Learning Search Control Knowledge for Equational Theorem Proving

Stephan Schulz

Institut für Informatik, Technische Universität München,
D-80290 München, Germany, schulz@informatik.tu-muenchen.de

Abstract. One of the major problems in clausal theorem proving is the control of the proof search. In the presence of equality, this problem is particularly hard, since nearly all state-of-the-art systems perform the proof search by saturating a mostly unstructured set of clauses. We describe an approach that enables a superposition-based prover to pick good clauses for generating inferences based on experiences from previous successful proof searches for other problems. Information about good and bad search decisions (useful and superfluous clauses) is automatically collected from search protocols and represented in the form of *annotated clause patterns*. At run time, new clauses are compared with stored patterns and evaluated according to the associated information found. We describe our implementation of the system. Experimental results demonstrate that a learned heuristic significantly outperforms the conventional base strategy, especially in domains where enough training examples are available.

1 Introduction

The last few years have seen an impressive improvement in the power of first-order theorem provers, and a corresponding increase in the use these systems in research and development. They are being used for the verification of protocols [Sch97,Wei99,GP00], and the retrieval of mathematical theorems [DW97] or software components [FS97] from libraries. Theorem provers are also used to synthesize larger programs from standard building blocks and to prove the correctness of the resulting program systems [SWL$^+$94,BRLP98]. For these purposes, various systems have been embedded into larger, interactive proof systems like ILF [DGHW97], KIV [RSS95] or VSE [HLS$^+$96].

The most visible success of automatic theorem provers (or *ATP systems*) today is the celebrated proof of the Robbins algebra problem by EQP [McC97]. Successes like this 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, even if these tasks can be encoded in first-order logic in a natural way. Moreover, many of the more impressive successes require an experienced human user who selects a suitable prover configuration, often by trial and error.

The main reason for this is that a theorem prover has to search for a proof in a usually infinite search space with a very high branching factor, i.e. a very high number of possible choices at each choice point. Much previous work in theorem proving has been targeted at the development of refined calculi that reduce this branching factor by restricting the number of possible inferences. However, the semi-decidability of the underlying problem for most interesting logics restricts the potential for this approach, and even the most refined calculi typically are highly non-deterministic. This is particularly true for calculi that deal with equality, as the congruence properties of this relation imply a very large number of possible inferences. Moreover, all calculi for equational logic that have been successfully implemented and show good performance operate on unstructured clause sets, and lack the inherent goal-orientedness of e.g. model elimination calculi or the set-of-support strategy for non-equational logic.

Most current theorem provers therefore use a small set of highly parameterized *heuristic evaluation functions* to guide the proof search. The selection of a proper evaluation function and set of parameters for a given problem (or problem domain) is based on experience of the human user, often supported by large and tedious sets of experiments. It is often encoded in a so-called *automatic mode*, where the prover selects one of a relatively small set of pre-configured strategies based on properties of the current proof problem. While such automatic configuration can improve the performance of a prover if used by a non-expert, the development of *new* strategies is still a manual process that requires a lot of work and attention by an expert user or even developer, and is hence very expensive in terms of man power.

In this paper we suggest a way to automatically *learn* information about good and bad search decisions from examples of successful proof searches, and to use the learned knowledge to define new search control heuristics. A proof search is represented by a clause dependency graph, and clauses contributing to a proof as well as superfluous clauses are identified. A suitable subset of these clauses is used to represent good and bad search decisions. We encode these clauses as *patterns* that abstract from irrelevant information and use the information about the value of a decision in later proof searches to compute an informed evaluation of new clauses.

For a broad overview of related works, see [DFGS99]. Our own previous work in this direction (see [DS96,DS00]) was restricted to the unit-equational case, where the suitable encoding of search decisions is much easier. Moreover, these previous approaches were based on positive search decisions only, and, due to the relatively small number of unit-equational proof problems publically available, we did not perform an experimental evaluation of the same rigidity. Other related work has been reported by Fuchs [Fuc96,Fuc97]. The main differences are that Fuchs again focuses on the unit-equational case, that his clause generalizations are weaker, and that he tries to replay search decisions from a single proof problem. Our approach, on the other hand, gathers experience from an arbitrary number of proof searches.

