Automatic Learning of Proof Methods in Proof Planning

MATEJA JAMNIK, *University of Cambridge Computer Laboratory, J.J. Thomson Avenue, Cambridge CB3 0FD, England, UK.*
www.cl.cam.ac.uk/~mj201

MANFRED KERBER, *School of Computer Science, The University of Birmingham, Birmingham B15 2TT, England, UK.*
www.cs.bham.ac.uk/~mmk

MARTIN POLLET, *Fachbereich Informatik, Universität des Saarlandes, 66041 Saarbrücken, Germany.*
www.ags.uni-sb.de/~pollet

CHRISTOPH BENZMÜLLER, *Fachbereich Informatik, Universität des Saarlandes, 66041 Saarbrücken, Germany.*
www.ags.uni-sb.de/~chris

Abstract

In this paper we present an approach to automated learning within mathematical reasoning systems. In particular, the approach 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 approach consists of an abstract representation for methods and a machine learning technique which can learn methods using this representation formalism. We present an implementation of the approach within the ΩMEGA proof planning system, which we call LEARNΩMATIC. We also present the results of the experiments that we ran on this implementation in order to evaluate if and how it improves the power of proof planning systems.

Keywords: automated reasoning, theorem proving, proof planning, knowledge acquisition, machine learning

1 Introduction

Proof planning [3] is an approach to theorem proving which uses so-called 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 expands to a number of individual inference rules. For example, an induction strategy 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 [4, 20]. The advantage is that search with methods can be much better structured according to the particular requirements of mathematical domains.

Proof planning also allows reuse of the same proof methods for different proofs, and, moreover, generates proofs where the reasoning patterns of proofs are transparent. When methods are designed appropriately, the level of proof plans can capture the level of communication of proofs amongst mathematicians. Hence proof plans can offer an intuitive appeal to a human mathematician.

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. Methods are typically implemented and added by the developer of a system. The development and encoding of proof methods by hand, however, is a laborious task. In this work, we show how a system can learn new methods automatically given a number of well-chosen (positive) 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.[1] In this paper we therefore present a hybrid proof planning system LEARNΩMATIC [14], which combines the existing proof planner ΩMEGA [1] with our own machine learning system [13]. This enhances the ΩMEGA system with an automated capability to learn new proof methods.

Automated 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.

1. Although machine learning techniques have been around for a while, they have been relatively little used in reasoning systems. Making a reasoning system learn proof patterns from examples, much like students 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.
2. 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.
3. Unlike in some of the existing related work (see Section 5), we are not aiming to improve ways of directing proof search within a fixed set of primitives. Rather, we aim to learn the primitives themselves, and to investigate whether this improves the framework and reduces the search space within the proof planning environment. Instead of searching amongst numerous low-level proof methods, a proof planner can now search with a newly learnt proof method which encapsulates several of these low-level primitive methods.

[1] Note that in this paper, we do not provide a systematic and automated way of choosing good examples – in our system, this is still the user's task, which does require some expert knowledge. Choosing good examples automatically is discussed as future work in Section 6.

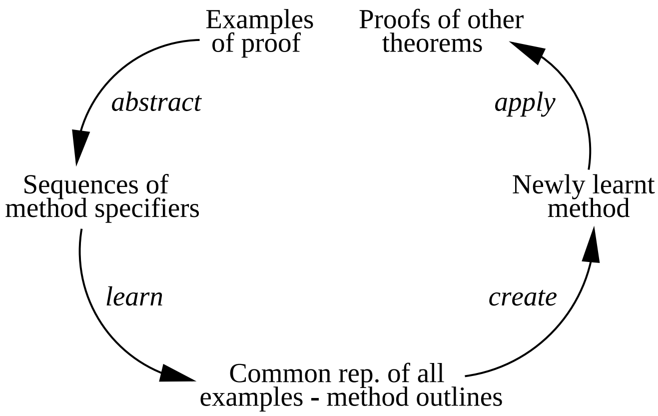

FIG. 1. Approach to learning proof methods.

FIGURE 1 gives the structure of our approach to learning proof methods, and hence an outline of the rest of this paper. In Section 2 we examine what needs to be learnt and give some examples of proofs that use a similar reasoning pattern. Then, in Section 3, we present the entire learning process. First, in Section 3.1, we simplify the method representation to ease the learning task. Second, we present our machine learning algorithm in Section 3.2. Third, in Section 3.3 we revisit our method representation and enrich it so that the newly learnt methods can be used in the ΩMEGA proof planner for proofs of other theorems. In order to assess the success of our approach, we go on in Section 4 to present some results of the evaluation tests that we ran on LEARNΩMATIC. Finally, we relate our work to that of others in Section 5, and conclude with some future directions in Section 6.

2 Motivation with Examples

A proof method in proof planning consists of a triple – preconditions, postconditions and a tactic.[2] 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 corresponding to the lowest-level methods) have to be applied in order to prove a given theorem. It often happens that a pattern of lower-level methods is applied time and time again in proofs of different problems. In this case it is sensible and useful to encapsulate this reasoning pattern in a new proof method. Such a higher-level proof method based on lower-level methods can be implemented and added to the system either by the user or by the developer of the system. However, this is a very knowledge intensive task. Hence, we present an alternative, namely a framework in which these methods can be learnt by the system automatically.

[2]This is an idealised view of a proof method. In practise, postconditions of proof methods are typically determined by executing the tactic part of the methods. So, when we speak of postconditions, it would be more appropriate and precise to speak of the effects of a method.

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 a 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 with 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. On the other hand, if a learnt method is applicable only seldom, then this may have negative effects on some performance criteria of the system (e.g., run time behaviour), but may not negatively affect others (e.g., even in the worst case, when a learnt method is not applicable or does not lead to a valid proof plan, the length of the generated proof plan does not increase, but remains unchanged). 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 can be expected to be performed in a shallower, but also bushier search space. In order to measure this trade-off between the increased search space and better directed search, an empirical study was carried out and is reported in Section 4. Typically, shorter proofs have a general advantage, since they are better suited for a user-adaptive presentation. We discuss this in Section 4.5.

The methods that LearnΩmatic learns are on a higher-level than the existing ones. Hence, the proofs constructed using them are not overwhelmed with unintuitive low-level proof steps, and can therefore be presented at a more abstract level. In this sense, such proofs reflect a higher-level idea of the proof, and can therefore be viewed as more human-oriented.

We demonstrate our ideas with examples that we used to develop and test LearnΩmatic. Most of the example conjectures can be automatically planned for in Ωmega with the Multi proof planner [19]. However, they demonstrate our approach, namely, they show how a proof planner can learn new methods automatically.

2.1 Group theory examples

The proofs of our first set of examples consist of simplifying an expression using a number of primitive simplification methods such as both (left and right) axioms of identity, both axioms of inverse, and the axioms of associativity (where e is the identity element, i is the inverse function, and LHS $\Rightarrow$ RHS stands for rewriting LHS to RHS).

$$\begin{array}{rcll} (X \circ Y) \circ Z & \Rightarrow & X \circ (Y \circ Z) & (\textsf{assoc-r}) \\ X \circ (Y \circ Z) & \Rightarrow & (X \circ Y) \circ Z & (\textsf{assoc-l}) \\ e \circ X & \Rightarrow & X & (\textsf{id-l}) \end{array} \qquad \begin{array}{rcll} X \circ e & \Rightarrow & X & (\textsf{id-r}) \\ X \circ X^i & \Rightarrow & e & (\textsf{inv-r}) \\ X^i \circ X & \Rightarrow & e & (\textsf{inv-l}) \end{array}$$

Here are two examples of proof steps which simplify given expressions and the inferences that are used:

$$\begin{array}{l} a \circ ((a^i \circ c) \circ b) \\ \quad \Downarrow (\textsf{assoc-l}) \\ (a \circ (a^i \circ c)) \circ b \\ \quad \Downarrow (\textsf{assoc-l}) \\ ((a \circ a^i) \circ c) \circ b \\ \quad \Downarrow (\textsf{inv-r}) \\ (e \circ c) \circ b \\ \quad \Downarrow (\textsf{id-l}) \\ c \circ b \end{array} \qquad \begin{array}{l} a^i \circ (a \circ b) \\ \quad \Downarrow (\textsf{assoc-l}) \\ (a^i \circ a) \circ b \\ \quad \Downarrow (\textsf{inv-l}) \\ e \circ b \\ \quad \Downarrow (\textsf{id-l}) \\ b \end{array}$$

Other examples include proofs for theorems such as $(a \circ (((a^i \circ b) \circ (c \circ d)) \circ f)) = (b \circ (c \circ d)) \circ f$. These three examples can be summarised in the following proof traces which are lists of method identifiers:

1. [assoc-l,assoc-l,inv-r,id-l],
2. [assoc-l,inv-l,id-l],
3. [assoc-l,assoc-l,assoc-l,inv-r,id-l].

