Using Term Space Maps to Capture Search Control Knowledge in Equational Theorem Proving

Stephan Schulz and **Felix Brandt**
Institut für Informatik, Technische Universität München, Germany
{schulz,brandtf}@informatik.tu-muenchen.de

Abstract

We describe a learning inference control heuristic for an equational theorem prover. The heuristic selects a number of problems similar to a new problem from a knowledge base and compiles information about good search decisions for these selected problems into a *term space map*, which is used to evaluate the search alternatives at an important choice point in the theorem prover. Experiments on the TPTP problem library show the improvements possible with this new approach.

Introduction

Automated theorem provers (ATP systems) are programs that try to prove the validity of a given statement under the assumption of a set of axioms. They are currently beginning to make inroads into industrial and scientific fields outside the core deduction community. Systems like DISCOUNT (Denzinger, Kronenburg, & Schulz 1997) and SETHEO (Letz *et al.* 1992) are being used for the verification of protocols (Schumann 1997), the retrieval of software components (Fischer & Schumann 1997) and mathematical theorems (Dahn & Wernhard 1997) from libraries. Recent successes of theorem provers, most visibly the proof of the Robbins algebra problem by EQP (McCune 1997), demonstrate the power of current theorem proving technology. However, despite the fact that ATP systems are able to perform basic operations at an enormous rate and can solve most simple problems much faster than any human expert, they still fail on many tasks routinely solved by mathematicians.

We believe that this is due to the differences in how humans and computers search for proofs. Human beings usually develop both conscious and intuitive knowledge about which operations to apply in a given situation to reach a given target. Most theorem proving programs, on the other hand, use very little of this kind of *search control knowledge* and rely on a set of fixed, preprogrammed search control heuristics.

Optimization of the theorem prover for a given set of problems consists in the selection of an existing heuristic (with suitable parameters), or even in the manual coding of a new heuristic based on the experience of a user with the domain. Both tasks are tedious, and expensive in terms of time and manpower. Our aim is to adapt a theorem prover to a domain or a problem by learning from examples of successful proof searches.

For this purpose, we store information about good search decisions for problems in a given domain. For each new problem, we select a couple of previous examples with similar features and compile the associated information into a *term space map*, which in turn defines a search guiding heuristic for the new problem. This work solves some problems encountered with a similar approach without example selection (Denzinger & Schulz 1996a).

In this paper, we first give a very short introduction into equational theorem proving and the associated search problem. We then describe how we generate and store examples of good search decisions. The next section describes how we select training examples for a given new problem and how we use these examples to create a suitable heuristic evaluation function. Finally, we present experimental results with the theorem prover DISCOUNT 2.1/TSM and conclude.

Equational Theorem Proving

The aim of equational theorem proving is to show that two terms s and t can be transformed into each other by the application of equations from a set of axioms E, i.e. they try to show that $s = t$ is a logical consequence of E. This problem is only semi-decidable, therefore all proof procedures have to search for a proof in an infinite search space. Most successful theorem provers (e.g. DISCOUNT or Waldmeister (Hillenbrand, Buch, & Fettig 1996)) for this kind of deduction are based on *unfailing completion* (Bachmair, Derschowitz, & Plaisted 1989). We assume that the reader is familiar with most basic terms and only give a very short introduction to the necessary concepts. See (Baader & Nipkow 1998) for a more comprehensive introduction.

The set $Term(F, V)$ of *terms* over a finite set of function symbols F (with associated arities) and an enumerable set of variables V is defined as usually. An equation $s = t$ is a pair of terms. We consider equations to be symmetrical. A rule $l \rightarrow r$ is an oriented equation such that all variables in r also occur is l. A *ground reduction ordering* $>$ is a Noetherian partial ordering that is stable with respect to the term structure and substitutions and total on ground terms. A rule $l \to r$ is said to be compatible with $>$ if $l > r$. Rules and equations can be applied to terms by matching one side onto a subterm and replacing this subterm with the instantiated other side. We usually only allow *simplifications*, i.e. applications of rules and equations that replace larger terms by smaller terms.

Our prover, DISCOUNT, takes a set of equations E, a goal $s = t$ and a ground reduction ordering $>$ as input. It tries to decide the equality of s and t modulo E by incrementally generating a *ground confluent* and terminating set of rules and equations equivalent to E. If certain fairness criteria are ensured, it can be guaranteed that any valid equation $s = t$ can be proven after a finite number of inferences by simplifying s and t as far as possible (i.e. to compute their *normal forms*) with each successive system of rules and equations.

