K" plot ends in Fig. 1 (left). We then increased the lowest limit to 16 000 Mi and continued in an analogous fashion for 155 iterations and 5.7 more days ("updated 16K") and eventually increased the lowest limit to 64 000 Mi ("updated 64K") until the end.

Figure 1 (right) is a scatter plot showing the totality of 1096 strategies that we finally obtained and how they individually perform. The primary order on the x axis is by the limit and allows us to make a rough comparison of the number of strategies in each limit group (2000 Mi, 4000 Mi, ..., 256 000 Mi, from left to right). It is also clear that many strategies (across the limit groups) are, in terms of problem coverage, individually very weak, yet each at some point contributed to solving a problem considered (at that point) challenging.

3.2 Problem Sampling

While the guiding principle of random probing is to constantly aim for solving an as-of-yet unsolved problem, we modified this criterion slightly to produce a set of strategies better suited for an unbiased estimation of schedule performance on unseen problems (as detailed in the second half of this paper).

Namely, in each iteration i, we "forgot" a random half P_i^F of all problems P, considered only those strategies (discovered so far) whose witness problem lies in the remaining half $P_i^R = P \setminus P_i^F$, and aimed for solving a random problem in P_i^R not covered by any of these strategies. This likely slowed the overall growth of coverage, as many problems would need to be covered several times due to the changing perspective of P_i^R. However, we got a (probabilistic) guarantee that any (not too small) subset $P' \subseteq P$ will contain enough witness problems such that their corresponding strategies will cover P' well.

3.3 Strategy Sampling

We sampled a random strategy by independently choosing a random value for each option. The only exception were dependent options. For example, it does not make sense to configure the AVATAR architecture (changing options such as `acc`, which enables congruence closure under AVATAR) if the main AVATAR option (`av`) is set to `off`. Such complications can be easily avoided by following, during the sampling, a topological order that respects the option dependencies. (For example, we sample `acc` only after the value `on` has been chosen for `av`.)

Even under the assumption of option independence, the mean time in which a random strategy solves a new problem can be strongly influenced by the value distributions for each option. This is because some option values are rarely useful and may even substantially reduce the prover performance, for example, if they lead to a highly incomplete strategy.[5] Nevertheless, not to preclude the

[5] An extreme example is turning off *binary resolution*, the main inference for non-equational reasoning. This can still be useful, for instance when replaced by *unit resulting resolution* [15], but our sampling needs to discover this by chance.

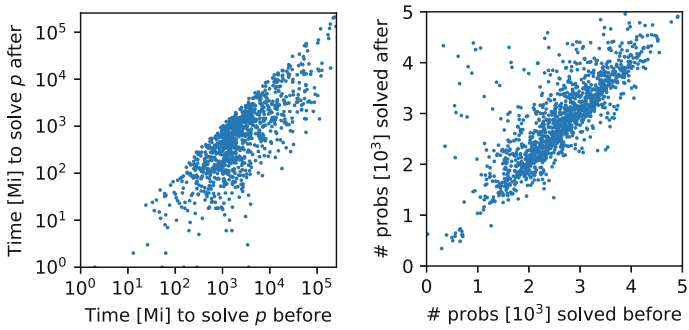

Fig. 2. Strategy optimization scatter plots. *Left:* time needed to solve strategy's witness problem (a log–log plot). *Right:* the total number of problems (in thousands) solved.

possibility of discovering arbitrarily wild strategies, we initially sampled every option uniformly where possible.[6]

Once we collected enough strategies,[7] we updated the frequencies for sampling finite-domain options (which make up the majority of all options) by counting how many times each value occurred in a strategy that, at the moment of its discovery, solved a previously unsolved problem. (This was done before a strategy got optimized. Otherwise the frequencies would be skewed toward the default, especially for option values that rarely help but almost never hurt.)

The effect of using an updated sampling distribution for strategy discovery can be seen in Fig. 1 (left). We ran two independent versions of the discovery process, one with the uniform distribution and one with the updated distribution. We abandoned the uniform one after approximately 5 d, by which time it had covered 6324 problems compared to 6607 covered with the help of the updated distribution at the same mark. We can see that the rate at which we were able to solve new problems became substantially higher with the updated distribution.

3.4 Impact of Strategy Optimization

Once random probing finds a new strategy s that solves a new problem p, the task of optimization (recall Sect. 2.1) is to search the option-value neighborhood of s for a strategy s' that solves p in as few instructions as possible and preferably uses default option values (where this does not compromise performance on p).

The impact of optimization is demonstrated in Fig. 2. On the left, we can see that, almost invariably, optimization substantially improves the performance of the discovered strategy on its witness problem p. The geometric mean of the improvement ratio we observed was 4.2 (and the median 3.2). The right

[6] Exceptions were: 1. The ratios: e.g., for `age_weight_ratio` we sampled uniformly its binary logarithm (in the range −10 and 4) and turned this into a ratio afterward (thus getting values between 1 : 1024 and 16 : 1); 2. Unbounded integers (an example being the *naming threshold* [20]), for which we used a geometric distribution instead.