It is clear that all three examples have a similar structure which could be captured in a new simplification method. Informally, one application of such a simplification method could be described as follows:

Precondition: *There are subterms in the initial term that are inverses of each other, and that are not separated by other subterms, but only by brackets.*

Tactic:

1. *Apply associativity* (assoc-l) *for as many times as necessary (including* 0 *times) to bring the subterms which are inverses of each other together, and then*
2. *apply inverse inference rule* (inv-r) *or* (inv-l) *to reduce the expression, and then*
3. *apply the identity inference rule* (id-l).

Postcondition: *The initial term is reduced, i.e., it consists of fewer subterms.*

The formal representation of the learnt method in our framework will be presented in Section 3.2.1.

Note that this is not the most general simplification method, because it does not use methods such as (assoc-r) and (id-r), but it is the one that is the least general generalisation of the given examples above. Note also that the application of this method may fail if the precondition is not strong enough. For instance, two terms may have to be brought together by the application of the (assoc-r) rule, which is not covered by the learnt method, since no example of this type has been provided. Also, should we want our system to learn a repeated application of this simplification method, then this can be achieved in another round of learning with suitable examples and methods. Alternatively, our set of initial examples that the system is learning from needs to include proofs of theorems such as $(c \circ (b \circ (a^i \circ (a \circ b^i)))) \circ (((d \circ a) \circ a^i) \circ f) = c \circ (d \circ f)$ which applies the above described simplification method three times.

2.2 Residue classes conjectures

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 [18]. Here are examples of three residue class theorems: (where $\mathbb{Z}_i$ is the residue class of integers modulo i)

1. *commutative-under*$(\mathbb{Z}_2, +)$
2. *associative-under*$(\mathbb{Z}_3, \times)$
3. *commutative-under*$(\mathbb{Z}_3, +)$

The pattern of reasoning to prove them is as follows. First, the definitions (e.g., *commutative-under, associative-under,*) are expanded (defn-exp), and quantifiers eliminated ($\forall_i$-sort). 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 (convert-resclass-to-num). Then, the proofs diverge: if the statements are universally quantified an exhaustive case analysis over all elements of the set is carried out (using a combination of elimination of disjuncts (or-e-rec), simplification (simp-num-exp), and reflexivity (reflex)). If the statements are existentially quantified, then all elements of the set are examined until one is found for which the statements hold (using a combination of disjunction introduction from left or right (ori-r, ori-l), simplification and reflexivity; see choose method in Section 3.2.1). Note that the three example theorems above are all universally quantified, but the set of theorems used in the evaluation tests (see Section 4) contains the existentially quantified theorems as well.

The proof trace for the above three theorems consist of a list of method identifiers used in the proof plans:

1. [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, reflex, reflex, reflex, reflex]
2. [defn-exp, $\forall_i$-sort, $\forall_i$-sort, $\forall_i$-sort, convert-resclass-to-num, or-e-rec, simp-num-exp, reflex, . . . , reflex]
3. [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, reflex, reflex, reflex, reflex, reflex, reflex, reflex, reflex, reflex]

The learnt generalisations for these proof traces are presented in Section 3.2.1.

2.3 Set theory conjectures

Another problem domain that we experimented with includes some theorems and non-theorems from set theory:

1. $\forall x, y, z. (x \cup y) \cap z = (x \cap z) \cup (y \cap z)$
2. $\forall x, y, z. (x \cup y) \cap z = (x \cup z) \cap (y \cup z)$
3. $\forall x, y, z. (x \cap y) \cap z = x \backslash (y \cup z)$

Although these problems are not very hard for automated theorem provers if a suitable representation is chosen, they may be hard to prove or disprove for existing automated theorem provers if attempted in a naive way. Their proofs consist of eliminating (introducing, in backwards reasoning) the universal quantifiers ($\forall_\mathsf{i}$), then applying set extensionality (set-ex) and definition expansions (defni) in order to get propositional or first order clauses (i.e., transforming statements about sets to statements about elements of sets), and then proving (with the Otter theorem prover, atp-otter) or disproving (with the Satchmo model generator, counterex-satchmo) these clauses. Here are the abstracted lists of method identifiers that describe these proofs:

1. $[\forall_\mathsf{i}, \forall_\mathsf{i}, \forall_\mathsf{i}, \mathsf{set\text{-}ext}, \forall_\mathsf{i}, \mathsf{defni}, \mathsf{defni}, \mathsf{atp\text{-}otter}]$
2. $[\forall_\mathsf{i}, \forall_\mathsf{i}, \forall_\mathsf{i}, \mathsf{set\text{-}ext}, \forall_\mathsf{i}, \mathsf{defni}, \mathsf{defni}, \mathsf{counterex\text{-}satchmo}]$
3. $[\forall_\mathsf{i}, \forall_\mathsf{i}, \forall_\mathsf{i}, \mathsf{set\text{-}ext}, \forall_\mathsf{i}, \mathsf{defni}, \mathsf{defni}, \mathsf{defni}, \mathsf{counterex\text{-}satchmo}]$

The learnt generalisations for these proof traces are presented in Section 3.2.1.

3 Learning

The representation of a problem is of crucial importance for the ability to solve it – a good representation of a problem often renders the search for its solution easy [25]. The difficulty is in finding a good representation. Our problem is to devise a mechanism for learning methods. Hence, the representation of a method is important and should make the learning process easy enough that we can learn useful information.

We start by presenting in Section 3.1 a simple representation formalism which abstracts away some detailed information in order to ease the learning process. Then, in Section 3.2 we describe the learning algorithm. Finally, we show in Section 3.3 how the necessary information is restored as much as possible so that the proof planner can use the newly learnt method. Some information may be irrecoverably lost. In this case, extra search in the application of the newly learnt methods will typically be necessary.

3.1 Method outline representation

The methods we aim to learn are complex and are beyond the complexity that can typically be tackled in the field of machine learning. Therefore, we first simplify the problem and aim to learn (using a variation of an existing learning technique) the so-called *method outlines*, and second, we reconstruct the full information as far as possible. Method outlines are expressed in the language that we describe here.

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 $list = (l_1, \ldots, l_k)$ such that $l_i \in L$ and $1 < i \leq k$, 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.[3] Let the set of primitives P be {assoc-l, assoc-r, inv-l, inv-r, id-l, id-r}. Using this language, the tactic of our simplification method described by the three group theory examples above can be expressed as:

$$\mathsf{simplify} \equiv \big[\mathsf{assoc\text{-}l}^*, [\mathsf{inv\text{-}r}|\mathsf{inv\text{-}l}], \mathsf{id\text{-}l}\big].$$

We refer to expressions in language L which describe compound methods as *method outlines*. simplify is a typical method outline that we aim our system to learn automatically.

3.2 Machine learning algorithm

Method outlines are abstract methods which have a simple representation that is amenable to learning. We now present an algorithm which can learn method outlines from a set of well-chosen examples. The algorithm is based on least general generalisation [23, 24], and on the generalisation of the simultaneous compression of well-chosen examples.

