Internal Guidance for Satallax

Michael Färber[1(✉)] and Chad Brown[2]

[1] Universität Innsbruck, Innsbruck, Austria
michael.faerber@uibk.ac.at
[2] Czech Technical University in Prague, Prague, Czech Republic

Abstract. We propose a new internal guidance method for automated theorem provers based on the given-clause algorithm. Our method influences the choice of unprocessed clauses using positive and negative examples from previous proofs. To this end, we present an efficient scheme for Naive Bayesian classification by generalising label occurrences to types with monoid structure. This makes it possible to extend existing fast classifiers, which consider only positive examples, with negative ones. We implement the method in the higher-order logic prover Satallax, where we modify the delay with which propositions are processed. We evaluated our method on a simply-typed higher-order logic version of the Flyspeck project, where it solves 26 % more problems than Satallax without internal guidance.

1 Introduction

Experience can be described as knowing which methods to apply in which context. It is a result of experiments, which can show a method to either fail or succeed in a certain situation. Mathematicians solve problems by experience. When solving a problem, mathematicians gain experience, which in the future can help them to solve harder problems that they would not have been able to solve without the experience gained before.

Fully automated theorem provers (ATPs) attempt to prove mathematical problems without user interaction. A thriving field of research is how to make ATPs behave more like mathematicians, by learning which decisions to take from previous proof attempts, in order to find more proofs in shorter time, and to prove problems that were previously out of reach for the ATP. Machine learning can help advance that field, for it provides techniques to model experience and to compare the quality of possible decisions. Machine learning approaches to improve ATP performance include:

- **Premise selection**: Preselecting a set of axioms for a problem can be done as a preprocessing step or inside the ATP at the beginning of proof search. Examples of this technique are the Sumo INference Engine (SInE) [HV11] and E.T. [KSUV15].
- **Internal guidance**: Unlike premise selection, internal guidance influences choices made during the proof search. The *hints* technique [Ver96] was among

N. Olivetti and A. Tiwari (Eds.): IJCAR 2016, LNAI 9706, pp. 349–361, 2016.
DOI: 10.1007/978-3-319-40229-1_24

the earliest attempts to directly influence proof search by learning from previous proofs. Other systems are E/TSM [Sch00], an extension of E [Sch13] with term space maps, and MaLeCoP [UVŠ11] respectively FEMaLeCoP [KU15], which are versions of leanCoP [Ott08] extended by Naive Bayesian learning.
- **Learning of strategies**: Finding good settings for ATPs automatically has been researched for example in the Blind Strategymaker (BliStr) project [Urb15].
- **Learning of strategy choice**: Once one has found good ATP strategies for different sets of problems, it is not directly clear which strategies to apply for which time when encountering a new problem. This problem was treated in the Machine Learning of Strategies (MaLeS) [Kü14].

In this paper, we show an internal guidance algorithm for ATPs that use (variations of) the given-clause algorithm. Specifically, we study a Naive Bayesian classification method, introduced for the connection calculus in FEMaLeCoP, and generalise it by measuring label occurrences with an arbitrary type having monoid structure, in place of a single number. This generalisation has the benefit that it can handle positive and negative occurrences. As a proof of concept, we implement the algorithm in the ATP Satallax [Bro12], using no features at all, which already solves 26 % more problems given the same amount of time, and which can solve about as many problems in 1 s than without internal guidance in 2 s.

2 Naive Bayesian Classifier with Monoids

2.1 Motivation

Many automated theorem provers have a proof state in which they make decisions, by ranking available choices (e.g. which proposition to process) and choosing the best one. This is related to the classification problem in machine learning, which takes data about previous decisions, i.e. which situation has led to which choice, and then orders choices by usefulness for the current situation.

For example, let us assume that the state of the theorem prover is modelled by the set of constants appearing in the previously processed propositions or in the conjecture. Let our conjecture be $x + y = y + x$ and let our premises include

$$\forall P.[P(0) \implies (\forall x.P(x) \implies P(s(x))) \implies \forall x.P(x)], \tag{1}$$

$$x + 0 = x. \tag{2}$$

If we first process Eq. 1, the prover state is characterised by $F = \{+, s, 0\}$. If we then continue to process Eq. 2 and it turns out that this contributes to the final proof, we register that in the situation F, Eq. 2 was useful.

