Learning Domain Knowledge to Improve Theorem Proving

Jörg Denzinger[1] and Stephan Schulz[2]

[1] Fachbereich Informatik, Universität Kaiserslautern, Postfach 3049, D-67663 Kaiserslautern, Germany, +49/6301/205-2181, denzinge@informatik.uni-kl.de
[2] Institut für Informatik, Technische Universität München, Arcisstrasse 20, D-80290 München, Germany, +49/89/521096, schulz@informatik.tu-muenchen.de

Abstract. We present two learning inference control heuristics for equational deduction. Based on data about facts that contributed to previous proofs, evaluation functions learn to select equations that are likely to be of use in new situations. The first evaluation function works by symbolic retrieval of generalized patterns from a knowledge base, the second function compiles the knowledge into abstract term evaluation trees. We analyze the performance of the two heuristics on a set of examples and demonstrate their usefulness. We also show that these strategies are well suited for cooperation in the framework of the knowledge based distribution method teamwork.

1 Introduction

The last years have seen a steady increase in the power of automatic theorem provers. There are a couple of reasons for this trend. The most significant ones are hardware improvements, refined inference engines, and stronger guiding heuristics. However, despite these advances, theorem provers still cannot rival humans. They perform basic inferences much faster than mathematicians, but are unable to cope with the complexities of the search space for many interesting problems.

One possible solution to this problem is to enable automatic systems to acquire knowledge about good proof strategies by learning from examples. This resembles human behaviour: Basic operations of the calculus are gained first, then proof strategies are learned from presented examples and own experiences. In this paper we address this approach for a distributed equational theorem prover. The required concepts are introduced in section 2: Unfailing completion ([HR87], [BDP89]), the teamwork method for distributing search processes ([AD93], [De95]), and proof recording and extraction ([DS94a], [DS94b]).

The task of learning heuristics for theorem provers is highly non-trivial. The first problem is the selection of the choice points to be controlled by a learning heuristic. If this selection has been made, training examples of good and bad decisions have to be obtained. In particular, it is necessary to distinguish among good and bad decisions and to find an adequate representation for these decisions. We solve this problem by analyzing proof protocols and by representing knowledge about proofs in the form of generalized *annotated patterns* of facts contributing to proofs. These problems are discussed in section 3.

Proof systems exhibit a very complex behaviour. We cannot expect a single chunk of knowledge to lead to good behaviour in all situations. Instead, we need to find ways to deal with conflicting information and to combine many different pieces of knowledge. We will address this problem in two ways: By employing the teamwork distribution method to combine different strategies, and by using strategies that integrate a strong general purpose heuristic with learned knowledge (section 4).

In order to evaluate our approach we present a couple of experiments in section 5 before concluding with some final remarks.

2 Distributed Equational Theorem Proving

2.1 Unfailing Completion

We introduce some concepts of equational reasoning, using standard notations. The reader is assumed to be familiar with rewriting techniques.

An equational theorem prover is a system trying to deal with the following problem: Given a set **E** of equations, is an equation $s = t$ a logical consequence of **E** (written as $s =_{\mathrm{E}} t$)?

A completion based prover tries to decide the **E**-equality of two terms by generating a (ground) confluent and terminating system of rules and equations for **E**. Then any valid equation can be proved by reducing both sides into a common normal form. By following certain fairness criteria one can guarantee that any valid equation will be proved by an intermediate system after only a finite number of inference steps.

Our DISCOUNT system ([ADF95]) is based on the inference rules for *unfailing completion* ([BDP89], [HR87]). We use three sets of term pairs to represent the current state of a completion process: A set **E** of processed, but unorientable equations, a set **R** of rules (processed and oriented equations) and a set **CP** of unprocessed equations. Rules are generated by orienting equations according to a *ground reduction ordering* $>$. New equations are generated by building *critical pairs* between existing rules and equations.

The completion algorithm will start out with empty sets **R** and **E**, and the initial axioms in **CP**. It will examine each equation in **CP**, reduce it to normal form with respect to **E** and **R**, use it to build new critical pairs (to be added to **CP**) and to eliminate redundancies from **R** and **E** by simplification. It will then be added to either **R** (if it can be oriented according to $>$) or **E**.

To build a prover on top of this algorithm the goal will be reduced to normal form with respect to each **E** and **R**. If these normal forms are identical or subsumed by an equation from **E**, the goal is proved. Both completeness and efficiency of the proof process depend on the order in which the equations from **CP** are considered, with both goals often conflicting.

The selection of critical pairs is usually guided by an evaluation function. An example is the *AddWeight* function, which simply counts function symbols and variables in an equation (compare [Hu80]). We found it useful to count function symbols with twice the weight of variable symbols.

Fuchs in [Fu95a] and [Fu95b] uses previous proofs to adapt parameters for generic evaluation functions by assigning different weights to different function symbols. Promising results are reported, however, a single suitable training example (with a matching signature) has to be provided by the human user.