The proof procedure of DISCOUNT is based on two basic inference rules: Ordered unit paramodulation (the building of *critical pairs*) and rewriting. Ordered unit paramodulation generates new equation by overlapping a maximal side of one rule or equation into a maximal side of another rule or equation. Rewriting, on the other hand, is a contracting inference. It does not create new equations, but allows the simplification of an existing rule or equation if certain conditions are fulfilled. 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. 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 in turn, 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.

The order in which equations from CP are processed is one of the most crucial points for the performance of the prover. This order is determined by an *heuristic evaluation function*, which assigns a weight to each fact. The prover always selects the fact with the lowest weight for processing. Experimental results show that all proofs found by DISCOUNT at all can be reproduced in sub-second times if a good evaluation function is used. However, using standard search heuristics (weighting equations according to the number of symbol in the terms) the prover typically spends more than 99% of the processing time on inferences not contributing to the proof (see (Denzinger & Schulz 1996b) for more detailed results). Our aim is to improve the overall performance of the prover by controlling this choice point with a learning evaluation function.

Knowledge Acquisition and Representation

Learning search control knowledge for theorem provers is based on the hypothesis that experience from previous proof searches is useful in guiding new proof searches. Given this hypothesis, the basic questions are which parts of a proof search should be used in learning, what kind of knowledge should be learned, and what learning algorithm should be employed.

Our approach to learning for DISCOUNT tries to extract search control knowledge from listings of inference steps. Our basic assumption is that the equations occurring in a successful proof search contain enough information to describe the proof adequately for reproduction, and that information about the exact structure of the proof is less important. This assumption is supported by the success of *learning by pattern memorization* (Denzinger & Schulz 1996a) particularly in reproducing proofs. We believe that the most important reason for this effect is that much of the relevant structure of the proof is given implicitly by the calculus (inferences can only be performed after all the necessary preconditions are fulfilled).

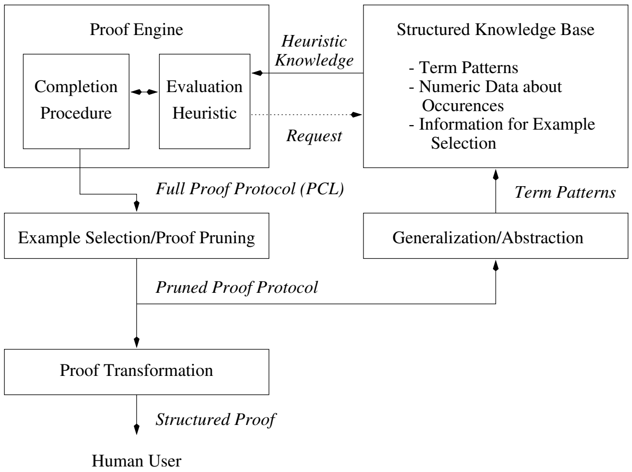


Figure 1: The DISCOUNT system

Our knowledge acquisition algorithm is structured into two phases (compare figure 1). First, a proof protocol of a successful proof search is analyzed. We determine the set of equations actually used in the proof. In the second phase, the selected equations are generalized and stored in a knowledge base, indexed by a set of features describing the proof problem.

Proof Recording and Analysis

One of the strengths of the DISCOUNT system is its ability to record proof searches in the PCL format (Denzinger & Schulz 1996b). PCL is a generic language for describing completion based proof processes. Figure 2 shows some example code. PCL protocols can be automatically analyzed, structured and transformed into a human-readable form. They also serve as the basis for a number of learning approaches, including the one presented in this paper.

```
...
21:tes-eqn : f(x,f(y,f(x,y))) = e() : cp(20,L,9,L)
22:tes-eqn : f(x,f(x,y)) = f(e(),y) : cp(20,L.1,9,L)
23:tes-eqn : f(x,f(x,y)) = y : tes-red(22,R,7,L)
39:tes-lemma : f(x,f(x,y)) → y : orient(23,u)
...
```

Figure 2: Example PCL code

Full PCL listings contain an entry for each inference done by the prover, describing both the inference and the resulting fact. Despite the fact that DISCOUNT's inference engine is by now rather dated and cannot compare with e.g. Waldmeister in speed of execution, a typical protocol for a hard problem will contain about 400 000 such entries and take more than 50 MB of disk space. As a first step of abstraction, we discard all facts except for the axioms and those contributing to the final proof. The resulting pruned listing typically contains between 50 and 500 facts and inferences (see (Denzinger & Schulz 1996b) for specific examples). As we are only interested in the facts themselves, we discard all structural information and keep only the equations to represent the proof process[1].

