EPiC Series in Computing
Volume 50, 2017, Pages 121–133
GCAI 2017. 3rd Global Conference on Artificial Intelligence

Automated Invention of Strategies and Term Orderings for Vampire*

Jan Jakubův[1], Martin Suda[2], and Josef Urban[1]

[1] Czech Technical Univeristy in Prague, Prague, Czech Republic
{jakubuv,josef.urban}@gmail.com
[2] Technische Universität Wien, Vienna, Austria
msuda@forsyte.at

Abstract

In this work we significantly increase the performance of the Vampire and E automated theorem provers (ATPs) on a set of loop theory problems. This is done by developing EmpireTune, an AI system that automatically invents targeted search strategies for Vampire and E. EmpireTune extends previous strategy invention systems in several ways. We have developed support for the Vampire prover, further extended Vampire by new mechanisms for specifying term orderings, and EmpireTune can now automatically invent suitable term ordering for classes of problems. We describe the motivation behind these additions, their implementation in Vampire and EmpireTune, and evaluate the systems with very good results on the AIM (loop theory) ATP benchmark.

1 Introduction

Modern automated theorem provers (ATPs) are routinely used to attack a wide range of formally expressed problems coming from areas such as mathematics, program analysis, or verification. To achieve their performance, state-of-the-art ATP systems such as Vampire [20] and E [28] employ and combine a large number of complicated search strategies. Each strategy is typically described by many search options, sometimes using dedicated domain-specific languages for their hierarchical combinations.

Inventing targeted proof search strategies for a specific prover and specific problem sets is a difficult task, which may however be very rewarding. Several machine learning systems that invent strategies automatically for ATPs were proposed recently [26, 21], including our BliStr(Tune) system [30, 11] which uses the ParamILS [10] implementation of iterated local search. These systems were typically used on ATP problems extracted from the large libraries of today's interactive theorem prover (ITP) systems [4, 3, 5, 13, 14]. All the above systems

*Jan Jakubův and Josef Urban were supported by the *AI4REASON* ERC Consolidator grant number 649043, and by the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/0000466 and the European Regional Development Fund. Martin Suda was partially supported by ERC Starting Grant 2014 SYMCAR 639270 and by the Austrian research projects FWF S11403-N23 and S11409-N23.

```
Function BliStrLoop($\beta_{\mathrm{eval}}, \beta_{\mathrm{cutoff}}, \beta_{\mathrm{imp}}, \beta_{\mathrm{min}}, \beta_{\mathrm{max}}, \beta_{\mathrm{tops}}, \beta_{\mathrm{bests}}$)
  $All \leftarrow$ Initials;        // Initialize current generation to the initial strategies
  loop
    Evaluate($All$, $\beta_{\mathrm{eval}}, \beta_{\mathrm{min}}, \beta_{\mathrm{max}}$);   // Evaluate all strategies with timeout $\beta_{\mathrm{eval}}$ (1)
    $G \leftarrow$ Reduce($All$, $\beta_{\mathrm{tops}}, \beta_{\mathrm{bests}}$);   // Select only $\beta_{\mathrm{tops}}$ best-performing strategies (2)
    $S \leftarrow$ Select($G$);   // Select a strategy to improve (3)
    if $S$ is undefined then return $G$;   // Terminate when all strategies are improved
    $S' \leftarrow$ Improve($S, \beta_{\mathrm{cutoff}}, \beta_{\mathrm{imp}}$);   // Improve $S$ on best-performing problems (4)
    $All \leftarrow All \cup \{S'\}$;   // Extend current generation
  end
```

Figure 1: An outline of the BliStr strategy invention loop.

have so far targeted the E prover, and none of the systems has so far focused on automated invention of suitable term orderings, which is an extremely important part of the proof search.

In this work we develop EmpireTune, a system that invents strategies both for E and Vampire. Vampire has been for a long time the strongest ATP system, consistently having best performance in CASC (the CADE Automated System Competition). To be able to automatically invent new term orderings, we extend Vampire by several new options and mechanisms allowing both fixed and weighted ordering schemes. Several other changes are implemented, which primarily allow us to extend the strategy-invention loop to ATP architectures that are (like Vampire's recent AVATAR [31, 23]) not purely saturation based. For the evaluation we choose an ATP benchmark where rewriting (and thus good term orderings) is very important. This is the AIM benchmark used in the CASC 2016 ATP competition [29], consisting of 1020 training and 200 evaluation problems coming from a large theorem proving project in loop theory [16].