2.2 The Teamwork Method

The teamwork method is a knowledge-based distribution concept for a certain kind of search processes (see [De95]). Unfailing completion is a process of this kind. Our teamwork-based system for unfailing completion employs four types of components that use different kinds of knowledge and work together following a very strict interaction scheme.

Experts are completion procedures using different selection heuristics for critical pairs, thus exploring different parts of the search space. *Specialists* classify sets of equations and can add known consequences of a recognized set. *Referees* judge whole sets of equations (e.g. the work done by an expert), as well as single equations. The *supervisor* periodically generates a new common starting point for the experts by integrating the search state of the best expert, the best equations of the other experts, and the results of the specialists. It can also select new experts and specialist for work in the team. This task is controlled by *reactive planning*, and allows the team to adapt itself to a given problem (see [DK94]).

The interaction of the components is organized as a cycle. In the working phase of this cycle, the experts and specialists work independently and in parallel. In the judgment phase, the first phase of a *team meeting*, the referees evaluate the results of the experts and the specialists. This again can be done in parallel. In the second phase of a team meeting, the planning phase, the supervisor receives the results of the specialists and the referee reports and generates the new starting state. It then uses information from the referee reports, results of the specialists, and planning knowledge to select the next team.

We found the teamwork method to be an ideal foundation for theorem provers that want to use learned knowledge. The referees allow the judgment of the different parts of knowledge that are represented by the experts. Specialist can be used for checking whether a chunk of knowledge can be applied in a given situation. The use of contradictory control knowledge is also possible, since multiple experts can be used and their work is evaluated. Teams can contain experts with strong general purpose heuristics. These experts provide the means to "glue together" the work of other, more specialized experts.

2.3 Recording and Extracting Proofs

One basic requirement for learning from successful proof attempts is the ability to analyze a proof in detail. Not every protocol method that is used by theorem provers provides for this case. Getting sufficiently complete data from a distributed prover is not trivial. We solved this problem as follows (also see [DS94b]):

In default mode, our prover protocols only enough information to reproduce the proof. When we require a detailed proof listing, the necessary information is gathered in a *reproduction run* based on the information from the previous run. During the reproduction run each inference that is made is recorded as a PCL-step[3] The resulting file (or files in case of a distributed run) can serve as a base for learning approaches using both positive and negative examples. A complete protocol is, however, too large for manual analysis, and contains much information needed only in special cases. Thus, starting at the end of the listing, our proof extraction tool strips the original file of all steps not contributing to the proof. The resulting file is used in our learning approach (which requires positive examples only). It also can be transformed into a proof suitable for human understanding

3 Knowledge Acquisition for Proof Heuristics

As we already stated in section 2.1, the order in which unprocessed equations (mainly critical pairs, but also initial axioms, and results from the interreduction process) are selected for processing is the single most important choice for the efficiency of a completion based proof system. Experiments and actual profiling show us, that even operations with a high theoretical complexity, like unification and rewriting, only need small percentages of the total time in practical applications.

The heuristics for the selection of critical pairs, on the other hand, typically influence the time needed to find a proof by several orders of magnitude. Simple analysis and some experiments show us, that, given an optimal evaluation function for critical pair selection, even the most complex among our test problems can be solved in trivial amounts of time. In order to improve the overall efficiency of a proof system, it is therefore sufficient to improve the evaluation function used to select the next facts to process[4]. Concentrating on this single most important choice point in the completion algorithm has a number of advantages:

1. The learning component has a very clean and well-defined interface to the inference engine. Changes to the internal structure of the proof system are not necessary.
2. Conventional heuristics use the same interface. Thus, learning and non-learning heuristics can be integrated easily.
3. Examples for the learning algorithm can be easily generated. As only equations need to be evaluated, no context information is required. Equations useful in a proof are positive examples, while useless equations are negative examples. Positive examples can easily be weighted according to their importance in a proof.

The choice of a critical pair evaluation function as the target for learning also has some less advantageous consequences.

1. The number of critical pairs evaluated during a typical proof run is very high. Thus, to avoid a large overhead for the learning strategy, a single evaluation

[3] PCL (Proof Communication Language) is a format that provides enough information for detailed analysis.
[4] Similar results can be found for resolution based provers and other systems.

has to be very cheap in terms of CPU time. This fact will influence our choice of the knowledge representation.
2. As only a high level choice point is controlled by the learning strategy, many quite complex operations (normal form calculation, critical pair generation) are performed as usual. This implies that even an optimal selection strategy will still produce some overhead. We accept this for two reasons: First, while the percentage of additional work can be quite large, the total amount of time involved is rather small. As we stated above, given an optimal critical pair selection strategy, most known problems can be solved in a few seconds despite this overhead. The more important reason is, however, that any strategy that tries to control the completion at a lower level needs to make very much implicit knowledge about the state of completion explicit. The resulting flood of information would be completely unmanageable.

3.1 Representing Knowledge about Proofs

When deciding on an appropriate representation for the knowledge about proofs we have to keep the following points in mind.

