FAKULTÄT FÜR INFORMATIK
der Technischen Universität München
Lehrstuhl VIII
Forschungsgruppe Automated Reasoning

Learning from Previous Proof Experience: A Survey

Report AR-99-4

Jörg Denzinger
Fachbereich Informatik
Universität Kaiserslautern
Germany

Matthias Fuchs
RSISE
Australian National University
Canberra, Australia

Christoph Goller
iXEC Executive Information
Systems GmbH
Hallbergmoos, Germany

Stephan Schulz
Institut für Informatik
TU München
Germany

TU München, Arcisstr. 21, 80290 München, Germany

Learning from Previous Proof Experience: A Survey

Jörg Denzinger
Fachbereich Informatik
Universität Kaiserslautern
Germany
`denzinge@informatik.uni-kl.de`

Matthias Fuchs
RSISE
Australian National University
Canberra, Australia
`fuchs@arp.anu.edu.au`

Christoph Goller
iXEC Executive Information Systems GmbH
Hallbergmoos
Germany
`c.goller@ixec.com`

Stephan Schulz
Institut für Informatik
Technische Universität München
Germany
`schulz@informatik.tu-muenchen.de`

Abstract

We present an overview of various learning techniques used in automated theorem provers. We characterize the main problems arising in this context and classify the solutions to these problems from published approaches. We analyze the suitability of several combinations of solutions for different approaches to theorem proving and place these combinations in a spectrum ranging from provers using very specialized learning approaches to optimally adapt to a small class of proof problems, to provers that learn more general kinds of knowledge, resulting in systems that are less efficient in special cases but show improved performance for a wide range of problems. Finally, we suggest combinations of solutions for various proof philosophies.

1 Introduction

Over the years, automated deduction systems have reached an impressive level of performance. There are three main reasons for this. Firstly, increasingly efficient inference engines with good search organization have been employed. Secondly, logical calculi became more elaborate and allowed a more efficient treatment of many proof problems. However, a third reason becomes visible if we look at the recent successes of theorem provers, as for example the proof of the Robbins algebra problem by EQP [Mc97] or the verification of hard- and software by various interactive provers (see [Re+97] for an example). These successes have been made possible only by guiding efforts of human users. The knowledge used in this explicit guidance by users results from the analysis of many experiments that provided the users with the necessary experience to tackle interesting tasks. So *learning* both by the developers and the users of theorem provers has played a major role in solving difficult problems.

When looking at the provers themselves, however, we do not find many mechanisms that make use of learning. In most cases, the best one can expect in a fully automated theorem prover is a selection of various different parameterized search-guiding heuristics. In this case the user is forced to learn which heuristic and which set of parameters is likely to perform well in a given proof situation. In interactive provers, programmable *tactics* allow the prover to learn (by being *taught*) about certain properties of the user, the domain, and the proof tasks. The user is offered a growing number of control pieces he can use in proof attempts. However, these tactics still have to be detected, implemented, and selected by the developers and users.

Although the potential of learning for automatic theorem proving systems was acknowledged very early in the history of deduction (see [CL73], pp. 154ff), progress in several areas of artificial intelligence and automated deduction was necessary to produce the first convincing solutions. Only in recent years, approaches have been developed for different deduction systems, different proof philosophies, and different tasks in theorem provers.

We believe that now is the time to compare and analyze these approaches in order to provide a classification for interested developers and users, and to characterize the general problems that have to be addressed when building a learning prover. Guided by our experience stemming from several funded projects in the area of learning theorem provers we have observed the following classes of problems evolving in this context:

First, there are the problems concerning the learning process itself (*learning phase*):

- Whom and what to learn from?
- What to learn?
- How to represent and store the learned knowledge?
- What learning method to use?

The second class of problems is concerned with the usage of learned knowledge (*application phase*):

- How to detect applicable knowledge?
- How to apply knowledge?
- How to detect and deal with misleading knowledge?
- How to combine knowledge from different sources?

And finally, there is a central problem in machine learning. Its answer influences all solutions to the above problems:

- Which concepts of similarity are helpful?

In this article, we will use these questions as a general framework for learning provers. For each question/problem we will characterize general classes of answers/solutions and for each class we will present the approaches from literature that fit into it. We will also point out additional possible answers that have not been considered so far.

We will see that it is not easy to borrow solutions from other areas of AI, since very often some preconditions required by these solutions are not valid in automated deduction. One example is the basic premise of case-based reasoning "*small differences in problems result in small differences in their solutions*", which, at least for intuitive difference measures, is not fulfilled in most logics and calculi. Therefore substantial modifications have to be made to use such an approach in automated deduction. We will also show that AI techniques other than learning, for example planning or multi-agent systems, often are useful for building a learning theorem prover or for adding learning components to existing provers.

Naturally, all approaches have strengths and weaknesses, and the solutions to our problem classes are interrelated. Furthermore, the proof problems that one is interested in lead to a preference for certain solutions.

The emphasis of this paper is to highlight the general problems that have to be solved, and to classify the possible solution classes. However, in order to build a complete learning prover the individual solutions to our nine problems have to be combined. Therefore, we will analyze possible combinations of solutions to our problem classes and present the expectations for provers employing these combinations with respect to the grade of specialization and efficiency. As can be expected, there is a trade-off between specialization and efficiency. This results in a spectrum of possibilities that different combinations of techniques can be placed in. We will also analyze the different requirements imposed by various proof paradigms, in particular the effect of saturating bottom-up versus analytical top-down procedures, and the influence of fully automated theorem provers versus interactive ones.

This article is organized as follows: After this introduction, we give a short overview of automated deduction, theorem provers, and their characteristics influencing the task of learning. In Section 3, we concentrate on the general questions already mentioned, present possible answers, and analyze the known approaches to learning in theorem proving with respect to these questions. In Section 4, we highlight some successes of learning provers and analyze the known approaches with respect to the specialization-efficiency spectrum and the basic prover characteristics. Finally, in Section 5 we give some remarks on our expectations for the future development of learning theorem provers.

2 Automated Deduction

The task of automated deduction is the (partial) mechanization of the following problem:

> Assume a set of terms over a fixed signature, a set of predicate symbols forming atomic formulae (also called literals) by using these terms as arguments, and a set of quantifiers and logical connectives that allow us to combine atomic formula into larger aggregates (well-formed formulae). Assume further a set of *interpretations* (i.e. functions) mapping terms to semantic objects, predicate symbols to predicates over these objects, and formulae to truth values by a common interpretation of the connectives and quantifiers. Then our problem is to determine whether a given formula is interpreted by all the interpretations to a particular truth value.

We call the set of well-formed formulae together with the set of interpretations and the truth value a *logic*. The stated problem is the *validity problem* for a given formula. Instances of the validity problem will also be called *proof problems* in the following. The formulae defining a proof problem are often split up into *goal* formulae which define the *conjecture* that has to be proven and *axioms* which define the relevant theory or domain, usually a set of algebraic structures.

There are many different logics. Examples are propositional logic, first-order logic or the logic resulting by allowing only the initial model of a specification as interpretation. Different logics can be used to specify and solve the same problem. If all interpretations of a logic interpret a symbol as the same semantic object, we call this symbol an *interpreted symbol* of this logic. Often it is possible to define the laws that describe such a semantic object in another logic that also incorporates the other syntactic structures of the first logic (we then say that the first logic is more specialized). So, semantic information can be expressed in a syntactic way. The best-known example for an interpreted symbol is the equality predicate in first-order logic with equality. By adding the laws of reflexivity, symmetry, transitivity and substitutivity for this symbol, validity problems for a formula of first-order logic with equality can be solved by using pure first-order logic (although the cost, in terms of additional search space to traverse, is often prohibitive for practical application).

The mechanization of the problem solving process for the validity problem is based on logical *calculi*, which are mainly sets of (inference) rules that describe how to transform sets of formulae or other structures based on formulae (as for example tableaux or sequents) into new sets or structures. In addition, one needs a transformation of the given formula into a start state and a condition that determines final states. Solving a given proof problem then means finding a sequence of applications of inference rules that transforms its start state into a final state. This is a typical description of a classical AI *search problem.*

There are two general types of calculi, *generating* calculi and *analytical* calculi, with some newer calculi combining features from both approaches. The two general classes require usually quite different learning techniques. We will therefore discuss them in some detail.

Generating theorem proving calculi use sets of formulae to represent search states. Typically, there are two types of inference rules: generating rules and contracting rules [De90]. Generating inferences add new formulae (in the rest of this paper simply called *facts*) to the set of formulae that represents the actual state. Contraction inferences either delete facts or they substitute facts with other (hopefully simpler) ones. The search process in generating calculi is often called *forward chaining*, because the search proceeds from a set of axioms until a certain desired formula is generated. Examples for generating calculi for first-order logic are resolution (see [Ro65]) and hyper-resolution used e.g. by the theorem prover Otter ([Mc94]). Further examples are superposition ([BG94]) for first-order logic with equality or proof by consistency ([Ba88]) for inductive equational logic. The theorem prover Discount (see [De+97b]), in which many learning approaches have been tried out, is based on unfailing completion (see [Ba+89]), a generating calculus for pure equational logic.

In analytical calculi the basic principle is to divide the goal formula (conjecture) by means of the inference rules into other (hopefully easier to prove) formulae. This process is repeated until each subgoal is obviously solved by an axiom from the problem specification. The search state is represented by trees or graphs. Due to the fact that the inference process starts with the goal, this kind of search is often called *backward chaining.* Examples for analytical calculi for first-order logic are the analytical tableau method (see [Sm68]) or model elimination (see [Lo69]), which is used by the theorem prover Setheo ([Le+92]). Further examples of analytical calculi are SLD resolution (as used in Prolog, see e.g. [Ll69]) for Horn logic, or the rippling method for equational logic (see [Hu96]).