[7] This was done in an earlier version of the main experiment.

scatter plot shows the overall performance of each strategy.[8] Here, the observed improvement is ×1.09 on average (median 1.03), and the improvement is solely an effect of setting option values to default where possible (without this feature, we would get a geometric mean of the improvement 0.84 and median 0.91). In this sense, the tendency to pick default values regularizes the strategies, making them more powerful also on problems other than their witness problem.

3.5 Parsing Does Not Count

When collecting the performance data about the strategies, we decided to ignore the time it takes Vampire to parse the input problem. This was also reflected in the instruction limiting, so that running Vampire with a limit of, e.g., 2000 Mi would allow a problem to be solved if it takes at most 2000 Mi on top of what is necessary to parse the problem.

The main reason for this decision is that Vampire, in its strategy scheduling mode, starts dispatching strategies only after having parsed the problem, which is done only once. Thus, from the perspective of individual strategies, parsing time is a form of a sunk cost, something that has already been paid.

Although more complex approaches to taking parse time into account when optimizing schedules are possible, we in this work simply pretend that problem parsing always costs 0 instructions. This should be taken into account when interpreting our simulated performance results reported next (in Sect. 4, but also in Sect. 6.2).

4 One Schedule to Cover Them All

Having collected our strategies, let us pretend that we already know how to construct a schedule (to be detailed in Sect. 5) and use this ability to answer some imminent questions, most notably: How much can we now benefit?

Figure 3 plots the cumulative performance (a.k.a. "cactus plot") of schedules we could build after 2 h, 6 h, 1 day, and full 20.1 days of the strategy discovery. The dashed vertical line denotes the time limit of 256 000 Mi, which roughly corresponds to a 2-minute prover run. For reference, we also plot the behavior of Vampire's default strategy. We can see that already after two hours of strategy discovery, we could construct a schedule improving on the default strategy by 26% (from 4264 to 5403 problems solved). Although the added value per hour spent searching gradually drops, the 20.1 days schedule is still 4% better than the 1 day one (improving from 6197 to 6449 at 256 000 Mi).

The plot's x-axis ends at 8·256 000 Mi, which roughly corresponds to the time limit used by the most recent CASC competitions [27] in the FOF division (i.e., 2 min on 8 cores). The strongest schedule shown in the figure manages to

[8] In a previous version of the main experiment, we evaluated each strategy both before and after optimization, which gave rise to this plot.

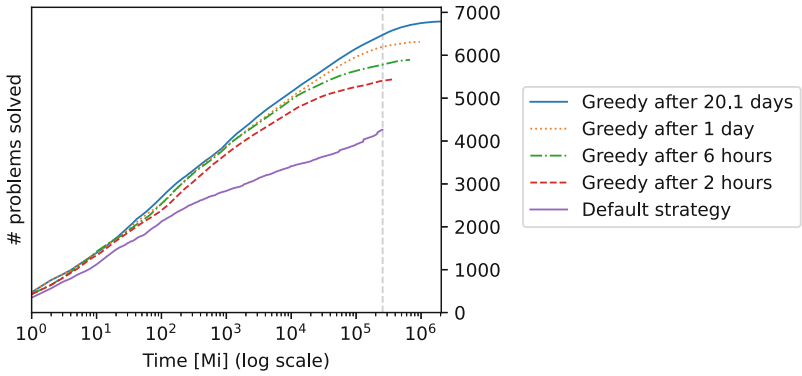

Fig. 3. Cumulative performance of several greedy schedules, each using a subset of the discovered strategies as gathered in time, compared with Vampire's default strategy

solve 6789 problems (of the 6796 covered in total) at that mark.[9] We remark that this schedule, in the end, employs only 577 of the 1096 available strategies, which points towards a noticeable redundancy in the strategy discovery process.

One way to fit all the solvable problems below the CASC budget would be to use a standard trick and split the totality of problems P into two or more easy-to-define syntactic classes (e.g., Horn problems, problems with equality, large problems, etc.) and construct dedicated schedules for each class in isolation. The prover can then be dispatched to run an appropriate schedule once the input problem features are read. We do not explore this option here. Intuitively, by splitting P into smaller subsets, the risk of overfitting to just the problems for which the strategies were discovered increases, and we mainly want to explore here the opposite, the ability of a schedule to generalize to unseen problems.