1. Knowledge retrieval at run time should be cheap. As much work as possible should be done off-line, while building the knowledge base.
2. Completion-based proof systems are usually limited by memory constraints. Most proof attempts fail not for lack of time, but for lack of memory. To avoid an additional burden on the available memory, the knowledge representation should be as compact as possible.
3. The knowledge should be stored in a generalized form and abstract from irrelevant details like the concrete signature used in a given proof attempt.

In our experience, the structure of an equation is much more important for the evaluation than the concrete function symbols and their role in the axiomatization[5]. We therefore decided to store knowledge about proofs in the form of *annotated patterns* of positive examples (equations that contributed to a proof) only. Storing negative examples was ruled out by the memory constraints. The number of unnecessary equations in a medium difficulty proof is typically about 3 orders of magnitude greater than the number of contributing equations (compare [DS94a] and [DS94b] for exact numbers). Patterns serve both as an easily computable abstraction of equations and allow us to implement efficient knowledge retrieval mechanisms.

Definition 1 : Representative Term Patterns. Let t be a term over some signature. Let $T = \{f_1, f_2, \ldots\}$ and $X = \{x_1, x_2, \ldots\}$ be two disjoint ordered sets of symbols. The *representative pattern* of t with regard to T and X is constructed by substituting the function symbols and the variables in t with symbols from T and X, respectively, in their order of appearance. Thus, the representative pattern of the term $t \equiv f(g(x), g(y))$ with respect to T and X would be $t\prime \equiv f_1(f_2(x_1), f_2(x_2))$. We call the representative pattern of a term t $\mathtt{pat}(t)$.

[5] Later experiments confirmed this view and showed us that even stronger abstractions are possible. Compare section 4.2.

The two orderings on T and X can be combined into a total ordering $>$ on $T \cup X$ by assuming $f_i > x_i$ for all $f_i \in T$ and all $x_i \in X$. This total ordering obviously induces a total lexicographical ordering $>_{pat}$ on term patterns as well[6]. $>_{pat}$ is extended to a quasi-ordering on all terms by assuming $s \geq_{pat} t$ iff $\mathtt{pat}(s) \geq_{pat} \mathtt{pat}(t)$ for arbitrary terms s and t. Using this ordering, we can define representative patterns for term pairs (rules or equations).

Definition 2 : Representative Patterns for Equations. Assume T and X as above. The *representative pattern* for an equation $s = t$ is generated by first orienting the equation according to $>_{pat}$ and then substituting the function symbols and the variables in $s = t$ with symbols from T and X as above. We again write $\mathtt{pat}(s = t)$ to denote the representative pattern of an equation.

Please note that the representative patterns for both terms and equations are unique. $>_{pat}$ is extended to patterns for equations by lexicographic composition. It remains total on these patterns. Thus the ordering can be used to efficiently implement data structures for the retrieval of patterns (see next section).

As we stated above, we aim at a compact knowledge representation. Thus, for each contributing equation we insert the corresponding pattern into the knowledge base, annotated with the proof it occurred in and the number of times the equation was used in this proof. If the same pattern appears more than once, the resulting annotations are simply concatenated. Thus, our knowledge about proofs consists of a set of patterns for equations, annotated with information about the occurrence of the corresponding equations in the proofs known to the system. The number of different proofs an equation occurred in can be used as an indicator for its versatility (i.e. its ability to contribute to multiple proofs), while the total number of references to an equation is a measure for the overall importance of the equation in a domain.

3.2 Structure of the Knowledge Base

The collected knowledge of many proofs is organized in a way to allow efficient retrieval and easy maintenance. For this purpose, we keep three different sets of annotated patterns and a list of the original proofs that were used to generate the knowledge base.

The first set contains patterns organized by theorem. All equations used in proofs of theorems with the same structure are inserted into a single subset indexed by the pattern representing the theorem. A second set of equations is indexed not by the theorem, but by the axiomatization of problems these equations were successfully used in. Finally, we keep a flat set of patterns which are not associated with particular characteristics of the proof they occurred in.

Each subset of patterns is represented by an AVL tree (a partially balanced binary search tree) built using the ordering $>_{pat}$. As $>_{pat}$ can be calculated very fast (using pre-calculated weights) and as AVL trees are very efficient for

[6] For reasons of efficiency we use a slightly different ordering in the implementation, comparing first pre-calculated term weights and using the lexicographical ordering only to break ties.

data retrieval, patterns can be found in the sets at very little cost in CPU time. Moreover, we are able to store pre-calculated trees in a way that allows rebuilding of the trees without balancing operations. Thus, even the time needed to read a part of the knowledge base from disk is minimized.

AVL trees store data with very little memory overhead. Moreover, we only need to read the relevant part of the knowledge base into the computers memory. Thus, this organization of the knowledge allows us to conserve both CPU time and memory, fulfilling the two main constraints required for our system.

3.3 The Knowledge Acquisition Cycle

Most learning systems that aim at improving the performance of an agent can be described by a circular model. We will now present the *learning cycle* for the DISCOUNT system. Fig. 1 gives an overview over this model.

