



# Lemma Discovery and Strategies for Automated Induction

Sólrún Halla Einarisdóttir<sup>1</sup>, Márton Hajdu<sup>2</sup>, Moa Johansson<sup>1</sup>,  
Nicholas Smallbone<sup>1</sup>, and Martin Suda<sup>3</sup>

<sup>1</sup> Chalmers University of Technology, Gothenburg, Sweden

{slrn,jomoa,nicsma}@chalmers.se

<sup>2</sup> TU Wien, Vienna, Austria

marton.hajdu@tuwien.ac.at

<sup>3</sup> Czech Technical University in Prague, Prague, Czech Republic  
martin.suda@cvut.cz

**Abstract.** We investigate how the automated inductive proof capabilities of the first-order prover Vampire can be improved by adding lemmas conjectured by the QuickSpec theory exploration system and by training strategy schedules specialized for inductive proofs. We find that adding lemmas improves performance (measured in number of proofs found for benchmark problems) by 40% compared to Vampire’s plain structural induction as baseline. Strategy training alone increases the number of proofs found by 130%, and the two methods in combination provide an increase of 183%. By combining strategy training and lemma discovery we can prove more inductive benchmarks than previous state-of-the-art inductive proof systems (HipSpec and CVC4).

**Keywords:** Induction · Theory Exploration · Lemma Discovery · Strategies · Vampire

## 1 Introduction

We have experimented with augmenting Vampire’s capabilities for induction by injecting extra lemmas suggested by the theory exploration system QuickSpec [25] and by training strategy schedules specialized for inductive proofs. Our aim is to improve on the state of the art in automating proofs by induction.

Proofs by induction provide a challenge for automated theorem provers. Not only are there typically many choices of which induction scheme to use, but a proof may also require the conjecture to be generalized to strengthen the inductive hypothesis, or require additional auxiliary lemmas, themselves needing another induction to prove. For example, suppose we have a recursively defined function *rev* for reversing lists, defined using the append function *++*:

$$\begin{aligned} \text{rev} [] &= [] \\ \text{rev} (x : xs) &= (\text{rev } xs) ++ (x : []) \end{aligned}$$

where  $++$  is defined as follows:

$$\begin{aligned} [] ++ xs &= xs \\ (x : xs) ++ ys &= x : (xs ++ ys) \end{aligned}$$

and want to prove that  $\text{rev}(\text{rev}(xs)) = x$  for any list  $xs$ . When we ask Vampire to find a proof of this using structural induction it is unable to find a proof, even when given a long time. The induction hypothesis  $\text{rev}(\text{rev}(xs)) = xs$  is not strong enough to prove that  $\text{rev}(\text{rev}(x : xs)) = x : xs$ : we are missing some *lemmas*.

QuickSpec [25] is a system that produces equational conjectures from function definitions. Suppose we use QuickSpec to conjecture some lemmas about the  $\text{rev}$  and  $++$  functions. In under 1.5 s (running on a regular laptop<sup>1</sup>) QuickSpec outputs the following 9 equations as unproved conjectures:

1.  $\text{rev} [] = []$
2.  $x ++ [] = x$
3.  $[] ++ x = x$
4.  $\text{rev}(\text{rev } x) = x$
5.  $\text{rev}(x : []) = x : []$
6.  $(x ++ y) ++ z = x ++ (y ++ z)$
7.  $x : (y ++ z) = (x : y) ++ z$
8.  $\text{rev } x ++ \text{rev } y = \text{rev}(y ++ x)$
9.  $(xs ++ (y : (z : []))) = \text{rev}(z : (x : (\text{rev } xs)))$

Now suppose we add these equations to the input we give to Vampire, marking them as conjectured lemmas. Vampire may use such lemma in a proof, but only if it also proves it (e.g. by induction). Vampire instantly (in 6 ms, running on the same laptop) finds a proof of the original property, using (2), (6), and (8) above as lemmas, as well as proofs for the lemmas that were used. A closer investigation shows that only (6) and (8) are necessary to find a proof, where (8) is used in the proof of the original goal and (6) is used to prove (8).

Coming up with lemmas is a non-trivial task, and has sparked research into various lemma discovery techniques (see [17] for an overview). Lemma discovery can broadly be divided into two categories: *Top-down* techniques include attempting to generalize the current subgoal, or analyzing failed proof attempts to suggest a missing lemma. *Bottom-up* techniques focus on discovering potentially interesting lemmas about the definitions and concepts available, without considering any particular ongoing proof attempts. Bottom-up techniques can find a wider class of lemmas, but have the disadvantage that the system spends time working with conjectures that are not relevant to the goal. For example, the earlier system HipSpec [5] would first run QuickSpec (just as in the example above) but then attempt to prove *all* discovered conjectures before working on the main goal.

<sup>1</sup> The same laptop the experiments in Sect. 4 were run on, see more precise description there.

In this work, we use theory exploration, a bottom-up technique, in a more goal-directed manner. We use QuickSpec to suggest useful lemmas, but we will not prove *all* the suggestions, only those that are useful in the proof of the main goal. To do this we leverage Vampire’s AVATAR architecture [20, 29], which allows us to attempt (speculatively, in parallel) the proof of the main goal using any subset of the candidate lemmas. Lemmas used must also be independently proved, but if that turns out to be hard (or even impossible) other options of finishing a proof may also be possible. Non-useful conjectures can be ignored and need not be proved, saving time. Since automatic theorem provers (ATPs) like Vampire and cvc5 now natively support applying automated induction [11, 22] it is no longer necessary to use a specialized prover to apply induction before sending the resulting proof obligations to an ATP, as HipSpec did, and we examine the differences between the two approaches.