5 Greedy Schedule Construction

Having collected a set of strategies S and evaluated each on the problems in P, let us by $E_p^s : S \times P \to \mathbb{N} \cup \{\infty\}$ denote the *evaluation matrix*, which records the obtained solutions times (and uses ∞ to signify a failure to solve a problem within the evaluation time limit used). Given a time budget T, the *schedule construction problem* (SCP) is the task of assigning a time limit $t_s \in \mathbb{N}$ to every strategy $s \in S$, such that the number of covered problems

$$\left| \bigcup_{s \in S} \{p \in P \mid E_p^s \leq t_s\} \right|,$$

subject to the constraint $\sum_{s \in S} t_s \leq T$, is maximized.

[9] It is possible to solve all 6796 covered problems with a schedule that spans 2 582 228 Mi. This is the optimum length – no shorter schedule solving all covered problems exists.

Algorithm 1. Greedy schedule construction (base version)

Input: Problems P, strategies S, solution times E_p^s, time budget T
Output: Pre-schedule $t_s : S \to \mathbb{N}$

1: $t_s \leftarrow \mathbf{0}$ ▷ Start with zeros everywhere
2: $T' \leftarrow T, P' \leftarrow P$ ▷ Remaining budget, remaining problems
3: **while** the maximum just below is positive **do**
4: $\quad s, t \leftarrow \operatorname{argmax}_{s \in S, t \in \mathbb{N}, 0 < (t - t_s) \leq T'} |\{p \in P' \mid E_p^s \leq t\}| / (t - t_s)$
5: $\quad T' \leftarrow T' - (t - t_s)$ ▷ Update the remaining budget
6: $\quad t_s \leftarrow t$ ▷ Extend the pre-schedule
7: $\quad P' \leftarrow \{p \in P' \mid E_p^s > t\}$ ▷ Remove covered problems from P'

To obtain a schedule as a sequence (as defined in Sect. 2), we would need to order the strategies having $t_s > 0$. This can, in practice, be done in various ways, but since the order does not influence the predicted performance of the schedule under the budget T, we keep it here unspecified (and refer to the mere time assignment t_s as a *pre-schedule* where the distinction matters).

Although it is straightforward to encode SCP as a mixed integer program and attempt to solve it exactly (though it is an NP-hard problem), an adaptation of a greedy heuristic from a closely related (budgeted) maximum coverage problem [3,13] works surprisingly well in practice and runs in time polynomial in the size of E_p^s. The key idea is to greedily maximize the number of newly covered problems *divided* by the amount of time this additionally requires.

Algorithm 1 shows the corresponding pseudocode. It starts from an empty schedule t_s and iteratively extends it in a greedy fashion. The key criterion appears on line 4. Note that this line corresponds to an iteration over all available strategies S and, for each strategy s, all meaningful time limits (which are only those where a new problem gets solved by s, so their number is bounded by $|P|$).

Algorithm 1 departs from the obvious adaptation of the above-mentioned greedy algorithm for the set covering problem [3] in that we allow extending a slice of a strategy s that is already included in the schedule (that is, has $t_s > 0$) and "charge the extension" only for the additional time it claims (i.e., $t - t_s$). This *slice extension* trick turns out to be important for good performance.[10]

5.1 Do We Need a Budget?

A budget-less version of Algorithm 1 is easy to obtain (imagine T being very large). When running on real-world E_p^s (from evaluated Vampire strategies), we noticed that the length of a typical extension $(t - t_s)$ tends to be small relative to the current used-up time $\sum_{s \in S} t_s$ and that the presence of a budget starts affecting the result only when the used-up time comes close to the budget.

As a consequence, if we run a budget-less version and, after each iteration, record the pair $(\sum_{s \in S} t_s, |P \setminus P'|)$, we get a good estimate (in a single run) of how the algorithm would perform for a whole (densely inhabited) sequence of

[10] The degree of importance of slice extension can be observed in Fig. 5 in Appendix A.

relevant budgets. This is how the plot in Fig. 3 was obtained. Note that this would be prohibitively expensive to do when trying to solve the SCP optimally.

We can also use this observation in an actual prover. If we record and store a journal of the budget-less run, remembering which strategy got extended in each iteration and by how much, we can, given a concrete budget T, quickly replay the journal just to the point before filling up T, and thus get a good schedule for the budget T without having to optimize specifically for T.

6 Regularization in Schedule Construction

To estimate the future performance of a constructed schedule on previously unseen problems, we adopt the standard methodology used in statistics, randomly split our problem set P into a train set P_{train} and a test set P_{test}, construct a schedule for the first, and evaluate it on the second.

To reduce the variance in the estimate, we use many such random splits and average the results. In the experiments reported in the following, we actually compute an average over several rounds of 5-fold cross-validation [6]. This means that the size of P_{train} is always 80.0 % and the size P_{test} 20.0% of our problem set P. However, we *re-scale* the reported train and test performance back to the size of the whole problem set P to express them in units that are immediately comparable. We note that the reported performance is obtained though simulation, i.e., it is only based on the evaluation matrix E_p^s.