Term Patterns

Users of ATP systems often use the same symbol with different intended semantics in different proof problems, and similarly use multiple symbols with the same intended semantics. We have e.g. seen both *plus* and *add* to describe an additive operator, and likewise seen *product* as a binary function symbol or a ternary predicate symbol. For this reason, we abstract from the particular signature used for a given proof problem by transforming the equations occurring in the proof into *representative patterns*. A representative pattern for a *term* t is computed by normalizing the variables in the term and substituting its function symbols in the way that the ith original function symbol of arity j occurring in t is replaced by the new symbol f_{ji}[2]. As an example, $\mathtt{pat}(f(x,a,a)) = f_{31}(x_1, f_{01}, f_{01})$. A representative pattern for an equation is computed by first orienting the equation according to some ordering stable with respect to the pattern transformation, and then treating it as a single term with top symbol '='. For details consult (Denzinger & Schulz 1996a). It is important to note that representative patterns of both terms and equations are terms over a new signature with function symbols $\{f_{01} \dots f_{0n}, f_{11} \dots f_{1n}, f_{m1} \dots f_{mn}\}$ for suitably large numbers of n and m. Thus, all operations on terms (including learning algorithms) can be directly transferred to patterns.

As the transformation of equations into patterns may generate the same pattern more than once, we annotate each pattern with the number of equations corresponding to it. Thus, a proof is represented by a set of annotated patterns of equations.

Indexing Proof Examples

One of the main problems in learning search control knowledge for theorem provers is that there exist very few efficient algorithms for learning on or comparing arbitrary sized recursive structures. Therefore, even the very first approaches to learning in theorem proving resorted to represent terms by vectors of numerical features. The success of these approaches has been, however, limited, as finite vectors of simple numerical features necessarily ignore a lot of information about the structure of terms. We do not use numerical features to learn evaluation functions, however, we do use them to generate a fingerprint for a complete proof problem, similar to the approach described in (Fuchs 1997).

We selected the following values for the feature vector:

- Number of axioms in the original specification
- Average term depth of the axioms
- Standard deviation of the term depth of the axioms
- Term depth of the goal
- A vector describing the distribution of function arities in the signature of the problem.

The experiments described below showed that this set of features does not contain redundant information, i.e. dropping any of the features leads to decreased performance of the proof system.

Term Space Maps

Term patterns describe only the structure of individual terms and equations, and allow for very little generalization. Fixed size feature vectors, on the other hand, can describe properties of large sets of terms, but lack the ability to describe significant structural elements of the terms. *Term Space Maps* (TSMs), introduced in (Schulz 1998) as a generalization of *term evaluation trees* (Denzinger & Schulz 1996b), fall in between these two extremes. They are recursive structures that have the ability to describe some structural aspects of sets of terms or equations.

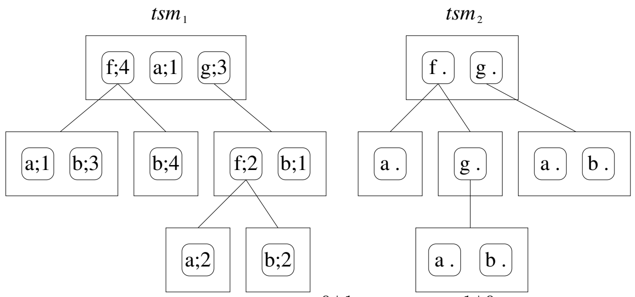
TSMs partition a set of terms according to an *index function* , i.e. a function $i : Term(F,V) \mapsto I$ that maps the set of terms onto an arbitrary (but fixed) index set I and has the property that for two terms $s = f(s_1 \dots s_n)$ and $t = g(t_1 \dots t_m)$ (where variables are treated as operators of arity 0) $i(s) = i(t)$ implies that $n = m$. This same operation is recursively applied to the subterms of the terms in each partition. Each partition (or *term space alternative, TSA*) in the TSM

[1] The approaches described in (Denzinger & Schulz 1996a) and (Schulz 1998) extract additional information about equations. However, we can treat a pruned PCL listing as a flat set of equations in this paper.

[2] This definition specializes the one given in (Denzinger & Schulz 1996a), ensuring that each new symbol is only used with one arity even in independently generated patterns.

may be annotated with the a representation of annotations of terms falling into this partition. Equations can be mapped onto TSMs in two ways, either by treating them as two separate terms or by treating them as a single term with the special top symbol =.