In other proof searches, processing Eq. 2 in a certain prover state will not contribute towards the final proof. We call such situations negative examples.

Intuitively, we would like to apply propositions in situations that are similar to those in which the propositions were useful, and avoid processing propositions in situations similar to those where the propositions were useless. In general, examples (positive and negative) can be characterised by a prover state F and a proposition l that was processed in state F. This makes it possible to treat the choice of propositions as classification problem. In the next section, we show how to rank choices based on previous experience.

2.2 Classifiers with Positive Examples

A classifier takes pairs (F, l), relating a set of features F with a label l, and produces a function that, given a set of features, predicts a label. Classifiers can be characterised by a function $r(l, F)$, which represents the relevance of a label wrt a set of features. For internal guidance, we use r to estimate the relevance of a clause l to process in the current prover state F.

A Bayesian classifier estimates the relevance of a label by its probability to occur with a set of features, i.e. $P(l \mid F)$. By using the Naive Bayesian assumption that features are conditionally independent, the conditional probability is:

$$P(l \mid F) = \frac{P(l)P(F \mid l)}{P(F)} = \frac{P(l)\prod_{f\in F} P(f \mid l)}{P(F)} \propto P(l) \prod_{f\in F} P(f \mid l).$$

To increase numerical stability, we use sums of logarithms. Furthermore, we weight the probabilities with the inverse document frequency (IDF) of the features, and we omit the constant factor $P(F)$. The resulting classifier then is:

$$r(l, F) = \log P(l) + \sum_{f\in F} \log(\mathrm{idf}(f_i)) \log P(f \mid l).$$

In FEMaLeCoP, the simplified probability functions[1] are approximated by

$$P(l) \approx D_l, \qquad P(f \mid l) \approx \begin{cases} c & \text{if } D_{l,f} = 0 \\ \frac{D_{l,f}}{D_l} & \text{otherwise} \end{cases}$$

where $D_{l,f}$ denotes the number of times l appeared among the training examples in conjunction with f, D_l denotes how often l appeared among all training examples, and c is a constant.

2.3 Generalised Classifiers

In our experiments, we found negative training examples to be crucial for internal guidance. Therefore, we generalised the classifier to represent the type of occurrences as a *commutative monoid.*

Definition 1. *A pair $(M, +)$ is a* monoid *if there exists a neutral element* $0 \in M$ *such that for all* $x, y, z \in M$, $(x + y) + z = x + (y + z)$ *and* $x + 0 = 0 + x = x$. *If furthermore* $x + y = y + x$, *then the monoid is* commutative.

[1] We omitted several constant factors. Furthermore, FEMaLeCoP considers also features of training examples that are *not* part of the features F, albeit this is a further derivation of the theoretical model.

The generalised classifier is instantiated with a commutative monoid $(M, +)$ and reads triples (F, l, o), which in addition to features and label now store the label occurrence $o \in M$. For example, if the classifier is to support positive and negative examples, then one can use the monoid $(\mathbb{N} \times \mathbb{N}, +_2)$, where the first and second elements of the pair represent the number of positive respectively negative occurrences, the $+_2$ operation is pairwise addition, and the neutral element is $(0, 0)$. A triple learnt by this classifier could be $(F, l, (1, 2))$, meaning that l occurs with F once in a positive and twice in a negative way. Commutativity imposes that the order in which the classifier is trained does not matter.