Many calculi for proving in an initial model have both generating and analytical features. Performing induction by dividing the problem into the base case and the induction step is clearly an analytical feature, while the generation of so-called lemmata (by *guessing* what may be needed) is a generating rule.

For most interesting logics and associated calculi, there are many different inferences applicable in a given search state. Therefore a theorem prover needs a *control strategy* that defines which of the possible inferences will be performed to determine the successor state. Typically, a single control strategy is only good for a specific (often small) set of proof problems. Applied to problems outside this class sequences leading to a final state cannot be found in acceptable time in most cases.

Naturally, the different types of calculi lead to different heuristics employed in search strategies. In generating theorem provers, nearly all search strategies try to employ the contracting inferences as often as possible. Under this assumption, the control of the generating inferences becomes crucial for the efficiency and eventual success of the inference process. Control heuristics for these inferences typically base their decision on (anticipated) syntactic aspects of the formulae that are the source (or result) of the rule applications. Size, parentage, or similarity to certain formulae are examples for these aspects. Typically, once a formula is added to the search state, it will only be removed by a contracting inference (although, of late, "forgetting" of formulae has proven to be very interesting, see [DF96] and [FF97a]).

The situation for analytical theorem provers is different. Most calculi used by analytical theorem provers are not (proof) *confluent*. Furthermore, since search states are represented by quite complex objects, analytical provers normally perform a *depth-first* search. Since it is in the general case impossible to decide which of the possible problem decompositions will lead to a proof, an analytical theorem prover (based on a non-confluent calculus) has to enumerate all possible decompositions. Therefore, *backtracking* to a prior state plays an important role. At each state an analytical theorem prover has to decide whether it should try to prove one of the open subgoals via further decomposition, or whether it should backtrack to a prior state and try some other decomposition of this state. Typical control heuristics in analytical theorem provers base this decision on measures such as the depth of the proof tree, the number and complexity of subgoals introduced, or the number of free variables occurring in the subgoals. *Bounds* on such measures are used to specify a finite initial part of the search space which is explored in a depth-first manner. If no proof is found, these bounds are incremented and the search starts on a bigger finite initial part of the search space. In this way a *breadth-first* search is simulated by an *iterative deepening* depth-first search. See e.g. [Le+92] for details.

Typically, even good heuristics for both types of provers will result in a high number of unnecessary applications of inference rules, i.e. inferences not necessary to solve a given validity problem. For problems of reasonable difficulty typically between one step per 1000 and one step per 100 000 (depending on the calculus) really contributes to the proof.

In so-called *interactive provers*, the help of the user is sought to minimize this discrepancy. Naturally, users are interested in minimizing the necessary amount of interaction. However, in order to give useful hints, the user has to understand the proof the prover tries to construct. This necessitates a sufficiently high degree of interaction. To limit the amount of interaction, the developers try to define certain situations in which the user is asked for help. Typically, a large part of the inferences is performed without interaction, and only the application of a few crucial inference rules is determined by the user. These crucial inferences often involve guessing steps, as for example the guessing of a useful lemma or the guessing of an appropriate case analysis.

As this discussion has made obvious, each theorem prover constantly faces the problem of making decisions. This starts with some aspects of the representation of the problem to solve, continues with concentrating on the important parts of that problem, and ends with the selection of the best step out of the set of possible inferences. And decision making is a task that can profit very much from experience and learning.

3 Learning and Automated Deduction

If one wants to use learning in automated theorem proving, a reasonable idea is to look at human learning in theorem proving. We can observe that aspects like *learning from examples*, *analogies between structures and situations*, and *a mix of many methods and levels of abstraction* play an important role for human mathematicians. If we want to examine these aspects of learning with regard to their appropriateness for automated theorem provers, a good starting point is to characterize them according to the level on which learning takes place. We can identify three distinct levels: The levels of *logic and specification*, *calculus*, and *control*. Note that this characterization is not completely sharp and there are some borderline cases.

Many of the aspects of human learning are on the level of logics and specification. Very often, humans transfer results and proof problems from one logic or specification to another. This process involves a lot of experience and learning. For example, a mathematician may first try to find a proof to a problem for a special interpretation of a logic only (and thus, according to our definition, in a separate logic). The proof in this restricted logic may then be generalized for all interpretations of the original logic, i.e. the mathematician learns from the restricted case for the more general one. Human learning also often involves the use of (counter) models to decide whether certain directions (conjectures) in a proof search should be rejected. Current theorem provers lack the ability to make use of such high-level semantic aspects of learning, since they are usually tied to a single logic and specification, and thus are not equipped to use meta-reasoning about different logics and their suitability for a given problem, or to translate knowledge and experiences from one logic to another. Therefore, learning for automated theorem provers so far has been restricted to the levels of calculus and control.

On the second level, the level of calculus, learning leads to new inference rules, lemmata, or additional axioms. Humans often rely on a large set of interpreted symbols with adequate decision procedures (or approximations of such procedures) that have been acquired during many experiences with these special operators. Such decision procedures can also be regarded as extensions of given calculi. However, more inference rules and more axioms, while useful in many situations, also result in a broader search space. This broader search space may quickly become unmanageable if not accompanied by additional control knowledge. Therefore, for automated theorem provers which learn on the calculus level, additional control knowledge or even learning on the control level is indispensable.

Human beings also develop good control strategies as they gain experience with a given calculus or in a given domain. New conditions about the use of certain rules or methods, as well as priorities among various possibilities, perhaps with semantic context conditions, are learned during the many exercises that a student of mathematics has to perform. Since coping with semantic information is no big problem for humans, they can easily control different solution approaches to problems and they may even use and control different calculi and logics for different parts of a proof. Automated theorem provers have to control inference rules of calculi that are not typical for human problem solving behavior. Therefore, human control techniques as well as learning techniques cannot be transferred directly to automated theorem proving systems. Nevertheless, most learning theorem provers use previous experience to learn on the control level.

As the previous paragraphs have shown, the three levels logic, calculus, and control are not very good candidates for a classification of learning theorem provers. In fact, it is impossible to find a classification such that each known learning prover fits in exactly one class while still having classes that contain more than one or two provers. The different calculi and proof paradigms already would result in so many classes that the particular aspects of learning would become veiled and the whole classification would be useless for practical purposes.

While it is very difficult to compare whole learning provers employing different logics and/or different calculi, it is nevertheless possible to define basic questions/problems that each learning prover has to solve. And for each of these problems certain classes of solutions can be identified that in most cases are independent of logic and calculus of a prover (although for some of the problems there is always the "interactive" solution, i.e. delegating it to the user, see, for example, Sections 3.5 and 3.7).

We have identified nine fundamental problems that arise in the context of learning for theorem proving. As already mentioned in Section 1, they can be assigned to one of two phases, namely the learning and the application phase. One problem, selecting suitable concepts of similarity, influences the solutions to all other problems and therefore is not confined to any one of the phases. In each of the following sections we will deal with one of the fundamental problems by first motivating what the problem is (if necessary) and then presenting the general classes of solutions to the problem. For each class we will characterize the solution idea to the problem (if not obvious) and then present the known instances of the solution idea from literature.

3.1 Whom and What to Learn From?

The first problem, when dealing with learning, is to determine whom and what to learn from. As this phrasing already suggests, there are different possibilities both for where to get data from and for what this data exactly is. These different possibilities can be exploited both by human beings and computer systems. Let us first look at who can provide data. In principle, a learning system can learn by itself or with the aid of a teacher. For a theorem proving system this offers three different possibilities:

- Learning from its own experiences.
- Learning from data provided by a human being.

- Learning from experiences of other systems.

Due to the, for a human being, unusual calculi used by theorem provers, most systems learn from their own experiences (e.g. [SE90], [KW94], [Go94], [Fu95], [DS96b], [De+97a], [Fu97c], and [Go97]). In interactive provers, those experiences naturally involve user interactions so that in these cases we have a mixture of the first and second possibility [Me95]. However, the emphasis is clearly on learning from own experiences (with users providing input leading to own experiences).

Learning from data provided by the user can, in a rudimentary form, be observed in OTTER ([Mc94]) in form of the so-called hot-list. In [Wo96] it is reported, that OTTER uses text book proofs by adding the formulae occurring in these proofs to a hot-list. Whenever OTTER encounters a formula from this list during its search, it puts more emphasis on this formula and its descendants. While one cannot call this exactly learning, it can be thought of as *tutoring* by the user, and it may be a good starting point for developing a learning prover (see also [Ve96]).

Learning from experiences of other systems is also not much exploited. In order to employ this type of learning, a system has to understand the output of the system that it learns from. Unfortunately, for most systems the documentation of their output is not very detailed and therefore it is difficult to interpret this output for other systems. The provers DISCOUNT (see [De+97b]) and WALDMEISTER (see [Hi+97]) produce their proofs in the PCL format (see [DS96a]), and therefore DISCOUNT can learn from WALDMEISTER proofs. In the ILF system (see [Da+94]) proofs of several systems can be transformed into proofs in the *block calculus* (see [DW96]). This may give access to (parts of) the experience of all these provers in the future.

