Automatic Learning in Proof Planning

Mateja Jamnik[1,2] and **Manfred Kerber**[2] and **Martin Pollet**[3,2]

Abstract. In this paper we present a framework for automated learning within mathematical reasoning systems. In particular, this framework enables proof planning systems to automatically learn new proof methods from well chosen examples of proofs which use a similar reasoning pattern to prove related theorems. Our framework consists of a representation formalism for methods and a machine learning technique which can learn methods using this representation formalism. We present the implementation of this framework within the ΩMEGA proof planning system, and some experiments we ran on this implementation to evaluate the validity of our approach.

1 Introduction

Proof planning [3] is an approach to theorem proving which uses proof methods rather than low level logical inference rules to find a proof of a theorem at hand. A proof method specifies a general reasoning pattern that can be used in a proof, and typically represents a number of individual inference rules. For example, mathematical induction can be encoded as a proof method. Proof planners search for a proof plan of a theorem which consists of applications of several methods. An object level logical proof may be generated from a successful proof plan. Proof planning is a powerful technique because it often dramatically reduces the search space, since the search is done on the level of abstract methods rather than on the level of several inference rules that make up a method. Therefore, typically, there are fewer methods than inference rules explored in the search space. Proof planning also allows reuse of proof methods, and moreover generates proofs where the reasoning strategies of proofs are transparent, so they may have an intuitive appeal to a human mathematician. Indeed, the communication of proofs amongst mathematicians can be viewed to be on the level of proof plans.

One of the ways to extend the power of a proof planning system is to enlarge the set of available proof methods. This is particularly beneficial when a class of theorems can be proved in a similar way, hence a new proof method can encapsulate the general reasoning pattern of a proof for such theorems. A proof method in proof planning basically consists of a triple – preconditions, postconditions and a tactic. A tactic is a program which given that the preconditions are satisfied transforms an expression representing a subgoal in a way that the postconditions are satisfied by the transformed subgoal. If no method on an appropriate level is available in a given planning state, then a number of lower level methods (with inference rules as the lowest level methods) have to be applied in order to prove a given theorem. Alternatively, a new method can be added by the developer of a system. However, this is a very knowledge intensive task and hence, presents a difficulty in applying a proof strategy to many domains.

In this work, we show how a system can learn new methods automatically given a typically small number of well chosen examples of related proofs of theorems. This is a significant improvement, since examples (e.g., in the form of classroom example proofs) exist typically in abundance, while the extraction of methods from these examples can be considered as a major bottleneck of the proof planning methodology.

The idea is that the system starts with learning simple proof methods. As the database of available proof methods grows, the system can learn more complex proof methods. Inference rules can be treated as methods by assigning to them pre- and postconditions. Thus, from the learning perspective we can have a unified view of inference rules and methods as given sequences of primitives from which the system is learning a pattern. We will refer to all the existing methods available for the construction of proofs as primitive methods. As new methods are learnt from primitive methods, these too become primitive methods from which yet more new methods can be learnt. Clearly, there is a trade-off between the increased search space due to a larger number of methods, and increasingly better directed search possibilities for subproofs covered by the learnt methods. Namely, on the one hand, if there are more methods, then the search space is potentially larger. On the other hand, the organisation of a planning search space can be arranged so that the newly learnt, more complex methods are searched for first. If a learnt method is found to be applicable, then instead of a number of planning steps (that correspond to the lower level methods encapsulated by the learnt method), a proof planner needs to make one step only. Generally, proof plans consisting of higher level methods will be shorter than their corresponding plans that consist of lower level methods. Hence, the search for a complete proof plan is shallower, but also bushier. In order to measure this trade-off between the increased search space and better directed search, an empirical study is carried out in the second part of this paper.

Here, we present a hybrid proof planning system LEARNΩMATIC, which uses the existing proof planner ΩMEGA [1], and combines it with our own machine learning system. This enhances the ΩMEGA system by providing it with the learnt methods that ΩMEGA can use in proving new theorems.