The rest of the paper is organized as follows. Section 2 discusses the invention of proof search strategies, focusing on our BliStr architecture and its EmpireTune extensions. Section 3 describes the Vampire proof strategies in general, and Section 4 focuses specifically on term orderings, their Vampire implementation and the extensions needed for their automated invention. The system is evaluated in several ways in Section 5, showing very significant improvements of both Vampire and E on the AIM benchmark. Section 6 describes the EmpireTune configuration parameters together with a practical guidelines on their initial setting, and Section 7 concludes.

2 Invention of Theorem Proving Strategies

The BliStr strategy invention loop [30] improves a given ATP system with given initial strategies (Initials) on a given set of training problems (Problems). This is done by gradually evolving new strategies specialized for classes of the training problems. The specialization of a strategy on a specific subset of problems is done by the ParamILS [10] automated algorithm configuration framework. ParamILS is based on iterated local search in parameter configuration space and hence requires the space of ATP strategies to be represented in the ParamILS syntax. Possible representations have been previously described for E Prover [30, 11] and in this work we present Vampire representation (Section 3).

Figure 1 provides an outline of the BliStr strategy invention loop which is common for all strategy invention methods in the BliStr family (BliStr [30], BliStrTune [11], and EmpireTune introduced in this work). The loop consists of four basic steps whose implementation might differ in a specific method. Because no strategy can be improved more than once on the same set of problems (see Step 3) and the set of all possible strategies is typically finite, the loop must eventually terminate.[1] A more detailed explanation of the four basic steps follows.

Step 1: Generation Evaluation. In the first phase, all strategies (*All*) are evaluated on all training problems Problems. The ATP is run on each problem with time limit β_{eval} yielding (1) the overall result (solved/unsolved) and (2) a number measuring the length of the proof search. In EmpireTune the proof search length measured in millions of instructions is newly used as an effective measure of the proof search performance, whereas the previous versions of BliStr used the number of given clauses processed by E Prover. This change is needed to accommodate different proof search approaches that might not be purely saturation-based. This is particularly important for Vampire's new AVATAR architecture, which tightly integrates a superposition theorem prover with a SAT or an SMT solver. In more detail, we now consider the ATP-independent measure of CPU instructions, using the `perf` Linux tool.[2]

For each strategy S, we compute its set of *best-performing problems* P_S ($\subseteq$ Problems), that is, all the problems where S outperforms all other strategies. The set P_S is further restricted to contain only the problems provable with the proof search length ranging between β_{min} and β_{max}. This is to later avoid improving S on problems which are "too easy" (because there is not much to improve) and which are "too hard" (to speed up the learning process).

Step 2: Generation Reduction. In the next step, we reduce the strategies invented so far (*All*) to contain only the strategies S with at least β_{bests} best-performing problems (that is, with $|P_S| \geq \beta_{\text{bests}}$). From the remaining strategies, we take only β_{tops} best strategies, where the strategies are compared by the number of best-performing problems ($|P_S|$). The first restriction keeps only the best-performing strategies (that is, the strongest individuals), while the second reduces their count keeping the size of G within the selected bound ($|G| \leq \beta_{\text{tops}}$). This is done to prevent overfitting on the training problems by having a large number of overspecialized strategies.

Step 3: Strategy Selection. The next step is to select a strategy to be improved on its best-performing problems. As a rule, no strategy can be improved on the same problems more than once within one execution of the BliStrLoop function. Because the sets of best-performing problems vary in time, the same strategy can be improved more than once but only on different problems. Our selection approach is to prefer improving strategies on diverse problems. Hence we prefer to improve strategies whose best-performing problems have not been used for improving so often.[3] If no strategy can be selected, the algorithm terminates with the current generation G as the result.

Step 4: Strategy Improvement. The strategy improvement is done by the ParamILS [10] automated algorithm configuration framework. Given a strategy S and a set of problems

[1] Note however that the set of all possible strategies is usually astronomically large, and relatively fast termination is due to more complicated factors influenced by the BliStr settings.

[2] `Perf` is a performance analysis tool for Linux, used to count the number of CPU instructions. A precise user-space counting ("`perf stat -e instructions:up`") rounded up to millions provides a reliable and deterministic measure for a particular machine.

[3] In more detail, for each problem p, we keep a counter c_p which is increased by $1/|P_{S_0}|$ whenever a strategy S_0 is improved on p (that is, when $p \in P_{S_0}$). We select the strategy S with (currently) the lowest average c_p over the best-performing problems P_S. In the case of equal values, we prefer the strategy with higher $|P_S|$.

P ($\subseteq$ PROBLEMS), PARAMILS attempts to find an ATP strategy S' with possibly the best performance on P. ParamILS employs an *iterated local search* (ILS) from the initial strategy S, occasionally perturbing a strategy to escape from a local optimum. ParamILS is a generic framework which is capable of finding well performing configurations for an arbitrary algorithm. In our case, the algorithm is an ATP with a time evaluation limit β_{cutoff}.