Training Strategy Sets. We retroactively simulate the effect of discovering strategies only for current training problems P_{train}. Given our collected pool of strategies S, we obtain the training strategy set S_{train} by excluding those strategies from S whose witness problem lies outside P_{train} (cf. Sect. 3.2). When a schedule is optimized on the problem set P_{train}, the training data consists of the results of the evaluations of strategies S_{train} on problems P_{train}.

6.1 Regularization Methods

We propose several modifications of greedy schedule construction (Algorithm 1) with the aim of improving its performance on unseen problems (the test set performance) while possibly sacrificing some of its training performance.

With the base version, we observed that it could often solve more test problems by assigning more time to strategies introduced into the schedule in early iterations, at the expense of strategies added later (the latter presumably covering just a few expensive training problems and being over-specialized to them). Most of the modifications described next assign more time to strategies added during early iterations, each according to a different heuristic.

Slack. The most straightforward regularization we explored extends each non-zero strategy time limit t_s in the schedule by multiplying it by the multiplicative slack $w \geq 1$ and adding the additive slack $b \in \{0, 1, \ldots\}$. For each

$t_s > 0$, the new limit t'_s is therefore $t_s \cdot w + b$. To avoid overshooting the budget, we keep track of the total length of the extended schedule during the construction (implementation details are slightly more complicated but not immediately important). The parameters w and b control the degree of regularization, and with $w = 1$ and $b = 0$, we get the base algorithm.

Temporal Reward Adjustment. In each iteration of the base greedy algorithm, we select a combination of strategy s and time limit t that maximizes the number of newly solved problems n per time t. Intuitively, the relative degree to which these two quantities influence the selection is arbitrary. To allow stressing n more or less with respect to t, we exponentiate n by a regularization parameter $\alpha \geq 0$, so the decision criterion becomes $\frac{n^\alpha}{t}$.

For small values of α, the algorithm values the time more and becomes eager to solve problems early. For large values of α, on the other hand, the algorithm values the problems more and prefers longer slices that cover more problems. For example, for $\alpha = 1.5$, the algorithm prefers solving 2 problems in 5000 Mi to solving 1 problem in 2000 Mi. Compare this to $\alpha = 1$ (the base algorithm), which would rank these slices the other way around.

Diminishing Problem Rewards. By covering a training problem with more than one strategy, we cover it robustly: When a similar testing problem is solved by only one of these strategies, the schedule still manages to solve it. However, the base greedy algorithm does not strive to cover any problem more than once: as soon as a problem is covered by one strategy, this problem stops participating in the scheduling criterion. This is the case even when covering the problem again would cost relatively little time.

Regularization by diminishing problem rewards covers problems robustly by rewarding strategy s not only by the number of *new* problems it covers but also by the problems covered by s that are already covered by the schedule. This is achieved by modifying the slice selection criterion. Instead of maximizing the number of new problems solved per time, we maximize the total reward per time, which is defined as follows: Each problem contributes the reward β^k, where k is the number of times the schedule has covered the problem and β is a regularization parameter ($0 \leq \beta \leq 1$). We define $0^0 = 1$ so that $\beta = 0$ preserves the original behavior of the base algorithm.

For example, for $\beta = 0.1$, each problem contributes the reward 1 the first time it is covered, 0.1 the second time, 0.01 the third time, etc. Informally, the algorithm values covering a problem the second time in time t as much as covering a new problem in time $10 \cdot t$.

These modifications are independent and can be arbitrarily combined.

6.2 Experimental Results

We evaluated the behavior of the previously proposed techniques using three time budgets: 16 000 Mi (≈8 s), 64 000 Mi (≈32 s), and 256 000 Mi (≈2 min).

Optimal Schedule Constructor. In the existing approaches to the construction of strategy schedules [7,24], it is common to encode the SCP (see Sect. 5) as a mixed-integer program and use a MIP solver to find an exact solution. We implemented such an optimal schedule construction (OSC) by encoding the problem[11] in Gurobi [5] (ver. 10.0.3) and compared OSC to the base greedy schedule construction (Algorithm 1) on 10 random 80 : 20 splits.

For the budget of 256 000 Mi, it takes Gurobi over 16 h to find an optimal schedule, whereas the greedy algorithm finds a schedule in less than a minute. The optimal schedule solves, on average, 45.0 (resp. 8.5) more problems than the greedy schedule on P_{train} (resp. on P_{test}) when re-scaled to $|P|$. For the 16 000 Mi and 64 000 Mi budgets, Gurobi does not solve the optimal schedule within a reasonable time limit. For example, after 24 h, the relative gaps between the lower and upper objective bound are 5.38 % and 1.43 %, respectively. This makes the OSC impractical to use as a baseline for our regularization experiments.[12]

Regularization of the Greedy Algorithm. To estimate the performance of the proposed regularization methods, we evaluated each variant on 50 random splits (10 times 5-fold cross-validation). We assessed the algorithm's response to each regularization parameter in isolation. For each parameter, we evaluated regularly spaced values from a promising interval covering the default value ($b = 0$, $w = 1$, $\alpha = 1$, $\beta = 0$). Figure 4 demonstrates the effect of these variations on the train and test performance for the budget 64 000 Mi.[13]

Temporal reward adjustment was the most powerful of the regularizations, improving test performance for all the evaluated values of α between 1.1 and 2.0. Surprisingly, the values 1.1 and 1.2 also improved the train performance, suggesting that the default greedy algorithm is too time-aggressive on our dataset.