Automatic learning by reasoning systems is a difficult and ambitious problem. Our work demonstrates one way of starting to address this problem, and by doing so, it presents several contributions to the field. First, although machine learning techniques have been around for a while, they have been relatively little used in reasoning systems. Making a reasoning system learn proving strategies from examples, much like children learn to solve problems from examples demonstrated to them by the teacher, is hard. Our work makes an important step in a specialised domain towards a proof planning system that can reason *and* learn.

[1] University of Cambridge Computer Laboratory, J.J. Thomson Avenue, Cambridge, CB3 0FD, UK, http://www.cl.cam.ac.uk/~mj201
[2] School of Computer Science, The University of Birmingham, Birmingham B15 2TT, England, UK, http://www.cs.bham.ac.uk/~mmk
[3] Fachbereich Informatik, Universität des Saarlandes, 66041 Saarbrücken, Germany, http://www.ags.uni-sb.de/~pollet

Second, proof methods have complex structures, and are hence very hard to learn by the existing machine learning techniques. We approach this problem by abstracting as much information from the proof method representation as needed, so that the machine learning techniques can tackle it. Later, after the reasoning pattern is learnt, the abstracted information is restored as much as possible.

Third, unlike in some of the existing related work [5, 14], we are not aiming to improve ways of directing proof search within a fixed set of primitives. In theorem proving systems these primitives are typically inference steps or tactics, and in proof planning systems these primitives are typically proof methods. Rather, we aim to learn the primitives themselves, and hence improve the framework and reduce the search within the proof planning environment. Namely, instead of searching amongst numerous low level proof methods, a proof planner can now search for a newly learnt proof method which encapsulates several of these low level primitive methods.

Running Example We demonstrate our approach with running examples. There is a large class of residue class theorems in group theory that can be proved using the same pattern of reasoning. Their use is well documented in [10]. Here are examples of three residue class theorems (where $\mathbb{Z}_i$ is the residue class of integers modulo i, and the lambda expression is the operation over this set):

1. *closed-under*$(\mathbb{Z}_3, (\lambda x\, \lambda y_{\blacksquare}\, x \bar{+} (x \bar{+} y)))$
2. *associative-under*$(\mathbb{Z}_3, (\lambda x\, \lambda y_{\blacksquare}\, (x \bar{\times} y)))$
3. *commutative-under*$(\mathbb{Z}_2, (\lambda x\, \lambda y_{\blacksquare}\, (x \bar{+} y)))$

The pattern of reasoning to prove them is as follows. First, the definitions are expanded (e.g., *closed-under, associative-under, commutative-under*). Then, all of the statements on residue classes are rewritten into corresponding statements on integers by transferring the residue class set into a set of corresponding integers. Then, the proofs diverge: if the statements are universally quantified, then an exhaustive case analysis over all elements of the set is carried out. If the statements are existentially quantified, then all elements of the set are examined until one is found for which the statements hold.

The aim of our work is to learn and generalise such patterns into proof methods that can then be used for proofs of other related theorems. We now show how this learning is done automatically within our framework.

2 Learning

The methods we aim to learn are complex and beyond the complexity that can typically be tackled in the field of machine learning. Thus, we first simplify the problem and aim to learn so-called *method outlines*. For this purpose we use a simple representation formalism which abstracts away as much information as possible for the learning process, which is described next. Second, we restore the necessary information as much as possible so that the proof planner can use the newly learnt method using the mechanism described in §3.[4]

Let us define the following language L, where P is a set of known identifiers of primitive methods used in a method that is being learnt:

- for any $p \in P$, let $p \in L$,
- for any $l_1, l_2 \in L$, let $[l_1, l_2] \in L$,
- for any $l_1, l_2 \in L$, let $[l_1|l_2] \in L$,
- for any $l \in L$, let $l^* \in L$,
- for any $l \in L$ and $n \in \mathbb{N}$, let $l^n \in L$,
- for any $list$ such that all $l_i \in list$ are also $l_i \in L$, let $T(list) \in L$.