In order to employ ParamILS, the space of ATP strategies must be described in the ParamILS syntax, yielding the ParamILS *configuration space* Θ. The configuration space Θ is described by a finite set of parameters where each parameter is assigned a finite domain of possible values. A *configuration* $\theta \in \Theta$ is then a finite assignment of specific values to all parameters from space Θ. ParamILS additionally allows to specify *conditional arguments* and *forbidden values* (see [10] for details). ParamILS also needs to be provided with a *performance metric* which estimates the quality of individual ATP runs. In EmpireTune we use the same metric as in Step 1, i.e., millions of CPU instructions.

We always launch ParamILS to improve a strategy S on its currently best-performing problems P_S. Either a single run of ParamILS can be used, as in BliStr [30], or, alternatively, several ParamILS runs can be combined to improve a strategy in a hierarchical manner, as in BliStrTune [11]. In both cases, ParamILS is always launched for a specific time limit, which is provided as the input parameter β_{imp}.

3 Inventing Strategies for Vampire

In order to use an ATP like E or Vampire within the BliStr loop, the space of ATP's proof search strategies must be described as a ParamILS configuration space Θ, as explained in Section 2. The configuration space does not necessarily need to cover all possible ATP strategies and, in practice, it usually covers only a subset. This subset, however, determines strategies that can possibly be invented by the BliStr loop. Hence this subset should be reasonably large. A subset of E Prover strategies has been previously described as ParamILS configuration space and used within the BliStr loop [30, 11]. In this section, we show how to describe a subset of Vampire strategies, carefully selected by a human expert.

The most recent version of Vampire (version 4.2) provides more than one hundred options for setting up a particular proof search strategy. Most of the options are either boolean or finite multi-valued, but many have also integer or even floating point domains. Vampire's CASC mode, a portfolio mode of preselected strategies designed to score well on a wide range or problems, makes active use of around half of these options. For our experiment, we picked a subset of the 32 most important options. We selected a small representative set of values for those options which have in principle unbounded domains. Also, we made use of ParamILS' conditional arguments and forbidden values to block combinations of options which are incompatible due to dependencies. This way we obtained a ParamILS configuration space for Vampire Θ_V.

Let us give a brief overview of the options used for defining Θ_V.[4]

1. There are several options influencing *preprocessing* of the input problem and related to normal form transformation [24].

2. A few options define whether and how the *Sine axiom selection* scheme [9] is used.

3. The next and probably the most important category is related to the *saturation algorithm*. Vampire implements the standard Otter and Discount loops, along with the limited resource strategy (LRS) [25]. Independently, the InstGen [18] paradigm can be used instead

[4] For a full list check our implementation at http://github.com/ai4reason/EmpireTune.

of these. Further options belonging to this category are the age-weight ratio option determining *clause selection*, a multi-valued option for controlling *literal selection* [8], and an option `sp` for determining the symbol precedence and thus the *term ordering* used during saturation.

4. There are options for enabling and tweaking the AVATAR architecture [31, 23].

5. Finally, there are options for enabling and disabling a set of various inference rules and redundancy checks, such as extensionality resolution [7], (forward and backward) subsumption and demodulation, and several others.

It should be noted that many option configurations render Vampire incomplete. However, this is always detected so the prover never reports a wrong result. (Instead, the prover reports "Unknown" in the case of finite saturation.) Moreover, incomplete strategies have been shown invaluable for solving many problems which cannot be solved otherwise.

4 Strategies and Term Orderings

4.1 Term Orderings and Their Importance in Proof Search

Term orderings are a well-known device used for restricting and guiding proof search of first-order automated theorem provers (ATPs). In more detail, a term ordering parametrises the resolution and the superposition calculi [1, 22] employed by a prover, taking part in determining which exact inferences should be performed. The two most commonly used classes of orderings are the Lexicographic Path Ordering (LPO) [15] and the Knuth-Bendix Ordering (KBO) [17]. To get a concrete instance of an ordering from either class one needs to specify a symbol *precedence*, a total order on the predicate and function symbols occurring in the given problem.[5]

It is well known that the choice of the ordering can have a huge impact on the success or failure of the subsequent proof attempt. For example, giving a high precedence to symbols introduced during clausification as names of sub-formulas [24], will effectively lead to their immediate elimination during saturation and thus give rise to an exponential clause set. However, there is no obvious general way for choosing a good ordering in advance and ATPs typically provide only a few *heuristic schemes* for the automatic selection of the precedence.[6] Note that this leaves the majority of the $n!$ possibilities of how to choose a precedence (for a problem with a signature of size n) inaccessible to a typical theorem proving strategy. The alternative of allowing to *explicitly specify* a full precedence out of the $n!$ possibilities is, nevertheless, only rarely used, because each problem comes, in principle, with its own signature which means that precedences are problem-specific and the ensuing strategy cannot be easily transferred.