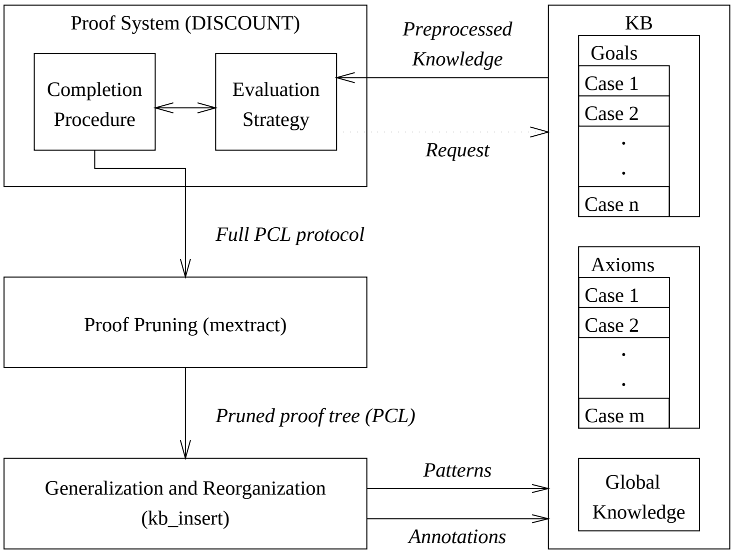

Fig. 1. Learning Model

In the first step, the prover is used to successfully prove a theorem, using either a learning strategy, a conventional one, or, in the case of a distributed teamwork proof, a combination of multiple strategies, all of which can be either learning or conventional. The steps actually contributing to the proof are then extracted from the PCL protocol of the proof (see [DS94b]). The equations used in these steps are generalized to their representative patterns and are inserted into the knowledge base as described in the previous sections. This updated knowledge base can now be used for proof reproduction purposes or to aid in the finding of new proofs.

4 Using the Knowledge Base

The previous section described the construction of the knowledge base. Now we present two very different strategies that make use of the accumulated knowledge.

We will, in this paper, concentrate on strategies that use the whole knowledge base and make no use of the indexing by theorem or axioms. The indexing techniques only become important when the knowledge base grows significantly.

4.1 Simple Retrieval

We call the first learning evaluation strategy `global_learn`. It searches the knowledge base for an entry equivalent to the representative pattern of the equation to be judged and uses the annotations to compute an evaluation.

Assume $s = t$ is the critical pair to be evaluated. The strategy assigns a weight to this critical pair as follows: If $\texttt{pat}(s = t)$ is found in the knowledge base, then it returns the weight

$$W_{gl}(s = t) = AddWeight(s = t) - (scale \times (to \times to_w + pr \times pr_w + av \times av_w)) \quad (1)$$

where $AddWeight(s = t)$ is the term weight of the equation (as described above), $scale$ is an arbitrary weighting factor (set to 20 in our implementation), and to, pr and av are the total number of times the fact has been referenced in any proof, the number of proofs it was used in, and the average number of applications it had in a proof. to_w, pr_w and av_w serve as adjustable weights for the previous three components. If the pattern is not found, the strategy simply returns $AddWeight(s = t) + pen$, where pen is a penalty large enough to ensure that critical pairs found in the knowledge base are nearly always preferred.

Thus, if all the weighting factors are positive numbers, the strategy will prefer known pairs to unknown ones, and it will furthermore show a preference for pairs that appeared more often in proofs covered by the knowledge base.

We found this strategy to be excellent for the reproduction of proofs already covered by the knowledge base. It also leads to improved behaviour for many new, unknown proofs. However, this extrapolative power is most pronounced in cases where a lot of previous examples from a domain are known. It is quite limited if only a few examples have been inserted into the knowledge base. For experimental results see section 5.

4.2 Strong Abstraction

The `global_learn` strategy uses learned knowledge only to evaluate term pairs that correspond to existing patterns. It uses a conventional back-up strategy in all other cases. We now present a strategy that uses the knowledge base to define measures of quality on *all* terms.

For the `1_tet_learn` strategy, the complete knowledge is compiled into a single *term evaluation tree* (*TET* for short). This TET is then used as a template to evaluate both sides of the equation to be judged. The results are combined into a single weight for the equation.

A TET represents a number of possible alternatives for the structure of terms. Each of these alternatives stands for all terms with a given number of principal subterms and is represented by a *term arity tree* or TAT. The alternatives for a given subterm are again represented by a TET. More exactly, TETs and TATs are defined by mutual recursion:

Definition 3 : TETs and TATs. A TET is a (possibly empty) set of TATs. A TAT is a tuple $(arity, count, info, (tet_1, \ldots, tet_{arity}))$, where the tet_i are non-empty TETs, and $arity$, $count$ and $info$ are elements from $\mathbb{N}$. We call $info$ an *annotation* to the tree node. $arity$ gives the arity of the TAT.