"[" and "]" are auxiliary symbols used to separate subexpressions, "," denotes a *sequence*, "|" denotes a *disjunction*, "$*$" denotes a *repetition* of a subexpression any number of times (including 0), n a fixed number of times, and T is a constructor for a branching point ($list$ is a list of branches), i.e., for proofs which are not sequences but branch into a tree.[5] We refer to expressions in language L which describe compound methods as *method outlines*.

Learning Technique Method outlines are abstract methods which have a simple representation that is amenable to learning. The algorithm that learns method outlines can find an adequate minimal and least general generalisation within the given language restrictions. It is based on the generalisation of the simultaneous compression of well-chosen examples.

Our learning technique considers some typically small number of positive examples which are represented in terms of sequences of identifiers for primitive methods, and generalises them so that the learnt pattern is in language L. The pattern is of *smallest size* with respect to a defined heuristic measure of *size* [6], which essentially counts the number of primitives in an expression. The intuition for it is that a good generalisation is one that reduces the sequences of method identifiers to the smallest number of primitives (e.g., $[a^2]$ is better than $[a, a]$). The pattern is also *most specific* (or equivalently, least general) with respect to the definition of specificity *spec*. *spec* is measured in terms of the number of nestings for each part of the generalisation [6]. Again, this is a heuristic measure. The intuition for this measure is that we give nested generalisations a priority since they are more specific and hence less likely to over-generalise.

We take both, the size (first) and the specificity (second), in account when selecting the appropriate generalisation. If the generalisations considered have the same rating according to the two measures, then we return all of them. For example, consider two possible generalisations constructed by the learning algorithm described below: $[[a^2]^*]$ and $[a^*]$. According to size, $size([[a^2]^*]) = 1$ and $size([a^*]) = 1$. However, according to specificity, *spec*$([[a^2]^*]) = 2$ and *spec*$([a^*]) = 1$. Hence, the algorithm selects $[[a^2]^*]$.

Here is the learning algorithm. Given some number of examples (e.g., $e_1 = [a, a, a, a, b, c]$ and $e_2 = [a, a, a, b, c]$):

1. For every example e_i, split it into sublists of all possible lengths plus the rest of the list. We get a list of pattern lists pl_i, each of which containing patterns p_i.[6] E.g.:
   for e_1: $\{[[a], [a], [a], [a], [b], [c]]$, $[[a, a], [a, a], [b, c]]$, $[[a], [a, a], [a, b], [c]]$, $[[a, a, a], [a, b, c]]$, $[[a], [a, a, a], [b, c]], \dots\}$
   for e_2: $\{[[a], [a], [a], [b], [c]]$,$[[a, a], [a, b], [c]]$, $[[a], [a, a], [b, c]]$, $[[a, a, a], [b, c]]$, $[[a], [a, a, b], [c]], \dots\}$
2. If there is any branching in the examples, then recursively repeat this algorithm on every element of the list of branches.
3. For every example e_i and for every pattern list pl_i find sequential repetitions of the same patterns p_i in the same example. Using an exponent denoting the number of repetitions, compress them into p_i^c and hence pl_i^c. E.g.:
   $pl_1^c = \{[[a]^4, [b], [c]], [[[a, a]^2], [b, c]], \dots\}$

[4] Some information may be irrecoverably lost. In this case, some extra search in the application of the newly learnt methods will typically be necessary.

[5] Note the difference between the disjunction and the tree constructors: for disjunction the proofs covered by the method outline consist of applying either the left or the right disjunct. However, with the tree constructor every proof branches at that particular node to all the branches in the list. Note also, that there is no need for an empty primitive as it can be encoded with the use of the existing language. E.g., let ϵ be an empty primitive and we want to express $[a, b, [\epsilon|c], d]$. Then an equivalent representation without the empty primitive is $[a, [b|[b, c]], d]$. We avoid using the empty primitive as it introduces a large number of unwanted generalisation possibilities.

[6] Notice that there are n mod m ways of splitting an example of length n into different sublists of length m. Namely, the sublists of length m can start in positions $1, 2, \dots, n$ mod m.