In this section, we propose and evaluate a new approach for exploring the space of possible term orderings. We extended Vampire to allow specifying linear combinations of the previously available heuristic schemes (see Section 4.3)). Moreover, we allow symbols to be assigned user-specified weights, by which the precedence is in the end sorted. We evaluate the new approach in a simplified setting in which all problems share the same signature. This allows us to ask the question whether orderings can be found which work well for a whole set of related problems such as those describing related conjectures within the context of a fixed theory.

[5] To fully determine a KBO, one also needs to specify an admissible weight function.
[6] Cf. the option `sp` mentioned before.

4.2 General Framework for Ordering Invention

We have proposed and developed the following general modification of the BliStr loop for inventing useful orderings in EmpireTune.

In Step 4 of the BliStrLoop (Section 2) a strategy S is improved on its best-performing problems P_S. Our previous experience with designing and parameterising complex E strategies [12, 11] has shown that running a single ParamILS instance to improve S becomes inefficient when the set of possible strategies is too large. Since the additional parameter space needed for term-ordering invention is in general going to be of nontrivial size, we employ a hierarchical invention similar to BliStrTune [11]. The improvement is done in two phases:

1. In the first phase, ParamILS is launched to improve S on P_S, but it is not allowed to change term ordering specified by S. Only the elements of S which are not specific to term ordering can be updated. The result of the first phase is a strategy S' which specifies the same term ordering as S, but in which other parts may differ.

2. In the second phase, ParamILS is launched to improve S' on P_S, but this time only term ordering settings might be adjusted. The result of the second phase is a strategy S'' which shares with S' the details unrelated to term ordering, but the term ordering setting might differ. Strategy S'' is returned as the overall result of the strategy improvement step (Step 4 of the BliStrLoop).

4.3 Vampire Term Orderings and Their Invention

The term ordering implemented in Vampire is the standard Knuth-Bendix Ordering, optionally extended to the transfinite case using predicate levels [19]. Version 4.2 fixes variable and symbol *weight* to 1 and offers an option (`sp`) for computing symbol *precedence* by one of the following schemes:

occurrence By default, precedence orders symbols by the order of their occurrence in the input problem.

arity Symbols are ordered by their arity.

frequency Symbols are ordered by the number of occurrences in the input.[7]

There are also "reverse" versions for the latter two values, meaning that both ascending and descending order is available.

To allow for a more refined tuning of the precedence, we replace the multi-valued option `sp` by three integer options `spoc`, `spac`, `spfc` to work as coefficients for mixing contributions from "occurrence", "arity" and "frequency", respectively. In this new setting, the precedence is constructed as follows. First, Vampire computes the precedences $P_{occurrence}$ and $P_{frequency}$ that could have been used in the original setting. Notice that these can be understood as mappings, assigning each symbol s from the signature Σ values from the range $0, \dots, n-1$ where $n = |\Sigma|$ is the size of the signature. Then, each symbol $s \in \Sigma$ is assigned a final value

$$val(s) = \texttt{spoc} \cdot P_{occurrence}(s) + \texttt{spac} \cdot arity(s) + \texttt{spfc} \cdot P_{frequency}(s), \tag{1}$$

where $arity(s)$ is the arity of s. The final precedence is then obtained by sorting the symbols of Σ by the final value $val(s)$. This gives us the expressive power to define the originally available

[7] This scheme has been a recent valuable addition adopted from E Prover [27].

precedences,[8] but also many others obtained by linear combination of the contribution values. Note that the contributions of occurrence and frequency are "normalised" by using the values from the respective precedences, rather than the actual values. We keep arity intact, as its values are already small non-negative integers.

To allow for explicitly setting an arbitrary precedence, we add one more numeric option coefficient `spuc`, which stands for "user". Additionally, there is an option for explicitly assigning user values $uv(s)$ to every symbol.[9] The sum (1) is extended with an additional term taking into account the user coefficient and user values. We are aware that the approach with symbol values and sub-sequent sorting as a means for obtaining a precedence actually gives us more degrees of freedom than would be strictly needed (specifically, there is n^n meaningful assignments, but only $n!$ precedences). We believe, however, that this gives rise to a setup better suited for parameter tuning. In particular, we can expect local changes in a value to have a local influence on the final precedence. This is in contrast with explicitly navigating the space of all permutations, which seems to lack this natural notion of "continuity".