We now formally define D_l (occurrences of label), $D_{l,f}$ (co-occurrences of label with feature) and idf (inverse document frequency):

$$\begin{aligned} D_l &= \sum \{o \mid (F, l', o) \in D, l = l'\}, \\ D_{l,f} &= \sum \{o \mid (F, l', o) \in D, l = l', f \in F\}, \\ \mathrm{idf}(f) &= \frac{|D|}{|\{(F, l', o) \mid (F, l', o) \in D, f \in F\}|} \end{aligned}$$

With this, our classifier for positive and negative examples can be defined as follows:

$$P(l) = \frac{|p - n|}{p + n}(c_p p + c_n n), \qquad P(f_i \mid l) = \begin{cases} c & \text{if } D_{l,f} = 0 \\ c_p \frac{p_f}{p} + c_n \frac{n_f}{n} & \text{otherwise} \end{cases}$$

where $(p, n) = D_l$, $(p_f, n_f) = D_{l,f}$, and c, c_p, and c_n are constants. The term $\frac{|p-n|}{p+n}$ represents *confidence* and models our intuition that labels which appear always in the same role (say, as positive example) should have a greater influence than more ambivalent labels. For example, if a label occurs about the same number of times as positive and as negative example, confidence is approximately 0, and when a label is almost exclusively positive or negative, confidence is 1.

We call D_l, $D_{l,f}$, and idf classification data. They are precalculated to allow fast classification. Furthermore, new training examples can be added to existing classification data efficiently, similarly to [KU15].

3 Learning Scenarios

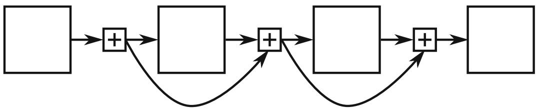
In this section, we still consider ATPs as black boxes, taking as input a problem and classification data for internal guidance, returning as output training data (empty if the ATP did not find a proof).

We propose two different scenarios to generate training data and to use it in subsequent proof searches, see Fig. 1:

– On-line learning: We run the ATP on every problem with classification data. For every problem the ATP solves, we update the classifier with the training data from the ATP proof.

– Off-line learning: We first run the ATP on all problems without classification data, saving training data for every problem solved. We then create classification data from the training data and rerun the ATP with the classifier on all problems.

While the second scenario can be parallelised, thus taking less wall-clock time, it has to treat every problem twice in the worst case (namely when every problem fails), thus taking up to double the CPU time of the first scenario.


(a) Online learning.

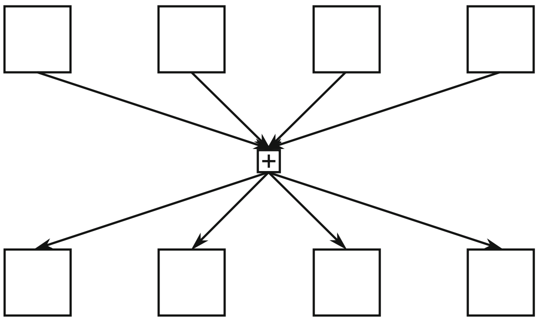

(b) Offline learning.

Fig. 1. Comparison of online and offline learning. The large boxes symbolise an ATP proof search, which takes classifier data and returns training data (empty if no proof found). The small "+" boxes combine classifiers and training data, returning new classifier data.

4 Internal Guidance for Given-Clause Provers

Variants of the given-clause algorithm are commonly used in refutation-based ATPs, such as Vampire [KV13] or E [Sch13].[2] We introduce a simple version of the algorithm: Given an initial set of clauses to refute, the set of *unprocessed* clauses is initialised with the initial set of clauses, and the set of *processed* clauses is the empty set. At every iteration of the algorithm, a *given clause* is selected from the unprocessed clauses and moved to the processed clauses, possibly generating new clauses which are moved to the unprocessed clauses. The algorithm

[2] Technically, our reference prover Satallax does not implement a given-clause algorithm, as Satallax treats terms instead of clauses, and it interleaves the choice of unprocessed terms with other commands. However, for the sake of internal guidance, we can consider Satallax to implement a version of the given-clause algorithm. We describe the differences in more detail in Sect. 6.

terminates as soon as either the set of unprocessed clauses is empty or the empty clause was generated.