In our case, we use an index function that maps a term to its top symbol[3]. The left picture (tsm_1) in figure 3 shows an example term space map, where terms are annotated with simple integers and partitions with the sum of these annotations. The TSM corresponds to the set $\{(f(a,b);1),(f(b,b);3),(a;1),(g(b);1),(g(f(a,b);2)\}$. We use term space maps not on the original terms and equations, but rather on their corresponding representative patterns (which, as stated above, are terms over a new signature).



$$\delta_{TSM}(tsm_1, tsm_2) = \frac{\frac{\frac{0+1}{2}+1}{2} + 1 + \frac{\frac{1+0}{2}}{1}}{3} = \frac{3}{4}$$

Figure 3: Example TSMs, δ_{TSM}

As in (Schulz 1998) we use term space maps to compute evaluations for terms and equations. However, we also use them to compute a distance measure for proof problems (see below).

Knowledge Application

In the application phase the prover retrieves some example proofs from the knowledge base and uses them to generate an evaluation function for the new problem.

Selection of Example Proofs

Previous approaches of learning with TSM-like structures described good initial successes, but also showed that performance of the prover peaks at a relatively small number of training examples and drops off as more example are added. We strongly believe that the main reason for this is that compiling too many and to diverse training examples into a single term space map leads to an undifferentiated map in which all nodes have a very similar weight. On the other hand, having a wide range of different training examples is obviously desirable to cover a large number of cases. We solve this dilemma by selecting only a few suitable examples from a much larger knowledge base, using two different similarity measures: First, we use a weighted Manhattan distance on the feature vectors describing each problem. Secondly, we used a tree distance measure $\delta_{TSM} : TSM \times TSM \mapsto [0;1]$ on the TSMs generated by the axiomatization of two problems.

δ_{TSM} recursively compares the indices of the TSAs of two TSMs. At each level, it adds 1 for each index occurring in only one of the two corresponding TSMs. It also adds the distances for the sub-TSMs for each TSA, normalized by dividing them by the number of TSMs belonging to the TSA. The resulting value is again normalized by dividing it by the number of TSAs in the larger of the two TSMs. Figure 3 shows an example calculation. Please note that the annotations in tsm_1 are only required for the evaluation of terms. It can be shown that δ_{TSM} is a metric on the space of TSMs, and that $\delta_{TSM}(tsm_1, tsm_2) = 1$ exactly if there is no term that can be completely mapped onto both TSMs. See (Brandt 1998) for more details on both feature and TSM distance measures.

We use both a maximal number of examples and a threshold for the distance to limit the number of examples selected.

Computing Evaluations

After selecting a number of proof examples for a given new problem, we compile the annotated patterns of the useful equations into a TSM. The annotations in the resulting TSM denote how many useful equations have been mapped to this node during the example proof searches. To evaluate a new equation, the two sides are mapped onto the TSM. Each term node receives an evaluation according to the annotation stored at the corresponding TSM node. We compute the weight for a node as $W(node) = w_{base} * (1 - limit * occur/norm)$, where w_{base} is 1 for variable nodes and 2 for operator nodes, *limit* is a scaling factor determining the maximal effect of the learned knowledge (set to 0.5 in our experiments), *occur* is the annotation of the node (i.e. the number of useful equations mapped to this TSA) and *norm* is the sum over all annotations of the corresponding TSM, i.e. the maximal number of equations which potentially might have been mapped to this TSA. Term nodes not corresponding to any TSM node just receive the base weight. To weight a term, we sum over all term nodes, and to weight an equation we sum over both terms.

As an example, consider the term $f(a, X)$ evaluated against tsm_1 from figure 3, with *limit* set to 0.5. The top node with the operator f is mapped onto the left most TSA with the annotation 4. The total number of terms mapped to the TSM is $4 + 1 + 3 = 8$. Therefore, the weight of the node is $2 * (1 - 0.5 * 4/8) = 1.5$. The weight of the node a is $2*(1-0.5*1/4) = 1.75$. As there is no TSA corresponding to the last node containing the variable X, we assign a weight of 1, giving a total

[3]Term evaluation trees result as a special case if we map each term to the arity of its top symbol.

weight of 4.25 for the complete term.

Experimental Results

We used the set of all unit-equality problems from the TPTP problem library, version 2.1.0 (Suttner & Sutcliffe 1997), as a test case. The set of training examples contains 201 proofs for problems that could be found by DISCOUNT within a 180 second time limit using DISCOUNT's best conventional strategy, *AddWeight*[4].

Table 1 shows results for *AddWeight*, for *Occnest* (a goal-directed strategy), and 4 different TSM-based strategies: A strategy without example selection, a strategy with random example selection, a strategy using the feature vector distance measure, and a version using the δ_{TSM} distance measure. The time limit for these tests was again 180 seconds.