A second dimension of the central question of this section, namely whom and what to learn from, is how many different experiences or lessons from teachers are actually used for learning how to solve a particular new proof problem. Approaches which are in the broadest sense related to *theorem proving by analogy* (see e.g. [KW94], [Me95], [Fu95], and [Fu96]) select experiences from a single solved problem (called *source problem*) to help proving a new problem. We will deal with some implications of this in Sections 3.3, 3.5, and 3.7. Other approaches (see e.g. [DS96b], [Go94], and [Go97]) are capable of abstracting and generalizing experiences stemming from several solved problems (often called *training problems* in this context) and of using this abstracted knowledge efficiently for further decisions.

Now, let us take a closer look on *what experiences* can be used to learn from. Obviously, the steps taken by a prover in order to solve a problem may form a useful experience. However, information about steps not taken, as well as information about steps taken, but not contributing to a proof, can be valuable as well. Therefore, an important classification of experiences used for learning is the split into

- positive experiences and
- negative experiences,

where *positive* means experiences that result from steps useful in a proof attempt, while *negative* means information about steps that were not useful. In this sense, counter models successfully used to curtail parts of the search space (as used in [DP97]) have to be classified as positive experiences, since they are useful in showing that a formula is not valid, while useless attempts at finding such a model would be classified as negative experiences. In the terminology of learning, positive or negative experiences are often also called *positive* or *negative (training) examples*.

Methods related to theorem proving by analogy (see above) traditionally use only positive experience (see [Fu95] and [Fu96] for exceptions). [DS96b] presents two approaches for abstracting and generalizing from several training examples, but uses only positive examples. It should be noted that negative examples are often essential for avoiding over-generalization[1]. Approaches where positive and negative examples are used for learning are presented in [Fu96], [SE90], [Go94], and [Go97]. While most approaches only use examples stemming from one proof per training problem, the approaches described in [Go94] and [Go97] use examples stemming from several different proofs per problem.

The kind of experiences used for learning determines the way in which experiences are extracted from successful proof attempts in order to provide *training data* (positive and/or negative training examples) for learning.

A listing of all inference steps performed by a prover during a successful proof attempt (together with their contexts) allows the retrieval of all types of positive experiences, and also contains some negative experiences. However, not all possible negative experiences can be extracted from such a listing. The set of steps that are useless for proving a particular proof problem is, in most cases, infinite, while a single proof attempt can only cover a finite number of steps. Nevertheless, the number of useless (negative) steps performed in a successful proof attempt is, in most cases, extremely large. Thus it is not practical to directly use a complete listing of all inference steps for learning. Therefore, one normally restricts the set of negative steps to those that are *close* to positive ones in the sense that only negative steps requiring at most a given number of other negative steps as preconditions are considered.

Such (restricted) listings of inference steps can be regarded as *finite initial parts of the search space* of the automated deduction system for the respective training or source problem. [Fu95] and [Fu96] document their use in learning for DISCOUNT.

In [Go94] and [Go97], in continuation of [SE90], such listings are used to train *neural networks* which implement a control heuristic for the theorem prover SETHEO. However, the listings used in [Go94], and [Go97] can contain positive examples from several different proofs per problem. This means that representations of several different proof attempts can be combined. It has to be noted that the user has to decide which proofs are acceptable or desirable[2] (positive examples) and which parts of the search space should be avoided (negative examples). This decision influences considerably what can

[1]Over-generalization in this case means that too many new situations are mistakenly taken for positive ones in the application phase

[2]The decision can e.g. be based on the size of the proofs.

and what cannot be learned.

If a complete listing of all steps performed during a proof attempt is stripped of all useless steps, only the proof remains. Proofs are the basic form of experience for many of the learning provers, although, due to the different calculi used, they may be represented very differently. Complete proofs are used in most of the learning DISCOUNT variants (see [De+97a], [DS96b], and [Fu97c]), in [Me95], and in ABALONE (see [MW97]). In [DS96b], the described system abstracts from everything but the facts occurring in any proof known to the system, but nevertheless proofs form the input to the learning algorithm.

Instead of using all steps that contributed to a proof for learning, the goal that has been proven together with the axioms (and perhaps a few additional lemmata) that have been needed in the proof are used in [KW94], [KW95], and [KW96].

As we have seen, quite a large number of solutions exist even for the initial problem of selecting possible input sources for learning. We will see in the following sections that the particular solution chosen for this problem has some influence on the other fundamental problems.

3.2 What to Learn?

In this section, we discuss the question of "What to learn?", that is the question of the goal of learning. We distinguish three major degrees of generality of the knowledge captured by learning:

- Specific knowledge about individual problems.
- Knowledge about domains or specifications.
- General knowledge about calculus and proof procedure.

Human experts often use analogies for solving new problems. In this case, very specific knowledge about the solution for one problem is transferred to a new problem. Domain knowledge is also often used consciously by human beings. An example is a rule like "$x^2 = (-x)^2$ *is a central lemma in many proofs*". Domain knowledge is specific to certain domains. However, as many domains of interest to humans share common traits, parts of this knowledge apply to many different domains.

The third kind of knowledge, knowledge about properties of the proof process, is often used only subconsciously by humans. Examples for knowledge of this kind is e.g. "Prefer facts with small terms" or "Overlap only maximal terms". For human mathematicians, analog knowledge would be something like "*always factor polynomials*" or "*always compute normal forms for all fractions*". This knowledge is usually not specific to a given domain, but to a given calculus. It may, however, be specific to a given problem class (that may or may not correspond to a domain of interest).

As we have already pointed out on page 7, learning in automated theorem provers takes place either on the calculus level or on the control level. This results in the following possibilities.

- Learning on the calculus level:
  - lemmata
  - proof schemata or meta inference rules
- Learning on the control level:
  - proof plans
  - control strategies or control heuristics

On both levels knowledge of all three degrees of generality can be learned.

Since theorem provers based on generating calculi produce new knowledge, such as clauses and equations, during a normal proof attempt, it can be argued that they already perform some kind of learning on the calculus level (they learn about new, valid facts). This also holds for *lemma mechanisms* in analytical calculi. However, in both cases the generated knowledge is used only during the proof attempt in which it is generated. It is not used for solving other problems.

In the late 80's, EBL/EBG (*explanation based learning / explanation based generalization*) (see [Mi+86] and [DM86]) was very popular for logic-based symbolic reasoning systems. Stated in terms of theorem proving, EBL/EBG is a sophisticated lemma mechanism. For theorem proving, the related concept of *reusing proofs* [KW94] has been introduced. A goal (conjecture) that has been proven together with the axioms (and perhaps a few additional lemmata) that have been needed in the proof are generalized (using second-order generalization, see Section 3.4.1) and stored in a so called *proof shell.* This can be seen as learning of a *proof schema* or *meta inference rule.*

The benefit of learning on the calculus level, i.e. the advantage of storing lemmata or proof schemata, is that their application yields a result (e.g. a proof for a subproblem) while avoiding the search needed for their deduction. An uncontrolled application of such mechanisms, however, leads to numerous new (and very specialized) lemmata or rules, and a tremendous increase of *redundancies*. It contradicts the fundamental intention of deduction, which is to store general knowledge and to derive answers for special cases, because it more or less leads to storing many *special cases* and *reformulations* of the domain theory (see also the discussion on memorization and symbolic representations in Sections 3.3 and 3.4). Indeed, in many cases performance can degrade. Examples and a theoretical analysis for the case of EBL/EBG can be found in [Mi90]. Mechanisms for identifying important lemmata, such as those recently developed in [Fu97d] or [Dr98a, Dr98b], could provide some help. However, in order to actually improve efficiency, learning on the calculus level has to be combined with learning on the control level.

Using analogy in theorem proving (see [Br+88] and [Me95]) is an example for learning on the control level. The goal in analogy is to extract a kind of generalized *proof plan* from the proof used for learning (source problem). Analogy does not reuse the final result of a proof for proving a new problem like the approach of [KW94] does. Instead, it tries to construct a proof for a new problem, guided by the decisions taken to construct the proof for the source problem (see Sections 3.3.3 and 3.6). Thus, analogy is a more flexible way of reusing proofs. If the exact correspondence between source and target proof breaks down, analogy-based proof processes can resort to various patching strategies like skipping or repeating individual steps, changing their order, or carefully introducing new steps. However, analogy suffers from similar problems as lemma mechanisms: Uncontrolled generation of proof plans can lead to partially overlapping plans and thus introduces redundancies into the proof search.

For theorem provers based on generating calculi, learning on the control level often takes the form of learning some kind of *heuristic evaluation function* for the new facts that are generated during the proof process. Such evaluation functions compute *numerical ratings* which are used to determine the order in which facts are used for further applications of inference rules. Examples are the learning DISCOUNT variants, which are described in [Fu95], [Fu96], and [DS96b]. In these cases, the *context*, (i.e. axioms and the goal that has to be proven), is not taken into account by the evaluation functions, since the facts are evaluated individually. Instead, the context is used for the detection of applicable knowledge, that is for the selection of one of multiple heuristic evaluation functions or the selection of training problems, as described in Sections 3.3 and 3.5.

For systems based on analytical calculi, the situation is rather different. In case of model-elimination or SLD-resolution (PROLOG), for example, the number of available input clauses (and hence the maximum number of possible inference steps in each state) is fixed[3]. Thus, an evaluation of individual input clauses is insufficient. At least part of the context in which the input clauses are to be applied, e.g. some features of the current *tableau*, has to be taken into account by the heuristic evaluation functions. Such a tableau, which represents the current proof attempt, is a possibly quite complex tree of literals. Its root usually is the goal that has to be proven. The heuristic evaluation functions learned by the learning SETHEO variants (see [SE90] and [Go94]) evaluate input clauses together with a very coarse description of the current SETHEO tableau. In [Go97] the whole tableau itself is evaluated. This results in learning a heuristic evaluation function for whole *proof attempts* or *proof structures*.