The integration of our internal guidance method into an ATP with given-clause algorithm involves two tasks: The recording of training data, and the ranking of unprocessed clauses, which influences the choice of the given clause. To reduce the amount of data an ATP has to load for internal guidance, we process training data and transform it into classification data outside of the ATP. We describe these tasks below in the order they are executed when no internal guidance data is present yet.

4.1 Recording Training Data

Recording training data can be done in different fashions:

- **In situ**: Information about clause usage is recorded every time an unprocessed clause gets processed. This method allows for more expressive prover state characterisation, on the other hand, we found it to decrease the proof success rate, as the recording of proof data makes the inference slower.
- **Post mortem**: Only when a proof was found, information about clause usage is reconstructed. As this method does not place any overhead on the proof search, we resorted to post-mortem recording, which is still sufficiently expressive for our purposes.

For every proof, we save: conjecture (if one was given), axioms A (premises given in the problem), processed clauses C, and clauses C_+ that were used in the final proof ($C_+ \subseteq C$). We call such information for a single proof a *training datum*. We ignore unprocessed clauses, as we cannot easily estimate whether they might have contributed to a proof.

4.2 Postprocessing Training Data

In our experiments, we frequently encounter clauses that are the same, differing only by containing different Skolem constants. To this end, we process the training data before creating classification data from it. We tried different techniques to handle Skolem constants, as well as other postprocessing methods:

- **Skolem filtering**: We discard clauses containing any Skolem constants.
- **Consistent Skolemisation**: We normalise Skolem constants inside all clauses, similarly to [UVŠ11]. That is, a clause $P(x, y, x)$, where x and y are Skolem constants, becomes $P(c_1, c_2, c_1)$.
- **Consistent normalisation**: Similarly to consistent Skolemisation, we normalise *all* symbols of a clause. That is, $P(x, y, x)$ as above becomes $c_1(c_2, c_3, c_2)$. This allows the ATP to discover similar groups of clauses, for example $a + b = b + a$ and $a * b = b * a$ both map to $c_1(c_2, c_3) = c_1(c_3, c_2)$, but on the other hand, this also maps possibly different clauses such as $P(x)$ and $Q(z)$ to the same clause. Still, in problem collections which do not share a common set of function constants (such as TPTP), this method is suitable.

– **Inference filtering**: An interesting experiment is to discard all clauses generated during proof search that are not part of the initial clauses.

We denote the consistent Skolemisation/normalisation of a clause c described above as $\mathcal{N}(c)$.

4.3 Transforming Training Data to Classification Data

For a given training datum with processed clauses C and proof clauses C_+, we define the corresponding classifier data to be:

$$\{(\mathcal{F}(c), c, (1, 0)) \mid c \in C_+\} \cup \{(\mathcal{F}(c), c, (0, 1)) \mid c \in C \setminus C_+\},$$

where $\mathcal{F}(c)$ denotes the features of a clause. We use the monoid $(\mathbb{N} \times \mathbb{N}, +_2, (0, 0))$ introduced in Sect. 2, storing positive and negative examples. The classifier data of the whole training data is then the (multiset) union of the classifier data of the individual training data.

4.4 Clause Ranking

This section describes how our internal guidance method influences the choice of unprocessed clauses using a previously constructed classifier.

At the beginning of proof search, the ATP loads the classifier. Some learning ATPs, such as E/TSM [Sch00], select and prepare knowledge relevant to the current problem before the proof search. However, as we store classifier data in a hash table, filtering irrelevant knowledge to the problem at hand would require a relatively slow traversal of the whole table, whereas lookup of knowledge is fast even in the presence of a large number of irrelevant facts. For this reason we do not filter the classification data per problem.

Then, at every choice point, i.e. every time the ATP chooses a clause from the unprocessed clauses C, it picks a clause $c \in C$ that maximises the clause rank $R(c, F)$, where

$$R(c, F) = r_{\mathrm{ATP}}(c) + r(\mathcal{N}(c), F)$$

and:

– $r_{\mathrm{ATP}}(c)$ is an ATP function that calculates the relevance of a clause with traditional means (such as weight, age, ...),
– F is the current prover state,
– $r(c, F)$ is the Naive Bayesian ranking function as shown in Sect. 2, and
– $\mathcal{N}(c)$ is the normalisation function as introduced in Subsect. 4.2.