The performance of ATPs like Vampire is heavily influenced by the use of proving *strategies* and their combinations into *schedules* [15, 27, 28, 30]. In addition to investigating the influence of adding lemmas from theory exploration, we also experiment with various learned strategies tailored for inductive proofs. A specialized strategy may allow Vampire to invent some easy lemmas itself, by applying generalization of a suitable subterm in a goal, lessening the need for theory exploration. However, finding strong targeted strategies is a time consuming endeavour which requires a set of problems with similar characteristics to those which we are interested in proving. For regular users, who typically just want to apply Vampire out of the box, this might not be an option.

## 2 Background

We propose the following design for an inductive theorem proving system:

1. We first use QuickSpec for theory exploration on the theory in question, generating equational conjectures about the theory.
2. The theory file including the original goal plus the conjectures from QuickSpec is sent to Vampire to attempt to find a proof.

Using our tools these two steps can be performed fully automatically, taking a problem file in the TIP [7] format as input and returning the proof found by Vampire as output.

### 2.1 QuickSpec

As seen in Sect. 1, QuickSpec is a system that produces equational *conjectures* about a theory. The conjectures are not guaranteed to be true, but have been tested to hold on 1000 randomly-generated test cases.<sup>2</sup> QuickSpec was originally designed to make conjectures about Haskell programs, but has been adapted to problems in inductive theorem proving.

<sup>2</sup> In automated reasoning terms, this means that 1000 ground instances of the conjecture have been shown to hold.

Conjecturing equations is difficult because of combinatorial explosion: even if we consider only quite simple equations and theories, there are many millions of possible conjectures. For example, if we identify a set of  $n = 10,000$  interesting *terms*, then there are  $n^2 = 100,000,000$  candidate equations which could be built from those terms. Generating and testing all of them is out of the question.

QuickSpec uses a more sophisticated approach which scales with the number of *terms* (e.g. 10,000) rather than the number of possible *equations* (e.g. 100,000,000). We enumerate terms in order of size (these terms may end up being the left or right hand side of an equational conjecture). We consider each term one by one, building up two sets as we go:

- The set of *discovered conjectures* between the terms considered so far.
- The set of *representative terms*. This consists of the set of terms considered so far, except that when several terms are equal, only one of them will be chosen as a representative. Therefore no two representative terms are equal.

Each time we consider a new term  $t$ , we answer the following question: *Is it equal to any representative term?* We do this in two steps:

1. *Pruning*. We check if the discovered conjectures imply that  $t$  is equal to a representative term  $r$ . If so, we simply ignore  $t$  and move on to the next term.
2. *Testing*. We test  $t$  against all representative terms. If it seems to be equal to some representative term  $r$ , we produce the conjecture  $t = r$ . Note that, since no two representative terms are equal, we only ever produce one conjecture per term.

If neither case holds, we add  $t$  as a representative term. The idea here is that, in case (1), the equation  $t = r$  is redundant – we knew it already – whereas in case (2), it is new information and hence a potentially useful conjecture.

For example, suppose we take the list append function  $++$  and consider the following terms, where  $x$ ,  $y$  and  $z$  range over lists:  $[], x, y, z, x ++ [], y ++ [], x ++ (y ++ z), x ++ (x ++ y), (x ++ y) ++ z, (x ++ x) ++ y$ .<sup>3</sup> Initially, the set of discovered equations and the set of representative terms are empty. The algorithm proceeds as follows:

- $[]$ . We add  $[]$  as a representative term.
- $x, y, z$ . None of these terms are equal to each other or  $[]$ , so we add them as representative terms.
- $x ++ []$ . The testing step reveals that this term is equal to  $x$ . We produce the conjecture (1):  $x ++ [] = x$  (and do not add  $x ++ []$  as a representative term).
- $y ++ []$ . The pruning step shows that this term is equal to  $y$ , by conjecture (1). We discard the term.
- $x ++ (y ++ z), x ++ (x ++ y)$ . We add these terms as representatives.

---

<sup>3</sup> In reality we enumerate terms in a systematic way, and a further refinement to the algorithm, *schemas*, eliminates many terms that differ from an existing term only in choice of variables.

- $(x ++ y) ++ z$ . Testing shows that this term is equal to  $x ++ (y ++ z)$ . We produce the conjecture (2):  $(x ++ y) ++ z = x ++ (y ++ z)$ .
- $(x ++ x) ++ y$ . Pruning shows that this term is equal to  $x ++ (x ++ y)$ , by conjecture (2). We discard the term.

At the end we have produced the conjectures (1)  $x ++ [] = x$  and (2)  $(x ++ y) ++ z = x ++ (y ++ z)$ . Note that these conjectures are *complete* with respect to the enumerated terms, in the sense that any true equation between two such terms follows from the conjectures. In general, the QuickSpec algorithm produces a complete set of equations in this sense (though not necessarily sound, i.e. we may have false equations if we are unlucky in the testing).

It is perhaps not obvious why this algorithm should be fast. We point out the following reasons:

- The runtime of the algorithm scales with the *number of terms considered*, not the number of possible equations. This is because, with careful data structures and algorithms, in both the pruning and testing steps we can efficiently compare a new term against *all representative terms at once*, in close to constant time. For pruning, we use unfailing completion [1] as implemented in Twee [24] to build a rewrite system from the discovered conjectures. We then keep the set of representative terms normalized with respect to this rewrite system. To prune a new term, we just normalize it and see if this normal form appears in the set of representative terms. For testing, we build a decision tree which allows us to, with a few (typically  $< 10$  test cases), either show that a new term  $t$  is not equal to any representative term, or find precisely one term  $r$  that it *might* be equal to, whereupon we can test the single equation  $t = r$  more thoroughly.
- In the common case, it takes a tiny amount of time ( $\ll 1$  ms) to consider each term. That is because: (1) in the pruning step, the term is just normalized, an operation taking microseconds; (2) in the testing phase, typically the term is not equal to any representative term, in which case (as mentioned above) the term is evaluated on only a few test cases. The only expensive case is when testing reveals that the new term is equal to a representative case – but this is precisely the case where we have discovered a new conjecture!