Learning on the control level, either of domain-specific knowledge or even of general knowledge about the calculus, can also be regarded as learning of some kind of *search strategy* or *calculus refinement*. Examples for very successful refinements, which were found by humans, are the various refinements of resolution such as hyper-resolution, linear resolution, and input resolution, or the restriction of paramodulation to maximal terms. The nature of such strategies and refinements is that they eliminate redundancies while still guaranteeing *completeness* at least for special problem classes. This

[3]Reduction steps in case of model-elimination are a rare exception to this.

means that some solutions (proofs) of problems may be lost, but at least one proof for every provable problem remains possible. The primary goal of learning in [Go97] are such class- or domain-specific strategies and calculus refinements. This is achieved by providing experiences from several proofs for each training problem. The learning algorithm has the possibility (freedom) to concentrate on some solutions while sacrificing others if this helps to distinguish a minimum of "good" situations (leading to at least one proof) from "bad" ones, that is if it helps to keep the heuristic evaluation function simple.

So, the question of what to learn is in almost all cases connected with the control of the theorem prover. However, the aspects of the control that are influenced by the learned knowledge are very different. This variety is also demonstrated by the many possibilities known for representing and storing learned knowledge.

3.3 How to Represent and Store the Learned Knowledge?

In this section, we examine how learning theorem provers represent and store the learned knowledge. We will analyze the possible knowledge representations according to the following criteria:

- Representational power (abstraction, generalization)
- Storage space requirements
- Understandability
- Cost of retrieval
- Availability of learning methods
- Cost of learning and maintenance/updating

We will now discuss these criteria briefly. Obviously, one of the most important characteristics of a knowledge representation scheme is its representational power, i.e. what can and what cannot be represented. Consider, as an example, the problem of representing terms. The set of terms over a finite signature is, of course, infinite except for trivial cases. Consequently, a fixed number of finite attributes cannot uniquely represent all these terms, and any such representation has to abstract from certain properties of the terms. However, the ability to uniquely represent all possible inputs is not necessarily important for a given learning task. Learning involves abstraction and generalization. The more important question is, therefore, which *concepts* or *properties* of the input can be represented. Unfortunately, up to now there is only very limited knowledge about which properties are important for representing knowledge to guide theorem provers. Therefore, a high representational power is desirable, as it is more likely to allow the description of relevant concepts.

High representational power may also allow a more compact knowledge representation, as it may allow the learning system to abstract from individual examples in favor of a compact *hypothesis*. However, more powerful representation schemes also allow for a much wider set of hypotheses, and thus lead to a very serious increase in the difficulty of finding good ones.

Besides representational power, understandability is a second important property, since most users and developers want to be able to understand what has been learned and to profit from this knowledge. Symbolic and explicit representations (e.g. enumerations of finite sets of facts) are easier to analyze as e.g. numerical values as found in a neural net.

The next important topic in discussing knowledge representation is the cost of retrieving relevant pieces of knowledge. Questions concerning retrieval are also discussed in Sections 3.5, 3.6, and 3.7. However, the representation of the learned knowledge can influence the cost of retrieval considerably. For an unsuitable representation or inefficient implementation, retrieval cost can become the dominating factor in the proof process, and can thus reverse any progress achieved in the proof search. In addition to the basic representation scheme, proper organization of the learned knowledge in a data base (or *knowledge base*), can help to reduce retrieval cost. We distinguish three different degrees of such *additional organization of learned knowledge.*

- Flat storage, i.e. unstructured knowledge
- Organization by proof
- Organization by hierarchical classes

Additional organization of the knowledge base is particularly important for learning approaches which refrain from strong generalization and abstraction, and hence store individual training examples or knowledge about individual training problems separately. For such learning approaches the knowledge base grows continuously with the number of examples or problems, and retrieval can become very expensive.

Learning approaches which are capable of generalizing and abstracting from several examples do a lot of organization during learning themselves. This holds especially if generalization and abstraction from examples stemming from several problems takes place and domain-specific knowledge or even general knowledge about the calculus is learned. In such cases additional organization of the learned knowledge is often not necessary. See e.g. [SE90], [Fu95], [Go94], and [Go97]. However, in case of domain-specific knowledge additional organization according to different problem domains is useful in order to have a fully automated system.

The second kind of organization, storing knowledge by proof, is mainly implemented by learning theorem provers based on analogy ([Me95] and [Fu97c]) or reuse [KW96].

The last basic approach, organization by hierarchical classes, actually covers a large spectrum of possible realizations. Knowledge can be organized in classes distinguished by certain aspects. These aspects can include quite complex ones, like the axiomatization, the signature, or the conjecture of a given proof problem (see [DS96b] and [De+97b]), or they can be based on simpler properties, like the number of function symbols or axioms or even the growth characteristics of the search space (see [Fu96], [Fu97a], and [FF97a]). Hierarchies of classes can be built by moving from more general to more specific cases, particularly for complex aspects like axiomatization or signature, or by considering different properties in a given order. Recent versions of the WALDMEISTER system [Hi+97, Hi+99] use such (hand-coded) representations to select good search strategies for a given domain.

As we stated above, more powerful knowledge representations usually make learning more difficult, as they allow for a wider range of hypotheses. Many learning algorithms generate only hypotheses from a very limited set. Therefore it is desirable to match the power of the learning algorithm with that of the selected knowledge representation scheme. For many powerful knowledge representation schemes, appropriate learning methods are currently not available. On the other hand, for a lot of approaches for learning theorem provers, it was the availability of a particular learning method that determined the choice of a knowledge representation. We will discuss the various learning methods that have been used so far in Section 3.4.

Finally, the cost of learning and the cost for maintaining and updating the learned knowledge have to be considered. Obviously, these costs depend on the learning method that is used. However, the knowledge representation already determines the cost of learning to some degree. Knowledge representations which allow a lot of abstraction and generalization naturally require a lot of resources for learning, since suitable generalizations and abstractions have to be found. Moreover, as the resulting compact hypotheses are usually based on many training examples, revising them becomes hard. Normally, a complete restructuring of an already existing knowledge base is required in order to incorporate new training examples, at least if these new examples contradict what has been learned before. On the other hand, for knowledge representations which perform little generalization and abstraction, learning usually is relatively cheap. Especially if variants of *memorization* (see Section 3.4.1) are applied and every (training) example is stored separately, the incorporation of new examples is very cheap.

One should keep in mind that minimizing the cost of retrieving learned knowledge is essential for the success of a learning theorem prover, as this step is performed for each new proof problem and possibly even for each decision during the search process. The cost for learning is less important, as knowledge learned once can be applied for a large number of new problems. However, the cost of learning still plays a major role in testing and benchmarking learning systems.

We now analyze the knowledge representations that so far have been used for learning theorem provers with respect to the criteria discussed above. We distinguish three different classes of knowledge representations: *symbolic* representations, *hybrid symbolic/numeric* representations, and *numerical* representations.

3.3.1 Symbolic Representations

Symbolic representations are attractive because they can easily handle the recursive, arbitrary sized objects typical for most theorem proving situations. Furthermore, symbolic representations can easily be understood and handled by humans. Symbolic representations – especially if they allow the introduction (invention) of new predicates (concepts) that have not been defined by the user – are very powerful. However, efficient learning methods for general symbolic representations do not yet exist. *Inductive logic programming* (see e.g. [MR94]) provides some very first answers to this problem, but the learning methods proposed there have not been used for learning theorem provers so far, and are usually not very well suited for dealing with approximative and probabilistic knowledge.

The symbolic representations prevalent in learning theorem provers allow only limited generalization and abstraction. So far there have been no attempts at generalizing and abstracting from multiple training examples (problems) or for identifying characteristics which are common to several training examples (problems). Typically, variants of memorization are applied and therefore large amounts of data are accumulated. Because of this accumulative nature, and because of possibly conflicting information about a given proof situation, the application of the learned knowledge often requires a complex search process of its own (see Sections 3.5, 3.6, and 3.7).

Most approaches for learning in theorem proving which are related to analogy use symbolic representations. Traditionally (see [Kl71] or [Br+88]), proofs are stored as sequence of inference steps allowing the deduction of a goal from the axioms. No generalization is performed during learning. However, generalization takes place when the learned knowledge is applied. Symbols of the source problem are mapped to symbols of the target problem, and with the help of this symbol mapping, proof steps of the source proof are transferred to the target proof. The necessary symbol mapping is often provided by the user.

The work on analogy which is described in [Me95] and [MW97] is inspired by earlier work done in the field of *planning*. Single inference steps are generalized to *proof schemata* and extended to *methods* which are complex, *frame-like* structures. They contain *meta-level information*, such as *pre- and postconditions* and *justifications*, which has to be specified by the user. Instead of storing a proof as sequence of inference steps, a so called *proof-plan* consisting of methods is stored. Proof-plans are stored in a *proof dictionary* (a list of proofs, possibly with efficient indexing methods), and are usually accessed by trying to find *(restricted) second-order matches* between conjecture and the axioms of the current proof task and recorded previous proofs. No abstraction or generalization takes place during learning. Instead, certain forms of abstraction, generalization, and reformulation, which are predefined by the user, occur when the learned knowledge is applied (see Section 3.6). Similar work on proof-plans and analogy can be found in [Si86] and [Bu88].