$pl_2^c = \{[[a]^3, [b], [c]], [[a], [a, a], [b, c]], \dots\}$

4. For every compressed pattern $p_i^c \in pl_i^c$ of every example e_i, compare it with p_j^c in all other examples e_j, and find matching p^c (called m_i) with the same constituent pattern, which may occur a different number of times. E.g.:
   $m_1 = (pl_1^{c1}, pl_2^{c1})$ due to $[a]^4$ and $[a]^3$
   $m_2 = (pl_1^{c2}, pl_2^{c2})$, due to $[b, c]$ and $[b, c]$, etc.
5. If there are no matches m_i in the previous step, then generalise the examples by joining them disjunctively using the "|" constructor.
6. For every p_i^c in a matching, generalise different exponents to a "$*$" constructor, and the same exponents n to a constant n. E.g.:[7]
   for m_1: $[a]^4$ and $[a]^3$ are generalised to $[a]^*$;
   for m_2: $[b, c]$ and $[b, c]$ are generalised to $[b, c]$
7. For every p_g of a match, transform the rest of the pattern list on the left and on the right of p_g back to the example list, and recursively repeat the algorithm on them. E.g.:
   for m_1 in e_1: LHS= [ ], $p_g = [a]^*$, repeat on RHS=$[b, c]$
   for m_1 in e_2: LHS= [ ], $p_g = [a]^*$, repeat on RHS=$[b, c]$
   for m_2 in e_1: repeat on LHS= $[a, a, a, a]$, $p_g = [b, c]$, RHS= [ ]
   for m_2 in e_2: repeat on LHS= $[a, a, a]$, $p_g = [b, c]$, RHS= [ ].
8. Repeat recursively this algorithm on all possible generalisations inside the method outline (i.e., "*", exponent n, "|" and ",") in order to find nested generalisations (such as $[[a]^2]^*$). Terminate, when no success.
9. If there is more than one generalisation remaining at the end of the recursive steps, then pick the ones with the smallest size and among these the ones with the largest specificity. Else, the examples cannot be generalised. E.g.: for the examples above, not applicable yet; after the algorithm is repeated on the rest of our examples, the learnt method outline will be $[[a]^*, [b, c]]$.

The learning algorithm was implemented in Standard ML v.110. Its input are the sequences of proofs constructed in ΩMEGA. Its output are method outlines which are passed back to ΩMEGA. The algorithm was tested on several examples of proofs and it successfully produced the required method outlines.

In particular, for our example of the residue class theorems above, the sequences of methods identifiers from the proofs of these theorems are as follows:

1. {*defn-exp, $\forall_i$-sort,$\forall_i$-sort, convert-resclass-to-num, defn-exp, or-e-rec, simp-num-exp, simp-num-exp, simp-num-exp, simp-num-exp, simp-num-exp, simp-num-exp, simp-num-exp, simp-num-exp, simp-num-exp*}
2. {*defn-exp, $\forall_i$-sort,$\forall_i$-sort, $\forall_i$-sort,convert-resclass-to-num, or-e-rec, simp-num-exp*}
3. {*defn-exp, $\forall_i$-sort,$\forall_i$-sort, convert-resclass-to-num, or-e-rec, simp-num-exp, simp-num-exp, simp-num-exp, simp-num-exp, simp-num-exp*}

The algorithm described here learnt the expected method outline:

$$\textit{tryanderror} \equiv \big[\textit{defn-exp}, [\forall_i\textit{-sort}]^*, \textit{convert-resclass-to-num}, [[\textit{or-e-rec}]|[\textit{defn-exp, or-e-rec}]], \textit{simp-num-exp}^*\big]$$

Due to lack of space the reader is referred for the discussion of some properties of our learning algorithm to [6]. There are some disadvantages to our technique, mostly related to the run time speed of the algorithm relative to the length of the examples considered for learning. The algorithm can deal with relatively small examples, which we encounter in our application domain, in an optimal way. The complexity of the algorithm is exponential in the worst case. Hence, we use some heuristics for large and badly behaved examples [6].

[7] Notice that here is a point where our generalisation technique can overgeneralise, namely when there is a pattern in the exponents – like prime number – this is ignored and just a $*$ is selected.