5 Tuning of Guidance Parameters

We employed two different methods to automatically find good parameters for internal guidance, such as c, c_p, and c_n from Sect. 2.

5.1 Off-Line Tuning

Off-line tuning analyses existing training data and attempts to find parameters that give proof-relevant clauses from the training data a high rank, while giving proof-irrelevant clauses a low rank. To do this, we evaluate for every training datum the following formula, which adds for every proof-relevant clause the number of proof-irrelevant clauses that were ranked higher:

$$\sum_{c_+ \in C_+} |\{c \mid R(c, F) > R(c_+, F_+), c \in C \setminus C_+\}|,$$

where C and C_+ come from the training datum (see Subsect. 4.1), F and F_+ are the features of the prover states when c respectively c_+ were processed (we reconstruct these from the training datum), and R is the ranking formula from Subsect. 4.4.

In the end, we sum up the results of the formula above for all training data, and take the guidance parameters which minimise that sum.

5.2 Particle Swarm Optimisation

Particle Swarm Optimisation [KE95] (PSO) is a standard optimisation algorithm that can be applied to minimise the output of a function $f(\boldsymbol{x})$, where $\boldsymbol{x}$ is a vector of continuous values. A *particle* is defined by a location $\boldsymbol{x}$ (a candidate solution for the optimisation problem) and a velocity $\boldsymbol{v}$. Initially, p particles are created with random locations and velocities. Then, at every iteration of the algorithm, a new velocity is calculated for every particle and the particle is moved by that amount. The new velocity of a particle is:

$$\boldsymbol{v}(t+1) = \omega \cdot \boldsymbol{v}(t) + \phi_p \cdot \boldsymbol{r}_p \cdot (\boldsymbol{b}_p(t) - \boldsymbol{x}(t)) + \phi_g \cdot \boldsymbol{r}_g \cdot (\boldsymbol{b}_g(t) - \boldsymbol{x}(t)),$$

where:

- $\boldsymbol{v}(t)$ is the old velocity of the particle,
- $\boldsymbol{b}_p(t)$ is the location of the best previously found solution among all particles,
- $\boldsymbol{b}_g(t)$ is the location of the best previously found solution of the particle,
- $\boldsymbol{r}_p$ and $\boldsymbol{r}_g$ are random vectors generated at every evaluation of the formula, and
- $\omega = 0.4$, $\phi_p = 0.4$, and $\phi_g = 3.6$ are constants.

We apply PSO to optimise the performance of an ATP on a problem set S. For this, we define $f(\boldsymbol{x})$ to be the number of problems in S the ATP can solve with a set of flags being set to $\boldsymbol{x}$ and with timeout t. We then run PSO and take the best global solution obtained after n iterations. We fixed $t = 1s$, $p = 300$, and $|S| = 1000$. The algorithm has worst-case execution time $t \cdot p \cdot n \cdot |S|$.

6 Implementation

We implement our internal guidance in Satallax version 2.8. Satallax is an automated theorem prover for higher-order logic, based on a tableaux calculus with extensionality and choice. It is written in OCaml by Brown [Bro12]. Satallax implements a priority queue, on which it places several kinds of proof search commands: Among the 11 different commands in Satallax 2.8, there are for example proposition processing, mating, and confrontation. Proof search works by processing the commands on the priority queue by descending priority, until a proof is found or a timeout is reached. The priorities assigned to these commands are determined by *flags*, which are the settings Satallax uses for proof search. A set of flag settings is called a *mode* (in other ATPs frequently called *strategies*) and can be chosen by the user upon the start of Satallax. Similar to other modern ATPs such as Vampire [KV13] or E [Sch13], Satallax also supports timeslicing via *strategies* (in other ATPs frequently called *schedules*), which define a set of modes together with time amounts Satallax calls each mode with. Formally, a strategy is a sequence $[(m_1, t_1), \ldots, (m_n, t_n)]$, where m_i is a mode and t_i the time to run the mode with. The total time of the strategy is the sum of times, i.e. $t_\Sigma(S) = \sum_{(m,t) \in S} t$.