Flexible re-enactment, which is described in [Fu96], [Fu97b], and [Fu97c] and which has been used for the systems DISCOUNT and CODE[4], is also based on the use of analogy. Instead of storing complex proof-plans, the proof is abstracted into the set of facts contributing to the proof. Knowledge retrieval is performed by searching in the knowledge base for the most similar case with respect to a suitable similarity measure (see Section 3.9). Abstraction on the level of facts is not performed, however, generalization to different signatures is performed by computing symbol mappings when the knowledge is applied.

In the PLAGIATOR system (see [KW94], [KW95], and [KW96]), the stored *proof shell* is a *second-order generalization* of a proof, i.e. a pattern or schema describing the conjecture and all necessary preconditions for the original proof (see also Section 3.4.1). For retrieval, the conjecture of a new problem is matched against the schematic conjecture of a proof shell (see 3.5). Proof shells are collected in a flat proof dictionary.

So, symbolic representations are usually chosen for learning proof systems that focus very much on the application of learned knowledge, while not putting much effort into the learning phase. As we will see in the following subsections, the more the representation shifts to non-symbolic ones the more the focus wanders to the learning phase.

3.3.2 Hybrid Symbolic/Numerical Representations

Hybrid symbolic/numerical representations such as *annotated term patterns* (see [DS96b] and [De+97b]) and *term space maps* (see [DS96b, SB99]) have been implemented for the learning variants of DISCOUNT. They are used to learn heuristic evaluation functions for equations. Both approaches currently use similar knowledge representation schemes. Instead of storing positive facts like flexible reenactment, sets of *annotated term patterns* are stored. Term patterns are generated from a fact by substituting function symbols in a fact (equation) with *second-order arity-preserving variables*, thus abstracting from the signature.

Annotations contain simple information about the role of a fact in the proof, e.g. the number of applications or distance from the final proof state (in inferences performed). In this way, these hybrid symbolic/numeric representations try to overcome the limitations of purely symbolic representations by compiling information from different (training) examples into a single numeric rating of the term pattern. Note that signature information is dropped and facts from different source proofs are no longer distinguished. Annotated patterns are kept in a data base organized for efficient retrieval and evaluation. The patterns can be indexed by the conjecture or the axiomatization of the proof problem they occurred in. If this index is used for retrieval, term patterns can be partially instantiated during the retrieval. However, good results have been reported even for totally flat knowledge bases.

Term patterns and term space maps are clearly more powerful representations than those used e.g. for flexible re-enactment. Therefore, space requirements are reduced considerably. However, there are also clear limitations to their representational power.

[4]CODE is a special prover for condensed detachment problems. It is based on a generating calculus.

For example the concepts 'occurrence of a specific subterm' or 'occurrence of a specific unification pattern' cannot be represented. Term patterns and term space maps are still relatively easy to interpret and understand for humans. As long as the values of annotations can be computed efficiently, learning times are very short, and incremental learning (updating the knowledge base with knowledge from new training examples) is usually easy. Retrieving the learned knowledge can be done by a single scan of the knowledge base (which is usually organized as an index tree), and no backtracking or patching is required.

3.3.3 Numeric Representations

In many approaches for learning theorem provers, numeric representations are used to learn *heuristic evaluation functions* for alternatives in proof situations. Numeric knowledge representations encode knowledge as a fixed-size set of numeric parameters, organized as tuples or matrices. These values are usually interpreted as parameters for a *functional model*, i.e. a function specified by the additional parameters which have to be adapted during the learning phase.

Numeric representations are attractive for the following three reasons. Firstly, numeric knowledge representations are very suitable to express the uncertain and inexact knowledge typical for heuristic evaluation functions. Secondly, there exists a lot of knowledge about concepts of *distance* and *similarity* for numeric representation (see also Section 3.9). Finally, there are a lot of well-understood and powerful learning methods from the field of *statistics* and *numeric optimization* (see Section 3.4.3), which can deal with numeric representations. Since these learning methods abstract and generalize based on sets of training examples, they can usually discover more compact and more general concepts than most symbolic learning algorithms. Learned knowledge is represented by a single, very compact collection of numeric values, compared to the mentioned tendency of many symbolic knowledge representations to grow monotonically with the addition of new examples. This compact representation also leads to an efficient application of knowledge – typically, no search or selection is needed in the application phase.

These advantages, however, are set off by some disadvantages. Numeric learning algorithms usually optimize the parameters of functional models which map one numeric representation to another one. Before being able to apply a numeric learning algorithm for learning heuristic evaluation functions, we therefore have to transform their input into a numeric representation. Knowledge about proof processes typically needs to deal with proof states, i.e. with recursively defined objects like first-order terms or even more complex structures (clauses, sets of clauses, tableaux) incorporating terms. The size of these structures cannot be predicted a priori. Therefore, there is no straightforward way of using numeric representations (usually *fixed-length tuples* of numeric values) to represent logical terms or clauses without loss of information. The traditional way for transforming such structures into numeric representations works by selecting certain features. A *feature* in this case is an easily computable abstraction of a formula, such as the number of symbols, its depth as a tree, or the number of variables. Most features used in automated theorem proving are numeric, that is, the possible feature values are (real or natural) numbers. Boolean features also occur, but can be easily mapped onto numbers as well. Given a set of features, a formula can be represented by its *feature vector*, a vector of feature values (in a given order).

However, the definition and selection of features already introduces a very strong *bias* and severely limits the class of functions and relations that can be expressed or learned for the respective domain of formulae. The reason for this is that aspects of the formulae which are important for the learning task may be either completely lost during the transition to the feature-representation, or may become very difficult to extract from it. Usually, very little or nothing about important features for heuristic evaluation functions is known in advance, and therefore the selection of features is based on secondary considerations like ease of computation, or the "hunches" of developers. The resulting loss of potentially important information is a severe disadvantage shared by all feature-based approaches for learning in theorem proving. The representational power of feature-based approaches is further restricted by the chosen functional model. Recent approaches, e.g. *folding architecture networks* (see below) manage to overcome these problems to a large degree.

A major disadvantage of all numeric knowledge representations is that the learning process is usually quite expensive in terms of processing time. While symbolic learning algorithms are often able to *construct* a hypothesis from the examples, learning for numeric representations most often involves a *search* for good parameters in a multi-dimensional space. This also means that the algorithms are usually not incremental, i.e. additional training examples usually require a complete re-learning. The fact that training examples are only used to guide a search process instead of being used directly in the construction of a hypothesis, as well as the fact that numeric knowledge representations are very compact, also mean that the learned knowledge is very hard to interpret for humans. The effect of individual training examples is usually small, and not at all obvious in the final representation.

Numeric knowledge representations are applied in the systems described in [SE90] and [Go94]. In both cases, proof situations are described by feature vectors, and *standard multi-layer perceptrons* [Ru+86] trained with the *error back-propagation algorithm* (see Section 3.4.3) are used as functional model. Three-layer perceptrons are capable of approximating any function from one finite-dimensional real vector space to another with arbitrary precision (see [Cy89] and [Ho+89]). Thus, the choice of the functional model does not limit the representational power here (although learning method and feature selection still do). A more traditional but less powerful computational model, linear polynomials of the feature vectors' components, is used in [SF71] and [CL73], pp. 154ff. Learning is of course easier than in case of multi-layer perceptrons.

The approach described in [Fu95] is based on the concept of a *weight function* for terms or formulae. Weight functions are used in many theorem provers, e.g. in DISCOUNT and OTTER, to measure the complexity of terms and formulae. In general, simple formulae, that is formulae with small weight, are preferred. A weight function can be seen as a special kind of heuristic evaluation function. The weight of a term is recursively defined as the weight of the top symbol plus a linear combination of the weights of the subterms. Free parameters, which are usually determined by the user, are the weights for the symbols of the respective signature, a special weight for variables, and the symbol-specific coefficients for linearly combining the weights of the subterms of a function or predicate symbol. The goal in [Fu95] is to learn good parameters for a weight function. Obviously, there are clear limitations to the representational power of weight functions.

Features can be used in direct ways to realize a heuristic evaluation function. A recent approach for the theorem prover DISCOUNT is described in [Fu96]. Given k features $f_1, \ldots, f_k$, let V_i denote the set of *permissible feature values*, i.e. the set of all feature values of positive steps (w.r.t. a feature f_i). Given a possible future step — which has to be assessed by a heuristic evaluation function — the *minimal distance* $\Delta_i = \min(\{|v_i - v| : v \in V_i\})$ of its feature value v_i and the permissible feature values w.r.t. feature f_i is computed. The *deviations* $\Delta_1, \ldots, \Delta_k$ are added up using coefficients $c_1, \ldots, c_k$ giving a global *penalty* $\sum_{i=1}^{k} c_i \Delta_i$. This penalty is used to modify one of DISCOUNT's standard weight functions. Learning this kind of heuristic evaluation functions is rather straight-forward (see Section 3.4.3) and can be done rather efficiently. However, the representational power is rather limited, and many concepts on feature vectors cannot be represented in this way. It is for example impossible to capture concepts based on combinations of values for different features.

In [Go97], *folding architecture networks* (see also [GK96] and [KG96]) are used for learning heuristic evaluation functions for the theorem prover SETHEO. Folding architecture networks are a relatively new neural network architecture for the processing of recursive structures such as *labeled, ordered trees* or *DAGs* (directed acyclic graphs), and hence can directly process logical terms and formulae. Folding architecture networks are a recursive generalization of both standard feed-forward multi-layer perceptrons and *discrete time recurrent networks*. They consist of two standard feed-forward multi-layer perceptrons: an *encoder network* which is recursively unfolded to encode (compress) a given structure (labeled tree, DAG or term), starting at the leaves, and a *transformation network* which computes the output for the compressed representation of an input object.