3 Using learnt methods

Method outlines that have been learnt so far do not contain all the information which is needed for the proof planner to use them. For instance, they do not specify what the pre- and postconditions of methods are, they also do not specify how the number of loop applications of methods is instantiated when used to prove a theorem. Hence, we need to restore the missing information.

For each learnt outline we automatically create a method for which its precondition is fulfilled if there is a sequence of methods that is an instantiation of the method outline, and each method of the sequence is applicable. The postcondition introduces the new open goals and hypotheses resulting from applying the methods of the sequence to the current goal. We will call this kind of method a *learnt method*.

Since methods in ΩMEGA may be very complex, the precondition of a learnt method cannot be extracted from the pre- and postconditions of the *uninstantiated* methods in the method outline. Hence, we actually have to apply a method to produce a proof situation for which we can test the preconditions of the subsequent method in the method outline. That is, we have to perform proof planning guided by the learnt outline.

The precondition test performs a depth first search on the learnt method outline. Besides the choice points of the learnt method, i.e., disjunctions and number of repetitions for "$*$", we also have to decide to which open goal every single method in the method outline should be applied. Additionally, for methods containing parameters an instantiation has to be chosen. More details on the application test and on the learnt method reuse can be found in [6].

4 Evaluation – experiments

In order to evaluate the success of our approach, we carried out an empirical study in different problem domains on a number of theorems. This test set includes the theorems from which new methods were learnt, but most of them are new and more complex. In particular, we tested our framework on examples of residue classes (e.g., like our running examples), set theory[8] and group theory.[9] The aim of these experiments was to investigate if the proof planner ΩMEGA enhanced with the learnt methods, can perform better than the standard ΩMEGA planner.[10] The learnt methods were added to the search space in a way that their applicability is checked first, before the existing standard methods.

The measures that we consider are: *coverage* – the ability to prove new theorems with the help of learnt methods; *timing* – the time it takes to prove a theorem; *proof length* – the number of steps in the proof plan; and *matchings* – the number of all true and false attempts

[8] Here are a set theory theorem and a non-theorem: $\forall x, y, z_{\blacksquare}((x \cup y) \cap z) = (x \cap z) \cup (y \cap z)$ and $\forall x, y, z_{\blacksquare}(x \cup y) \cap z) = (x \cup z) \cap (y \cup z)$.

[9] Here are some group theory examples: $a \circ (((a^{-1} \circ b) \circ (c \circ d)) \circ f) = (b \circ (c \circ d)) \circ f$ and $(c \circ (b \circ (a^{-1} \circ (a \circ b^{-1})))) \circ (((d \circ a) \circ a^{-1}) \circ f) = c \circ (d \circ f)$.

[10] There are several reasons for comparing the performance of ΩMEGA with and without a learning capability, rather than with the performance of other learning reasoning systems. First, there are only few proof planners around. In the future, we plan to apply our learning approach to *λClam* proof planner in order to assess the benefits of our approach in other systems. Second, as discussed in §5, little work has been done on applying machine learning techniques to automated theorem provers, hence a direct comparison with another approach is infeasible.

to match candidate methods. Checking the candidate methods that may be applied in the proof, i.e., *matchings*, is by far the most expensive part of the proof search, and is hence the best measure to indicate the validity of our approach.

Table 1 compares the values of *matchings* and *proof length* for the three problem domains.[11] It compares the values for these measures when the planner searches for the proof with the standard set of available methods (column marked with S), and when in addition to these, there are also our newly learnt methods available to the planner (column marked with L). "—" means that the planner ran out of resources (four hours of CPU time) and could not find a proof plan.

Theorem	Matchings		Length	
	S	L	S	L
assoc-z3z-times	651	113	63	2
assoc-z6z-times	4431	680	441	2
average res. class	1362.0	219.5	134.0	2.0
closed-z3z-plusplus	681	551	49	34
closed-z6z-plusplus	3465	2048	235	115
average res. class	1438.8	918.3	101.0	57.3
average set examples	33.5	12.5	13.0	2.0
average group theory (simple)	94.2	79.0	15.5	8.3
average group theory (complex)	—	189.6	—	9.8

Table 1. Evaluation results.