Therefore, the runtime of QuickSpec largely grows proportionally with the *number of discovered conjectures*, plus a small amount which is proportional to the number of explored terms. In practice, QuickSpec is able to handle theories with  $\approx 20$  functions and generate equations having  $\approx 10$  symbols on each side, after which the number of discovered conjectures typically becomes too huge.

## 2.2 Induction in Vampire

Vampire supports induction over both term algebras and integers. The former, used in this work, is based on a constructor-style and two infinite descent-style schemas [21] in addition to ad hoc schemas generated from well-founded recursive

functions in the search space [13]. When inducting on a term in a unit clause (a literal), an instance of a schema with the negation of the unit clause is added to the search space. A stronger (and also more explosive) feature is non-unit induction, which inducts on arbitrarily many occurrences of a term, possibly across many literals and clauses.

Some basic lemma generation techniques such as generalizations over complex terms and occurrences [12] as well as active occurrence heuristics are also supported. In the presence of function definitions or induction hypotheses, (unordered) paramodulation may be used to reach lemmas otherwise not reachable with ordered superposition [13]. For a more detailed description of induction in Vampire we refer to [11].

**Lemma Generation in Vampire.** Vampire uses the traditional top-down backward reasoning approach to generate lemmas. It tries to reduce goals into subgoals and apply inferences on them, interleaved with induction inferences applied to all intermediate consequences that result from this process. A new lemma may be conjectured by generalizing over one of the terms in a subgoal. This lemma generation approach in Vampire usually derives different lemmas than QuickSpec’s bottom-up theory exploration approach.

**Simplifications and Orderings in Superposition.** As superposition is tailored for first-order reasoning, it does not come as a surprise that some techniques that increase the efficiency of first-order reasoning are incompatible with inductive reasoning or higher-order reasoning in general. In particular, simplifications and orderings can affect a built-in induction within superposition.

Simplifications are inferences where one of the premises becomes redundant for further first-order reasoning and can be removed. For example, demodulation rewrites a clause into a smaller clause with an unconditional (unit) equation, and removes the original clause. In inductive reasoning things are not as simple, and any clause (even if it follows from smaller clauses) can be useful to generate interesting lemmas. For example, we might simplify a clause that would give rise to a crucial generalized lemma into a clause that does not give the same generalization anymore. Interestingly, given that simplification steps take up most of the inferences in a saturation run, in our experience this affects inductive reasoning less than expected.

### 3 Implementation

In order to perform our experiments we needed to integrate the lemmas conjectured by QuickSpec into Vampire’s proof search, and choose a promising proof search strategy.

### 3.1 Conjectured Lemmas, AVATAR, and Vampire’s Claims

Integrating conjectured lemmas into proof search poses a technical challenge as they must be proven before they can be soundly used in a proof. At the same time, trying to prove each suggested lemma before the main goal is even attempted can create a great deal of unnecessary work. As has been noted before [8, 21], this challenge can be smoothly overcome in the presence of the AVATAR architecture for clause splitting [20, 29].

AVATAR keeps track of information about which clause has been derived from which splitting assumption, and soundly propagates it through inferences. Deriving the empty clause conditioned on some assumptions then does not necessarily mean the search is successfully concluded, but merely signifies that the conjunction of the attached assumptions can no longer be maintained. (AVATAR then updates its propositional model to reflect this newly derived information through a call to an underlying SAT or SMT solver.)

Let us assume we want to accommodate a speculative proof with lemmas  $L_1, \dots, L_n$  under AVATAR, where each lemma  $L_i$  is a closed formula. As a first approximation to explaining how this can be done, let us imagine introducing and immediately splitting the tautologies  $L_i \vee \neg L_i$  for  $i = 1, \dots, n$ .<sup>4</sup> Each clause in the search then carries (independently for each  $i$ ) the information whether it depends on: 1) the assumption corresponding to  $L_i$  (proving with the help of lemma  $L_i$ ), 2)  $\neg L_i$  (trying to prove lemma  $L_i$ ), or 3) neither of these (currently ignoring lemma  $L_i$ ). Depending on the order in which (conditional) empty clauses get derived, the whole power set of possible scenarios is played out as if in parallel, in which some lemmas may already have been shown to suffice for proving the main conjecture, while themselves waiting to be proven (possibly with the help of other lemmas). The underlying SAT/SMT solver orchestrates the whole endeavour, decides which compatible subset of assumptions will be worked on next, and declares the proof attempt successful as soon as the first such scenario is complete. We remark that cyclic reasoning is automatically avoided by treating the assumptions of  $L_i$  and  $\neg L_i$  as mutually exclusive.

It is surprisingly easy to get access to this feature of speculative lemma use in Vampire under AVATAR. In fact, we can rely on a small adjustment of just the parser added by Andrei Voronkov already in 2011. In the TPTP language [26], this adjustment introduced a new custom formula role called the *claim*. Precisely as in our use case, a claim is a formula that most likely follows from the surrounding axioms and has a high chance of being useful for proving the given conjecture, but must be itself also proven by the system in a valid proof. For this work we extended Vampire’s SMT-LIB parser in an analogous way and added a custom construct `assert-claim` with the same semantics.

---

<sup>4</sup> In reality, both  $L_i$  and  $\neg L_i$  must also be skolemized and clausified, which in the prover happens before splitting. We return to this aspect further below.

Technically, when the parser reads a claim formula  $L$ , it picks a fresh propositional symbol  $p_L$  and passes on the equivalence  $p_L \leftrightarrow L$  as a standard axiom.<sup>5</sup> The equivalence  $p_L \leftrightarrow L$  is then clausified to

$$\{\neg p_L \vee C \mid C \in \text{CNF}(L)\} \quad \text{and} \quad \{p_L \vee D \mid D \in \text{CNF}(\neg L)\}.$$

AVATAR recognizes the  $p_L$  and  $\neg p_L$  as complementary ground components and will then always assert either  $p_L$  or  $\neg p_L$ . Thus the first-order part of the prover must work with either the clauses from  $\text{CNF}(L)$  or from  $\text{CNF}(\neg L)$ , while AVATAR keeps track of the respective dependencies.