In contrast to standard neural network approaches, features for representing terms or formulae are not needed. The user's or developer's task is reduced to assigning labels (fixed-length vectors of reals) to all symbols from the respective signature, with experimental results indicating good results for most obvious encoding labels (i.e. *one out of n* or a binary representation with a Hamming distance of 1). Folding architecture networks can be trained with an extension of the back-propagation algorithm called *back-propagation through structure* (Section 3.4.3). This learning method determines optimal weights not only for the transformation network but also for the encoder network. This means that encoding and transformation are combined into a single process. The encoding is optimized exclusively for the learning task at hand. Features that are relevant for the learning task are detected automatically.

It has been proven that any function from the domain of labeled ordered trees to vectors of real numbers can be approximated with arbitrary precision by a folding architecture network (see [Ha96] and [HS97]). Furthermore, it has been shown that for any *bottom-up tree automaton* there is a folding architecture network which simulates it (see [Kü98]). Note that e.g. arbitrary weight functions (as described above) can be computed by folding architecture networks.

As has been pointed out several times in this section, there are close connections between the representation used for learned knowledge and the method used to acquire this knowledge. Therefore, the following section on the concrete learning methods will have a similar structure as this section. Note however, that there are different learning methods that work on the same representations, and different representations to which multiple different learning methods can be applied. So, separating the two problems – as we have done – is necessary.

3.4 What Learning Method to Use?

In the preceding section, we have compared different ways for representing what is learned from previous proof experience. A crucial requirement for any useful representation scheme is the existence of a *learning method*, that is a procedure or algorithm which analyzes the (training) example(s), extracts the relevant information, and brings this information into a compact form suitable for the representation scheme. In analogy to the three different knowledge representation schemes, there exist three main classes of learning algorithms: *memorization-based* approaches, *example compilation* techniques, and *numeric optimization* procedures.

3.4.1 Memorization-Based Approaches

As already mentioned in Section 3.3, memorization of examples (often after some abstraction or generalization) is the predominant learning method in case of symbolic representations. Hybrid symbolic/numeric techniques compile large numbers of examples into a single, compact data structure while numeric learning algorithms achieve even more compact hypotheses by optimizing parameters of a functional model.

The variants of memorization that are used by learning theorem provers differ mainly with respect to the degree of abstraction and generalization that is performed during learning. *Memorization* in its pure form means *learning by heart*, that is every training example is stored separately with almost no generalization or abstraction taking place. The only step of abstraction that is performed is the selection of which parts of a proof search are stored (see also Section 3.2 and Section 3.3). Storing a separate entry for every example leads to continuously growing data bases. Pure memorization can e.g. be found in learning provers using analogy, particularly those inspired by *derivational analogy* (see [Ca86] or [CV88]). Two examples are detailed in [Me95], [MW97] (on the level of proof plans) and [Fu96], [Fu97b], respectively. In the first case, complete proof plans are memorized and their decision sequence is replayed during the application phase. This application phase allows various patching strategies if complete correspondence between the stored proof and the one for the new problem breaks down. In the second case, proofs are represented as sets of (relevant) facts, thus abstracting from the order in which they are generated. During application, facts similar to those in the source proof are preferred.

Systems using *explanation-based learning* (see [Mi+86] or [DM86]) and related techniques perform more work in the learning phase than systems relying on pure memorization. The basic idea of this learning approach is to analyze an existing proof and to generalize it by computing the *weakest possible precondition* necessary for the success of the proof, e.g. to drop unnecessary axioms and to abstract from instantiations enforced by the conjecture only, while still ensuring that the remaining preconditions logically imply the (generalized) goal. For theorem proving, the related concept of *reusing proofs* (see [KW94], [KW95], and [KW96]) has been introduced. In this case, the first-order generalization of the proof is restricted to finding the necessary case-specific axioms and omitting details about the application of the axioms. On the resulting *proof catch* a second-order generalization is performed, by substituting function symbols with function variables, where independent occurrences of the same function symbol can be substituted by different variables. The final result of the learning process is a *proof shell*, a pattern describing the conjecture and necessary preconditions of the original proof.

In many of the learning provers using memorization at the level of proofs, most of the abstraction and generalization is done during the application of learned knowledge (see Section 3.6), not during the learning phase. The application phase for a new problem can be split into two parts: finding the most similar case in the proof dictionary (see Section 3.5), and making the learned knowledge found there applicable for the new proof problem (see Section 3.6). Therefore, many approaches using memorization can be seen as a form of *Case-Based Reasoning/Learning* (CBR, see e.g. [Ko92]).

[Fu97a] describes a purely case-based approach, using features to characterize proof problems and the nearest neighbor method to retrieve information about which out of a fixed set of strategies is likely to perform well on a new problem (see also Section 3.5 and Section 3.9). [FF97b] and [FF97c] explicitly apply CBR to learning theorem provers.

3.4.2 Example Compilation Techniques

Learning by pattern memorization (see [DS96b] and [De+97b]) is a borderline case between symbolic and hybrid learning algorithms. In this approach, information from several facts (equations), which may even originate from proof searches for multiple different problems, can be compiled into the annotations of a single *representative term pattern*. Memorization is performed on the level of facts, not on the level of proofs. The basic learning algorithm for learning by pattern memorization starts with analyzing the listing of inferences to determine the set of facts actually leading to the proof[5]. For these facts, it then calculates some relevant properties (distance from the goal, number

[5]Work on the use of negative examples is reported as "in progress", a first prototype has been presented at the CASC-14 ATP system competition [SS97b].

of applications in the proof). The facts are then individually generalized into a term pattern by consistently substituting function symbols with function variables, and the properties are added as annotations. The resulting pattern is stored in a data base indexed for efficient retrieval. If multiple copies of the same term pattern are inserted into this data base, their annotations are combined.

Variants of the basic algorithm maintain a separate data base for each (generalized) axiomatization and each (generalized) goal, and substitute function symbols occurring in the axioms or the goal consistently throughout the data base. The result of the learning process is always a set of term patterns with abstract information about the usage of the original facts. These annotated term patterns can be used directly to evaluate facts with equivalent patterns during future proof searches.

Learning by term space mapping is an example for a compilation-based hybrid learning algorithm. A term space map maps (a finite subset of) the space of possible *term nodes* to evaluations of these nodes. The learning algorithm starts out with annotated patterns of facts derived from proofs as described for the previous approach. It constructs a recursive structure (a *term evaluation tree* or *term space map*) describing all growth alternatives[6] for these terms, starting at the root. For each node, an evaluation is computed from the properties of the patterns containing this node. The final term space map is then used to compute heuristic evaluations for new terms by collecting the annotations of nodes of the term space map corresponding to nodes in the term that has to be evaluated.

3.4.3 Numeric Optimization Procedures

The existence of powerful numeric optimization procedures is one of the main reasons why numeric representations have been considered for learning theorem provers. As already pointed out in Section 3.3.3, learning in this context means finding an approximation to an unknown target function (i.e. a heuristic evaluation function), which in most cases is a mapping from one tuple space of numeric values to another[7]. Training data for this kind of learning consists of a *finite sample*, that is a finite number of *training examples* (pairs of possible input tuples with the respective target output tuples). A further precondition for the application of numeric optimization procedures for learning is the selection of a functional model with *free numeric parameters*, and with the same domain and range as the function which has to be learned.

Applying a numeric optimization procedure further requires a *cost function*, or an *error-* or *fitness measure*. Such measures compare the target function to the function which is actually computed by the functional model with the current set of parameters. Since the target function is only known for the training sample, this comparison can only be done with respect to the training sample. An error measure of zero normally indicates that the model is perfectly adapted to the target function. However, this does not

[6]Currently published approaches only distinguish between function symbols of different arities, but other abstractions are possible and outlined as future work.

[7]In most cases the output is only a single value, interpreted as an evaluation of a fact or proof situation.

necessarily mean a low generalization error, that is a small expected error of the trained model with respect to new input. The trained model may be *over-fitted* to the training sample. Fortunately, for many functional models the following can be guaranteed: the bigger the size of the training sample, the higher the correspondence between the error on the training set and the generalization error. The exact relationship depends on the diversity of the functional model. See e.g. [Va95] for a theoretical foundation.

Learning with a numeric optimization procedure means finding parameters for the chosen functional model which minimize the error measure. Note that an error measure enables the examination of a parameter set and the corresponding heuristic evaluation function without actually testing it on the proof system (which usually would be far too expensive). In order to understand the following, the reader should keep in mind that the size of the training sample, that is the size of the finite initial part of the search space which generated the training sample (see Section 3.1), is usually very small compared to the search space that is explored by a theorem prover when solving the corresponding proof problem without heuristic guidance. Therefore, if the search guided by a heuristic evaluation function remains within this finite initial part and allows to find at least one of the proofs "contained" in it, such a heuristic evaluation function is regarded as perfect and an error of zero should be assigned to it.

In most approaches, standard error measures such as the *mean squared error* are used for learning heuristic evaluation functions. A general assumption for standard error measures is that the target values of individual training examples are independent of each other. Unfortunately, this is not true in case of training examples for a heuristic evaluation function. An error measure should e.g. disregard the rating of a heuristic evaluation function for a negative training example N if the heuristic evaluation function already assigns a bad rating to the predecessor of N in the search space (search tree). The reason for this is that the search process, if guided by this evaluation function, would never explore the state corresponding to N. Similar dependencies exist between positive training examples. Furthermore, it seems reasonable to use relative target values instead of absolute ones. For an analysis and a more adequate error measure for learning heuristic evaluation functions see [Go97].