We start the remainder of this paper with a very short introduction to superposition-based theorem proving and the organization of the search process. In the next section we describe our approach to learning and cover knowledge acquisition, representation, and application. Section 4 contains experimental results, and we conclude in section 5.

2 Superposition-Based Theorem Proving

The superposition calculus [BG94,BG98] and its variants are generally recognized as the most powerful calculi currently available to tackle theorem proving problems with equality. The well-known prover SPASS [WAB$^+$99] and our own system E [Sch99,Sch01] are based on particular instances of this calculus, and all other leading saturating theorem provers incorporate at least some important features of it. We will give a very short introduction, concentrating on aspects important for search control.

2.1 Calculus

The superposition calculus is a refutational calculus. It works on a set of clauses, and tries to show the unsatisfiability of the clause set by deriving the empty clause. A clause in this calculus is a multi-set of literals (positive or negative equations[1]) and is interpreted as the universally quantified disjunction of these equations. An equation is an unordered pair of first-order terms. Generating inferences add new clauses to the set, contracting inferences modify or remove existing clause. A theorem proving derivation is a sequence of generating and contracting inferences applied to a set of clauses. Most inference types are restricted by a *simplification ordering* on terms, which is extended to equations and clauses.

The most important generating inference in any superposition calculus is the superposition inference, a paramodulation inference restricted to maximal terms in maximal literals[2]. It can be seen as lazy conditional rewriting applied to instances of clauses resulting from unification.

Despite the restrictions imposed by ordering and literal selection, superposition inferences are typically responsible for more than 99% of all clauses generated during a proof search. In addition to superposition inferences, special factoring inferences and equality resolution inferences are necessary for the completeness of the calculus. However, the number of clauses generated by these inferences typically is miniscule.

One of the major strengths of the superposition calculus is its compatibility with a wide range of contracting inferences. Probably the most important of these contracting inferences is the *rewriting* of clauses i.e. the use of orientable

[1] Non-equational atoms are encoded as equations.

[2] In many cases, inferences can be further restricted by *literal selection*, i.e the restriction of paramodulation inferences to a single negative literal.

instances of positive unit clauses to replace terms with smaller terms. Other contracting inferences are the removal of trivial or duplicated literals. Tautological or subsumed clauses can be eagerly removed without affecting the completeness of a proof search (assuming a suitable definition of subsumption). These contracting processes typically take up a large part of the total work during a proof search, and are totally indispensable for the efficient work of the prover.

2.2 Organizing the Proof Search

The superposition calculus is complete, i.e. for any unsatisfiable clause set there is a theorem proving derivation that derives the empty clause. However, in order to guarantee that the empty clause is found, certain *fairness* constraints are imposed upon the search. A sufficient way to ensure that these constrains are met is to eventually perform all generating inferences between *persistent* clauses, i.e. to ensure that no possible inference is delayed forever. This is achieved by different variants of the *given-clause* algorithm. This algorithm uses two sets of clauses, a set of *unprocessed* clauses, which initially contains the input clauses, and a set of *processed clauses*, which initially is empty. The algorithm repeatedly picks one of the unprocessed clauses, performs all generating inferences between this *given clause* and all clauses in the processed set, adds the newly produced clauses into the set of unprocessed clauses, and moves the given clause to the set of processed clauses.

The selection of the given clause is typically done by a heuristic evaluation function which rates each clause. The most widely used and successful heuristics are based on symbol counting, and assign a low weight to clauses with a low number of symbols. This heuristic may be interleaved with a *first-in first-out* strategy that always picks the oldest remaining clause.

Different implementations of the given-clause algorithm differ mainly in the way in which contracting inferences are performed. In our version, the given clause is rewritten with the processed clause set and checked for subsumption after being picked. Then we purge clauses that can be subsumed by or rewritten with the given clause from the set of processed clauses. In addition, newly generated clauses are rewritten and checked for obvious tautologies. Other variants of the algorithm also keep the list of unprocessed clauses in normal form or use generated unit clauses more eagerly for rewriting processed clauses (backward contraction). We will abstract from contracting inferences in the following, for a discussion of their effect and handling see [Sch00].

3 Learning Search Control Knowledge

We have performed a detailed analysis of the given clause algorithm, supported by experimental data, and found that the selection of the given clause is the most critical choice point in this algorithm [Sch00]. In a typical proof search today, a high-performance prover generates up to a few million clauses within 5 minutes. Of these, up to about 10000 are processed. However, typically much less than 200 clauses are needed for the proof. If we assume an optimal oracle, processing these 200 clauses only takes minimal time for modern theorem provers – typically less then a second, and in virtually all cases less than 5 seconds.