As a side-effect of this work, we have extended Satallax with the capability of loading user-defined strategies, which was previously not possible as strategies were hard-coded into the program. Furthermore, we implemented modifying flags via the command line, which is useful e.g. to change a flag among all modes of a strategy without changing the flag among all files of a strategy. We used this extensively in the automatic evaluation of flag settings via PSO, as shown in Subsect. 5.2.

When running Satallax with a strategy S and a timeout t_{max}, then all the times of the strategy are multiplied by $\frac{t_{max}}{t_\Sigma(S)}$ if $t_{max} > t_\Sigma(S)$, to divide the time between modes appropriately when running Satallax for longer than what the strategy S specifies. Then, every mode m_i in the strategy is run sequentially for time t_i until a proof is found or the timeout t_{max} is hit.

An analysis of several proof searches yielded that on average, more than 90 % of commands put onto the priority queue of Satallax are proposition processing commands, which correspond to processing a clause from the set of unprocessed clauses in given-clause provers. For that reason, we decided to influence the priority of proposition processing commands, giving those propositions with a high probability of being useful a higher priority. The procedure follows the one described in Subsect. 4.4, but the ranking of a proposition is performed when the proposition processing command is put onto the priority queue, and the Naive Bayes rank is added to the priority that Satallax without internal guidance would have assigned to the command. As other types of commands are in the priority queue as well, we pay attention not to influence the priority of term processing commands too much (by choosing too large guidance parameters), as this can lead to disproportionate displacement of other commands.

To record training data, we use the terms from the proof search that contributed to the final proof. For this, Satallax uses `picomus` [Bie08] to construct a minimal unsatisfiable core.

To characterise the prover state of Satallax, we tried different kinds of features:

- Symbols of processed terms: We collect the symbols of all processed propositions at the time a proposition is inserted into the priority queue and call these symbols the features of the proposition. However, this experimentally turned out to be a bad choice, because the set of features for each proposition grows quite rapidly, as the set of processed propositions grows monotonically.
- Axioms of the problem: We associate every proposition processed in a proof search with all the axioms of the problem. In contrast to the method above, this associates the same features to all propositions processed during the proof search for a problem, and is thus more a characterisation of the problem (similar to TPTP characteristics [SB10]) than of the prover state.

In our experiments, just calculating the influence of these features without them actually influencing the priority makes Satallax prove less problems (due to the additional calculation time), and the positive impact of the features on the proof search does not compensate for the initial loss of problems. Therefore, we currently do not use features at all and associate the empty set of features to all labels, i.e. $\mathcal{F}(c) = \{\}$. However, it turns out that even without features, learning from previous proofs can be quite effective, as shown in the next section.

7 Evaluation

To evaluate the performance of our internal guidance method in Satallax, we used a THF0 [SB10] version (simply-typed higher-order logic) of the top-level theorems of the Flyspeck [HAB+15] project, as generated by Kaliszyk and Urban [KU14]. The test set consists of 14185 problems from topology, geometry, integration, and other fields. The premises of each problem are the actual premises that were used in the Flyspeck proofs, amounting to an average of 84.3 premises per problem.[3] We used an Intel Core i3-5010U CPU (2.1 GHz Dual Core, 3 MB Cache) and ran maximally one instance of Satallax at a time.

To evaluate the performance of the off-line learning scenario described in Sect. 3, we run Satallax on all Flyspeck problems, generating training data whenever Satallax finds a proof. We use the Satallax 2.5 strategy (abbreviated as "S2.5"), because the newest strategy in Satallax 2.8 can not always retrieve the terms that were used in the final proof, which is important to obtain training data.