It can be reasonable to occasionally refine the training sample by a new test run with the proof system (as suggested in [Fu95]). If e.g. at some point during learning the optimization procedure is not able to produce further improvement, this might be caused by the finite initial part of the search space that was selected to provide training data. Perhaps a simple heuristic with zero-error with respect to the sample does not exist. However, another finite initial part of the search space containing new proofs could perhaps allow a simple heuristic. The heuristic evaluation function that is computed with the currently best parameter set can be used to guide the prover and to determine such a new finite initial part for providing training data. Such an approach seems especially justified, if training data from several proof problems are combined and the currently best heuristic already provides good guidance for some of these proof problems. Since in this way better and better heuristic evaluation functions are found the whole procedure could be called *bootstrapping* (see [Go97]).

Various numeric optimization procedures have been used for learning heuristic evaluation functions for theorem provers. One of the first approaches was *least square estimation* (see [CL73], pp.154ff and [SF71]), applied to linear polynomials of the feature vector components. More advanced feature-based approaches (as [SE90] or [Go94]) use multi-layer perceptrons (neural networks) trained by the *error back-propagation algorithm*. Multi-layer perceptrons compute continuously differentiable functions. The error back-propagation algorithm (see [We74] and [Ru+86]) is a very efficient method for computing the *gradient* of the error measure[8] with respect to the parameters (weights) of a multi-layer perceptron. The gradient computation has the same complexity as the computation of the error measure itself. This is one of the reasons why multi-layer perceptrons are so widely applied. Having a gradient to control search is certainly a big advantage for a learning method since it gives at least locally the best direction for improving the current hypothesis. However, *gradient-based search* has also disadvantages. It can e.g. get stuck in local minima of the error function. Therefore, though the function which has to be learned may be computable by the chosen network, it is not guaranteed that learning will converge to it.

Besides multi-layer perceptrons, *folding architecture networks* (see [GK96] and [KG96]) have been used for learning heuristic evaluation functions for the theorem prover SETHEO (see [Go97]). As already described in Section 3.3.3, folding architecture networks compute mappings from labeled trees or general DAGs to real vector spaces, and they can be trained by an extension of the back-propagation algorithm, called *back-propagation through structure* (BPTS). As in the case of standard error back-propagation, gradient computation with BPTS has the same time complexity as the computation of the error measure itself. The time complexity of both computations is linear in the number of nodes in the graph-representation of the structures (e.g. logic formulae) from the training sample. Folding architecture networks always produce the same result for a tree- and a DAG- representation of a logical formula. Therefore, choosing a *minimal DAG-representation* for the logic formulae from the training sample can lead to an exponential reduction of the time complexity of the gradient computation. In practice, factors between five and fifty are reported.

BPTS is a gradient descent optimization just as standard error back-propagation. In contrast to the positive theoretical results on the representational power of folding architecture networks (see Section 3.3.3), convergence to a solution can only be guaranteed for very special learning tasks. However, a couple of publications show that in practice BPTS is a very efficient learning method: artificial term classification problems, in [GK96], [KG96], and [Sc97], predicting quantitative structure activity relationships in chemistry in [SG98], and theorem proving in [Sc+97] and [Go97].

The approach described in [Fu95] uses a *genetic algorithm* (see e.g. [Go69]) to optimize the parameters of a weight function for a given proof problem. Instead of keeping only one hypothesis and trying to constantly improve it, genetic algorithms use a set of candidate hypotheses (called a population of individuals) that are constantly evaluated by a fitness function. New individuals are generated by combination (*crossover*) or

[8]Note that in order to apply gradient descent techniques, continuous differentiable error measures are needed.

modification (*mutation*) of old ones. Fitter individuals are given a higher probability of generating new individuals, however, due to the use of random effects sometimes less fit individuals can survive and produce new individuals as well. This makes genetic algorithms more expensive than e.g. gradient descent. However, genetic algorithms are usually better in handling local minima in the parameter space than gradient-based search algorithms.

In [Fu96], a set of permissible values for each feature is used to determine a minimal distance for each component of a feature vector and a weighted sum of these minimal distances determines the final output of the heuristic evaluation function (see Section 3.3.3). Therefore, learning in this context means finding a set of permissible values and a vector of coefficients for each feature. Suitable coefficients are determined algorithmically using both positive and negative examples. The basic idea is to choose the coefficients so that negative examples receive a penalty which is (much) larger than a possible penalty of positive examples. As a consequence, positive examples will be preferred by the heuristic which takes the penalty into account. However, the penalty should be moderate, so that some of the negative examples — which might play the role of positive examples for a proof problem that is rather similar to the training or source problem — will not be delayed too long. Hence a suitable trade-off must be found. See [Fu96] and [Fu97c] for details.

As this section has shown, many of the learning methods known in machine learning have been also integrated in learning theorem provers. However, due to the structure of the objects the provers deal with, theorem proving offers some challenges for these methods. The success of a learning theorem prover does not solely depend on a good learning method. The application of the learned knowledge and a good handling of the problems that have to be solved for such an application is of equal importance.

3.5 How to Detect Applicable Knowledge?

The question of how to detect applicable knowledge addresses the problem of finding those pieces of knowledge that are most likely to be useful for solving a given new proof problem. While this is of little importance for general knowledge concerning the calculus or the proof procedure, it becomes more important for domain-specific knowledge and is of prime importance if problem-specific knowledge has been learned. Of course, a method for detecting applicable knowledge strongly depends on the method used to apply it. Detecting applicable knowledge is a crucial step in the problem-solving process of any learning problem solver, since it provides the application phase with the necessary input. Recent works have acknowledged the importance of detecting applicable knowledge *automatically* and have produced several techniques for dealing with this issue.

In Section 3.3, we already argued that it is profitable to organize the learned knowledge in a knowledge- or data base, which may e.g. be organized in a hierarchical manner. Detecting applicable knowledge then means retrieving it from such a data base using some sort of "search key". Since the knowledge to be retrieved is to be used to solve a given problem, this very problem is the most obvious and suitable search key. Therefore, all methods for detecting (retrieving) applicable knowledge center, in one way or another, on comparing the problem to be solved with data base entries using some kind of similarity measure (see also Section 3.9). Essentially, the following alternatives for detecting applicable knowledge (within a suitably designed data base) are most widely used:

- Requesting user input.
- (Higher-order) matching.
- Using similarity criteria.

Asking for user advice is often done implicitly, by having the user manually select a knowledge base or a set of training problems for tuning the prover. Of course this alternative is only acceptable in the experimental stages of research or, in some cases, for interactive provers. (The "data base" then corresponds to the user's experience.) Sooner or later automated methods using matching or (general) similarity criteria and "real" data bases are called for. The fact that most learning provers we are aware of, except for those explicitly discussed in this section, still rely on user input shows that we are still in a relatively early stage in the development of learning provers.

The majority of methods for detecting applicable knowledge center on matching. Certain parts of solved problems are stored in a data base and are matched against the corresponding parts of a new problem. Given that a (proof) problem is specified by a set of axioms and a goal to be proven, relevant parts can include the goal only, the axioms only, or both axioms and goal.

[KW94] uses only the goal as a key to detect applicable knowledge. Found proofs (see Section 3.2) are generalized, resulting in formulae (proof shells) substituting *function variables* for function symbols. Given a new problem with a goal G, a proof shell represents applicable knowledge if its (generalized) goal $\hat{G}$ satisfies $\pi(\hat{G}) \equiv G$ for some second-order match π. The method for applying the knowledge (see Section 3.6) justifies this kind of retrieval. Note that second-order matching allows function symbols of different arity to match each other. For instance, the second-order term $F(x, y)$ — F being a function variable — may be instantiated as $s(x)$ or $s(y)$, where s is a unary function symbol, and the second or first argument of F is ignored. Similarly, the second-order term $S(x)$ may be instantiated with $f(x, x)$.

In [DP97] a second-order matching approach is used as well, but both goal and axioms of the new proof problem form the search key. In this case a partial solution of the matching problem will produce a set of lemmata that have to be proven additionally.

The approach presented in [DS96b, SB99] is designed to abstract and generalize from a huge amount of training examples stemming from different training problems. Therefore, there is basically no need for detecting applicable knowledge. However, it does make sense to restrict the amount of usable knowledge for various practical reasons. Usually, the applicable (eligible) knowledge is limited to the domain (e.g. *rings* or *lattice-ordered groups*) to which the problem to be solved belongs. This is reasonable especially when domains differ substantially. Domains can be identified by examining the set of axioms. Naturally, there are many ways to formalize the same theory, and the problem to identify equivalent theories is undecidable. For practical use, a given problem is classified as a member of a certain domain if all the axioms describing the domain can be shown to hold under the axioms of the problem within a limited calculus (in [DS96b], only *normal forms* with respect to the axioms are computed, see also [DK96]). In order to abstract from specific names for function symbols, *signature matching* between the second-order variables used in the knowledge base and the problem's signature is performed.

Similarity measures are the third and most general approach for detecting applicable knowledge. They too are sometimes based on first- or second-order matching. More generally, these measures represent "agreement" of axioms and goals of proof problems with numeric values (see Section 3.9). These values can then be employed to define notions like "sufficiently similar" and "more similar than". Based on these notions, retrieval can identify knowledge that — according to the similarity measure — appears to be (sufficiently) applicable. Naturally, these ("quasi") matching criteria are merely a heuristic guideline for detecting applicable knowledge. However, often there is no certain way to determine applicability a priori. To put it another way, the use of similarity measures heavily relies on the assumption that similar problems (most likely) provide applicable knowledge — an assumption that is reasonable for "appropriately designed" similarity measures.