Only finite TETs are of interest to us, as only finite TETs are needed to represent finite sets of terms. Terms are inserted into a TET by a recursive function.

Definition 4 : Inserting terms into TETs. Let $T = (tat_1, \ldots, tat_m)$ be a TET and let $t \equiv f(t_1, \ldots, t_n)$ be a ground term with n principal subterms. Assume $info \in IN$. We call a tuple $(t, info)$ a *term with additional data.* The function `ins` accepts a TET and a ground term with additional data and returns a TET. If none of the tat_i has arity n, a new TAT is added and the subterms are inserted recursively:

$\texttt{ins}(T, (t, info)) := T \cup$
$\quad \{(n, 1, info, (\texttt{ins}(\{\}, (t_1, info)), \ldots, \texttt{ins}(\{\}, (t_n, info))))\}$

Otherwise, there exists a $tat_i \equiv (n, count, info', (tet_1, \ldots, tet_n))$. In this case the annotation of the term is merged into the tat_i. Again, the subterms are inserted by recursion:

$\texttt{ins}(T, (t, info)) := \{tat_1, \ldots, tat_{i-1}, tat_{i+1}, \ldots, tat_m\} \cup$
$\quad \{(n, count+1, info+info'), (\texttt{ins}(tet_1, (t_1, info)), \ldots, \texttt{ins}(tet_n, (t_n, info)))\}$

`ins` is extended to non-ground terms by treating variables as function symbols of arity 0.

Insertion of terms in TETs is associative, the resulting tree depends only on the terms (and data) inserted into it, not on the particular order of the terms. Thus, it makes sense to speak of the representative TET for a set of equations.

Definition 5 : Representative Term Evaluation Trees. Assume a finite set $S = \{(t_1, info_1), \ldots, (t_n, info_n)\}$ of terms with additional data. $\texttt{rep\_TET}(S)$ is the unique TET generating by inserting the terms from S into an empty TET. For a set $D = \{(s_1 = t_1, info_1), \ldots, (s_n = t_n, info_n)\}$ of equations with data $\texttt{rep\_TET}(D) = \texttt{rep\_TET}(\{(s_1, info_1), (t_1, info_1), \ldots, (s_n, info_n), (t_n, info_n)\}$.

The `1_tet_learn` strategy builds the representative TET of the patterns found in the knowledge base, using the number of proofs the corresponding equations appeared in as additional data. This TET is used to evaluate equations using a number of measures.

Definition 6 : TET-defined term measures. Let $T = \{tat_1, \ldots, tat_m\}$ be a TET and let $t = f(t_1, \ldots, t_n)$ be a term whose top function symbol has arity n. T defines three measures on t: If there is no tat_i with arity n in T, then

- $\texttt{eval}_{\texttt{W}}(t, T) = pen_u \times AddWeight(t)$
- $\texttt{eval}_{\texttt{C}}(t, T) = 0$
- $\texttt{eval}_{\texttt{I}}(t, T) = 0$

Otherwise there is a $tat_i = (n, count, info, (tet_1, \ldots, tet_i)$ in the TET and

- $\texttt{eval}_{\texttt{W}}(t, T) = \sum_{i=1}^{n} \texttt{eval}_{\texttt{W}}(t_i, tet_i)$
- $\texttt{eval}_{\texttt{C}}(t, T) = count + \sum_{i=1}^{n} \texttt{eval}_{\texttt{C}}(t_i, tet_i)$
- $\texttt{eval}_{\texttt{I}}(t, T) = info + \sum_{i=1}^{n} \texttt{eval}_{\texttt{W}}(t_i, tet_i)$

The three measures calculate the weight of subterms not represented in the TET (with a penalty factor pen_u), the number of times an inserted term was mapped to TET nodes corresponding to the term and the sum over the information stored at these TET nodes, respectively.

The `l_tet_learn` strategy calculates the weight of a term from three components: First, the conventional term weight, then the weight of the term nodes not covered by the TET and finally the information collected. We empirically found the following equation to yield good results:

$$W_{tet} = AddWeight(s=t) + eval_W(s,T) + eval_W(t,T) - l_{infl} \tag{2}$$

$$\times \left\lfloor \frac{\sqrt{p_w * (\texttt{eval}_{\texttt{I}}(s,T) + \texttt{eval}_{\texttt{I}}(t,T))}}{(s_{tet} * \texttt{eval}_{\texttt{C}}(s,T) + 1 + \texttt{eval}_{\texttt{C}}(t,T) + 1)} \right\rfloor$$

In this equation, l_{infl} is a weighting factor for the knowledge found in the TET, p_w is a weighting factor for the collected addition data (number of proofs) and s_{tet} is a factor for the number of terms inserted into the TET.

The strategy resulting from this weight function has excellent generalization capabilities. A few proofs will suffice to generate a strong heuristic. However, the strategy is less good at pure reproductive tasks and will break down if too much knowledge is used in the TET. The reasons for this are not yet clear and will be subject of further work. Experimental results will be presented in the next section.