Residue class domain: In the domain of residue classes, we gave our learning mechanism examples such that it learnt two new methods: *tryanderror* (as demonstrated in our running examples), and *choose*. The first two theorems in Table 1 (about associativity) and the average for all theorems of a similar type that we tested, can use the *tryanderror* method, if it is available to the planner. It is clear from Table 1 that the number of candidate methods that the planner has to check if they can be applied in the proof (i.e., *matchings*) is reduced by roughly a factor of six in the case where our newly learnt methods are available. As expected, the *proof length* is much shorter when using newly learnt methods, since the learnt methods encapsulate a pattern in which a number of other methods are used in the proof. In fact, *tryanderror* can in most cases prove the entire theorem.

The next two theorems (about the closed property) and the average of all theorems of a similar type, can use the *choose* method. *choose* can only prove a subpart of a theorem, hence a number of other methods need to be used in addition to *choose* in order to prove the theorem. Hence, the proofs of this type are longer than the proofs of the former type (that use *tryanderror*), as is evident also from Table 1. As expected, the improvement in the number of *matchings* reflects this, and is a factor of two on average. In general, the more complicated the theorem, the better is the improvement made by the availability of the learnt methods.

Set theory domain: A similar trend can be noticed in the case of set theory conjectures. We gave our learning mechanism examples from which it learnt one new method. This method consists of eliminating the universal quantifiers, then transforming statements about sets to statement about elements of sets, and then proving (with the Otter theorem prover) or disproving (with the Satchmo model generator) these statements. Since all of the theorems on which we tested our proof planner are of a similar type, we only give the average figures for the number of *matchings* and *proof length* in Table 1. As in the residue class case, there is a reduced number of *matchings* required when a learnt method is available. This number is roughly reduced by a factor of two. As expected, the *proof length* is smaller when a planner can use our learnt method.

[11] Note that assoc-z3z-times and closed-z3z-plusplus in Table 1 are our second and first running example, respectively.

Group theory domain: We notice an improvement also in the case of the group theory examples. Our learning mechanism learnt five new methods, but since some are recursive applications of others, we only tested the planner by using two newly learnt complex recursive methods. The methods simplify group theory expressions by applying associativity left and right methods, and then reduce the expressions by applying appropriate inverse and identity methods. The entries in Table 1 refer to two types of examples. First, we give the average figures for simple theorems that can be proved with standard *and* with learnt methods. Second, we give the average figures for complex theorems that can *only* be proved within the resource limits when our learnt methods *are* available to the planner.

It is evident from Table 1 that the number of *matchings* is improved, but it is only reduced by about 15%. We noticed that for some very simple theorems, a larger number of *matchings* is required if the learnt methods are available in the search space. However, for more complex examples, this is no longer the case, and an improvement is noticed. The reason for this behaviour is that additional methods increase the search space, so when there are only a few ways to apply methods in the case of simple theorems, this causes some overheads. In the case of complex examples, there are many more possible ways to apply methods, hence the presence of complex learnt methods causes small or no overheads, or in fact, a reduced number of *matchings*.

The success of our approach is also evident from the fact, that when our learnt methods are not available to the planner, then it cannot prove some complex theorems, i.e., the *coverage* of the system that uses learnt methods is increased. When trying to apply methods such as associativity left or right, for which the planner has no control knowledge about their application, it runs out of resources. Our learnt methods, however, provide control over the way the methods are applied in the proof, and enable a planner to generate a proof plan. Again, the *proof length* is reduced by using learnt methods, as expected.

On average, the *time* it took to prove theorems of reside classes with the newly learnt methods was up to 50% shorter than without such methods. For conjectures in set theory the proof search with learnt methods was about 15% shorter. The search in group theory took approximately 100% longer than without the learnt methods. The time results reflect in principle the behaviour of the proof search measured by method *matchings*, but also contain the overhead due to the current implementation for the reuse of the learnt methods (see §3). For example, the current proof situation is copied for the applicability test of the learnt method, and the new open goals and hypotheses resulting from a successful application are copied back into the original proof. This overhead could be reduced in later versions.