As the off-line learning scenario involves evaluating every problem twice (once to generate training data and once to prove the problem with internal guidance), it doubles runtime in the worst case, i.e. if no problem is solved. Therefore, a user

[3] The test set, as well as our modified version of Satallax and instructions to recreate our evaluation, can be found under: http://cl-informatik.uibk.ac.at/~mfaerber/satallax.html.

might like to compare its performance to simply running the ATP with double timeout directly: When increasing the timeout from 1 s to 2 s, the number of solved problems increases from 2717 to 3394. However, this is mostly due to the fact that Satallax tries more modes, so to measure the gain in solved problems more fairly, we create a strategy "S2.5_1s" which contains only those modes that were already used during the 1 s run, and let each of them run about double the time. This strategy proves 2845 problems in 2 s.

We now compare the different postprocessing options introduced in Subsect. 4.2. For this, we create a classifier from the training data gathered during the 1 s run. We then run Satallax with internal guidance in off-line learning mode with 1 s timeout and the Satallax 2.5 strategy. We perform this procedure for each postprocessing option. We call a problem "lost" that Satallax with guidance could not solve and Satallax without guidance could. Vice versa for "gained". The results are given in Table 1. We perform best when influencing only the priority of axioms (inference filtering), solving 786 problems that could not be solved by Satallax in 1 s without internal guidance.

Table 1. Comparison of postprocessing options.

Postprocessing	Solved	Lost	Gained
Consistent normalisation	1911	920	114
Consistent Skolemisation	1939	885	107
None	2166	688	137
Skolem filtering	3395	98	776
Inference filtering	3428	75	786

To evaluate online learning, we run Satallax on all Flyspeck problems by ascending order, accumulating training data and using it for all subsequent proof searches. We filter away terms in the training data that contain Skolem variables. As result, Satallax with online learning, running 1 s per problem, solves 3374 problems (59 lost, 716 gained), which is a plus of 24 %.

In the next experiment, we evaluate the prover performance with the "S2.5_1s" strategy and a timeout of 30 s. For this, we use an 48-core server with 2.2 GHz AMD Opteron CPUs and 320 GB RAM, running 10 instances of Satallax in parallel. First, we run Satallax without internal guidance for 30 s, which solves 3097 problems. Next, we create from the training data a classifier with Skolem filtering, which takes 3 s and results in a 1.8M file. Finally, we run Satallax with internal guidance in off-line learning mode using the classifier. This proves 4028 problems in 30 s, which is a plus of 30 %. Results are shown in Fig. 2. The "jumps" in the data stem from changes of modes.

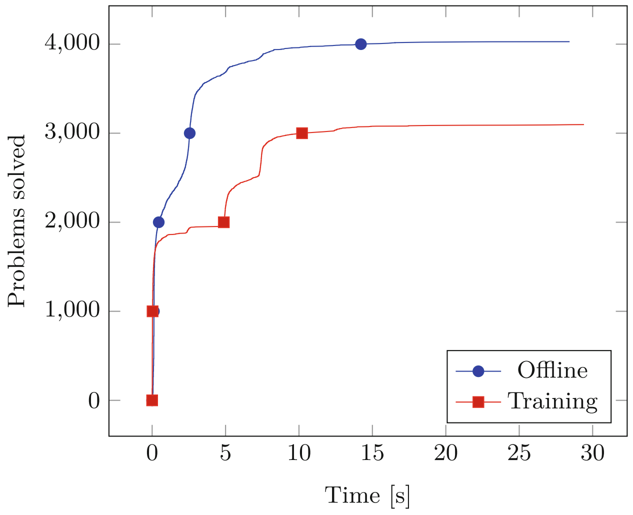


Fig. 2. Problems solved in a certain time (Color figure online).

8 Conclusion

We have shown how to integrate internal guidance into ATPs based on the given-clause algorithm, using positive as well as negative examples. We have demonstrated the usefulness of this method experimentally, showing that on a given test set, we can solve up to 26 % more problems. ATPs with internal guidance could be integrated into hammer systems such as Sledgehammer (which can already reconstruct Satallax proofs [SBP13]) or HOL(y)Hammer [KU14], continually improving their success rate with minimal overhead. It could also be interesting to learn internal guidance for ATPs from subgoals given by the user in previous proofs. Currently, we learn only from problems we could find a proof for, but in the future, we could benefit from considering also proof searches that did not yield proofs. Furthermore, it would be interesting to see the effect of negative examples on existing ATPs with internal guidance, such as FEMaLeCoP. We believe that finding good features that characterise prover state are important to further improve the learning results.