### 3.2 Proving Strategies and a New Induction Schedule

A theorem prover typically has many parameters (in Vampire called *options*) that can be changed to adjust the proof search characteristics. In Vampire, there are more than 100 options for configuring the preprocessing steps, the saturation algorithm, generating and simplification rules, proof search heuristics and also induction. By a *strategy* we mean a concrete assignment of values to such options. It is long known [27, 31] that the success rate of an ATP can be dramatically improved by arranging a number of different proving strategies of complementary characteristics into a *strategy schedule*, a sequence of strategies with assigned time budgets, to be executed in sequence (or in parallel).

In this work, we constructed a strategy schedule specifically targeting inductive theorem proving on the TIP benchmarks (see Sect. 4.1). We followed the strategy discovery recipe pioneered by the Spider system [30]. This consists of

1. Randomly sampling strategies to try to solve a previously unsolved problem (or possibly to improve the solution time on a problem already known to be solvable).
2. Optimizing the found strategy on that problem using local search (in which, for each option in turn, different values are tried out and a new value is committed to, if the corresponding change leads to an improved time or the time stays the same, but the value becomes default).
3. Evaluating the optimized strategy on all problems, to update the information about which problems are solvable and in what best time.

In our case, we sampled strategies from a space defined by a total of 115 base Vampire options and 19 dedicated induction options. Most of these options are Boolean, many are finite enumerations of discrete values and a few are numeric. It is clear that the totality of all strategies is astronomically large and random sampling is a way to have access to all the strategies, at least in principle. We searched for strategies in parallel on 60 cores of our server<sup>6</sup> for several days. In

<sup>5</sup> The only extra effort is to mark and a protect the new symbol  $p_L$  against potential elimination during preprocessing, as, after all, the new equivalence would otherwise qualify as an unused predicate definition and could be discarded.

<sup>6</sup> Equipped With Intel®Xeon®Gold 6140 CPU @ 2.3 GHz and 500 GB RAM.

the end, we collected 246 strategies covering 236 of the 486 TIP benchmarks that we used for training.

Once a sufficiently large set of strategies has been discovered (or when the rate of solving new problems becomes too low to make search for additional strategies worth the effort), schedule construction can be formulated as an integer programming task, in which running times are assigned to individual strategies to cover the union of as many problems as possible while not exceeding a given overall time bound [15, 23]. We instead adopted a greedy algorithm [4] to a weighted set cover formulation of the problem: starting from an empty schedule, we iteratively add a new strategy  $s$  for additional  $t$  units of time if this step is currently the best in terms of  $1/t$ —“the number of problems that will additionally get covered”. This greedy approach does not guarantee an optimal result, but runs in polynomial time and is really easy to implement. (See also [3].)

Our final schedule makes use of 66 of the discovered strategies and should be able to solve all of the covered 236 problems in under 12s (per problem). For our later experiment we prepared a second schedule, specialized to also take into consideration the versions of the TIP benchmarks with the added lemmas (cf. label T in Sect. 4 below). This schedule makes use of 86 strategies, aims to cover a total of 522 problems (version with and without lemmas counted separately) and runs to completion after approximately 24s.

## 4 Evaluation

In our evaluation we compare several variants of Vampire. We start with two baseline versions without strategy scheduling:

- (V): Vampire with the following flags for structural induction:

`-ind struct -indoct on -nui on -to lpo -drc off`

The option `-ind struct` enables using structural induction (constructor-based induction axioms for term algebras), and with `-indoct on` these axioms are based on generalizing over any term, not just Skolem constants. Moreover, the induction axioms are generated from any clause set using `-nui on`. Finally, `-to lpo` and `-drc off` enable a simplification ordering which is well-suited for handling recursive functions.

- (V + L): Vampire with the same flags active as in (V), plus conjectures from QuickSpec added to the problem files as claims as described in Sect. 3.1.

The idea is that (V) serves as a baseline for what kinds of inductive proofs Vampire is capable of. By comparing (V) with (V + L), we see whether the lemmas discovered by QuickSpec help Vampire.

Next we add versions of Vampire with specialized strategy schedules:

- (S): Vampire with the specialized strategy schedule for inductive problems described in Sect. 3.2.

- (S + L): Vampire with both the strategy schedule as in (S) above and conjectures from QuickSpec added to the problem files as claims.

By comparing (S) with (V+L), we can see the relative importance of strategy scheduling and lemmas. In (S+L) we can see whether the two strategies complement each other. We expect that (S) may see less benefit from lemmas than (V) because the learned strategies may be better at e.g. generalising subgoals. Note that the strategy schedule is tuned without seeing the lemmas, so it is even possible that (S+L) might perform worse than (S) due to the extra lemmas disturbing the strategy scheduler.

Since the strategy schedule is tuned without seeing the lemmas, (S+L) illustrates what we can get by taking an existing prover with a built-in strategy schedule, and adding new lemmas to it. Notice also that any problems that require lemmas will not be proved during training, so will not influence the schedule. We can use these problems as a kind of test set for S+L, as they are effectively unseen during training.

To investigate the limits of our approach we add a third family:

- (T): Vampire with a specialized strategy schedule for inductive problems, trained on the TIP problems after conjectures from QuickSpec have been injected into the problem files.
- (T + L): Vampire with the lemma-specialized schedule (T) as above and conjectures from QuickSpec added to the problem files as claims.

Note that the strategy used by (T) and (T+L) may be prone to overfitting, as all the test problems are seen during training and influence the schedule.<sup>7</sup> The results for (T) and (T+L) are useful as a benchmark to compare the other provers against, and an indication of what a perfectly-tuned strategy schedule could do.