5 Experimental Evaluation of EmpireTune

We evaluate EmpireTune[10] on the CASC 2016 AIM benchmark [29]. AIM is a long-term large-scale project [16] in applied automated deduction concerned with proving open algebraic (loop theory) conjectures by Kinyon and Veroff. In the CASC 2016 competition, 1020 problems (and their Prover9 proofs) were provided for the training of ATPs, while additional 200 problems were used for the evaluation. The AIM problems have a fixed signature with only 10 symbols, which is useful for the evaluation of our term ordering invention methods.

We first (Section 5.2) evaluate EmpireTune on AIM both for E and Vampire without the invention of term orderings introduced in Section 4. Then (Section 5.3) we evaluate the additional impact of invention of new term orderings in Vampire. While EmpireTune can be used to invent strategies for E and Vampire for an arbitrary benchmark set, our method for term ordering invention (Section 4) has so far been implemented only for Vampire and only for the fixed AIM problems signature.

5.1 Common Evaluation Setup

The evaluations in both sections share a common setup. We take 820 problems out of the 1020 CASC training problems and we use them as the training problems for EmpireTune (the set Problems from Section 2). The remaining 200 CASC training problems are kept for a later independent cross-validation of the invented strategies. As the initial strategies (set Inits), we take 10 well-performing strategies extracted from the CASC mode in the case of Vampire, and from the auto-schedule mode in the case of E. EmpireTune must be currently run separately for E and separately for Vampire. In the E Prover mode, EmpireTune currently works similarly to BliStrTune [11], however using our new evaluation criteria (see Section 2). For each prover mode (E or Vampire) we run several EmpireTune instances with various parameter settings (parameters β_x). The selection of the EmpireTune parameters is briefly discussed in Section 6. Although an EmpireTune run can be terminated after a specific time (for example, after one

[8] For instance, the reversed frequency precedence is obtained by setting `spoc` $= 0$, `spac` $= 0$, and `spfc` $= -1$.

[9] Symbols introduced during preprocessing, namely the Skolem functions and sub-formula names, have their own meta-identifier and can thus be given one (shared) value each.

[10] Our implementation is available at `http://github.com/ai4reason/EmpireTune`. The strategies developed in the experiments described here are at `http://forsyte.at/static/people/suda/gcai-protos.tar.gz`.

ATP	*runs*	*tune time*	*invented*	*greedy*	**solved**	V+	E+
E 2.0 (EmpireTune)	*4*	*15.4 days*	*1010*	*15*	**74**	+64%	+139%
Vampire 4.2 (EmpireTune)	*16*	*12.8 days*	*1597*	*7*	**52**	+16%	+68%
Vampire 4.2 (CASC mode)	-	-	-	-	**45**	+0%	+45%
E 2.0 (auto-schedule)	-	-	-	-	**31**	-31%	+0%

Table 1: Evaluation of the best EmpireTune strategies on 200 AIM testing problems with 300 seconds overall time limit per problem.

day), in these experiments we keep all the runs running until they terminate. As the final set of strategies we do not use only the strategies in the final generation (the returned G) but collect all the strategies invented during all the EmpireTune runs. As this can give us hundreds of strategies, we use for final strategy selection cross-validation on the 200 independent problems which were not used for training but were kept aside previously.

The purpose of cross-validation is to limit the number of invented strategies. We evaluate all the invented strategies on all 200 cross-validation problems with 10 seconds time limit. From the results, we greedily compute a subset of strategies which is enough to cover all the solved problems as follows. We start with the set P of 200 cross-validation problems. We select the strategy which solves the most problems from P and we remove them from P. This process is iterated as long as there is at least one strategy solving some problems from P. The process of greedy coverage is described in more details in our previous work [11] and typically leaves us with dozens of strategies (depending on the size of set PROBLEMS and other factors).

Finally, we evaluate the strategies from the greedy cover on the 200 CASC testing problems with an overall 300 seconds time limit per problem. This overall time limit is evenly divided between the strategies from the greedy coverage. This allows us to directly compare the number of solved problems with the automated modes of E and Vampire run also for 300 seconds.

5.2 Evaluation of EmpireTune Strategies

Results of the evaluation of strategies invented by EmpireTune for E and Vampire are presented in Table 1. The best invented E and Vampire strategies are shown in Appendix A. The column *runs* shows the number of different EmpireTune runs (where applicable), the column *tune time* shows the sum of runtimes of all EmpireTune runs in days, the column *invented* presents the total number of strategies invented by all the runs, and the column *greedy* shows the number of strategies in the greedy coverage described above. Finally, the column *solved* shows the number of AIM test problems solved, and the column *V+* (or the column *E+* respectively) shows the gain in percents on Vampire's CASC mode (or on E's auto-schedule mode respectively). The table contains four rows for the four ATP systems, that is, for (1) E with strategies invented by EmpireTune, (2) Vampire with strategies invented by EmpireTune. (3) Vampire's CASC mode, and (4) E's auto-schedule mode. All ATPs were given 300 seconds time limit per problem and we are using Vampire version 4.2 and E version 2.0.