5 Experiments

We have performed a variety of experiments to evaluate the learning strategies, both isolated and as members of a team for the distributed prover.

Problems for (equational) theorem provers are often generated in a rather *ad hoc* fashion. A large number of small, unrelated problem classes did exist, but there was a distinct lack of problems scalable to various difficulties, or of a domain with a variety of problems covering a wide range of difficulties.

However, we are now in a more fortunate position. The DISCOUNT system was, among other provers, used for the construction of a proof pad for mathematicians (the ILF project – see [Da+94]). ILF was used on the test case of lattice ordered groups. In the course of the evaluation of ILF, a large number of problems from the domain of lattice ordered groups have been generated. We have, for some time now, used these examples from a field of active interest to mathematicians in the evaluation of DISCOUNT. The examples used have also been accepted into the TPTP problem library ([SSY95]) and appear in the latest release.

5.1 The Sequential Case

While our ultimate goal is to integrate the learning experts into a distributed teamwork team, the analysis of their performance is much easier for the sequential case. The influence of a single strategy can be studied in isolation. The sequential prover does not rely on critical timing for cooperation and its behaviour is therefore not influenced by the load imbalances which are not avoidable in a multi-user, multi-tasking environment. Results can be reproduced more easily. Moreover, a sequential test run ties up less system resources, allowing multiple test runs at the same time.

Methodology. We will use three different modes for the test runs. In all modes, DISCOUNT tries to prove all of the 86 selected examples. The difference is the composition of the knowledge base. *Explorative* mode aims at simulating a stand-alone learning system without any influence by the user. In a completely closed-loop approach, the examples are handed to the prover in a fixed order. If a proof is found, exactly this proof is handed back to the learning component, otherwise the knowledge base is unchanged.

The second, *persistent* mode, probably is the most realistic simulation of a proof system actually used by humans in a normal environment. A proof is added to the knowledge base regardless of whether the proof was found by the learning strategy or by any other means, including team runs (with and without learning team members) and proof attempts for more than 1000 seconds. Note that there are still 10 examples for which *no* proof was available at all. In these cases the knowledge base is obviously not changed.

Finally, in *reproductive* mode the prover starts with a knowledge base stocked with all the knowledge available about the domain. The knowledge base is constructed at the beginning of the test and does not change during it. This simulates an environment where the same problems reappear multiple times after it has been solved once.

For comparison we will also present the results for two conventional strategies: The *AddWeight* strategy (compare section 2.1) and the *Occnest* strategy ([DF94]). The *AddWeight* strategy is a very strong general purpose strategy that also serves as the base for the learning heuristics. The *Occnest* strategy is a goal oriented heuristic hand-tuned for strong performance in the domain of lattice ordered groups.

Results and Discussion. Table 1 shows the results of both the non-learning and the learning strategies in the three different modes. The first column gives the strategy and, for the learning strategies, the mode of learning. The second column gives the number of proofs found in 1000 seconds, and the final column gives the total CPU time for all examples, counting failures with 1000 seconds. All times are measured on a SPARCstation ELC.

Fig. 2 shows the performance of some selected strategies in more detail. In this figure we plot the total time spend on all examples versus the number of examples.

The `global_learn` heuristic in explorative mode performs similar to the standard *AddWeight*. Except for one case, both strategies prove the same examples.

Table 1. Results on 86 problems in the lattice ordered groups domain

Strategy		Successes	Total time
AddWeight		59	29564
Occnest		67	19569
`global_learn`[7]	(explorative)	58	28445
	(persistent)	67	19442
	(reproductive)	75	11266
`1_tet_learn`[8]	(explorative)	68	21833
	(persistent)	68	22607
	(reproductive)	57	31563

In this one case, a very difficult example cannot profit from the preceding, easy examples, and the (quite small) overhead of the learning strategy increases the total proof time to over 1000 seconds. Despite this one case, the overall performance of **`global_learn`** is slightly better than the performance of *AddWeight*.

In the more realistic persistent mode, the general purpose **`global_learn`** strategy performs significantly better and even beats the specialized *Occnest* strategy by a small margin. Both strategies do not prove the same set of problems – each strategy proves 4 problems the other is unable to prove. In reproductive mode the strategy performs still better. It can reproduce 74 of the inserted proofs very fast (all 75 proofs were found in only 266 seconds), and even solves one example where no prior proof had been found by any strategy. Two of the proofs inserted cannot be reproduced within the time limit. It should be noted that only TET-based strategies have been able to find proofs for these problems.

We found that the strategy based on term pair retrieval works better with more data. If only a few examples are available, the strategy works for examples similar to the ones in the knowledge base because the known term pairs are preferred. If more examples are inserted into the knowledge base, the strategy learns to stronger differentiate among the term pairs that occurred in known proofs. In this case the strategy also gains extrapolative power. Fig. 2 shows this effect. At first, the graph for the learning strategy in reproductive mode closely hugs the graph of *AddWeight*. Only with increasing knowledge the learning strategy outperforms the conventional one.