Acknowledgements. We would like to thank Sebastian Joosten and Cezary Kaliszyk for reading initial drafts of the paper, and especially Josef Urban for inspiring discussions and inviting the authors to Prague. Furthermore, we would like to thank the anonymous IJCAR referees for their valuable comments.

This work has been supported by the Austrian Science Fund (FWF) grant P26201 as well as by the European Research Council (ERC) grant AI4REASON.

References

[Bie08] Biere, A.: PicoSAT essentials. JSAT **4**(2–4), 75–97 (2008)

[Bro12] Brown, C.E.: Satallax: an automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 111–117. Springer, Heidelberg (2012)

[HAB+15] Hales, T.C., Adams, M., Bauer, G., Dang, D.T., Harrison, J., Le Hoang, T., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, A.H.T., Tran, T.N., Trieu, D.T., Urban, J., Vu, K.K., Zumkeller, R.: A formal proof of the Kepler conjecture. CoRR, abs/1501.02155 (2015)

[HV11] Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 299–314. Springer, Heidelberg (2011)

[KE95] Kennedy, J., Eberhart, R.: Particle swarm optimization. In: IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948, November 1995

[KSUV15] Kaliszyk, C., Schulz, S., Urban, J., Vyskocil, J.: System description: E.T. 0.1. In: Felty, A.P., Middeldorp, A. (eds.) CADE-25. LNCS (LNAI), vol. 9195, pp. 389–398. Springer, Heidelberg (2015)

[KU14] Kaliszyk, C., Urban, J.: Learning-assisted automated reasoning with Flyspeck. J. Autom. Reasoning **53**(2), 173–213 (2014)

[KU15] Kaliszyk, C., Urban, J.: FEMaLeCoP: fairly efficient machine learning connection prover. In: Davis, M., et al. (eds.) LPAR-20 2015. LNCS, vol. 9450, pp. 88–96. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48899-7_7

[KV13] Kovács, L., Voronkov, A.: First-order theorem proving and VAMPIRE. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013)

[Kü14] Daniel, A.K.: Machine learning for automated reasoning. Ph.D. thesis, Radboud Universiteit Nijmegen, April 2014

[Ott08] Otten, J.: leanCoP2.0 and ileanCoP1.2: high performance lean theorem proving in classical and intuitionistic logic (system descriptions). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 283–291. Springer, Heidelberg (2008)

[SB10] Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. J. Formalized Reasoning **3**(1), 1–27 (2010)

[SBP13] Sultana, N., Blanchette, J.C., Paulson, L.C.: LEO-II, Satallax on the Sledgehammer test bench. J. Appl. Logic **11**(1), 91–102 (2013)

[Sch00] Schulz, S.: Learning Search Control Knowledge for Equational Deduction. DISKI, vol. 230. Akademische Verlagsgesellschaft Aka GmbH Berlin, Berlin (2000)

[Sch13] Schulz, S.: System description: E 1.8. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR-19 2013. LNCS, vol. 8312, pp. 735–743. Springer, Heidelberg (2013)

[Urb15] Urban, J.: BliStr: the blind Strategy maker. In: Gottlob, G., Sutcliffe, G., Voronkov, A. (eds.) GCAI 32015, Global Conference on Artificial Intelligence. EPiC Series in Computing, vol. 36, pp. 312–319. EasyChair (2015)

[UVŠ11] Urban, J., Vyskočil, J., Štěpánek, P.: MaLeCoP machine learning connection prover. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011. LNCS, vol. 6793, pp. 263–277. Springer, Heidelberg (2011)

[Ver96] Veroff, R.: Using hints to increase the effectiveness of an automated reasoning program: case studies. J. Autom. Reasoning **16**(3), 223–239 (1996)