Table 1 compares the performance of notable configurations of the greedy algorithm. Specifically, we include evaluations of the base greedy algorithm and the best of the evaluated parameter values for each of the regularizations. The table also illustrates the effect of regularizations on the computational time of the greedy schedule construction: $\beta > 0$ slows the procedure down and $\alpha > 1$ speeds it up.

In a subsequent experiment, we searched for a strong combination of regularizations by local search from the strongest single-parameter regularization ($\alpha = 1.7$). This yielded a negligible improvement over $\alpha = 1.7$: The best observed test performance was 5707 ($\alpha = 1.7$ and $b = 30$), compared to 5704 of $\alpha = 1.7$.

Finally, we briefly explored the interactions between the budget and the optimal values of the regularization parameters. For each of the three budgets of interest and each of the regularization parameters, we identified the best param-

[11] We used a straightforward encoding similar to the encoding described by Holden and Korovin [7].

[12] A better encoding and solver settings may improve on this. However, we suspect the problem to be hard; we tried some modifications with similar (or worse) results.

[13] This budget seems to be the most practically relevant (e.g., for the application in interactive theorem provers). The other two budgets are detailed in Appendix A.

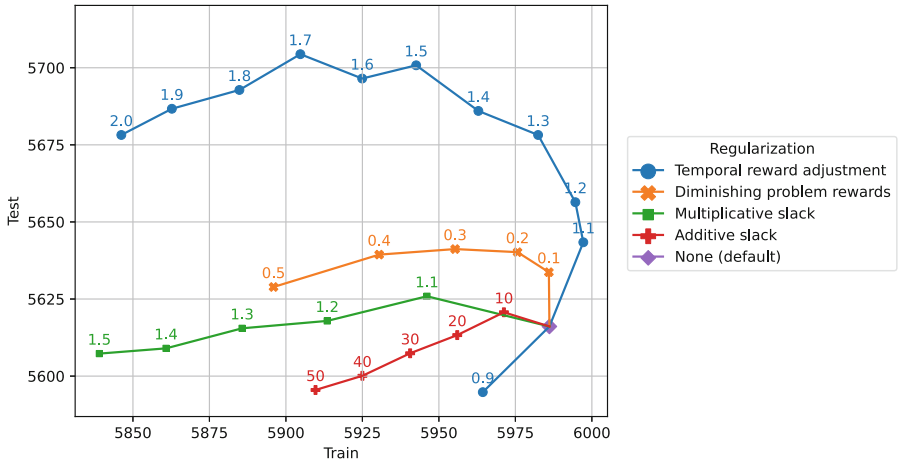

Fig. 4. Train and test performance of various regularizations of the greedy schedule construction algorithm for the budget 64 000. Performance is the mean number of problems solved out of 7866 across 50 splits. The label of each point denotes the value of the respective regularization parameter.

eter value from the evaluation grid. Table 2 shows that the best configurations of all the regularizations except multiplicative slack vary across budgets.[14]

7 Related Work

Outside the realm of theorem proving, strategy discovery belongs to the topic of *algorithm configuration* [22], where the task is to look for a strong configuration of a parameterized algorithm automatically. Prominent general-purpose algorithm configuration procedures include ParamILS [8] and SMAC [17].

To gather a portfolio of complementary configurations, Hydra [36] searches for them in rounds, trying to maximize the marginal contribution against all the configurations identified previously. Cedalion [25] is interesting in that it maximizes such contribution *per unit time*, similarly to our heuristic for greedy schedule construction. Both have in common that they, a priori, consider all the input problems in their criterion. BliStr and related approaches [7,9,11,12,33], on the other hand, combine strategy improvement with problem clustering to breed strategies that are "local experts" on similar problems. Spider [34] is even more radical in this direction and optimizes each strategy on a single problem.[15]

[14] See a more detailed comparison in Appendix A.

[15] Although the preference for default option values as a secondary criterion, at the same time, helps to push for good general performance (see Sect. 3.4).

Table 1. Comparison of regularizations of the greedy schedule construction algorithm for the budget 64 000 Mi. Performance is the mean number of problems solved out of 7866 across 50 splits. Time to fit is the mean time to construct a schedule in seconds.