Based on these results, we consider the choice of the next clause to process the most crucial choice point in the given-clause algorithm. We try to improve the proof search by learning good evaluations for clauses based on experiences from previous proof searches.

In order to be able to meet this goal, we have developed a general framework for incorporating feedback from previous proof searches into the selection of the *given clause*. The corresponding learning theorem prover is organized as follows:

- In a first phase, a protocol of the inference steps is turned into a *proof derivation graph*. This graph is analyzed, and clauses contributing to the proof as well as clauses that are in certain sense *close* to the proof are selected as training examples.
- These clauses are then transformed into *annotated patterns*, where a pattern is a unique representation for a class of structurally identical clauses, and where an annotation describes the role of the clauses corresponding to the pattern in previous proof searches. The resulting sets of patterns are stored in a knowledge base and indexed by a feature vector describing the original proof problem.
- In the application phase, we use this feature vector to decide on a set of proof experiences to use to guide the new proof search. The corresponding patterns are recalled and fed into a learning algorithm. Newly generated clauses are evaluated using the resulting learned knowledge representation.

In the instance of the general framework described in this paper, the learning algorithm simply stores the clause patterns with their corresponding annotations. If a new clause matches a known pattern, its standard evaluation is modified based on the annotations.

Our implementation of the learning process is fully automatic. The only human intervention necessary is the a-priory selection of parameters for both the theorem prover and the analysis module that determines which clauses are used to represent a given proof search.

Fig. 1 shows a general sketch of this learning cycle as implemented in our prover E. We will discuss the different aspects of this cycle in the next sections.

3.1 Knowledge Acquisition

The first problem is to identify *which* search decisions from a given proof search should be used as input for a learning prover. As we wrote above, the starting point for our learning system is a complete listing of all inferences performed by the prover during a successful proof search. Such a listing basically consists of a number of steps, each of which contains a clause (the result of the step) and a justification for this clause. A justification is either *"initial"* for clauses from the problem specification, or a description of the inference generating this clause, including pointers to all clauses used in the premise of the inference.

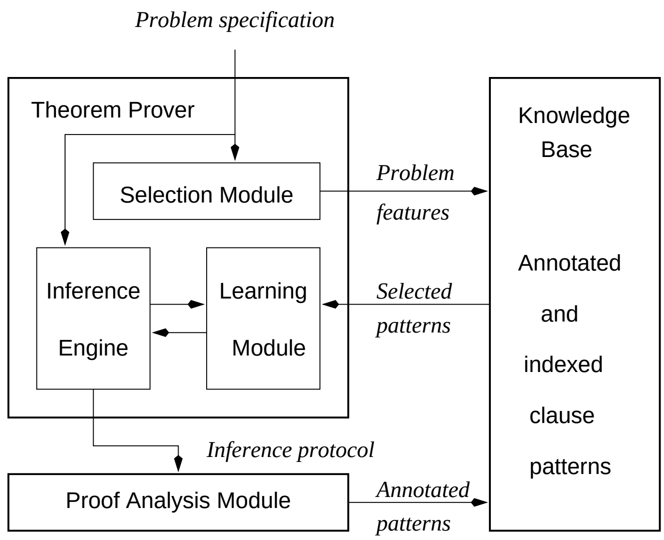

Fig. 1. Architecture of the learning theorem prover

An inference listing defines a directed *clause dependency graph*. In this graph, nodes correspond to the clauses (where different occurrences of the same clause are considered to be distinct objects and are represented by different nodes) and edges go from the premises of an inference to the conclusion. A path from an initial clause to the empty clause is called a *proof path*, clauses on a proof path are called *contributing* (to the proof) and all other clauses are called *superfluous*. The subgraph containing exactly the nodes and edges of proof paths is called the *proof object* or simply *proof*[3].

In a typical non-trivial proof search, superfluous clauses outnumber contributing clauses by a factor of at least 100. To keep the knowledge base manageable, and to balance the number of contributing and superfluous clauses we use as training examples, we select an appropriate number of superfluous clauses from those that are close to the proof, i.e. that can be derived in at most a small number of inferences from the contributing clauses. Additionally, we discard clauses that have never been selected as the *given clause*, as these clauses did not have a chance to contribute to a proof at all, and hence we have no information on the impact they would have had on the proof search.