As with compression algorithms in general, we have to compromise the expressive power of the language used for compression with the time and space efficiency of the compression process. Optimal compression – in the sense of Kolmogorov complexity – can be achieved by using a Turing-complete programming language. However, optimal compression is not computable in general, that is, there is no algorithm which finds the shortest program to represent any particular string. As a compromise we selected regular expressions with explicit exponents and branching points, which seem to offer a framework that is on the one hand, general enough for our purpose, and on the other

[3] 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 – this is commonly known as the OR branch. However, with the tree constructor every proof branches at that particular node to all the branches in the list – this is commonly known as the AND branch.

Note also, that there is no need for an empty primitive as it can be encoded with the use of 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.

hand, (augmented with appropriate heuristics) sufficiently efficient.[4] 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 such as those we presented without the use of any heuristic.

Our learning technique considers some number of positive examples[5] which are represented in terms of lists 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 this defined measure of size, which essentially counts the number of primitives in an expression (where $l_1, l_2, p \in L$, $p \in P$, $n \in \mathbb{N}$ for some finite n, len is length of a list function, and $list$ is a list of expressions from L).

$$\begin{aligned}
size([l_1, l_2]) &= size(l_1) + size(l_2)\\
size([l_1|l_2]) &= size(l_1) + size(l_2)\\
size(T(list)) &= \textstyle\sum_{i=1}^{len(list)} size(l_i) \text{ where } l_i \in list\\
size(l_1^n) &= size(l_1)\\
size(p) &= 1\\
size(l_1^*) &= size(l_1)
\end{aligned}$$

This is a heuristic measure of size and 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 $spec$ which is measured in terms of the number of nestings for each part of the generalisation.

$$\begin{aligned}
spec([l_1, l_2]) &= 1 + spec(l_1) + spec(l_2)\\
spec([l_1|l_2]) &= 1 + spec(l_1) + spec(l_2)\\
spec(T(list)) &= 1 + \textstyle\sum_{i=1}^{len(list)} spec(l_i) \text{ where } l_i \in list\\
spec(l_1^n) &= 1 + spec(l_1)\\
spec(p) &= 0\\
spec(l_1^*) &= 1 + spec(l_1)
\end{aligned}$$

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 overgeneralise.

In our experiments, we take both, the size first (choose smallest size), and the specificity second (choose highest specificity) into account when choosing the generalisation. If the generalisations considered have the same rating according to the

[4] Our chosen language L (see Section 3.1) cannot express all method outlines. For example, we cannot express an outline that a method m_1 (e.g., definition unfolding) should be applied as often as possible, then a different method m_2 should be applied, and finally a third method m_3 (e.g., definition folding) should be applied exactly as often as the first method m_1. In our language we would have to overgeneralise this to $[m_1^*, m_2, m_3^*]$ (unless we know the number of method applications explicitly and this stays the same in all example proofs).

[5] We use positive examples only, because these are readily available. Negative examples are useful only if they are "near-misses" of proof attempts which uncover important features of the proof. Constructing and choosing informative negative examples is non-trivial, requires a lot of analysis and reasoning, and detracts from the main goal of our research. However, it would be an interesting topic for future research.

two measures, then we return all of them. For example, consider two possible generalisations: $[[a^2]^*]$ 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 chooses $[[a^2]^*]$.

Note that there are other ways of selecting a generalisation and finding a different compromise between size (keeping learnt expressions small) and specificity (keeping learnt expressions close to the examples). For instance, one could vary the value of the following formula $\alpha \cdot size(l_i) + (1 - \alpha) \cdot spec(l_i)$ by changing the value of α in order to select a suitable generalisation l_i. The value of α could depend on the degree to which the generalisation should be concise and general/specific (e.g., sometimes it may be beneficial to overgeneralise). Moreover, there are other possible heuristic measures to select a generalisation. We defined and chose size and specificity that are suitable measures in our problem domains and with our set of theorems. In the range between specificity and generality, we tend to (slightly) overgeneralise, but the test results in Section 4 demonstrate that our choice is a suitable one.

Here is the learning algorithm. Given some number of examples e_i (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 contains patterns p_i.[6] E.g.:
   - for e_1: $\{[[a], [a], [a], [a], [b], [c]],\ [[a, a], [a, a], [b, c]],\ [[a], [a, a],\ [a, b],\ [c]],\ [[a, a, a], [a, b, c]],\ [[a], [a, a, a],\ [b, c]], \ldots\}$
   - for e_2: $\{[[a], [a], [a], [b], [c]],\ [[a, a], [a, b], [c]],\ [[a],\ [a, a],\ [b, c]],\ [[a, a, a],\ [b, c]],\ [[a], [a, a, b], [c]], \ldots\}$
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.:
   - $pl_1^c = \{[[a]^4, [b], [c]], [[[a, a]^2], [b, c]], \ldots\}$
   - $pl_2^c = \{[[a]^3, [b], [c]], [[a], [a, a], [b, c]], \ldots\}$
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 m_k 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_k 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, and hence obtain p_g. E.g.:[7]
   - for m_1: $[a]^4$ and $[a]^3$ are generalised to $p_g = [a]^*$
   - for m_2: $[b, c]$ and $[b, c]$ are generalised to $p_g = [b, c]$

[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 $1, 2, \ldots, n$ mod m.

[7] Notice that here is a point where our generalisation technique can overgeneralise. For instance, when there is a pattern in the exponents, e.g., all exponents are prime numbers, then this is ignored and just a $*$ is selected.

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. 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. E.g.: after the algorithm is repeated on the rest of our examples, the learnt method outline will be $[[a]^*, [b, c]]$.

The learning algorithm is implemented in SML. Its inputs are the sequences of method identifiers from proofs that were 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.

3.2.1 Learnt method outlines for the examples

For the examples introduced in Section 2 our learning algorithm generates the following method outlines:

- Group theory:

$$\mathsf{simplify} \equiv \big[\mathsf{assoc\text{-}l}^*, [\mathsf{inv\text{-}r}|\mathsf{inv\text{-}l}], \mathsf{id\text{-}l}\big].$$

- Residue classes:

$$\begin{aligned}\mathsf{tryanderror} \;\equiv\; & \big[\mathsf{defn\text{-}exp}, [\forall_i\text{-}\mathsf{sort}]^*, \mathsf{convert\text{-}resclass\text{-}to\text{-}num},\\ & [[\mathsf{or\text{-}e\text{-}rec}]|[\mathsf{defn\text{-}exp},\ \mathsf{or\text{-}e\text{-}rec}]], \mathsf{simp\text{-}num\text{-}exp}^*, \mathsf{reflex}^*\big]\end{aligned}$$

- Set theory:

$$\mathsf{learnt\text{-}set} \;\equiv\; \big[[\forall_\mathsf{i}]^3, \mathsf{set\text{-}ext}, \forall_\mathsf{i}, \mathsf{defni}^*, [\mathsf{atp\text{-}otter}|\mathsf{counterex\text{-}satchmo}]\big]$$

As mentioned before, the method outline simplify for the group theory examples is not the most general one, as the examples that it was learnt from did not contain the use of the right identity method, for example. Furthermore, it is only a single application of simplification. However, we tested our learning algorithm also on examples that use this single simplify method several times. As expected, the learning mechanism learnt a method outline which is a repeated application of simplify, namely rep-simplify = simplify*. We also tested the learning mechanism on examples that use methods such as right identity and right associativity, and altogether learnt five new method outlines, some of which are repeated applications of others.

In the domain of residue classes, the learning mechanism also learnt another method outline called choose. When fully fleshed into an ΩMEGA method (see Section 3.3), this method proves a subpart of proofs for theorems of residue classes. Namely, given a theorem with an existential quantifier, statements on integers are combined using a disjunction in a particular normal form (from the right side). Then, each disjunct has to be checked until one is found that is true for the statement. Hence, the method choose starts inspecting the right disjuncts until either the right (ori-r) or the left (ori-l) one is true, which is then followed with the rest of the proof, in this case with the application of reflexivity (reflex). This proof pattern is learnt and captured in the method outline:

$$\mathsf{choose} = \big[\mathsf{defn\text{-}exp},\ \mathsf{ori\text{-}r}^{*}, [\mathsf{reflex} \mid [\mathsf{ori\text{-}l},\ \mathsf{reflex}]]\big]$$

The method corresponding to the third method outline *learnt-set*, i.e., for set theory examples, transforms a higher-order problem into a propositional logic one, which is much easier to prove or disprove, since it is a decidable problem. The method does not eliminate search altogether, but makes it, in this case, much more tractable. Notice also, that the method outline learnt-set applies the elimination of the universal quantifier ($\forall_{\mathsf{i}}$) only three times. This is consistent with the examples from which the method outline was learnt, but in general the quantifier elimination would be applied any required number of times, which could be denoted with a star construct in the method outline. This shows that the quality of a method outline learnt from the examples depends on the quality of the input examples. Hence, it is important to use well-chosen examples when learning new methods. Note, however, that sometimes a slight over-generalisation might be beneficial. Also note that any learning can work only if in the domain there is some structure or regularity which can be exploited.