We evaluated our methods on the TIP benchmark set. For all methods the time limit was set to 30s. Since the strategy schedules are randomized and may not find the same proofs every time they are run, we ran each one 5 times on each problem. The experiments were run on a Dell Inc. Latitude 5320 with an 11th Gen Intel®Core™i5-1145G7 @ 2.60GHz × 8 processor and 16GB RAM. Scripts used to run experiments and process results are available at <https://github.com/solrun/vampspec>.

## 4.1 TIP Benchmarks

TIP is a collection of benchmarks specifically for inductive theorem provers [6]. The problems are expressed in a syntax very similar to SMT-LIB [2], and come

---

<sup>7</sup> Why not use a training/test split? Because there are not very many problems in total, and more importantly, because many problems are related, which makes it hard to design an uncontaminated test set, since we need to avoid having related problems where one is in the training set and one is in the test set.

with tools to translate the problems into various formats (including standard SMT-LIB) as well as built-in support for lemma generation using QuickSpec.

TIP consists of several subsets: the *prod* set contains 50 theorems and 24 lemmas about lists and natural numbers defined in [16], the *IsaPlanner* set defined in [18] contains 86 properties originally designed to test provers that use the rippling heuristic. The *prod* and *IsaPlanner* problems have previously been used to evaluate a number inductive theorem provers [5, 9, 22] so experiments with them enable comparison to previous work.

The *TIP2015* set contains a further 326 problems and was added as many existing provers, like HipSpec, could solve almost all problems in the previous two sets. It includes a variety of problems such as various sorting algorithms with correctness properties expressed in alternative ways, properties of regular expressions, binary search trees, integers implemented on top of natural numbers, natural numbers in binary representation, and properties of various functions on lists and natural numbers. Some of the problems were not known to have been automated at the time of their publication [6] and are offered as challenges.

## 4.2 Results

Table 1 shows the number of proofs found for the 486 TIP benchmarks, by the different methods that we described previously. We count a proof as found if it was found in any of the 5 proof attempts using that strategy. We found that Vampire with structural induction enabled, (V), finds proofs to 102 of the problems, which increases to 143 with the addition of lemmas from QuickSpec. The specialized strategy schedule, (S), finds 236 proofs, more than twice as many as (V). The specialized strategy schedule finds some more proofs with the addition of lemmas but the increase is not so great, from 236 to 263. The strategy schedule trained on problems already containing lemmas, (T), finds 237 proofs (the same proofs as (S) and one additional proof), which increases to 288 with the addition of lemmas. The bottom line of Table 1 shows the number of proofs found by each method with or without lemmas added.

**Table 1.** The number of proofs found for the 486 TIP benchmarks when testing the different proof methods in the presence and absence of generated lemmas.

| Proofs found       | (V) | (S) | (T) |
|--------------------|-----|-----|-----|
| no lemmas          | 102 | 236 | 237 |
| with lemmas (+ L)  | 143 | 263 | 288 |
| Total proofs found | 153 | 269 | 289 |

Although all methods find more proofs with lemmas than without, a number of proofs can only be found without the additional lemmas and are lost after lemmas are added. Most often when using ATPs, different strategies or parameterizations might both gain and lose some proofs rather than one simply

being strictly better than the other. Table 2 shows this for our three strategies, with and without added lemmas. As mentioned above there are a small number of proofs (10 for (V), 6 for (S) and 1 for (T)) which are found by the each strategy without lemmas, but not after lemmas are added. Since the added lemmas increase the size of the proof search space, we are not surprised that they may in some cases prevent the strategy from finding a proof in time. In the case of the specialized strategy schedule (T) which has already seen the problems with added lemmas in its training, the added lemmas only hinder it from finding a proof in one instance. In the case where (T+L) loses the proof (T) found, *TIP2015/regexp\_RecAtom*, the number of conjectured lemmas added to the problem file is very large (459) which probably causes the search space to explode. For this particular problem both strategies (V) and (S) also found a proof, but no strategy found a proof for the problem with lemmas added.

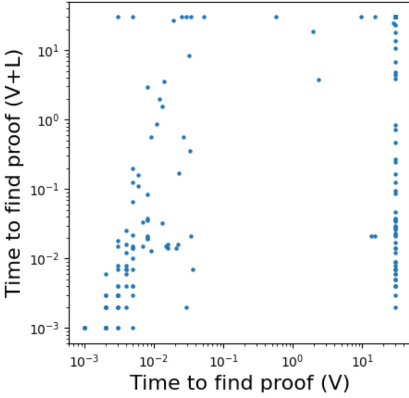
Note that (T+L) finds 53 proofs not found by (S), showing the improvement achievable by adding QuickSpec’s lemma conjecturing and using a strategy schedule specialized to make use of those lemmas, compared to only using strategy schedule training as with (S). Of these 53 problems, (S+L) finds proofs for 33 of them, making use of the added lemmas without having seen them in its training. As mentioned above, the strategy schedule of (S+L) is effectively not trained on any problems that require lemmas, so we can view the 53 problems as test problems, unseen in the training data, all solvable with a perfect strategy, and say that (S+L) solves 62% of those problems. Thus we get an indication that the strategy schedule is generalizing to unseen problems.

**Table 2.** Here each column shows the number of unique proofs found by the respective method (column label) but not by one of the other methods (row label).