The **`1_tet_learn`** strategy performs very well in both explorative and persistent mode. It is the strategy that finds the most proofs. Again, the examples found by this strategy are not a pure superset of the examples found by *Occnest*. The strategy solves 4 examples no other strategy is able to prove. However, the strategy performs much worse in reproductive mode. The current version of a TET-based strategy cannot cope with too much knowledge. The performance graph of the strategy in extrapolative mode also demonstrates this. Most of its performance advantage is found near the middle of the graph, where enough knowledge for meaningful learning is available, not at the right side, where the

[7] We used `global_learn` with $to_w = 20$, $av_w = 0$ and $pr_w = 0$ for these results.
[8] For `1_tet_learn` the parameters were $l_{infl} = 10$, $p_w = 1$ and $pen_u = 5$.

strategies weakness for coping with too much knowledge becomes visible. We suspect one main reason for this weakness. Most terms inserted into the TET have a certain size limit. This leads to a very strong preference for terms below this size and thus makes the strategy overly conservative.

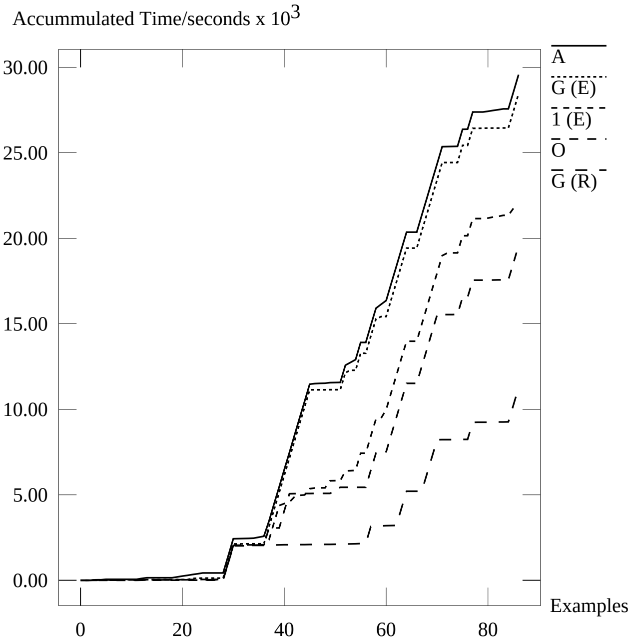

Fig. 2. Comparative behaviour of different strategies. Plotted are times for *AddWeight* (A), *Occnest* (O), `1_tet_learn` in extrapolative mode (I (E)) and `global_learn` in extrapolative (G (E)) and reproductive (G (R)) mode.

To summarize, we found the learning strategies to be quite powerful and very valuable even for a sequential prover. Even in explorative mode the learning strategies were able to find 4 previously unknown proofs, reducing the number of unsolvable examples for the sequential prover from 18 to 14. Moreover, the experiments in persistent mode show that the learning strategies can even profit from proofs found by very different strategies. Finally, the performance of `global_learn` in reproductive mode (where one additional proof was found) show that this strategy seems to scale very well and that further improvements can be expected for larger knowledge bases.

5.2 The Distributed System

In order to increase the power of the prover we integrate the learning experts into a teamwork team, to combine the improvements of the learning experts with the known synergetic benefits of teamwork. As a DISCOUNT expert is defined mainly by its evaluation strategies, this is easy to accomplish. The team specification language has been extended to allow the incorporation of the learning experts, and a team incorporating them has been created for test runs.

Evaluation Method. It is not reasonable to run a distributed team on trivial examples, because the overhead of starting the team would dominate the times. Thus, we split the set of examples into 2 parts – training examples and test examples. The training examples are all examples that a conventional strategy proves in less then 4 seconds. We used a constant knowledge base consisting of the 53 training examples and run both a learning team and a conventional team on the remaining 26 examples. Both teams use 2 processors. The conventional team uses a preliminary version of reactive planning ([DK94]) to select experts. The learning team adds `1_tet_learn` and `global_learn` as possible alternatives.

Due to the higher amount of resources required for a team run and the critical timing for distributed runs, we allowed 500 seconds CPU time on the initial team process for a single proof attempt.

Results. Table 5.2 shows the results for some test examples. Neither of the two teams could prove any of the other problems within the given time. Reported are seconds of real time, including communication overhead and idle times.

Table 2. Run times for the standard team and the learning team

Example	Standard	Learning	Example	Standard	Learning
lat2a	25.7	8.6	lat2b	33.3	9.1
lat4	–	81.1	p02a	–	–
p02b	–	–	p03a	9.9	9.2
p03d	9.8	8.0	p05a	39.2	7.6
p05b	39.7	7.8	p08b	–	97.1
p09a	55.8	31.0	p09b	57.9	30.6
p10	–	–	p11	–	–
p18	–	–	p21	–	–
p22b	20.6	6.6	p39a	25.3	15.9
p39b	17.3	15.5	p39c	17.6	15.4
p39d	26.0	15.7	p8_9a	–	–
p8_9b	7.7	–			

As the table shows, the learning team outperforms the conventional team for most examples, despite the fact that only simple examples are present in the knowledge base. The learning experts cooperate very well both with each other and with the conventional strategies.