3.2.2 Properties

Let us look at some properties of our learning algorithm:

PROPERTY 3.1
Given a number of examples, the algorithm learns a generalisation which is at least as general as all examples.

In order to see this property let a language expression r stand for the set of all expressions that are just sequences of primitive expressions. Then an expression r_1 is more general than another r_2 if each primitive expression of the set of sequences for r_2 is a proper subset of that for r_1. In the algorithm only the steps (5) and (6) are critical since all others do not change the generality of the expressions. Only steps (5) and (6) perform a generalisation, (5) in form of a disjunction, (6) in form of a star. Since a disjunction covers each of its disjuncts, and a star each of its components as well, the property follows.

PROPERTY 3.2
The learning algorithm is exponential.

In terms of computational complexity, the algorithm is quadratic in step (1)[8] and is exponential in step (7), since we try every possible combination here. All other steps are linear. The complexity of step (7) could be improved by using the initially computed information about all sublists of an example list, rather than recomputing it in every recursive step.

[8] An example list of length n is split into all together n^2 different sublists: there are n sublists of length 1, $n-1$ sublists of length 2, $n-2$ of length 3, $n-k+1$ of length k and so on, and 1 sublist of length n. Hence, in total, there are n^2 sublists of different fixed lengths. Notice that there exist algorithms, e.g., suffix trees, which run this step in linear time.

Since step (7) is exponential, our learning algorithm does not run efficiently for large examples.[9] In case the algorithm needs to be used for very large examples, we implemented some heuristic optimisations. These prune the number of generated matches. Good heuristics are those which select matches that make a big impact on the size of the final generalisations. For example, a good heuristic is to pick a pattern match whose pattern of smallest size forms a maximal sublist of the original example. This enables the algorithm to deal with very large examples (e.g., lists of length 2000) which are way beyond the length of examples that we expect for learning our method outlines. Clearly, using such heuristic learning may miss the best generalisation (according to the measures defined above). The user of our LearnΩmatic system can choose whether to use the heuristic optimisations in the learning mechanism or not. Users could also define their own heuristics, but this is left for future work.

3.3 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. In our approach, we restore the missing information by search.

In the particular case of our implementation in the Ωmega proof planning system, important information which is needed for the application of methods – but which is lost in the abstraction process – are parameters for the methods that constitute the newly learnt method. Concretely, the methods which make up the new learnt method in Ωmega take some (or no) parameters. These can be in the form of position information indicating where in the expression the method is applied, or a term naming the concept for which the definition should be expanded, or instantiating a term used by the method, etc. The parameters of a method are supplied by control-rules to reduce and to direct the search performed by the proof planner. For example, the parameter in the definition expansion method *defn-exp* names the concept that should be expanded. The possible relevant control-rules can be of the form 'Expand only definitions of the current theory' or 'Prefer definition expansion of the head symbol of the formula to be proved'.

A set of methods together with a set of control-rules defines a planning strategy of Ωmega's multi-strategy proof planner Multi [19]. Note that control-rules of a strategy are used not only for determining the parameters of methods, but also to prefer or reject methods according to the current proof situation.

For each learnt method outline we automatically build a method. The precondition of a learnt method employs search that is guided and structured by the method outline; that is, we perform search guided by the method outlines in order to analyse whether the learnt method 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*.

[9] However, we argue that proof methods that are being learnt typically do not consist of a large number of low-level methods. Indeed our algorithm runs efficiently on all the tested examples.

The precondition of a learnt method cannot be extracted from the pre- and post-conditions of the *uninstantiated* methods in the method outline, because the formulae introduced by the postcondition depend on the formulae that fulfil the preconditions. 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 pattern which is captured by the method outline.[10]

In detail, the applicability test is realised by the following algorithm:

1. Copy the current proof situation. Initialise a stack with a pair $(P_0; \emptyset)$, where P_0 is the initial learnt method outline and $\emptyset$ stands for the empty history.
2. Take the first pair from the stack:
   (a) If this pair is $([[P_1|P_2], P']; \mathcal{H})$, then put $([P_1, P']; \mathcal{H})$ and $([P_2, P']; \mathcal{H})$ back on the stack. For $([P^n, P']; \mathcal{H})$ put $([P, P^{n-1}, P']; \mathcal{H})$ on the stack. In the case of $([P^*, P']; \mathcal{H})$, return $(P'; \mathcal{H})$ and $([[P, P^*], P']; \mathcal{H})$.[11]
   (b) If the pair is $([m, P']; \mathcal{H})$ where m is a method-name, then test the precondition of m for all open goals (and for all possible instantiations of method parameters, if the method contains parameters). Each satisfied test of preconditions results in a partial matching μ_i of m for the corresponding goal (and parameter). The partial matchings $([\mu_1, P'], \mathcal{H}), \ldots, ([\mu_n, P'], \mathcal{H})$ are put on the stack. If m is not applicable, then backtrack the difference between the current history $\mathcal{H}$ and the history of the next pair of the stack.
   (c) If the pair is $([\mu_i, P']; \mathcal{H})$ where μ_i is a partially instantiated method, then apply the postconditions of μ_i to the copied proof and put $(P'; (\mu_i, \mathcal{H}))$ on the stack.
   (d) If the pair is $([\ ]; \mathcal{H})$ where $[\ ]$ denotes the empty outline, an instantiation of the learnt outline is found. That means, a particular sequence of methods, corresponding to the method outline, has been successfully applied and be found in $\mathcal{H}$.
3. If the stack is empty, then it was not possible to apply the learnt method outline; otherwise continue with step (2).

Notice that the application of the method introduces new open lines and new hypotheses resulting from the application of methods in $\mathcal{H}$ into the proof.

The learnt method may contain other learnt methods. That is, the applicability test in (2)(b) may recursively call this same algorithm again within the applicability test of an embedded learnt method.

Our implementation of the applicability test causes an overhead in the run time behaviour of the system. This is because the current proof is firstly copied in step (1) of the applicability test of the learnt method, and secondly in case of an application of the method the new open goals and hypotheses are copied back into the original proof. These two copying steps are carried out in order to avoid an interference between the planning process of Multi in the current proof situation, and the planning process

[10] One may suggest that our system learns tactics rather than methods as we have not mechanised the learning of preconditions. Such a suggestion is not entirely correct, since we can use the learnt methods in proof planning. It is true, however, that because of the increasing complexity of methods the originally clear difference between tactics and methods is getting increasingly blurred – not only in our approach but in proof planning in general.

[11] There is a counter for the operator $*$, the evaluation of this operator is only performed until an upper bound is reached. This guarantees the termination of the applicability test.

inside the applicability test of the method outline. The inefficiency due to overhead could be avoided in a complete re-implementation of the MULTI proof planner.

4 Evaluation and Experiments

In order to evaluate our approach, we carried out an empirical study in different problem domains. In particular, we tested our framework on examples of group theory, residue classes and set theory. 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. The learnt methods were added to the search space in conjunction with a heuristic (control-rule) specifying that their applicability is checked first, that is, before the existing standard methods.

The measures that we consider are:

1. *matchings* – the number of all true and false attempts to match methods that are candidates for application in the proof plan;
2. *proof length* – the number of steps in the proof plan;
3. *timing* – the time it takes to prove a theorem;
4. *coverage* – the ability to prove theorems.