We can see that both EmpireTune versions outperform their respective built-in automated modes. From this we can conclude that both E and Vampire largely benefit from invention of targeted strategies on the AIM problems. E Prover, however, produces better results and thus seems to be more suitable for strategy invention. This should be most likely attributed to the size of the strategy space, because E provides a richer language to parametrize strategies. Furthermore, in our current implementation, an average size of the ParamILS configuration space for E is around 10^{50} while for Vampire it is only 10^{16}. This is partly due to our previous attention to E and partly because of the richness of E's native configuration space. The ParamILS configuration space size is reflected in longer average run times of EmpireTune runs (3.9 days for E while only 19.2 hours for Vampire). It should be noted that different EmpireTune runs can be parallelized and thus the results can be produced faster than indicated in the *tune time* column. The union of the two EmpireTune runs solves 77 problems altogether and these 77 problems cover all the problems solved by the automated modes of both E and Vampire. The best of the EmpireTune runs for Vampire was able to solve 375 training problems while the best run for E solved 464 training problems. It seems that EmpireTune produces better strategies for E than for Vampire. This might be related to the configuration space size mentioned above. The Vampire configuration space might be extended in the future version of EmpireTune, however, E simply provides more options for user configuration than Vampire.

The remarkable improvement of E's performance should be considered specific to the selected AIM benchmark problem set. Our previous results on MPTP benchmarks [11] also outperformed Vampire's CASC mode, but not so dramatically (+5% improvement). Both E's and Vampire's automated modes are being constantly developed by their maintainers and tuned for different problem sets. It might well be the case that these automated modes were not yet specialized for AIM (or similar) problems. This, however, increases the importance of EmpireTune which gives the user an opportunity to develop their own targeted strategies, without reliance on external sources.

It is interesting to note that the seven Vampire strategies selected by EmpireTune for the AIM problems do not make use of the whole repertoire of available option values. This can be explained by the special form of the AIM benchmarks. It seems plausible that a specific benchmark set requires specific techniques to be solved efficiently. For example, because the AIM problems are unit equality problem, splittable clauses never occur during derivation and thus it does not make sense to employ splitting. Indeed, none of the seven strategies enable the AVATAR mode. Furthermore, five of these strategies use the Otter saturation algorithm (used also by Prover9) which is not deemed to be superior in general. The remaining two use LRS, thus Discount is never used. Also, the InstGen, which is known not to be well suited for equational problems is not used. Besides, all the seven strategies agree on the following values for the following options: "backward demodulation" (preordered), "backward subsumption" (off), "backward subsumption resolution" (off), "forward subsumption" (on), "forward subsumption resolution" (off). This fact, however, does not seem to have a simple explanation.

5.3 Experiments with Term Orderings

The impact of the tuning of term orderings (described in Section 4) is presented in Table 2. The table has the same columns as Table 1 described in the previous section. We compare EmpireTune using term ordering tuning against EmpireTune used without term ordering tuning (that is, with the runs from Section 5.2) and against Vampire's CASC mode. As mentioned above, the tuning of term orderings is currently implemented only for Vampire and only for the fixed signature of AIM problems.

We can see that the tuning of term orderings has significantly increased an average EmpireTune runtime from 19.2 hours to 2.3 days. This is because of the additional term ordering tuning phase. Moreover, we can see that the version with the term ordering tuning invented more strategies but fewer strategies were actually needed for the greedy coverage. From the fact that a smaller number of strategies solved more problems, we can conclude that EmpireTune

ATP	*runs*	*tune time*	*invented*	*greedy*	**solved**	V+
EmpireTune (with ordering tuning)	*7*	*16.3 days*	*1651*	*6*	**57**	+27%
EmpireTune (no ordering tuning)	*16*	*12.8 days*	*1597*	*7*	**52**	+16%
Vampire 4.2 (CASC mode)	-	-	-	-	**45**	+0%

Table 2: Evaluation of the impact of EmpireTune term ordering tuning for Vampire.

with term ordering tuning produced more universal, and hence better, strategies. Finally, the version with term ordering tuning provided an additional valuable improvement of 11% over the Vampire's CASC mode. This means that the invention of term orderings is a promising area of future research. The best of the EmpireTune runs for Vampire with ordering tuning was able to solve 414 training problems, compared to 375 problems solved without ordering tuning.