[3] While proofs are often considered to be *trees*, generating provers reuse intermediate results, and thus each node (corresponding to a clause) can be referenced more than once, resulting in a general directed acyclic graph structure. The customary proof tree can be generated by recursively duplicating subgraphs referenced more than once, thus *unfolding* the graph. However, as this typically leads to an exponential explosion of the number of nodes, we consider the graph representation to be not only more natural, but also more suitable for our purposes.

All selected clauses are annotated with information about their role in the proof process. We found that the clause status (contributing or superfluous) and the distance of the clause from the closest proof path are useful for assigning evaluations to clauses.

3.2 Knowledge Representation

The next problem we have to tackle is to find an adequate representation of search decisions. This representation has to strike a fine line between generality and specificness. Equivalent search decisions should be represented by similar structures, but significantly different ones should still be distinct.

We achieve this compromise by the *generalization* of clauses into clause patterns. This serves two purposes. First, remember that clauses are multi-sets of literals, and are hence unordered. Thus, there are 2^n different ways to order the literals in a clause with n literals. Moreover, equational literals are again unordered pairs of terms, and can be arranged independently in each literal. Thus, any given clause with n literals has $n!\,2^n$ different syntactic representations. We want to have a unique representation for all these strictly equivalent but syntactically different clause representations.

Secondly, the transformation into patterns abstracts from the given signature of the problem. In our experience, specifications of problems from the same domain are often encoded using different function symbols. Moreover, there are analogous substructures even within a single specification. As an example, consider algebraic fields, which contain both an additive and a multiplicative group. We are therefore convinced that the loss of information resulting from the abstraction of signature information is more than compensated for by the ability to generalize to analogous situations. Earlier experiments for the unit-equational case [DS96,DS00] as well as the experimental results presented in this paper support this conviction.

We achieve these two goals by systematically replacing function symbols with new symbols from the set $\{f_{ij} | i, j \in \mathbf{N}\}$ (where the first index encodes arity, the second one distinguishes between different symbols of the same arity), and by reordering the terms and literals in the clause representation, until a representation is found that is minimal in a suitable ordering.

We use a modified lexicographic ordering that first compares terms based only on their size and structure (ignoring the symbols) and then compares terms lexicographically. The ordering is then lexicographically lifted to literal and clause representations. The search for the minimal clause representation is conducted using a conventional backtracking algorithm, which in the practical implementation is supported by a pre-ordering of terms and literals with the part of the ordering that is independent of function symbols and depends on the structure of terms only. The final result is then encoded as a first order term over an extended signature.

We use the reserved symbols *eq/2* and *neq/2* to encode positive and negative equations, respectively. The disjunction of the literals of a clause is encoded by using the reserved symbols *or/2* and *nil/0* as standard list constructors.

As an example, the clause

$$b = f(a, a) \vee a \neq b$$

is represented by the pattern

$$or(eq(f_{20}(f_{00}, f_{00}), f_{01}), or(neq(f_{00}, f_{01}), nil))$$

The knowledge base simply stores patterns of clauses with the annotations computed in the proof search analysis. Patterns are indexed by a vector of 15 feature values describing aspects of the original problem specification, and by arity frequency vectors for function symbols and predicate symbols. Features used include e.g. the number of unit, Horn and general clauses, the average size and depth of terms, and the average number of positive and negative literals per clause. The arity frequency vectors describe how many symbols of each arity are present in the problem signature. For a detailed description of the problem selection mechanism see [Sch00].

All relevant data is stored in the form of plain ASCII text files in a subdirectory on the hard disk. While it is possible to modify the knowledge base by hand, we have implemented tools to create a knowledge base and to automatically insert new or remove old proof examples.

3.3 Knowledge Application

The application phase is concerned with two main problems: The identification of suitable proof experiences and the application of the corresponding knowledge to the new proof search.

If a new proof problem is given to the learning theorem prover, the corresponding feature vector is computed. The prover then uses a standard distance measure (modified Euclidean distance) to determine a set of similar problems[4]. All clause patterns from selected proof experiences are stored into a pattern cache organized as a splay tree [ST85], a memory-efficient data structure suitable for very fast data retrieval. If the same pattern occurs more than once in the knowledge base, the corresponding annotations are combined. As a result, each of the patterns in the tree carries a tuple (p, d), where p describes the number of proofs the clauses corresponding to the pattern participated in, and d is the average distance of these clauses from the proof.