6 Conclusion

All in all we believe to have promising results. Term pair structures of equations contributing to old proofs definitely carry information about good decisions and can be used to induce good evaluation strategies. We found unfailing completion to be particularly well suited for learning heuristics, as a single important choice point controls the efficiency of the process and offers a clean and simple interface to the learning procedure.

An efficient access function for stored patterns allows a very cheap evaluation of terms even in large knowledge bases. Future work in this direction should focus on even larger, more complete collections of proofs. These tests will show whether filters for restricting access to a large knowledge base will become necessary.

The TET-based strategies, which abstract from most properties of terms, can still achieve good results. Moreover, they require only a few examples to induce a strong evaluation function. Further research will hopefully show us how to eliminate their weakness in cases where much knowledge is available.

Teamwork has proved to be an excellent integrative principle for both learning and conventional strategies. It allows for a seamless transition from proof reproduction to from-scratch proving. Better integration of learning experts and the planning component should lead to further performance gains. Term pair evaluation is also used in teamwork's referees. Using the principles outlined in this paper, it should be possible to apply machine learning to this task as well.

Obviously, we do not claim to have found a perfect solution for guiding theorem provers. However, we believe to have made a good step in the right direction.

References

[AD93] **Avenhaus, J.; Denzinger, J.:** *Distributing Equational Theorem Proving*, Proc. 5th RTA, Montreal, LNCS 690, pp. 62-76, 1993; also available as SEKI-Report SR-93-06, University of Kaiserslautern, 1993.

[ADF95] **Avenhaus, J.; Denzinger, J.; Fuchs, M.:** *DISCOUNT: A System For Distributed Equational Deduction*, Proc. 6th RTA, Kaiserslautern, LNCS 914, pp. 397-402, 1995.

[BDP89] **Bachmair, L.; Dershowitz, N.; Plaisted, D.A.:** *Completion Without Failure*, Coll. on the Resolution of Equations in Algebraic Structures, Austin (1987), Academic Press, 1989.

[Da+94] **Dahn, B.I.; Gehne, J.; Honigmann, T.; Walther, L.; Wolf, A.:** *Integrating Logical Functions with ILF*, Internal report, Institut für Reine Mathematik, Humbold-University, Berlin, 1994.

[De95] **Denzinger, J.:** *Knowledge-Based Distributed Search Using Teamwork*, Proc. ICMAS-95, San Francisco, AAAI Press, pp. 81-88, 1995.

[DF94] **Denzinger, J.; Fuchs, M.:** *Goal Oriented Equational Theorem Proving Using Teamwork*, Proc. 18th KI-94, Saarbrücken, LNAI 861, pp. 343-354, 1994; also available as SEKI-Report SR-94-04, University of Kaiserslautern, 1994.

[DK94] **Denzinger, J.; Kronenburg, M.:** *Planning for Distributed Theorem Proving: The Teamwork Approach*, SEKI-Report SR-94-09, University of Kaiserslautern, 1994.

[DS94a] **Denzinger, J.; Schulz, S.:** *Analysis and Representation of Equational Proofs Generated by a Distributed Completion Based Proof System*, SEKI-Report SR-94-05, University of Kaiserslautern, 1994.

[DS94b] **Denzinger, J.; Schulz, S.:** *Recording, Analyzing and Presenting Distributed Deduction Processes*, Proc. 1st PASCO, Hagenberg/Linz, pp. 114-123, World Scientific Publishing, 1994.

[Fu95a] **Fuchs, M.:** *Learning Proof Heuristics by Adapting Parameters*, Proc. 12th ML, Morgan Kaufmann, pp. 235-243, 1995; also available as SEKI-Report SR-95-02, University of Kaiserslautern, 1995.

[Fu95b] **Fuchs, M.:** *Experiments in the Heuristic Use of Past Proof Experience*, SEKI-report SR-95-10, University of Kaiserslautern, 1995.

[HR87] **Hsiang, J.; Rusinowitch, M.:** *On Word Problems in Equational Theories*, Proc. 14th ICALP, Karlsruhe, LNCS 267, pp. 54-71, 1987.

[Hu80] **Huet, G.:** *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, J. of ACM 27, No. 4, pp. 798-821, 1980.

[KB70] **Knuth, D.E.; Bendix, P.B.:** *Simple Word Problems in Universal Algebras*, Computational Algebra, J. Leech, Pergamon Press, pp. 263-297, 1970.

[SSY95] **Sutcliffe, G.; Suttner, C.B.; Yemenis, T.:** *The TPTP Problem Library*, Proc. 12th CADE, Nancy, Springer LNAI 814, pp. 252-266, 1994

This article was processed using the LaTeX macro package with LLNCS style