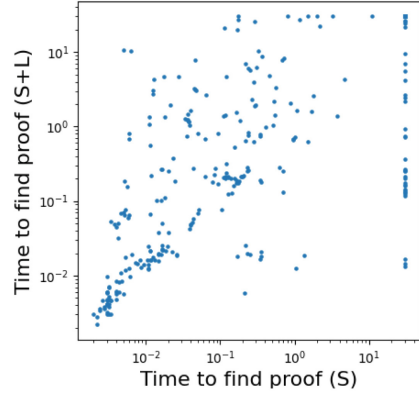
|           | (V) | (V + L) | (S) | (S + L) | (T) | (T + L) |
|-----------|-----|---------|-----|---------|-----|---------|
| - (V)     |     | 51      | 134 | 162     | 135 | 187     |
| - (V + L) | 10  |         | 109 | 124     | 109 | 145     |
| - (S)     | 0   | 16      |     | 33      | 1   | 53      |
| - (S + L) | 1   | 4       | 6   |         | 6   | 25      |
| - (T)     | 0   | 15      | 0   | 32      |     | 52      |
| - (T + L) | 1   | 0       | 1   | 0       | 1   |         |

In some cases, one of our methods performs strictly better than another, namely (S) and (T) are strictly better than (V), (T) is strictly better than (S), and (T+L) is strictly better than (S+L). Since (S) and (T) are specialized strategy schedules that can execute many different strategies for each proof attempt, including the strategy used by (V), it is unsurprising that they subsume (V). Since (S) and (T) are strategy schedules trained in the same manner, with the only difference being that (T) is trained on a superset of the problems (S) is trained on, and both evaluated on problems they have encountered in

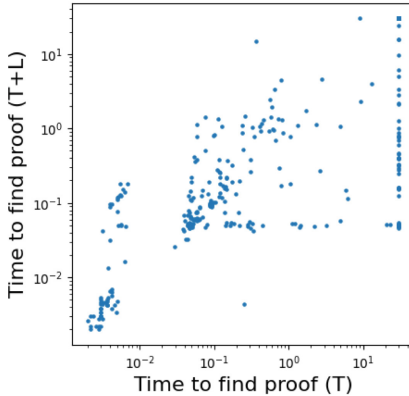
their training, we expect them to achieve a similar performance. Note that (T) only finds one proof not found by (S), so their performance is nearly equivalent. Since (T+L) is evaluated on problems with added lemmas that it has already seen during training while (S+L) is given previously unseen lemmas in its input problems, it would be surprising if (S+L) found a proof that (T+L) could not. In all cases these pairs of methods are evaluated on the same input problems (both are evaluated on problems with additional lemmas or both on the problems without lemmas).



(a) Time taken by (V+L) vs by (V).



(b) Time taken by (S+L) vs by (S).



(c) Time taken by (T+L) vs by (T).

**Fig. 1.** Time taken to find a proof with lemmas versus without them using the same strategy (on a log scale).

In cases where the same strategy could find a proof both with and without lemmas added to the problem file, we compare the time taken to find a proof as a metric of how easy the proof is to find. Figure 1 shows the plots of the time taken to find a proof with lemmas versus without them using the same strategy (on a

log scale). The points around the edges indicate that the respective method did not find a proof within the given time limit (30 s), so points along the right-hand edge indicate that a proof was found with lemmas and not without them, while points along the top edge indicate a proof was found without lemmas and lost after they were added.

For problems where both (V) and (V+L) found a proof, the average time for (V) was 0.67 s with a standard deviation of 3.56 s, while the average time for (V+L) was 1.04 s with a standard deviation of 4.30 s. We can see how in most cases where a proof was found both with and without lemmas added, the proof search went faster without them, indicated by how most of the points are to the left of the diagonal. For problems where both (S) and (S+L) found a proof, the average time for (S) was 0.20 s with a standard deviation of 0.49 s while the average time for (S+L) was 1.53 s with a standard deviation of 4.16 s. We see many points clustered around the diagonal, indicating both proof searches took a similar amount of time to find a proof, though many more points lie to the left of the diagonal than to its right, indicating a faster proof search without lemmas. For problems where both (T) and (T+L) found a proof, the average time for (T) was 0.59 s with a standard deviation of 2.39 s while the average time for (T+L) was 0.37 s with a standard deviation of 1.16 s, so as opposed to methods (V) and (S), the proof time goes down with the addition of lemmas. Since the schedule here was trained on problems containing lemmas, it prioritizes strategies that make use of the available lemmas, thus finding the proofs more efficiently with lemmas.

**Table 3.** The number of proofs found in different subsets of the TIP benchmarks, along with results for CVC4 and HipSpec for the same subsets.

| Set (size)             | (V) | (V+L) | (S) | (S+L) | (T) | (T+L) | CVC4 | HipSpec |
|------------------------|-----|-------|-----|-------|-----|-------|------|---------|
| <i>prod</i> (50)       | 7   | 29    | 27  | 46    | 28  | 49    | 39   | 47      |
| <i>IsaPlanner</i> (85) | 41  | 43    | 73  | 75    | 73  | 81    | 80   | 80      |
| <i>TIP2015</i> (326)   | 39  | 53    | 113 | 119   | 113 | 134   | –    | –       |

The problems from the *IsaPlanner* and *prod* subsets of TIP were also used for evaluation of HipSpec in [5]<sup>8</sup> and of inductive reasoning with CVC4 in [22]. The number of proofs they found are included in Table 3 along with the results of our experiments for those subsets.<sup>9</sup> We see a clear difference in results on the *prod*-

<sup>8</sup> In order to investigate whether the numbers for HipSpec would be better on a modern machine, we re-ran it on the *prod* benchmark. We found that it solved *fewer* problems, 44 in all, as a result of slight changes in HipSpec since the publication of [5]. No problems were solved by HipSpec today that were not solved back then.

<sup>9</sup> We tried but failed to run HipSpec on the *TIP2015* problems. HipSpec’s input format is a limited dialect of Haskell and, while TIP problems can be converted to Haskell, the dialect is not the same as HipSpec’s. As HipSpec is unmaintained, we were unable to go further.

subset, which is designed such that more complicated lemmas are needed for most proofs, and the *IsaPlanner*-subset, which contains easier problems which can often be solved without external lemmas (or with just lemmas coming from generalizations of a subgoal). On both subsets we only achieve results competitive with either CVC4 or HipSpec when we combine a specialized strategy schedule with lemmas (S+L) and (T+L). On the *TIP2015* subset, none of the methods we tested found proofs for even half of the problems, and we leave a closer examination of what is required to achieve better results there as future work.