Experiments were performed on a 233 MHz Pentium PC running Linux. All times and time limits are CPU times.

Heuristic	Solutions	Time/Sol.
Occnest	237	8.04 s
AddWeight	253	7.95 s
TSM (no selection)	248	7.81 s
TSM (random selection)	237	6.97 s
TSM (features)	259	7.30 s
TSM (δ_{TSM})	263	8.24 s

Table 1: Results

The system also participated in the CADE-15 ATP system competition and, despite the known weaknesses of DISCOUNT's base inference engine, completed in third place (of 8) in the unit equality category. See `http://www.cs.jcu.edu.au/~tptp/CASC-15/`.

Conclusion

Our results show that information about previous proofs can help in finding new proofs even for previously unsolvable problems. They also demonstrate that term space maps compiled from patterns of equations contain useful information both for the evaluation of search alternatives in theorem proving and for the selection of similar proof problems.

Our future work will concentrate on a couple of topics. First, we will work on improving TSM-related learning algorithms by aiming at stronger expressive power. Currently, TSM-based learning algorithms only allow us to express some rather simple concepts, namely distribution of certain function symbols and simple substructures in the training set. By using more general index functions on terms to discriminate between term space alternatives, we can express more complex concepts, and can extend term space mapping to include pattern memorization as a special case.

Another current focus of our work is the implementation of a more efficient inference engine capable of handling full clausal logic. This will allow us to evaluate our learning techniques for the more general case, and will hopefully also lead to an even more competitive prover.

[4]Proof recording is not yet implemented for problems with existentially quantified variables in the goal. *AddWeight* solves an additional 52 of these problems, which are not available for learning.

References

Baader, F., and Nipkow, T. 1998. *Term Rewriting and All That.* Cambridge University Press.

Bachmair, L.; Derschowitz, N.; and Plaisted, D. 1989. Completion Without Failure. In *Coll. on the Resolution of Equations in Algebraic Structures, 1987.* Academic Press.

Brandt, F. 1998. Example Selection for Learning in Automated Theorem Proving. Diplomarbeit in Informatik, Institut für Informatik, TU München. (available from the author at `brandtf@informatik.tu-muenchen.de`).

Dahn, B., and Wernhard, C. 1997. First Order Proof Problems Extracted from an Article in the MIZAR Mathematical Library. In *Proc. of the 1st FTP*, 58–62. RISC Linz, Austria.

Denzinger, J., and Schulz, S. 1996a. Learning Domain Knowledge to Improve Theorem Proving. In *Proc. CADE-13*, LNAI 1104, 62–76. Springer.

Denzinger, J., and Schulz, S. 1996b. Recording and Analysing Knowledge-Based Distributed Deduction Processes. *Journal of Symbolic Computation* 21:523–541.

Denzinger, J.; Kronenburg, M.; and Schulz, S. 1997. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning* 18(2):189–198.

Fischer, B., and Schumann, J. 1997. SETHEO Goes Software-Engineering: Application of ATP to Software Reuse. *Proc. CADE-14*, LNAI 1249, 65–68. Springer.

Fuchs, M. 1995. Learning Proof Heuristics by Adapting Parameters. In *Proc. 12th ML*, 235–243. San Mateo, CA: Morgan Kaufmann.

Fuchs, M. 1997. Automatic Selection of Search-Guiding Heuristics. In *Proc. of the 10th FLAIRS*, 1–5. Florida AI Research Society.

Hillenbrand, T.; Buch, A.; and Fettig, R. 1996. On Gaining Efficiency in Completion-Based Theorem Proving. *Proc. of the 7th RTA*, LNCS 1103, 432–435. Springer.

Letz, R.; Schumann, J.; Bayerl, S.; and Bibel, W. 1992. SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning* 1(8):183–212.

McCune, W. 1997. Solution of the Robbins Problem. *Journal of Automated Reasoning* 3(19):263–276.

Schulz, S. 1998. Term Space Mapping for DISCOUNT. In *CADE-15 Workshop on* Using AI methods in Deduction.

Schumann, J. 1997. Automatic Verification of Cryptographic Protocols with SETHEO. In *Proc. CADE-14*,LNAI 1249, 87–11. Springer.

Suttner, C., and Sutcliffe, G. 1997. The TPTP Problem Library (TPTP v2.1.0). Technical Report AR-97-01 (TUM), 97/04 (JCU), Institut für Informatik, TU München, Munich, Germany/Department of Computer Science, James Cook University, Townsville, Australia.