6 Tuning EmpireTune Configuration

This section briefly describes the EmpireTune configuration parameters together with a practical guidelines on their initial setting. The parameters can be split into the following three groups. (1) Parameters influencing the selection of best-performing problems (β_{min}, β_{max}). (2) Parameters influencing generation reduction (β_{tops} β_{bests}). (3) Time limits for strategy evaluation and improvement (β_{eval}, β_{cutoff}, β_{imp}). In practice, we usually run several testing runs, fixing all but one parameter, to determine an optimal value for each parameter.

Parameters β_{min} and β_{max} are used to limit the set P_S of best-performing problems of strategy S only to the problems which are neither too easy nor too hard. The number of instructions executed to solve a problem must be bounded by these parameters. The initial setting of β_{min} and β_{max} can be established from the evaluation of initial strategies on the training problems. We usually set the parameter span to cover 80% of problems solved by the initial strategies.

Parameters β_{tops} and β_{bests} influence the size of the current generation G and the size of individuals (that is, strategies). The size of G is required to be limited by β_{tops} ($|G| \leq \beta_{\text{tops}}$) while only strategies S which are best-performing on at least β_{bests} problems ($|P_S| \geq \beta_{\text{bests}}$) are allowed in G. Increasing β_{tops} or decreasing β_{bests} increases runtime because there are more strategies to improve. However, longer runtime does not necessarily mean better results. It seems that there is an optimal ratio between the size of the generation and the size of individuals. We usually fix one parameter and run several evaluation EmpireTune runs with different values for the second parameter to find out the best combination.

Parameters β_{eval} and β_{cutoff} set time limits for running ATPs in the evaluation and strategy improvement phase respectively. We set β_{eval} to a higher value than β_{cutoff}, usually $\beta_{\text{eval}} = 5$ and $\beta_{\text{cutoff}} = 1$ seconds. Parameter β_{imp} sets the time limit for running a single ParamILS run in the strategy improvement phase. In BliStr and BliStrTune, there is a fixed time limit for each ParamILS run, which is independent on the size of P_S (we use either 100 or 300 seconds). In practice, the size of P_S ranges from dozens to hundreds, hence it would make sense to adjust the time limit for ParamILS accordingly to the size of P_S.

Recently, we have developed an automated selection mode for parameter β_{imp}. In this mode, several ParamILS instances are launched in parallel (at least two instances are required). During ParamILS runs, each instance keeps reporting its progress in the form of a triple (θ, Q, n), where θ is the best configuration found so far, Q is the quality of θ where Q is based on evaluation of θ on n problems. The number n only increases in time as new results are acquired. In our automated mode for β_{imp}, we keep checking this progress reports, and wait for the first ParamILS instance to report a configuration θ_1 evaluated on all P_S problems (that is, $n = |P_S|$). Then we keep only this instance and restart all the other instances but from initial state θ_1. This process is repeated as long as the best quality Q keeps improving. In our implementation, this experimental mode is turned on by setting $\beta_{\text{imp}} = 0$.

7 Conclusions and Future Work

In this work we have started to automatically invent targeted strategies for Vampire, and to automatically invent new Vampire term orderings for classes of problems. We have focused on the AIM benchmark coming from loop theory, where the signature is small and allows us to explore various methods for term ordering invention. The overall result is a very significant improvement of the performance of E and Vampire on the AIM problems, making them much more useful for the AIM project.

The first results show that automated invention of term orderings indeed leads to stronger strategies that better generalize to unknown testing problems. We have also shown that the larger parameter space available for E prover makes a significant difference in comparison to Vampire, where the training phase converges much faster, but in general with worse strategies. The general enhancements needed for Vampire now allow any ATP system to be included in the EmpireTune, regardless of its particular ATP proof search methods.

Future work includes further extension of the term ordering invention to large signatures, arising e.g. in the problems coming from the ITP libraries. Another important direction particularly useful for the AIM project would be addition of strategy invention for Prover9 and its powerful search exploration methods such as proof sketches and proof hints.

References

[1] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[2] Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors. *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *LNCS*. Springer, 2011.

[3] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016.

[4] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

[5] Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for HOL4. In *Certified Programs and Proofs (CPP'15)*, LNCS. Springer, 2015. http://dx.doi.org/10.1145/2676724.2693173.

[6] 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*. EasyChair, 2015.

[7] Ashutosh Gupta, Laura Kovács, Bernhard Kragl, and Andrei Voronkov. Extensional crisis and proving identity. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia,*

November 3-7, 2014, Proceedings, volume 8837 of *Lecture Notes in Computer Science*, pages 185–200. Springer, 2014.