In order to perform these tests we have built different counters in the program. The counter *matchings* counts the successful and unsuccessful application tests of methods. It also contains the method matchings checked by the search engine in the applicability tests of learnt methods (see Section 3.3). *Matchings* provides an important measure, since on the one hand, it indicates how directed the search for a proof is. On the other hand, checking the candidate methods that may be applied in the proof is by far the most expensive part of the proof search. Hence, *matchings* is a good measure to compare the performance of the two approaches (i.e., with and without learnt methods) while it is also independent of potential implementation inefficiencies.

The development set usually consists of a small number of examples, in particular, for the examples in the domains discussed in this paper it consisted of three example theorems (see Section 2). The test set consists of a number of theorems, which are new, more complex, and significantly diverse from the development set. It excludes the proofs from which the new methods were learnt.

The size of our test sample was relatively small in group theory: we tested our learnt methods on 8 theorems, but large in other domains: we had 881 theorems of residue classes and 120 conjectures of set theory.

Moreover, we chose our test set to be characteristic of the problem domain in general. Furthermore, notice that some evaluation measures, e.g., *proof length* and *coverage* are independent of the size of the test set. Namely, some inspection of the approach clearly indicates that the proof plans that use learnt methods will be shorter, and from the domain of group theory, it is clear that new theorems are proved that otherwise could not be.

Table 1 compares the values of *matchings* and *proof length* for the three problem domains. In each problem domain we break down the results according to the type of theorems under consideration (e.g., how complex they are, what pattern of reasoning or proof methods their proofs may use, how many variables are in them). The table compares the values for these measures when the planner searches for the proof with

Domain	Type of Theorems	Matchings S	Matchings L	Matchings $\frac{S}{L}$	Length S	Length L	Length $\frac{S}{L}$
Group theory	simple	94.2	79.0	1.19	15.5	8.3	1.87
	complex	—	189.6	—	—	9.8	—
Residue Class $\mathbb{Z}_3$	choose	691.0	656.0	1.05	39.3	33.0	1.19
	tryanderror	425.3	82.1	5.80	38.6	2.0	19.30
	both	552.9	323.2	1.71	39.7	19.0	2.09
Residue Class $\mathbb{Z}_6$	choose	751.2	713.8	1.05	35.2	29.1	1.21
	tryanderror	2309.5	402.9	5.73	218.2	2.0	109.10
	both	2807.8	1419.3	1.98	185.9	73.0	2.55
Residue Class $\mathbb{Z}_9$	choose	1996.1	1640.5	1.22	111.2	78.4	1.42
	tryanderror	4769.1	1132.2	4.21	453.1	2.0	226.55
	both	6931.6	3643.4	1.90	438.7	163.0	2.69
Set theory	1 variable	26.9	42.0	0.64	6.6	6.6	1.00
	3 variables	45.6	14.9	3.06	10.9	2.0	5.45
	5 variables	48.1	28.7	1.68	12.7	4.0	3.17

TABLE 1. Evaluation results.

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 (i.e., four hours of CPU time) and could not find a proof plan.

4.1 Group theory domain

In the group theory domain, our learning mechanism learnt five new methods, but since some are repeated applications of others, we only tested the planner by using two newly learnt complex compound methods.[12]

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 (see Section 2.1). 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 be proved *only* when the planner has our learnt methods.

It is evident from Table 1 that the number of *matchings* is improved, but it is only reduced by about 15%. We noticed that the simpler the theorem, the smaller the improvement. In fact, for some very simple theorems, a larger number of *matchings* is required if the learnt methods are available in the search space. The reason for this behaviour is that there are only a few standard methods available initially in the group theory domain. Hence, any additional learnt method will noticeably increase the search space. Also, the application test for learnt methods may be expensive, especially when a learnt method is not applicable, but still all possible interpretations of

[12] In general, it is a good heuristic to keep the size of the set of applicable methods small. This can be achieved by subsuming specialised methods by more general ones. For example, as soon as the system has learnt repeated application of simplify in group theory (rep-simplify = simplify*), we can remove the proof method simplify.

the learnt method outline have to be checked by the search engine. However, for more complex examples, this is no longer the case, and an improvement is noticed. This is because the search within the applicability test of the learnt method is more directed compared to the search performed by the proof planner. The improvement increases when a larger number of primitive methods is replaced by the learnt methods.

As expected, the *proof length* is reduced by using learnt methods.

On average, the *time* it took to prove *simple* theorems of group theory was approximately 100% longer than without the learnt methods. Notice that this does not include the case of *complex* theorems, when the proof planner timed out without finding the proof plans of the given theorems. The reason for bad timing in the case of *simple* theorems is that the learnt methods are small and simple, and the proof search contains the overhead due to the current implementation for the reuse of the learnt methods (see Section 3.3).

On the other hand, in the case of *complex* group theory examples, the advantage of having learnt methods in the search space is evident from the fact, that when our learnt methods are not available to the planner, then it cannot prove some complex theorems. When trying to apply methods such as associativity left or right, for which the planner has no control knowledge about their application, then it cannot find a proof plan within the given resources (i.e., four hours of CPU time). Our learnt methods, however, encapsulate typical patterns of reasoning about these theorems, hence they provide control over the way the methods are applied in the proof and lead to successful proof plans.

4.2 Residue class domain

In the domain of residue classes, we gave our learning mechanism examples from the residue class $\mathbb{Z}_3$ domain such that it learnt two new methods: tryanderror (as demonstrated in our examples in Section 3.2), and choose.

We applied the standard set of methods and the set enhanced with the two learnt methods to randomly chosen theorems regarding the residue class sets $\mathbb{Z}_3$, $\mathbb{Z}_6$ and $\mathbb{Z}_9$. We subdivided the results in Table 1 according to whether only one of the learnt methods or both of them were applicable in the proof. The number of method matchings is also represented in FIGURE 2 and the length of proofs in FIGURE 3. The labels in these figures denote the class of theorems, for example, "choose L" stands for theorems where the learnt method choose was applicable and proved by a strategy containing the learnt methods, while "choose S" stands for the same class of theorems but now proved with the standard strategy (i.e., without the learnt methods).

There is an improvement in each residue class set when learnt methods are available. Since choose replaces only small subproofs, whereas tryanderror can prove the whole theorem in one step, the latter has clearly better results for *proof length* and *matchings*. The benefit in the search for proofs where both learnt methods are applicable lies between them.

In addition to comparing the absolute values for our measures within the different sub-domains of residue class theorems (i.e., $\mathbb{Z}_3$, $\mathbb{Z}_6$ and $\mathbb{Z}_9$) in Table 1, we also compare the relative improvement between the different sub-domains. This can be done by examining the ratio between the number of *matchings* in the standard (S) and the enhanced (L) sets of methods (and the same for *proof length*), and then comparing

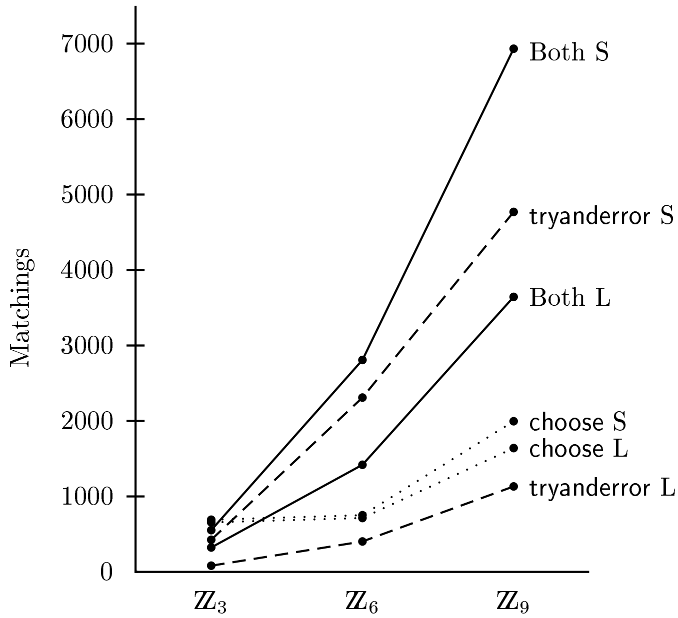

FIG. 2. Method matchings for residue classes.

the ratios for each type of theorems across sub-domains. Table 1 states these values. For example, the ratio for *proof length* in the case of theorems that use tryanderror method in $\mathbb{Z}_3$ is 19.30. This means that the proofs when only standard methods are available are 19.30 times longer than when learnt methods are available as well.