Regularization	Performance		Time to fit [s]
	Test	Train	
$\alpha = 1.7$	5704	5905	4
$\beta = 0.3$	5641	5955	67
$w = 1.1$	5626	5946	19
$b = 10$	5621	5971	20
None (default)	5616	5986	20

Table 2. Best observed values of regularization parameters for various budgets

Budget [Mi]	Best parameter value			
	b	w	α	β
16000	0	1.1	1.3	0.2
64000	10	1.1	1.7	0.3
256000	20	1.1	1.4	0.2

Once a portfolio of strategies is known, it may be used in one of several ways to solve a new input problem: execute all strategies in parallel [36], select a single strategy [9], select one of pre-computed schedules [7], construct a custom strategy schedule [19], schedule strategies dynamically [16], or use a pre-computed static schedule [12,24]. The latter is the approach we explored in this work.

A popular approach to construct a static schedule (besides solving SCP optimally [7,24]) is to greedily stack uniformly-timed slices [12].[16] Regularization in this context is discussed by Jakubuv et al. [10]. Finally, a different greedy approach to schedule construction was already proposed in *p-SETHEO* [35].

8 Conclusion

In this work, we conducted an independent evaluation of Spider-style [34] strategy discovery and schedule creation. Focusing on the FOF fragment of the TPTP library, we collected over a thousand Vampire proving strategies, each a priori optimized to perform well on a single problem. Using these strategies, it is easy to construct a single monolithic schedule which covers most of the problems

[16] However, uniformly-timed slices only get close to the performance of our greedy schedule at a small region depending on the slice time limit used.

known to be solvable within the budget used by the CASC competition. This suggests that for CASC not to be mainly a competition in memorization, using a substantial set of previously unseen problems each year is essential.

To construct strong schedules using the discovered strategies, we proposed a greedy schedule construction procedure, which can compete with optimal approaches. For a time budget of approximately 2 min, the greedy algorithm takes less than a minute to produce a schedule that solves more than 99.0% as many problems as an optimal schedule, which takes more than 16 h to generate. For shorter time budgets, optimal schedule construction is no longer feasible, while greedy construction still produces relatively strong schedules.

This surprising strength of the greedy scheduler can be further reinforced by various regularization mechanisms, which constitute the main contribution of this work. An appropriately chosen regularization allows us to outperform the optimal schedule on unseen problems. Finally, the runtime speed and simplicity of the greedy schedule construction algorithm and the regularization techniques make them attractive for reuse and further experimentation.

Acknowledgment. This work was supported by the Czech Science Foundation project no. 20-06390Y (JUNIOR grant), the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466, the project RICAIP no. 857306 under the EU-H2020 programme, the Grant Agency of the Czech Technical University in Prague, grant no. SGS20/215/OHK3/3T/37, and the Czech Science Foundation project no. 24-12759S.

A Experimental Results on Various Budgets

In addition to the budget of 64 000 Mi (approx 32 s), which we discussed in Sect. 6.2, we evaluated the schedule construction algorithms on the budgets of 16 000 Mi (approx. 8 s) and 256 000 Mi (approx. 2 min). Figure 5 shows the results of these evaluations. It shows namely that temporal reward adjustment is the most powerful of the regularizations under consideration for all of these budgets, and that the optimal values of most of the regularization parameters vary across budgets.

To demonstrate the effect of the slice extension trick described in Sect. 5, we also include two weaker versions of the base greedy algorithm: one without slice extension and one with the slice extension restricted to the most recent slice ("conservative slice extension"). Both of these modifications allow including any single strategy in the schedule more than once, which is implemented in a straightforward fashion.