Analysis: As it is evident from the discussion above, in general, the availability of newly learnt methods that capture general patterns of reasoning improves the performance of the proof planner. In particular, the number of *matchings* (which are the most expensive part of the proof search) is reduced across domains, as indicated in Table 1. Furthermore, as expected, learnt methods cause proofs to be shorter, since they encapsulate a number of other methods. Also, the *time* is in general reduced when using learnt methods. There are some overheads, and in some cases these are bigger than the improvements. The *time* should be related to the reduced number of *matchings*, but it is not in all our cases (group theory), this indicates that our implementation of the execution of learnt methods, as described in §3, is not the most efficient, and can be improved.

In general, the *coverage* when using learnt methods is improved, which is also indicated by the fact that using learnt methods ΩMEGA can prove theorems that it can otherwise not prove.

The reason for the improvements described above is due to the fact that our learnt methods provide a structure according to which the existing methods can be applied, and hence they direct search. This structure also gives a better explanation why certain methods are best applied in particular combinations. For example, the simplification method for group theory examples indicates how the methods about associativity, inverse and identity should be combined together, rather than applied blindly in any possible combination.

5 Related work

Our approach to learning methods is related to techniques for learning macro-operators, which was originally proposed for planning in artificial intelligence [13, 9]. But within theorem proving and proof planning related work is scarce. E.g., some work has been done on applying machine learning techniques to theorem proving, in particular on improving the proof search [5, 14]. However, not much work has concentrated on high level learning of structures of proofs and extending the reasoning primitives within an automated theorem prover.

Silver [15] and Desimone [4] used precondition analysis which learns new inference schemas by evaluating the pre- and postconditions of each inference step used in the proof. A dependency chart between these pre- and postconditions is created, and constitutes the pre- and postconditions of the newly learnt inference schema. These schemas are syntactically complete proof steps, whereas the ΩMEGA methods contain arbitrary function calls which cannot be determined by just evaluating the syntax of the inference steps.

Kolbe, Walter, Melis and Whittle have done related work on the use of analogy [11] and proof reuse [8, 7]. Their systems require a lot of reasoning with one example to reconstruct the features which can then be used to prove a new example. The reconstruction effort needs to be spent in every new example for which the old proof is to be reused. In contrast, we use several examples to learn a reasoning pattern from them, and then with a simple application, without any reconstruction or additional reasoning, reuse the learnt proof method in any number of relevant theorems.

In terms of a learning mechanism, more recent work on learning regular expressions, grammar inference and sequence learning [16] is related. Learning regular expressions is equivalent to learning finite state automata, which are also recognisers for regular grammars. Muggleton has done related work on grammatical inference methods [12]. The main difference to our work is that these techniques typically require a large number of examples in order to make a reliable generalisation, or supervision or an oracle which confirms when new examples are representative of the inferred generalisation. Furthermore, these techniques only learn sequences. However, our language is larger than regular grammars as it includes constant repetitions of expressions and expressions represented as trees.

Related is also the work on pattern matching in DNA sequences [2], as in the GENOME project, and some ideas on our learning mechanism have been inspired by this work.

6 Conclusion and future work

In this paper we described a hybrid system LEARNΩMATIC, which is based on the ΩMEGA proof planning system enhanced by automatic learning of new proof methods. This is an important advance in addressing such a difficult problem, since it makes first steps in the direction of enabling systems to better their own reasoning power. Proof methods can be either engineered or learnt. Engineering is expensive, since every single new method has to be freshly engineered. Hence, it is better to learn, whereby we have a general methodology that enables the system to automatically learn new methods. The hope is that ultimately, as the learning becomes more complex, the system will be able to find better or new proofs of theorems across a number of problem domains.

There are several limitations of our approach that could be improved in the future. Namely, the learning algorithm may overgeneralise, so we need to examine what are good heuristics for our generalisation and how suboptimal solutions can be improved. In particular, we want to address the question how to deal with noise in the examples. In order to reduce unnecessary steps, the preconditions of the learnt methods would ideally be stronger. Currently, we use an applicability test to search if the preconditions of the method outline are satisfied. In the future, preconditions should be learnt as well. Finally, in order to model the human learning capability in theorem proving more adequately it would be necessary to model how humans introduce new vocabulary for new (emerging) concepts.