As described in Sect. 3.2, the strategy schedules may not find the same proofs in every run. In our experiments we ran each schedule 5 times and found there was a handful of problems where the same strategy schedule would sometimes find a proof and not others, the exact numbers are shown in Table 4. In the results shown in Tables 1–3 we count that a proof was found if it was found in at least one of the five runs.

**Table 4.** Number of inconsistently found proofs by each strategy schedule with and without added lemmas.

| Method              | (S) | (S + L) | (T) | (T + L) |
|---------------------|-----|---------|-----|---------|
| Inconsistent Proofs | 1   | 9       | 4   | 6       |

## 5 Discussion

Modern day ATPs like Vampire have many moving parts. Slight changes in configuration often lead to some extra proofs being found while others are lost. This is particularly true when considering also proofs by induction, as here the potential for exploding the search space in unproductive directions is even larger. It is often difficult to know in advance what parameters and strategies will affect the capabilities of finding a proof within reasonable time.

Our first experiments tested the effect of adding lemma candidates for inductive proofs to a standard out-of-the-box variant of Vampire, simulating what a regular user might have at hand. Here, we see a clear improvement in the number of proofs for the *TIP prod*-subset, where more complicated lemmas are needed for most of the proofs, and a modest improvement on other subsets. Still, the results are well below both CVC4 and HipSpec. We conclude that simply adding lemmas from QuickSpec to Vampire (with a default induction strategy) is not sufficient to reach a state-of-the-art performance.

Secondly, we also experimented with training specialized strategies, customized to inductive proofs on TIP problems. This seems to be necessary for top performance. We experiment with two trained strategies: the first on proofs of TIP-problems without added lemmas (starting from the out-of-the-box Vampire setup). The second, to get an upper bound of how well Vampire could perform, we also trained on proofs *with the added lemmas* from QuickSpec. With a customized strategy for induction, already without lemmas Vampire performs much

better than before. We notice that the increase is larger for the customized strategy than it was for adding lemmas to the standard Vampire version! We conclude that specialized strategies have a larger effect on the number of proofs than just adding auxiliary lemmas.

Finally, we added the QuickSpec lemmas. Both strategies now improved even more, especially on the TIP *prod*-subset, where they both beat previous state of the art, proving 46 and 49 problems respectively. As expected, the strategy trained on proofs with lemmas (T) had a larger increase, being able to use the lemmas available more efficiently. Interestingly, on the TIP *IsaPlanner*-subset, only the (T) strategy beat the state of the art. We conclude that in the presence of auxiliary lemmas and together with specialized strategies, Vampire can indeed outperform previous state of the art systems CVC4 and HipSpec.

However, one might argue that the comparison is biased. To get state-of-the-art performance from Vampire requires a strategy optimised by seeing and trying the problems already! This might not be a viable option for all users. We do not know how well these schedules would perform on other types of inductive problems as they are likely overfitted to TIP to some degree. We could have divided the TIP problems into training and testing sets to try to avoid overfitting, but the TIP set is not very large, only 486 problems (of which only 60% could be solved using any method we tried), and many problems are similar to each other, so it is not clear that this would solve the problem. In short, there is too little data to train a general-purpose strategy, and we can not say how well the learned strategy generalizes to problems outside of TIP.

Even so, domain-specific strategies are reasonable in many applications. For example, in program verification, it is reasonable to run the prover over a set of problems multiple times, and find a strategy that works for just those kind of problems one is interested in verifying. Our results show that specially-tuned strategies are highly effective, and compatible with lemma discovery.

The search space for proofs where induction is allowed is inherently enormous and becomes particularly explosive when the ATP itself has to decide when to apply induction. Trained strategies seem to be necessary for competitive performance. HipSpec on the other hand was developed before CVC4 and Vampire supported induction, and thus handled the induction step outside the ATP, and only outsourced the resulting subgoals. One benefit of doing so is that the search space is much less explosive, which contributes to HipSpec's good performance. We thus leave the question of how to best implement automated induction partially unsolved: we either need highly specialized strategies trained on many attempts of proofs, or keeping the application of induction under strict control.

## 5.1 Future Work

We have many ideas for improvements when it comes to generating lemmas for inductive proofs. QuickSpec is limited to discovering equational conjectures that may have a predicate as a condition (if the theory being explored contains a function that returns a boolean value, the value of that function may be used as a predicate). However, many inductive proofs require more complex conditional

lemmas. In [10] we presented RoughSpec, a system that generates conjectures that match a user-defined input template. This could be used to conjecture lemmas for inductive proofs, using lemma templates likely to be useful learned from proof libraries. Another idea is developing better methods of only providing lemmas likely to be useful, limiting the number of lemmas given to the prover so that the search space does not explode. For example, there are some prominent examples of using simple syntactic conditions [14] inside the theorem prover or using machine learning [19] before the invocation of the theorem prover to mitigate this issue.

**Acknowledgments.** This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation. Martin Suda was supported by the Czech Science Foundation project no. 24-12759S and the project RICAIP no. 857306 under the EU-H2020 programme.