Table 1 clearly shows that the ratios for *proof length* increase across sub-domains (e.g., in case both learnt methods are used, the ratio increases from 2.09 to 2.55 and 2.69 across sub-domains). This indicates that the more complex the theorem (higher residue classes have longer and more complex proofs), the better the improvement when learnt methods are available to the planner.

In general, the same trend can be observed for the *matchings* ratios. An exception are the ratios for the type of theorems that can be proved using tryanderror method, which only marginally decrease across sub-domains (but we would expect them to increase as in the case for theorems that are proved using choose method). This can be explained by the fact that the theorems were randomly chosen across sub-domains, rather than using the theorems for the same properties but different residue classes. Namely, the random differences in the complexity of theorems in different sub-domains may be significant, e.g., the properties randomly chosen in $\mathbb{Z}_6$ may be more complex to prove than the ones chosen in $\mathbb{Z}_3$.

On average, the *time* it took to prove theorems of residue classes with the newly learnt methods was 50% shorter for proofs containing tryanderror than without such methods, 25% longer for both methods and 80% longer for choose. The time corresponds to the measured *matchings* but suffers from the overhead of the current implementation, especially for the smaller choose method. Since the learnt methods are tried before the standard set of methods, this effect increases for longer proofs.

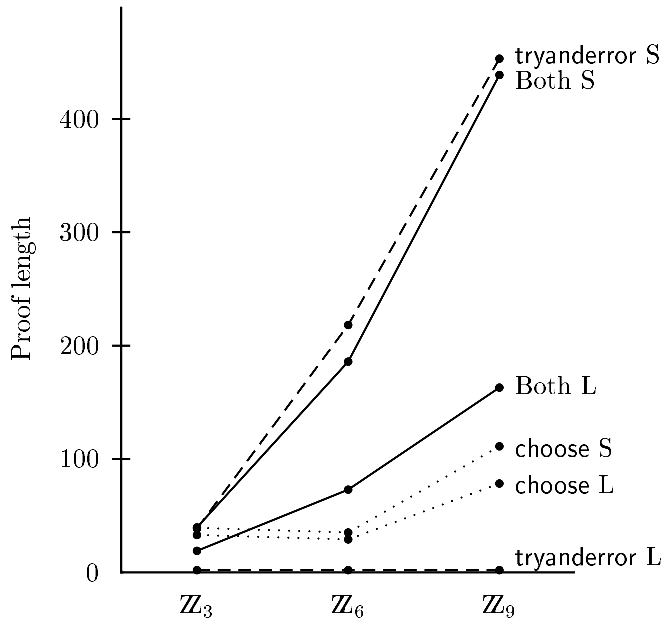

FIG. 3. Proof length for residue classes.

4.3 Set theory domain

The examples in this domain were selected to test the cost of learnt methods in the search process when they are not applicable. To this end, we added to the set of available methods the method learnt-set (see Section 3.2) that was learnt from theorems containing *three* variables. Note that since all the theorems in the development set have three variables, the universal quantification in learnt-set is eliminated (introduced in backward reasoning) exactly three times. Our development set is chosen deliberately in this restricted way in order to test the effect of a learnt method in situations where it is not applicable or applicable only in combination with other methods. In this way, we wanted to find out to which degree can learnt methods have a negative effect on the search space. Note that if we chose 'better examples' for learning, e.g., theorems that have one, three *and* five variables, then our learnt method learnt-set would be more powerful and applicable to all three types of theorems.

In order to test the restricted method learnt-set we added to our test set two types of theorems, namely, with one and with five variables. As expected, learnt-set is not applicable in the proofs of theorems with one variable. In the proofs of theorems with five variables learnt-set is applicable after two methods of the standard set are applied.

For theorems with three variables the proof search performs best, that is, the number of *matchings* is reduced by a factor of three when the learnt method is available. *proof length* is reduced by more than five times. The results for theorems with five variables are still better than without the learnt method, but as expected, not as good as with three variables. For theorems with one variable, where learnt-set is not applicable at all, the proof search clearly suffers from the additionally available learnt method, and hence the number of *matchings* is increased. Of course, *proof length* is not affected in this case.

The benefits and drawbacks of the availability of learnt methods can be seen very clearly in these evaluation results for the set theory examples. Namely, when a learnt method is applicable, then its availability improves the performance of the proof planner. However, when a learnt method is not applicable then the proof planner has to test a larger set of methods, and this will harm its performance.

On average, the *time* it took to prove or disprove conjectures in set theory with the newly learnt methods was about 40% faster for theorems with three variables, approximately 5% faster for theorems with five variables, and nearly 20% slower for theorems with one variable.

4.4 Analysis of results

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. Since 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 Section 3.3, is not as efficient as that of the ΩMEGA proof planner.

In our experiments, the *coverage* when using learnt methods is increased, which is also indicated by the fact that using learnt methods, ΩMEGA can prove theorems that it cannot prove otherwise. Since in our experiments proof plans were either found relatively quickly or not at all, we did not notice a possible effect where some proof plans that were found with the standard set of methods, now could no longer be found, because the learnt methods misled the proof search and increased planning time beyond the four hour limit.

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 better explanations why certain methods are best applied in particular combinations. For example, the simplification method for group theory examples indicates how the methods of associativity, inverse and identity should be combined together, rather than applied blindly in any possible combination.

A general performance problem of using learnt methods arises when a learnt method is not applicable. A learnt method is not applicable when there is no instantiation of the learnt sequence so that the methods of this instantiation are applicable. This means that every possible instantiation has to be tested and refuted. In the presented experiments, the learnt methods nearly always outperform the standard set of primitive methods. But there could be worst case scenarios where the learnt method is very general (contains many star operations) and a large part of the learnt sequence is applicable but the whole sequence is not. This has not happened in our experiments.

4.5 Analysis of general approach

The additional hierarchical structure of proofs constructed with learnt methods can also be beneficial for proof verbalisation and proof explanation tools like P.REX [8]. The information hidden within our learnt methods can now likewise be hidden in verbalisations, and expanded if appropriate or requested by the user. Namely, learnt methods encapsulate bigger and more abstract steps in proofs than smaller methods that make up our learnt methods. Hence, learnt methods provide a higher-level explanation of what is going on in the proof plan, and therefore they help to reflect the main idea of a proof by masking and grouping details in the proof. In combination, for instance, with the proof verbalisation tool P.REX it enables the proof planner ΩMEGA to automatically produce better explanations of the proofs which can be as high-level or as low-level as needed.

The preconditions of learnt methods are currently generated by the search engine for the reuse of methods described in Section 3.3. The engine searches for the instantiation of the method outline which is applicable in a given proof situation. This means that a small amount of search, which is guided by the method outline, needs to be carried out in the applicability test of the learnt method. Note that in the standard set of methods, i.e., not the learnt ones, the applicability test is carried out by checking if the *explicitly* and *declaratively* stated preconditions for the method hold or not in a given proof situation – similarly to the case with our learnt methods, this may also require search. The fact that the preconditions of the standard set of methods are declaratively stated, but the preconditions of our learnt methods need to be computed, does not change how proof methods are treated in the planning process. All methods, whether learnt or not, form part of the search space that the proof planner traverses in the process of finding a proof plan. Indeed, one of our motivations stated at the start of this paper was to devise a mechanism which is able to learn new primitives of the search space, rather than control the search within a fixed set of primitives. In the framework of proof planning the primitives of the search space are proof methods which we can now learn automatically. Even when some search needs to be carried out in order to compute the applicability condition of our learnt methods, this still is much better, that is, search is much pruned, than when such methods are not available to the planner. This is supported by the results of our evaluation demonstrated above in this section.