New clauses to be evaluated are transformed into patterns as well. If the resulting pattern is found in the cache, we use the annotation of the pattern to compute the evaluation. Otherwise, we use an annotation $(0, d_{max})$, where (d_{max}) is the largest distance in the cache, increased by one. In order to increase the stability of the prover and to help the system to fill gaps in the acquired knowledge, we combine a conventional heuristic with the learned knowledge.

[4] For pattern memorization, we get no benefit from selecting only similar problems, and hence always use all available training examples (see section 4 for a short discussion). However, if we use more sophisticated versions of *term space mapping* [Sch00], this selection becomes important.

Consider a clause C to be evaluated with an annotation of (p, d). A clause with a large value of p should get a better than standard evaluation (it has contributed to many proofs). A clause with a large value of d, on the other hand, usually has not contributed to proofs, and should receive a worse evaluation. Now assume that w is the weight assigned by a standard symbol counting evaluation function. The weight assigned by the learning heuristics is $lw(C) = w(1 + w_l\,norm\,(w_p\overline{p} + w_d\overline{d}))$, where $\overline{p}$ and $\overline{d}$ are normalized by dividing p and d by the largest corresponding values in any annotation in the pattern cache, where w_p and w_d determine the relative influence of these two variables, where *norm* rescales all evaluations for annotations in the cache to the interval $[0;1]$, and where finally w_l determines the overall influence of the learning component. The double renormalization effect allows us to independently control the influence of the two parameters p and d, as well as the relative weight given to the conventional and the learned parts of the evaluation.

Please note that for positive values of w_l the resulting strategy is *fair* if the conventional base strategy is, i.e. it will never prefer an infinite number of clauses to any given clause.

4 Experimental Results

We have implemented the above concepts as a special case of our learning theorem prover E/TSM. To evaluate the approach, we have tested the resulting system on the set of all 3558 clause normal form (CNF) problems from the TPTP problem library [SSY94,SS97], version 2.3.0[5].

Clause normal form problems in the TPTP are either *unsatisfiable*, which implies that the underlying first-order problem does have a proof, or *satisfiable*, which means that a model exists that can serve as a counterexample for the original first-order problem. A problem can be *proved* by a prover if it can show the unsatisfiability. Alternatively, a prover can (in our case rarely) show that no proof is possible, i.e. that a *model* for the specification exists. We consider both of these case to be a *success* of the proof search. Termination of the proof search without a definitive answer about the state of the problem, either due to reaching a time limit or due to running out of some other limited resource, are considered failures.

To test the success of learning, we compare the learning heuristic to a standard symbol counting heuristic (which always picks the clause with the least number of symbols). Both heuristics are interleaved with first-in-first-out (always pick the oldest clause) using a pick-given ratio of 5 to 1. For the symbol counting heuristic, this combination corresponds to a widely used standard heuristic implemented e.g. in Otter [MW97]and Vampire [RV99]. E controlled

[5] The results presented in the following have been obtained in compliance with the guidelines for use of the TPTP. TPTP input files were unchanged except for removal of equality axioms and syntax transformation. All experiments were performed on a cluster of SUN Ultra 60 workstations running at 300 MHz. Memory was limited to 192 megabyte for each proof attempt.

by this conventional heuristic can prove 1602 out of the 3558 CNF problems in TPTP within a 300 second time limit on our hardware. It also finds 88 models, for total of 1690 successes.

The learning evaluation function uses the same symbol counting heuristic as a base and uses values of 5 for w_l, -20 for w_p and 20 for w_d (note that only the relative size of w_p and w_d are relevant due to the normalization). After preliminary experiments, we have always chosen *all* problems in the knowledge base as training examples. It turns out that any strong decrease in the number of problems used also impairs the performance, while the presence of more, probably very different problems does not hurt the pattern memorization based heuristic at all. This may be due to the fact that very different proof problems also generate syntactically different clause patterns, and hence do not influence the proof search at all.