[8] Kryštof Hoder, Giles Reger, Martin Suda, and Andrei Voronkov. Selecting the selection. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 – July 2, 2016, Proceedings*, pages 313–329, Cham, 2016. Springer International Publishing.

[9] Krystof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Bjørner and Sofronie-Stokkermans [2], pages 299–314.

[10] 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.

[11] 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.

[12] Jan Jakubuv and Josef Urban. Extending E prover with similarity based clause selection strategies. In Michael Kohlhase, Moa Johansson, Bruce R. Miller, Leonardo de Moura, and Frank Wm. Tompa, editors, *Intelligent Computer Mathematics - 9th International Conference, CICM 2016, Bialystok, Poland, July 25-29, 2016, Proceedings*, volume 9791 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2016.

[13] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.

[14] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.

[15] S. Kamin and J.-J. Levy. Two generalizations of the recursive path ordering. Departement of Computer Science, University of Illinois, Urbana, IL, 1980.

[16] Michael Kinyon, Robert Veroff, and Petr Vojtěchovský. *Loops with Abelian Inner Mapping Groups: An Application of Automated Deduction*, pages 151–164. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[17] D. E. Knuth and P. B. Bendix. *Simple Word Problems in Universal Algebras*, pages 342–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[18] K. Korovin. Instantiation-based automated reasoning: From theory to practice. In R. A. Schmidt, editor, *22nd International Conference on Automated Deduction CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 163–166. Springer, 2009.

[19] Laura Kovács, Georg Moser, and Andrei Voronkov. On transfinite Knuth-Bendix orders. In Bjørner and Sofronie-Stokkermans [2], pages 384–399.

[20] 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.

[21] Daniel Kühlwein and Josef Urban. MaLeS: A framework for automatic tuning of automated theorem provers. *J. Autom. Reasoning*, 55(2):91–116, 2015.

[22] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[23] Giles Reger, Martin Suda, and Andrei Voronkov. Playing with AVATAR. In P. Amy Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 399–415, Cham, 2015. Springer International Publishing.

[24] Giles Reger, Martin Suda, and Andrei Voronkov. New techniques in clausal form generation. In Christoph Benzmüller, Geoff Sutcliffe, and Raul Rojas, editors, *GCAI 2016*, volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016.

[25] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003.

[26] Simon Schäfer and Stephan Schulz. Breeding theorem proving heuristics with genetic algorithms. In Gottlob et al. [6], pages 263–274.

[27] Stephan Schulz. E 2.0, user manual. `http://wwwlehre.dhbw-stuttgart.de/~sschulz/WORK/E_DOWNLOAD/V_2.0/eprover.pdf`. Accessed: 2017-07-20.

[28] 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.

[29] Geoff Sutcliffe. The 8th IJCAR automated theorem proving system competition - CASC-J8. *AI Commun.*, 29(5):607–619, 2016.

[30] Josef Urban. BliStr: The Blind Strategymaker. In Gottlob et al. [6], pages 312–319.

[31] Andrei Voronkov. AVATAR: The architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer International Publishing, 2014.

A Best obtained strategies

A.1 E Prover

```
--definitional-cnf=24 --split-aggressive --split-clauses=7 --forward-context-sr
 --destructive-er-aggressive --destructive-er --prefer-initial-clauses -tKBO6
 -Garity -F1 --delete-bad-limit=150000000 -WSelectMaxLComplexAvoidPosPred
 -H'(1*ConjectureSymbolWeight(PreferUnitGroundGoals,300,400,400,3,-1,4,3,0.2),
 1*StaggeredWeight(SimulateSOS,1),21*Refinedweight(SimulateSOS,200,-1,9999.9,1.5,0.2),
 8*ConjectureSymbolWeight(DeferSOS,10,18,10,200,2,3,0.1,0.1))'
```

A.2 Vampire without ordering

```
-av off -awr 2 -bce on -bd preordered -bs off -bsr off -cond off -drc off
 -erd input_only -fd all -fde unused -fs on -fsr off -gs off -gsp off -ins 8
 -lcm reverse -nm 64 -s -4 -sa otter -sas minisat -sd 0 -sgt 4 -sos off
 -sp reverse_frequency -ss axioms -st 2 -updr off -urr off
```

A.3 Vampire with ordering

```
-av off -bce on -bd preordered -drc off -fd preordered -fde unused
 -fp '"identity 0 6#multiply 2 9#associator 3 5#commutator 2 6#left_division 2 1#
 right_division 2 7#left_inner_mapping 3 8#right_inner_mapping 3 8#
 middle_inner_mapping 2 4# sK* 0 10"'
 -fsr off -nm 26 -s 1004 -sa otter -sas z3 -spoc 0 -spuc 1 -updr off -urr on
```