It is obvious that a new learnt method does not, in general, make some of the lower level methods obsolete while sustaining some notion of completeness. On the other hand, we cannot rule out this possibility completely for special cases. Determining such situations is challenging and requires proof theoretic methods based on derivability and admissibility criteria. This task can clearly not be addressed automatically in our approach.

We can see our approach as a mechanism that learns how to hierarchically structure the search through the search space. We built new methods that encapsulate guided search over some more primitive methods, and then add these new elements as a kind of *chunks of structured search* to our system. This contrasts with the idea of having only one global control layer in proof planning, since our learnt methods themselves can be seen as little planning processes consisting of a set of internal methods and control information on how to search with them.

The mechanism for reusing learnt methods described in Section 3.3 is specific to

ΩMEGA proof methods. On the other hand, the learning algorithm presented in Section 3.2 is general and can be used for learning in other automated reasoning systems, not just the ΩMEGA proof planner (see Section 6). The learning algorithm learns method outlines which with some enrichment could be used in other systems as inference rules, in ΩMEGA as proof methods, in λ*Clam* [27] as methodical expressions [26], etc. In fact, in some systems, like λ*Clam*, method outlines are exactly methodical expressions that the planner can use directly, so no enriching of the method outline representation is required. In other systems, enriched method outlines are just inference rules. This may give rise to the question of what is the difference between methods, methodical expressions and tactics. It seems that our method outlines offer a unified view of all these structures that are used in different automated reasoning system, e.g., inferencing systems, tactical theorems provers, and proof planners. Depending on the system, a different primitive of the search space is needed (e.g., inference rules, tactics, proof methods, methodical expressions). Hence, the enriching of the learnt method outline representation so that the new primitive can be used in the given system has to be carried out differently, or may indeed need no enriching at all. Studying how this process varies for different systems may give us some clues about the similarities and differences between such structures, but this is left for future work.

5 Related Work

Some work has been done in the past on applying machine learning techniques to theorem proving. Unfortunately, not much work has concentrated on high-level learning of structures of proofs and extending the reasoning primitives within an automated theorem prover.

For example, Schulz's work in [29], which is a continuation of previous work such as by Fuchs and Fuchs [9] and Denzinger and Schulz [6], investigates learning of heuristic control knowledge in the context of machine oriented theorem proving, more precisely, equational or superposition based theorem proving. Knowledge gained from the analysis of the inference process is used to learn important search decisions, which are represented as abstract clause patterns. These are employed in heuristic evaluation functions to better guide the search when attacking new proof problems. The selection of heuristic evaluation functions for a new problem at hand is guided by meta-data. The main difference with our work is that the learnt information in Schulz's work is not becoming a reasoning primitive, such as our learnt methods. It rather guides the search amongst the existing primitives at the global search layer instead of building up new, structured chunks of encapsulated search processes.

Silver [30] and Desimone [7] 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, Walther, Brauburger, Melis and Whittle have done related work on the use of analogy [21] and proof reuse [17, 16], that is, a sort of learning to solve new problems by using a similar existing problem. 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. Ireland [12] extends the applicability of the proof planning approach by patching failed proof plans by so-called proof critics.

A piece of related work in cognitive science is Furse's Mathematics Understander [10], MU, which stores mathematical domain and procedural knowledge in a contextual memory system, and tries to simulate how students learn mathematics from textbooks. MU builds up a uniform low-level data structure, while we build high-level hierarchical proof planning methods. Having explicit methods allows us to check proofs for their correctness, while in MU incorrect proof steps cannot be distinguished from correct ones. The hierarchical character of our methods also allows for a user-adaptive proof presentation.

In the field of machine learning there is a huge amount of relevant work and we mention only some that we deem most related to our work. In terms of a learning mechanism, more recent work on learning regular expressions, grammar inference and sequence learning by Sun and Giles [31] 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 [22] which automatically construct finite-state structures from trace information. His method IM1 is a general one and can describe all other existing grammatical inference methods. IM1 consists of first, generating a prefix tree from example traces, second, merging of states to get canonical acceptor states (which still describe only the example traces), and third, merging states which essentially does the generalisation of the structure. The generalisation, i.e., merging, is determined by a particular chosen heuristic measure. The existing state automata learning techniques differ depending on the heuristic that they employ for generalisation. 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, the heuristics described by Muggleton do not seem to be sufficient for generalisation in our case, as none of the states describing our proof traces would be merged. It is unclear what other heuristic could be employed to suffice the generalisation of our examples. Moreover, these techniques learn only sequences, i.e., regular expressions. However, our language is larger than regular grammars as it includes constant repetitions of expressions and expressions represented as trees.

There have been various approaches to incorporate learning in planning. In the PRODIGY system [32] a number of techniques for learning are available. The goal of the learning process is either to get control knowledge, that is, rules that describe which goal to tackle next and which method to prefer at the decision points of the planning algorithm, or learn planning operators from the change of planning states by observing an expert agent. The learning mechanism of LEARNΩMATIC differs in both aspects as its goal is to learn new operators that are learnt from other operators and could be compared to learning of macro operators of chunks [28]. Another difference is that learning from an analysis of the domain theory, in our case the set of methods, without the generation of examples appears to be difficult, since proof planning methods are complex and the postconditions are only available when a method is applied in a concrete proof situation. The abstraction from the proof to method names that is the input for the learning mechanism of LEARNΩMATIC is rather radical compared with abstractions in other planning systems, see [15]. There, a hierarchy of abstractions can be established by analysing the predicates of the domain theory. Some ideas for abstractions in method learning that retain possibly useful information are discussed in the next section.

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

6 Future Work

There are several aspects of our learning framework which need to be addressed in the future. With respect to the representation formalism, we have mainly considered sequential rewriting proofs. Other styles (different directions of reasoning) should also be considered.

Furthermore, we would like to apply our learning approach to other proof planners, such as λ*Clam* [27]. Since proof methods have a different structure in different proof planners, this task would require using the same learning mechanism, but probably, instead of our applicability test, a different reuse of methods approach than in the case of ΩMEGA.

The expressiveness of our language L for method outlines (see Section 3.1) could be studied further in order to determine if it should be extended. In particular, we could look into what type of ΩMEGA methods cannot be expressed using the current language L, and what other language constructs we would need. Moreover, we could examine if our language is sufficient to express primitives of the search space in other automated reasoning systems, like methodical expressions in λ*Clam* or inference rules in other theorem provers.

Regarding the learning algorithm itself, we need to examine what are good heuristics for our generalisation and how suboptimal solutions can be improved. While the learning mechanism is not efficient, we argue that we do not need a highly complicated and efficient technique for learning patterns, as in the GENOME project, for example. If we moved to larger example sets we could use a divide and conquer heuristic. Our learning algorithm without heuristics is sufficient for small patterns (e.g., less than 50 steps). We did not encounter larger patterns in our examples and do not expect very large ones for our application domain, since we assume well-chosen examples for the learning part. Our approach reflects the view that human mathematicians learn complex structures not in one single step but compose them step by step in a hierarchical way.

An interesting aspect that could be addressed in the future is whether a system could automatically learn the information that is abstracted from the proof traces and that has to be reconstructed by search performed in the applicability test when reusing learnt methods. What could this additional information that describes learnt methods more specifically be? When we take a look at the examples in group theory, it seems obvious that the *simplification* using associativity, inverse and identity methods are meant to act on the same subformula. This information is lost during abstraction, and hence, during the applicability test of the learnt method, associativity is applied at every possible place. So, the question is, could the smallest subterm of an expression to which the newly learnt method should be applied, i.e., the focus for the method, be learnt automatically and how? Future investigations could address such questions as well as identify additional pieces of information that describe learnt proof methods more specifically. In order to reduce search with the newly learnt methods it would also be good to learn meta-level control knowledge for them.