In all cases, we have used a Knuth-Bendix ordering with constant weight for all symbols and a precedence that is induced by the symbol arities (symbols with a higher arity are bigger than symbols with a lower arity, order between symbols of the same arity is chosen at random). We also used a literal selection strategy that always selects the negative literal with the largest size difference between the two terms in non-positive clauses.

To demonstrate the ability of the learning heuristic to generalize to unseen problems, we have adapted *10 fold stratified cross validation*, a standard technique from the field of machine learning. The set of all proof problems is partitioned into 10 approximately equal sized sets (or *folds)*. Within each set, the number of problems that can be solved by the conventional heuristic is again approximately equal. We then use the proofs from nine out of the ten folds as training examples for the learning heuristic and apply it to the remaining fold. In particular, this means that none of the new proof problems is already known to the prover. The resulting knowledge bases each contain about 1450 training examples, with about 70000 clauses. These are mapped onto about 14000 distinct clause patterns. The knowledge bases have a size of about 5 megabyte for the functional part and of approximately 20 megabyte if we include the full, uncompressed set of training data for archival purposes.

Table 1 shows the results of the cross evaluation. We again used a time limit of 300 seconds for each proof attempt. As we can see, in each fold the learning heuristic performs better than the conventional heuristic. If we consider the set of all problems, this improvement is statistically significant. We can also see that all of the improvement is due to the finding of more proofs. Indeed, the learning heuristic does loose one model compared to the base heuristic.

If we consider the performance of the 10 learning heuristics as a whole, we find that there are 19 problems that are only proved by the conventional heuristic, but not by the learning heuristic used on the given problem. On the other hand, there are 138 problems that are solved by the appropriate learning heuristic, but not by the conventional one. Thus, while the learning heuristics do not prove a strict superset of the problems solved by the conventional one, they can solve a fairly large number of new, hard problems.

Fold	Conventional			Learning		
	Proofs	Models	Successes	Proofs	Models	Successes
1	161	12	173	180	12	192
2	161	7	168	172	7	179
3	160	12	172	167	12	179
4	160	10	170	171	10	181
5	160	6	166	173	6	179
6	160	14	174	168	13	181
7	160	6	166	175	6	181
8	160	5	165	165	5	170
9	160	10	170	179	10	189
10	160	6	166	171	6	177
Average	160.2±0.422	8.8±3.190	169.0±3.266	172.1±4.886	8.7±3.020	180.8±6.088
Total	1602	88	1690	1721	87	1808

Table 1. Results of 10-fold stratified cross validation

We can also compare the different heuristics on subsets of the TPTP: Problems containing unit clauses only, non-unit Horn problems, and problems with general clauses. Each class is once more split according to the presence of an equational sub-theory. Table 2 shows the results.

Again we can see that the learning heuristic is superior nearly across the board. However, the gain is most significant for the case of non-horn-problems with equality (which also is generally considered to be the hardest class of problems in the TPTP), and is fairly small for unit problems. This may be correlated with the size of the corresponding class. The improvements achieved by the learning heuristic increase with the size of the problem class, which supports the plausible idea that more suitable training examples lead to better performance on unknown problems.

Problem class	Size	Conventional	Learning	Successes gained
Unit, no equality	11	11	11	0.00%
Unit with equality	447	289	292	1.04%
Horn, no equality	609	464	492	6.03%
Horn with equality	507	291	308	5.84%
General, no equality	766	307	333	8.47%
General with equality	1218	328	372	13.41%
Total	3558	1690	1808	6.98%

Table 2. Number of successes by problem class

Finally, we can compare both heuristics on the TPTP domains. A TPTP domain is a set of problems dealing with similar structures. However, TPTP domains are usually more diverse than typical application domains, because TPTP domains have been created *a posteriori* to organize a large number of existing proof problems, and do not result from a systematic axiomatization of a given application area.