## References

1. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. In: *Rewriting Techniques*, pp. 1–30. Elsevier (1989)
2. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017). [www.SMT-LIB.org](http://www.SMT-LIB.org)
3. Bártek, F., Chvalovský, K., Suda, M.: Regularization in spider-style strategy discovery and schedule construction. In: *IJCAR* (2024), accepted
4. Chvátal, V.: A greedy heuristic for the set-covering problem. *Math. Oper. Res.* 4(3), 233–235 (1979)
5. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Automating inductive proofs using theory exploration. In: Bonacina, M.P. (ed.) *CADE 2013. LNCS (LNAI)*, vol. 7898, pp. 392–406. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-38574-2\\_27](https://doi.org/10.1007/978-3-642-38574-2_27)
6. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: TIP: tons of inductive problems. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) *CICM 2015. LNCS (LNAI)*, vol. 9150, pp. 333–337. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-20615-8\\_23](https://doi.org/10.1007/978-3-319-20615-8_23)
7. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: The TIP format. <http://tip-org.github.io/format.html>
8. Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) *FroCoS 2017. LNCS (LNAI)*, vol. 10483, pp. 172–188. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-66167-4\\_10](https://doi.org/10.1007/978-3-319-66167-4_10)
9. Dixon, L., Johansson, M.: Isaplaner 2: A proof planner for isabelle (2007)
10. Einarsdóttir, S.H., Smallbone, N., Johansson, M.: Template-based theory exploration: Discovering properties of functional programs by testing. In: *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages, IFL 2020*, pp. 67–78. Association for Computing Machinery, New York (2021). <https://doi.org/10.1145/3462172.3462192>
11. Hajdu, M., Hozzová, P., Kovács, L., Reger, G., Voronkov, A.: Getting Saturated with Induction, pp. 306–322. Springer Nature Switzerland, Cham (2022)

12. Hajdú, M., Hozzová, P., Kovács, L., Schoisswohl, J., Voronkov, A.: Induction with generalization in superposition reasoning. In: Benzmüller, C., Miller, B. (eds.) *Intelligent Computer Mathematics*, pp. 123–137. Springer International Publishing, Cham (2020). [https://doi.org/10.1007/978-3-030-53518-6\\_8](https://doi.org/10.1007/978-3-030-53518-6_8)
13. Hajdú, M., Hozzová, P., Kovács, L., Voronkov, A.: Induction with recursive definitions in superposition. In: *Formal Methods in Computer Aided Design, FMCAD 2021*, New Haven, CT, USA, 19–22 October 2021, pp. 1–10. IEEE (2021)
14. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) *CADE 2011. LNCS (LNAI)*, vol. 6803, pp. 299–314. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22438-6\\_23](https://doi.org/10.1007/978-3-642-22438-6_23)
15. Holden, E.K., Korovin, K.: Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In: Kamareddine, F., Sacerdoti Coen, C. (eds.) *CICM 2021. LNCS (LNAI)*, vol. 12833, pp. 107–123. Springer, Cham (2021). [https://doi.org/10.1007/978-3-030-81097-9\\_8](https://doi.org/10.1007/978-3-030-81097-9_8)
16. Ireland, A., Bundy, A.: Productive use of failure in inductive proof. *J. Autom. Reason.* **16**, 79–111 (1996)
17. Johansson, M.: Lemma discovery for induction. In: Kaliszyk, C., Brady, E., Kohlhase, A., Sacerdoti Coen, C. (eds.) *CICM 2019. LNCS (LNAI)*, vol. 11617, pp. 125–139. Springer, Cham (2019). [https://doi.org/10.1007/978-3-030-23250-4\\_9](https://doi.org/10.1007/978-3-030-23250-4_9)
18. Johansson, M., Dixon, L., Bundy, A.: Case-analysis for rippling and inductive proof. In: *International Conference on Interactive Theorem Proving* (2010)
19. Kühlwein, D., Blanchette, J.C., Kaliszyk, C., Urban, J.: MaSh: machine learning for sledgehammer. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) *Interactive Theorem Proving*, pp. 35–50. Springer, Berlin Heidelberg, Berlin, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-39634-2\\_6](https://doi.org/10.1007/978-3-642-39634-2_6)
20. Reger, G., Suda, M., Voronkov, A.: Playing with AVATAR. In: Felty, A.P., Middeldorp, A. (eds.) *CADE 2015. LNCS (LNAI)*, vol. 9195, pp. 399–415. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-21401-6\\_28](https://doi.org/10.1007/978-3-319-21401-6_28)
21. Reger, G., Voronkov, A.: Induction in saturation-based proof search. In: *CADE* (2019). <https://api.semanticscholar.org/CorpusID:126940163>
22. Reynolds, A., Kuncak, V.: Induction for SMT solvers. In: D’Souza, D., Lal, A., Larsen, K.G. (eds.) *VMCAI 2015. LNCS*, vol. 8931, pp. 80–98. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46081-8\\_5](https://doi.org/10.1007/978-3-662-46081-8_5)
23. 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, Israel, 11 - 12 August, 2022. *CEUR Workshop Proceedings*, vol. 3201. CEUR-WS.org (2022). <https://ceur-ws.org/Vol-3201/paper8.pdf>
24. Smallbone, N.: Twee: An equational theorem prover. In: *CADE*, pp. 602–613 (2021)
25. Smallbone, N., Johansson, M., Claessen, K., Alghed, M.: Quick specifications for the busy programmer. *J. Funct. Program.* **27** (2017)
26. Sutcliffe, G.: The Logic Languages of the TPTP World. *Logic J. IGPL* (2022). <https://doi.org/10.1093/jigpal/jzac068>
27. Tammet, T.: Towards efficient subsumption. In: Kirchner, C., Kirchner, H. (eds.) *CADE 1998. LNCS*, vol. 1421, pp. 427–441. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0054276>
28. Urban, J.: Blistr: The blind strategymaker. In: Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.) *Global Conference on Artificial Intelligence, GCAI 2015*, Tbilisi, Georgia, 16–19 October 2015. *EPiC Series in Computing*, vol. 36, pp. 312–319. EasyChair (2015), [https://easychair.org/publications/volume/GCAI\\_2015](https://easychair.org/publications/volume/GCAI_2015)

29. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 696–710. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_46](https://doi.org/10.1007/978-3-319-08867-9_46)
30. Voronkov, A.: Spider: learning in the sea of options. In: Vampire23: The 7th Vampire Workshop (2023), <https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833>, to appear
31. 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, May 18–20, 1998, Sanibel Island, Florida, USA, pp. 142–146. AAAI Press (1998). <http://www.aaai.org/Library/FLAIRS/1998/flairs98-027.php>

**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.