Another interesting, but difficult idea for future work is to characterise well-chosen examples more precisely, so that these could be selected automatically, rather than depend on the user. It would be desirable to identify automatically the subparts of proof traces in several examples of proofs that contain the same reasoning pattern. In our framework, this has to be done by the user of the system. Techniques from data mining or algorithmic learning theory could perhaps be useful to tackle this difficult problem, however, they usually require very large data sets, which in proof planning we typically do not have. An idea is to apply our approach to mechanised theorem provers (rather than proof planners), for which we have large proof corpuses (e.g., MIZAR, ISABELLE, OTTER), and then use data mining techniques in order to get good examples from them.

The extraction of method sequences from proofs is currently implemented with respect to the chronological order of method applications during proof planning. There could be other orderings, e.g., the different linearisations of the proof tree, some of them could even result in more adequate learnt method outlines. For example, in a situation where the planner has more than one different subgoal that can be closed by the same sequence of method applications $[m_1, m_2]$, it depends on the search behaviour of the proof planner whether the proofs will have a trace like $[m_1, \ldots, m_1, m_2, \ldots, m_2]$ or $[m_1, m_2, \ldots, m_1, m_2]$. The learning mechanism will produce $[m_1^*, m_2^*]$ in the first case and $[m_1, m_2]^*$ in the second case. The latter will have a better search behaviour in the applicability test of the learnt method because only one instantiation for the star operator has to be found.

Finally, an idea for more long-term future research is to model the powerful human learning capability in theorem proving more adequately. For this, it would be necessary to model how humans introduce new vocabulary for new (emerging) concepts (e.g., representing associative expressions as lists of terms in the expressions, annotations in rippling [5, 11]). With our approach, we cannot do that, however. It is a very challenging question left for future projects.

7 Conclusion

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 significant 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.

A demonstration of LEARNΩMATIC implementation can be found on the following web page: `http://www.cs.bham.ac.uk/~mmk/demos/LearnOmatic/`.

Acknowledgements

The main part of this work was done when the authors were working for the School of Computer Science of The University of Birmingham. Many people were generous with their time and advice and their help was invaluable to us. In particular, we would like to thank Alan Bundy, Predrag Janičić, Achim Jung, Stephen Muggleton, and Julian Richardson for their helpful advice on our work, and Andreas Meier, and Volker Sorge for their help with some of the implementation in ΩMEGA. This work was supported by an EPSRC Advanced Research Fellowship GR/R76783, EPSRC grants GR/M22031 and GR/M99644, SFB 378 Project grant, and European Commission IHP Calculemus Project grant HPRN-CT-2000-00102.

References

[1] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a mathematical assistant. In W. McCune, editor, *14th Conference on Automated Deduction*, number 1249 in Lecture Notes in Artificial Intelligence, pages 252–255. Springer Verlag, 1997.

[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 E. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, number 310 in Lecture Notes in Computer Science, pages 111–120. Springer Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[4] A. Bundy. A critique of proof planning. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski*, number 2408 in Lecture Notes in Computer Science, pages 160–177. Springer Verlag, 2002.

[5] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[6] J. Denzinger and S. Schulz. Learning domain knowledge to improve theorem proving. In M.A. McRobbie and J.K. Slaney, editors, *13th Conference on Automated Deduction*, number 1104 in Lecture Notes in Artificial Intelligence, pages 62–76. Springer Verlag, 1996.

[7] R.V. Desimone. Learning control knowledge within an explanation-based learning framework. In I. Bratko and N. Lavrač, editors, *Progress in Machine Learning – Proceedings of 2nd European Working Session on Learning, EWSL-87*, Wilmslow, UK, 1987. Sigma Press. Also available from Edinburgh as DAI Research Paper 321.

[8] A. Fiedler. *P.rex*: An interactive proof explainer. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Automated Reasoning – Proceedings of the First International Joint Conference, IJCAR-01*, number 2083 in Lecture Notes in Artificial Intelligence, pages 416–420. Springer Verlag, 2001.

[9] M. Fuchs and M. Fuchs. Feature-based learning of search-guiding heuristics for theorem proving. *AI Communications*, 11:175–189, 1998.

[10] E. Furse. Learning university mathematics. In C.S. Mellish, editor, *Proceedings of the 14th IJCAI*, volume 2, pages 2057–2058. International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1995.

[11] D. Hutter. Guiding inductive proofs. In M.E. Stickel, editor, *10th Conference on Automated Deduction*, number 449 in Lecture Notes in Artificial Intelligence, pages 147–161. Springer Verlag, 1990.

[12] A. Ireland. The use of planning critics in mechanizing inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning, LPAR-92*, number 624 in Lecture Notes in Artificial Intelligence, pages 178–189. Springer Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[13] M. Jamnik, M. Kerber, and M. Pollet. Automatic learning in proof planning. In F. van Harmelen, editor, *Proceedings of 15th ECAI*, pages 282–286. European Conference on Artificial Intelligence, IOS Press, 2002.

[14] M. Jamnik, M. Kerber, and M. Pollet. LEARNΩMATIC: System description. In A. Voronkov, editor, *18th Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 150–155. Springer Verlag, 2002.

[15] C.A. Knoblock. An analysis of ABSTRIPS. In James Hendler, editor, *Artificial Intelligence Planning Systems: Proceedings of the First International Conference (AIPS-92)*, pages 126–135. Morgan Kaufmann, 1992.

[16] T. Kolbe and J. Brauburger. PLAGIATOR — A Learning Prover. In W. McCune, editor, *14th Conference on Automated Deduction*, number 1249 in Lecture Notes in Artificial Intelligence, pages 256–259. Springer Verlag, 1997.

[17] T. Kolbe and C. Walther. Reusing Proofs. In A. Cohn, editor, *Proceedings of the 11th ECAI*, pages 80–84. Wiley, New York, 1994.

[18] A. Meier and V. Sorge. Exploring properties of residue classes. In M. Kerber and M. Kohlhase, editors, *Symbolic Calculation and Automated Reasoning: The Calculemus 2000 Symposium*, pages 175–190, Natick, MA, 2001. A K Peters.

[19] E. Melis and A. Meier. Proof planning with multiple strategies. In J. Loyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L.M. Pereira, Y. Sagivand, and P. Stuckey, editors, *First International Conference on Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 644–659. Springer Verlag, 2000.

[20] E. Melis and J.H. Siekmann. Knowledge-based proof planning. *Artificial Intelligence*, 115(1):65–105, 1999.

[21] E. Melis and J. Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22(2), 1998.

[22] S. Muggleton. *Inductive Acquisition of Expert Knowledge*. Addison-Wesley, Reading, MA, 1990.

[23] G. Plotkin. A note on inductive generalization. In D. Michie and B. Meltzer, editors, *Machine Intelligence 5*, pages 153–164. Edinburgh University Press, Edinburgh, UK, 1969.

[24] G. Plotkin. A further note on inductive generalization. In D. Michie and B. Meltzer, editors, *Machine Intelligence 6*, pages 101–126. Edinburgh University Press, Edinburgh, UK, 1971.

[25] G. Pólya. *How to solve it*. Princeton University Press, Princeton, NJ, 1945.

[26] J.D.C. Richardson and A. Smaill. Continuations of proof strategies. In R. Gore, A. Leitsch, and T. Nipkov, editors, *Short Papers of International Joint Conference on Automated Reasoning, IJCAR-01*, pages 130–139, 2001.

[27] J.D.C. Richardson, A. Smaill, and I. Green. System description: proof planning in higher-order logic with lambdaclam. In C. Kirchner and H. Kirchner, editors, *15th Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 129–133. Springer Verlag, 1998.

[28] P.S. Rosenbloom, J.E. Laird, and A. Newell. *The Soar Papers: Readings on Integrated Intelligence*. MIT Press, 1993.

[29] S. Schulz. *Learning Search Control Knowledge for Equational Deduction*. Number 230 in DISKI. Akademische Verlagsgesellschaft Aka GmbH Berlin, 2000.

[30] B. Silver. Precondition analysis: Learning control information. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning 2*, Palo Alto, CA, 1984. Tioga Press.

[31] R. Sun and L. Giles, editors. *Sequence Learning: Paradigms, Algorithms, and Applications*, number 1828 in Lecture Notes in Artificial Intelligence. Springer Verlag, 2000.

[32] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

Received November 5, 2003