Problem class	Size	Conventional	Learning	Successes gained
ALG	10	2	2	0.00%
ANA	19	2	2	0.00%
BOO	68	55	60	9.09%
CAT	58	54	53	-1.85%
CID	4	1	1	0.00%
CIV	14	11	11	0.00%
COL	160	94	94	0.00%
COM	6	5	5	0.00%
FLD	281	24	53	120.83%
GEO	165	98	101	3.06%
GRA	1	1	1	0.00%
GRP	376	254	251	-1.18%
HEN	64	62	63	1.61%
KRS	17	17	17	0.00%
LAT	35	12	16	33.33%
LCL	503	262	296	12.98%
LDA	23	19	19	0.00%
MGT	0	0	0	–
MSC	13	10	9	-10.00%
NUM	309	33	37	12.12%
PLA	30	5	5	0.00%
PRV	9	8	8	0.00%
PUZ	60	53	53	0.00%
RNG	98	41	45	9.76%
ROB	36	14	14	0.00%
SET	695	171	212	23.98%
SYN	480	377	376	-0.27%
TOP	24	5	4	-20.00%
Total	3558	1690	1808	6.98%

Table 3. Number of successes by TPTP domain

Table 3 shows the results by domain. Now we see a more diverse picture. In many domains, especially smaller ones, learning does not improve the performance of the prover much, or even leads to slightly worse performance (although this is typically not significant). Most of the gain comes from a couple of larger domains. FLD, LAT and SET are particularly impressive. Again, we get the impression that an increased number of suitable training examples improves the performance of the learning heuristic. This makes it likely that the learning heuristics are particularly useful in domains like verification or software reuse, where a large number of similar problems have to be solved over and over again, possibly with slight modifications.

The successes of the learning heuristics are achieved despite a significant computational overhead. It takes about 15 seconds to read the knowledge base into memory, determine the set of similar problems, and to prepare the pattern cache with pre-computed evaluations. Moreover, the inference rate is about 20% lower as for the less complex conventional strategies. Profiling shows that most of this overhead is spent on the transformation of new clauses into patterns. As this operation depends only on the single clause, and not on the total size of the clause database (as e.g. superposition and subsumption), this cost is relatively less significant for large problems, and will likely become less important as advances in hardware allow us to deal with even bigger search spaces.

5 Conclusion

In this paper, we have described a fully automatic learning theorem prover based on the memorization of clause patterns from previous proof searches. The learning heuristics were able to significantly outperform a conventional base heuristic, especially on hard problems. Our results show that clause pattern memorization is an adequate method for learning search control knowledge in domains where enough previous proof experiences are available. We find that time spent on good search guidance is usually well spent. While our results have been obtained with a superposition-based prover, there is no reason to suppose that it would not carry over to other saturating systems, including e.g. inductive provers.

In the future, we will combine the basic learning cycle and the abstraction provided by clause patterns with other learning algorithms. We have already investigated various versions of *term space mapping* (strictly speaking, pattern memorization is subsumed as a special case) with good success [Sch00]. We may also use *folding architecture networks* [GK96,SKG97], a much more powerful, but rather slow learning algorithm based on the neural network paradigm.

We also will try to combine proof experiences generated using a wide variety of different search heuristics, and to work with much larger knowledge bases. In all of our experiments to date, the performance of heuristics based on pattern memorization has increases significantly with an increase in stored proof experiences. We therefore expect significant synergy effects from this approach.

Finally, our prototypical implementation can significantly benefit from a clean-up and partial redesign of the time-critical sections of the code.

References

[BG94] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[BG98] L. Bachmair and H. Ganzinger. Equational Reasoning in Saturation-Based Theorem Proving. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume 9 (1) of *Applied Logic Series*, chapter 11, pages 353–397. Kluwer Academic Publishers, 1998.

[BRLP98] J. Van Baalen, P. Robinson, M. Lowry, and T. Pressburger. Explaining Synthesized Software. In *Proc. of the 13th IEEE Conference on Automated Software Engineering, Honolulu*. IEEE, 1998.

[DFGS99] J. Denzinger, M. Fuchs, C. Goller, and S. Schulz. Learning from Previous Proof Experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999. (also to be published as a SEKI report).

[DGHW97] B.I. Dahn, J. Gehne, T. Honigmann, and A. Wolf. Integration of Automated and Interactive Theorem Proving in ILF. In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 57–60. Springer, 1997.

[DS96] J. Denzinger and S. Schulz. Learning Domain Knowledge to Improve Theorem Proving. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 62–76. Springer, 1996.

[DS00] J. Denzinger and S. Schulz. Automatic Acquisition of Search Control Knowledge from Multiple Proof Attempts. *Journal of Information and Computation*, 162:59–79, 2000.

[DW97] B.I. Dahn and C. Wernhard. First Order Proof Problems Extracted from an Article in the MIZAR Mathematical Library. In *Proceedings of the 1st FTP, Linz*, pages 58–62. RISC Linz, Austria, 1997.