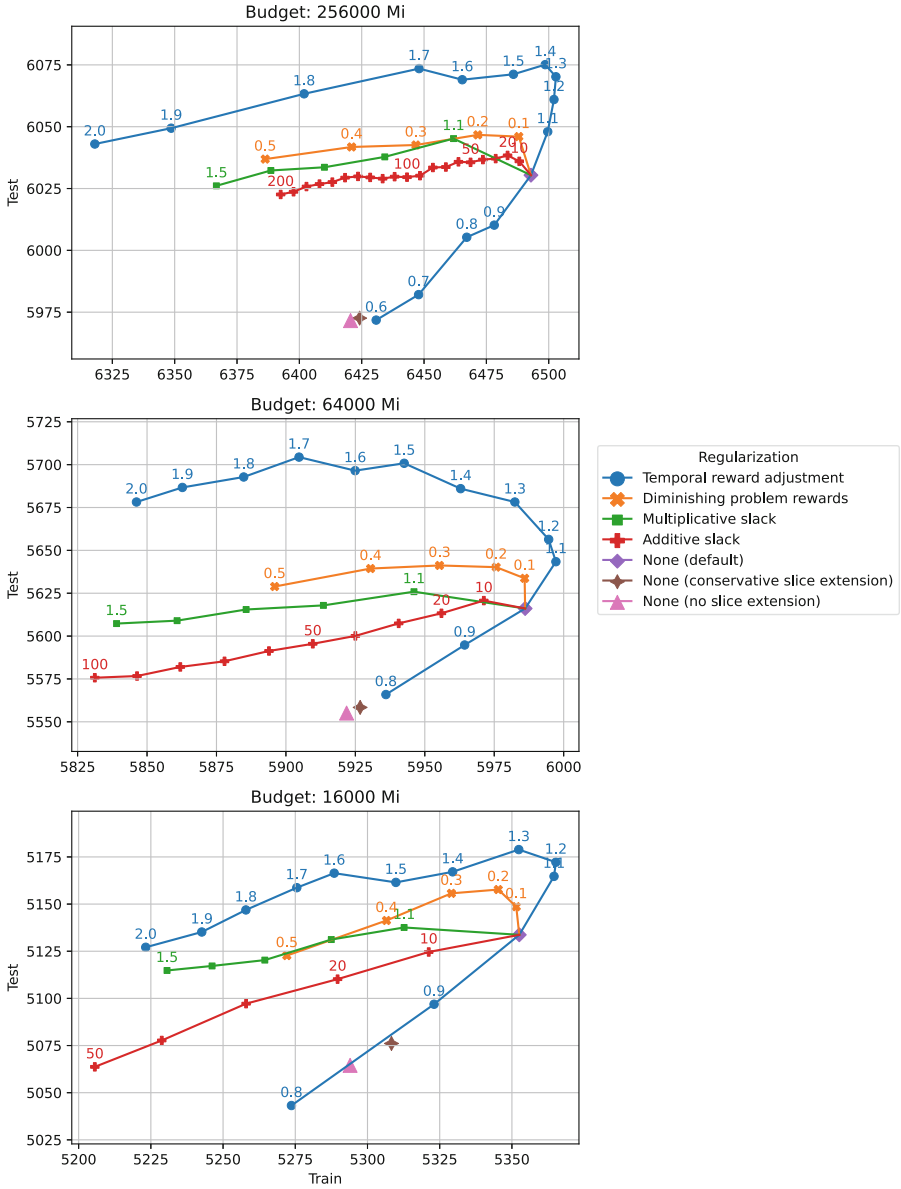

Fig. 5. Train and test performance of various regularizations of the greedy schedule construction algorithm for the budgets 256 000 Mi (*top*), 64 000 Mi (*middle*), and 16 000 Mi (*bottom*). Performance is mean number of problems solved out of 7866 across 50 splits. The label of each point denotes the value of the respective regularization parameter.

References

1. Bártek, F., Chvalovský, K., Suda, M.: Regularization in spider-style strategy discovery and schedule construction (2024). https://doi.org/10.48550/arXiv.2403.12869
2. Bártek, F., Suda, M.: Vampire strategy performance measurements (2024). https://doi.org/10.5281/zenodo.10814478
3. Chvátal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3), 233–235 (1979). https://doi.org/10.1287/moor.4.3.233
4. Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.): Global Conference on Artificial Intelligence, GCAI 2015, Tbilis, 16–19 October 2015, EPiC Series in Computing, vol. 36. EasyChair (2015). https://easychair.org/publications/volume/GCAI_2015
5. Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual (2023). https://www.gurobi.com
6. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning, 2nd edn. SSS, Springer, New York (2009). https://doi.org/10.1007/978-0-387-84858-7
7. Holden, E.K., Korovin, K.: Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In: Kamareddine, F., Coen, C.S. (eds.) CICM 2021. LNCS, vol. 12833, pp. 107–123. Springer, Cham (2021).https://doi.org/10.1007/978-3-030-81097-9_8
8. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. J. Artif. Intell. Res. **36**, 267–306 (2009). https://doi.org/10.1613/JAIR.2861
9. Hula, J., Jakubuv, J., Janota, M., Kubej, L.: Targeted configuration of an SMT solver. In: Buzzard, K., Kutsia, T. (eds.) CICM 2022. LNCS, vol. 13467, pp. 256–271. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-16681-5_18
10. Jakubuv, J., et al.: MizAR 60 for Mizar 50. In: Naumowicz, A., Thiemann, R. (eds.) 14th International Conference on Interactive Theorem Proving, ITP 2023, 31 July to 4 August 2023, Białystok. LIPIcs, vol. 268, pp. 19:1–19:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.ITP.2023.19
11. Jakubuv, J., Suda, M., Urban, J.: Automated invention of strategies and term orderings for Vampire. In: Benzmüller, C., Lisetti, C.L., Theobald, M. (eds.) GCAI 2017, 3rd Global Conference on Artificial Intelligence, Miami, 18–22 October 2017. EPiC Series in Computing, vol. 50, pp. 121–133. EasyChair (2017). https://doi.org/10.29007/XGHJ
12. Jakubuv, J., Urban, J.: BliStrTune: hierarchical invention of theorem proving strategies. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, 16–17 January 2017, pp. 43–52. ACM (2017).https://doi.org/10.1145/3018610.3018619
13. Khuller, S., Moss, A., Naor, J.: The budgeted maximum coverage problem. Inf. Process. Lett. **70**(1), 39–45 (1999). https://doi.org/10.1016/S0020-0190(99)00031-9
14. Korovin, K.: iProver—an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_24
15. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1