A demonstration of LEARNΩMATIC implementation can be found on the following web page: `http://www.cs.bham.ac.uk/~mmk/demos/LearnOmatic/`. Further information, also with links to papers with more comprehensive references can be found on `http://www.cs.bham.ac.uk/~mmk/projects/MethodFormation/`.

Acknowledgements We would like to thank Alan Bundy, Predrag Janičić, Achim Jung, and Stephen Muggleton for their helpful advice on our work, and Christoph Benzmüller, Andreas Meier, and Volker Sorge for their help with some of the implementation in ΩMEGA. This work was supported by EPSRC grant GR/M22031 and European Commission IHP Calculemus Project grant HPRN-CT-2000-00102.

REFERENCES

[1] C. Benzmüller, et al., 'ΩMEGA: Towards a mathematical assistant', in *14th Conference on Automated Deduction*, ed., W. McCune, number 1249 in LNAI, pp. 252–255, (1997). Springer.
[2] A. Brazma, 'Learning regular expressions by pattern matching', Technical Report TCU/CS/1994/1, Institute of Mathematics and Computer Science, University of Latvia, (1994).
[3] A. Bundy, 'The use of explicit plans to guide inductive proofs', in *9th Conference on Automated Deduction*, eds., E. Lusk and R. Overbeek, number 310 in LNCS, pp. 111–120, (1988).
[4] R.V. Desimone, 'Learning control knowledge within an explanation-based learning framework', in *Progress in Machine Learning – Proceedings of 2nd European Working Session on Learning, EWSL-87*, eds., I. Bratko and N. Lavrač, (1987). Sigma Press.
[5] M. Fuchs and M. Fuchs, 'Feature-based learning of search-guiding heuristics for theorem proving', *AI Comm.*, **11**, 175–189, (1998).
[6] M. Jamnik, M. Kerber, M. Pollet, and C. Benzmüller, 'Automatic learning of proof methods in proof planning', Technical Report CSRP-02-5, School of Computer Science, University of Birmingham, UK, (2002).
[7] T. Kolbe and J. Brauburger, 'PLAGIATOR — A Learning Prover', in *14th Conference on Automated Deduction*, ed., W. McCune, number 1249 in LNAI, pp. 256–259, (1997). Springer.
[8] T. Kolbe and C. Walther, 'Reusing Proofs', in *Proceedings of the 11th ECAI*, ed., A. Cohn, 80–84, Wiley, (1994).
[9] R.E. Korf, *Learning to Solve Problems by Searching for Macro-operators*, Pitman Publishing Ltd., 1985.
[10] A. Meier and V. Sorge, 'Exploring properties of residue classes', in *Symbolic Calculation and Automated Reasoning: The Calculemus 2000 Symposium*, eds., M. Kerber and M. Kohlhase, pp. 175–190, (2001). A K Peters.
[11] E. Melis and J. Whittle, 'Analogy in inductive theorem proving', *Journal of Automated Reasoning*, **22**(2), (1998).
[12] S. Muggleton, *Acquisition of Expert Knowledge*, Addison-Wesley, 1990.
[13] D. Ruby and D.F. Kibler, 'Learning subgoal sequences for planning', in *Proceedings of the 11th IJCAI*, ed., N.S. Sridharan, pp. 609–614, (1989). International Joint Conference on AI, Morgan Kaufmann.
[14] S. Schulz, *Learning Search Control Knowledge for Equational Deduction*, Ph.D. dissertation, Fakultät f. Informatik, TU München, 2000.
[15] B. Silver, 'Precondition analysis: Learning control information', in *Machine Learning 2*, eds., R.S. Michalski, et al, (1984). Tioga Press.
[16] R. Sun and L. Giles, eds. *Sequence Learning: Paradigms, Algorithms, and Applications*, number 1828 in LNAI, 2000. Springer.