[FS97] B. Fischer and J. Schumann. SETHEO Goes Software-Engineering: Application of ATP to Software Reuse. In W.W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 65–68. Springer, 1997.

[Fuc96] M. Fuchs. Experiments in the Heuristic Use of Past Proof Experience. In M.A. McRobbie and J.K. Slaney, editors, *Proc. of the 13th CADE, New Brunswick*, number 1104 in LNAI, pages 523–537. Springer, 1996.

[Fuc97] M. Fuchs. *Learning Search Heuristics for Automated Deduction*. Number 34 in Forschungsergebnisse zur Informatik. Verlag Dr. Kovač, 1997. Accepted as a Ph.D. Thesis at the Fachbereich Informatik, Universität Kaiserslautern.

[GK96] C. Goller and A. Küchler. Learning Task-Dependent Distributed Representations by Backpropagation Through Structure. In *Proc. of the ICNN-96*, volume 1, pages 347–352. IEEE, 1996.

[GP00] S. Gürdens and René Peralta. Validation of Cryptographic Protocols by Efficient Automatic Testing. In J. Etheredge and B. Manaris, editors, *Proc. of the 13th FLAIRS, Orlando*, pages 7–12. AAAI Press, 2000.

[HLS+96] D. Hutter, B. Langenstein, C. Sengler, J.H. Siekmann, W. Stephan, and A. Wolpers. Verication support environment (VSE). *Journal of High Integrity Systems*, 1(6):523–531, 1996.

[McC97] W.W. McCune. Solution of the Robbins Problem. *Journal of Automated Reasoning*, 3(19):263–276, 1997.

[MW97] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[RSS95] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Proc. of the 10th Annual Conference on Computer Assurance, Gaithersburg*. IEEE Press, 1995.

[RV99] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 292–2296. Springer, 1999.

[Sch97] J. Schumann. Automatic Verification of Cryptographic Protocols with SETHEO. In W. McCune, editor, *Proc. of the 14th CADE, Townsville*, number 1249 in LNAI, pages 87–11. Springer, 1997.

[Sch99] S. Schulz. System Abstract: E 0.3. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 297–391. Springer, 1999.

[Sch00] S. Schulz. *Learning Search Control Knowledge for Equational Deduction*. Number 230 in DISKI. Akademische Verlagsgesellschaft Aka GmbH Berlin, 2000. Ph.D. Thesis, Fakultät für Informatik, Technische Universität München.

[Sch01] S. Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, number 2083 in LNAI, pages 370–375. Springer, 2001.

[SKG97] S. Schulz, A. Küchler, and C. Goller. Some Experiments on the Applicability of Folding Architecture Networks to Guide Theorem Proving. In D.D. Dankel II, editor, *Proc. of the 10th FLAIRS, Daytona Beach*, pages 377–381. Florida AI Research Society, 1997.

[SS97] C.B. Suttner and G. Sutcliffe. The TPTP Problem Library (TPTP v2.1.0). Technical Report AR-97-01 (TUM), 97/04 (JCU), Institut für Informatik, Technische Universität München, Munich, Germany/Department of Computer Science, James Cook University, Townsville, Australia, 1997. Jointly published.

[SSY94] G. Sutcliffe, C.B. Suttner, and T. Yemenis. The TPTP Problem Library. In A. Bundy, editor, *Proc. of the 12th CADE, Nancy*, number 814 in LNAI, pages 252–266. Springer, 1994.

[ST85] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[SWL+94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive Composition of Astronomical Software from Subroutine Libraries. In A. Bundy, editor, *Proc. of the 12th CADE, Nancy*, number 814 in LNAI, pages 341–355. Springer, 1994.

[WAB+99] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, G. Jung, E. Keen, C. Theobalt, and D. Topic. System Abstract: SPASS Version 1.0.0. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 378–382. Springer, 1999.

[Wei99] C. Weidenbach. Toward an Automatic Analysis of Security Protocols in First-Order Logic. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, number 1632 in LNAI, pages 314–328. Springer, 1999.

The most recent version of the E equational theorem prover and the other programs used for the experiments described in this paper are available from `http://wwwjessen.informatik.tu-muenchen.de/~schulz/WORK/eprover.html`.