16. Kühlwein, D., Urban, J.: MaLeS: a framework for automatic tuning of automated theorem provers. J. Autom. Reason. **55**(2), 91–116 (2015). https://doi.org/10.1007/s10817-015-9329-1
17. Lindauer, M., et al.: SMAC3: a versatile bayesian optimization package for hyperparameter optimization. J. Mach. Learn. Res. **23**, 54:1–54:9 (2022). http://jmlr.org/papers/v23/21-0888.html
18. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Inf. Process. Lett. **47**(4), 173–180 (1993). https://doi.org/10.1016/0020-0190(93)90029-9
19. Mangla, C., Holden, S.B., Paulson, L.C.: Bayesian ranking for strategy scheduling in automated theorem provers. In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022. LNCS, vol. 13385, pp. 559–577. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-10769-6_33
20. Reger, G., Suda, M., Voronkov, A.: New techniques in clausal form generation. In: Benzmüller, C., Sutcliffe, G., Rojas, R. (eds.) GCAI 2016. 2nd Global Conference on Artificial Intelligence, 19 September–2 October 2016, Berlin. EPiC Series in Computing, vol. 41, pp. 11–23. EasyChair (2016). https://doi.org/10.29007/DZFZ
21. Schäfer, S., Schulz, S.: Breeding theorem proving heuristics with genetic algorithms. In: Gottlob et al. [4], pp. 263–274. https://doi.org/10.29007/gms9
22. Schede, E., et al.: A survey of methods for automated algorithm configuration. J. Artif. Intell. Res. **75**, 425–487 (2022). https://doi.org/10.1613/jair.1.13676
23. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: Fontaine, P. (ed.) CADE 27. LNCS, vol. 11716, pp. 495–507. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_29
24. Schurr, H.: Optimal strategy schedules for everyone. In: Konev, B., Schon, C., Steen, A. (eds.) Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), Haifa, 11–12 August 2022. CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022). https://ceur-ws.org/Vol-3201/paper8.pdf
25. Seipp, J., Sievers, S., Helmert, M., Hutter, F.: Automatic configuration of sequential planning portfolios. In: Bonet, B., Koenig, S. (eds.) Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, 25–30 January 2015, Austin, pp. 3364–3370. AAAI Press (2015). https://doi.org/10.1609/AAAI.V29I1.9640
26. Suda, M.: Vampire getting noisy: will random bits help conquer chaos? (system description). EasyChair Preprint no. 7719 (2022). https://easychair.org/publications/preprint/CSVF
27. Sutcliffe, G.: The CADE ATP system competition - CASC. AI Mag. **37**(2), 99–101 (2016). https://doi.org/10.1609/AIMAG.V37I2.2620
28. Sutcliffe, G.: The TPTP problem library and associated infrastructure: from CNF to TH0, TPTP v6.4.0. J. Automat. Reason. **59**(4), 483–502 (2017). https://doi.org/10.1007/s10817-017-9407-7
29. Suttner, C.B., Sutcliffe, G.: The CADE-14 ATP system competition. J. Autom. Reason. **21**(1), 99–134 (1998). https://doi.org/10.1023/A:1006006930186
30. Tammet, T.: Gandalf. J. Autom. Reason. **18**(2), 199–204 (1997). https://doi.org/10.1023/A:1005887414560
31. Tammet, T.: Gandalf c-1.1 (1998). https://www.tptp.org/CASC/15/SystemDescriptions.html#Gandalf. Accessed 25 Jan 2023
32. Tammet, T.: Towards efficient subsumption. In: Kirchner, C., Kirchner, H. (eds.) CADE-15. LNCS, vol. 1421, pp. 427–441. Springer, Cham (1998). https://doi.org/10.1007/BFb0054276

33. Urban, J.: BliStr: The blind strategymaker. In: Gottlob et al. [4], pp. 312–331. https://doi.org/10.29007/8n7m
34. Voronkov, A.: Spider: learning in the sea of options (2023). https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833. Unpublished. Paper accepted at Vampire23: The 7th Vampire Workshop
35. Wolf, A., Letz, R.: Strategy parallelism in automated theorem proving. In: Cook, D.J. (ed.) Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference, 18–20 May 1998, Sanibel Island, pp. 142–146. AAAI Press (1998). http://www.aaai.org/Library/FLAIRS/1998/flairs98-027.php
36. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hydra: automatically configuring algorithms for portfolio-based selection. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, 11–15 July 2010. AAAI Press (2010). http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1929

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.