A similarity measure also is necessary when embedding learning theorem provers into the framework of case-based reasoning (CBR). CBR is a well-established area of artificial intelligence that provides a general framework for analogy-based problem solving. Storage, retrieval, adaptation, and application are the key elements of CBR. These elements also appear in learning theorem provers. Employing CBR in order to create largely user-independent learning theorem provers therefore appears to be quite profitable. This hypothesis is supported by recent results (see [FF97b], [FF97c], and [De+97a]).

An approach to detecting applicable knowledge using similarity without falling back on matching criteria is the use of features and feature vectors (Section 3.3.3, p. 19) for describing proof problems. Similarity is defined via a distance measure on feature vectors (usually Euclidean distance or a variant). The most similar problems (*nearest neighbors*) are the ones most likely to provide applicable knowledge. This approach has been successfully applied to selecting appropriate search-guiding heuristics for solving a new problem (see [Fu97a]) given a data base of solved problems and the respective heuristics that were used to solve them.

The detection of applicable knowledge only sets the stage for the application of it. Note however that nearly all solutions presented in this section give no guarantee that the selected knowledge really is useful. We will deal with this problem in Sections 3.7 and 3.8.

3.6 How to Apply Knowledge?

There are three main lines of research dealing with the application of knowledge in connection with automated deduction:

- Transformational analogy.
- Derivational analogy.
- Search-control heuristics.

Naturally, there are again borderline cases and hybrids. However, all published approaches essentially revolve around these three lines of research.

Approaches based on *transformational analogy* employ knowledge in the form of solutions (i.e., proofs). A proof of a problem solved in the past (*source*) is transformed into a proof of the current problem (*target*) by means of an *analogy mapping* which is derived by comparing source and target problem. Naturally, this transformation process will not always succeed. Depending on the chosen analogy mapping it is possible that the result of the transformation process is not a proof, or it is a proof, but not the required one. The former reason for failure is the most frequent one and calls for patching strategies to recover from and repair failures (see also Section 3.7).

The work on *reusing proofs* described in [KW94], [KW95], and [KW96] can be classified as transformational analogy. In Section 3.5, we already outlined that applicable knowledge is detected using a second-order match π for the generalized goal $\hat{G}$ from a proof schema and the goal G from a given new problem such that $\pi(\hat{G}) \equiv G$. Let $\hat{Ax}$ denote the generalized axioms form the proof schema and Ax the axioms for proving G. If for every $\hat{a} \in \hat{Ax}$ there is a $a \in Ax$ so that $\pi(\hat{a}) \equiv a$, then G is proven by virtue of the proof of $\hat{G}$. (Note that it is possible that $\pi(\hat{a})$ still contains unbound (free) function variables, namely those which did not occur in $\hat{G}$. In this case a further second-order match ϕ must be found so that $\phi(\pi(\hat{a})) \equiv a$.)

Comparing [KW94] to the general outline of transformational analogy, it is obvious that the required analogy mapping corresponds to the second-order match π. In the case that there is an $a \in Ax$ for every $\hat{a} \in \hat{Ax}$ so that $\pi(\hat{a}) \equiv a$, the abstraction (generalization) method guarantees that we have a proof. If there are $\hat{a} \in \hat{Ax}$ so that there is no $a \in Ax$ satisfying $\pi(\hat{a}) \equiv a$, various patching strategies [KW95] may be applied. One may e.g. try to prove the missing $\pi(\hat{a})$ perhaps by applying another proof schema, and the process may continue recursively (see also [KW96] and Section 3.8). However, applying a proof schema in this way generates a search of its own. Thus applicability becomes undecidable and has to be controlled heuristically. The success of reusing proofs — like most approaches based on transformational analogy — critically depends on a high degree of similarity between source and target problem.

Note that the approach essentially utilizes generalized proofs (i.e., solutions) as a kind of (analytical) meta inference rule. The same is true for the approach of [DP97]. A major problem for this kind of learning is that it reduces the length of a proof at the expense of increasing the average branching factor in the search space. This effect has been analyzed for *explanation based learning* in the domain of planning systems (see [Mi90], [Et93], and Section 3.2).

Derivational analogy explicitly focuses on *how* a proof was found rather than the proof itself. It employs the decisions made and the actions taken which finally led to a proof. Thus, when trying to find a proof of a new problem, derivational analogy attempts to make analogous decisions and take analogous actions.

In [Br+88], such an approach is described. There, the user pairs facts (axioms) of the source problem with facts (axioms) of the target problem, if they appear to have an analogous meaning. Then, the automated prover applies the same inferences to facts of the target as were applied to their analogous counterparts of the source proof[9]. A fact resulting from such an inference is again considered to be analogous to the corresponding fact in the source proof. If the source proof no longer applies, then a "standard" prover (i.e., a prover employing some basic heuristic) takes over. Furthermore, [Br+88] also describes several methods to recover from breakdowns in analogy (e.g., if no "analogous" inference is applicable). The recovery mechanisms may involve backtracking, because facts of the source often can be paired with more than one analogous fact of the target, thus allowing the exploration of analogies in a depth-first manner (see Section 3.7). It may also involve a conventional proof search.

Hence, the approach presented in [Br+88] has two phases. First, it attempts to exploit a source proof by re-enacting its *analogous* steps in order to arrive at a target proof. The search possibly involved in this process is different from the search the standard prover would conduct. Second, if the source can no longer be exploited, a standard prover that does not make use of the source in any form takes over.

In [Me95], derivational analogy is applied to inductive theorem proving. The approach centers on proof planning using tactics and methods (see also Section 3.3.1, page 17). A proof plan can be considered as a set of instructions on how to obtain a proof. Naturally, during the application of a proof plan, failures can occur (e.g., a certain tactic or method suggested by the proof plan is not applicable). To a certain extent failures again can be compensated for by appropriate patching techniques (see also Section 3.7) such as *reformulations* of proof plans. However, these patching strategies again introduce a search process which has to be controlled heuristically.

Transformational and derivational analogy build on a deterministic knowledge transfer. Search is avoided as much as possible and is brought into play only through patching attempts. In contrast to this, *search-control heuristics* explicitly build on search for transferring knowledge. In most cases search-control heuristics are realized by *heuristic evaluation functions* (see also Section 3.2) which compute numeric ratings for alternatives in the search space. These ratings can then be used to define an order on the alternatives and thus they realize a search-control heuristic. However, they can also be combined with existing search-control heuristics or strategies in order to improve and refine them. The fact that search is explicitly involved leads to more flexibility. Less similarity between training and target problems is required than in case of ana-

[9][Br+88] refers to the *source problem* and *source proof* as as *guiding problem* and *guiding proof*, respectively.

logical reasoning. Search-control heuristics are related to derivational analogy as both approaches utilize information about *how* to find a proof. In both cases knowledge on the control level is transferred (see also Section 3.2).

[Fu95] centers on adapting or fine tuning the parameters of a weight function for the theorem prover DISCOUNT (see also Section 3.3.3). The resulting heuristic evaluation function is used to evaluate possible inference steps during the proof search. The approach described in [Fu96] and [Fu97c] yields a feature-based numeric penalty for each possible inference (see also Section 3.3.3) of DISCOUNT. This penalty is added to the weight which is assigned by a standard weight function and used to determine the (probably) best inference.

Similar to the approach in [Fu95], the neural approaches described in [SE90], [Go94], and [Go97], as well as the numeric learning algorithms described in [SF71] and [CL73] yield an easily applicable evaluation function, either in the form of a pre-trained neural network or as a weight vector for a given set of features. However, in contrast to the approaches mentioned in the last paragraph, these approaches deal with systems based on analytical calculi. As described in Section 2, such systems perform a depth-first iterative deepening search. Heuristic evaluation functions can be applied in two ways, i.e. *locally* or *globally*, to guide the search process. If they are applied locally (as in [CL73] and [SE90]), they are only used to determine an order on the possible successor states of the current state. Applying them globally means using their ratings also for the decision whether the system should backtrack (see [Go94] and [Go97]). The ratings are then applied in a similar way as traditional bounds (see Section 2) to realize an iterative deepening search.

Flexible re-enactment (see [Fu96], [Fu97b], [Fu97c], and [De+97a]), used for the systems DISCOUNT and CODE, is closely related to derivational analogy in that it attempts to re-enact proofs by focusing on the positive facts learned from a source problem when tackling a target problem. However, flexible re-enactment is a pure search-control heuristic. Re-enactment is accomplished by giving preference to (re-enactable) positive facts applying a signature mapping if necessary. In the case that there (temporarily) are no recognizable positive facts which can be re-enacted, the search proceeds nonetheless. Flexible re-enactment combines the idea of re-enactment, i.e., preferring positive facts, with a "standard" heuristic, i.e., a heuristic which is in general quite profitable and allows to conduct a reasonable search in the absence of re-enactable positive facts. A distinctive feature of flexible re-enactment is the preference of facts that are not positive facts themselves, but originate from positive facts. These facts have a relatively high likelihood to contribute to a proof. The preference given to these *descendants* of positive facts decreases as descendants become more and more remote w.r.t. their ancestors which are positive steps. Flexible re-enactment can also be combined with the feature-based approach of [Fu96].

The two approaches presented in [DS96b] exploit positive facts collected from successful proofs of *all* problems belonging to a certain problem domain. The generalized and stored abstract patterns of the data base (see Sections 3.3.2 and 3.4.2) are used in different ways. For the first approach, learning by pattern memorization, new facts are matched against the stored patterns (using an efficient data structure to minimize the overhead). If a corresponding pattern is found, an evaluation of the pattern is computed from the stored information, preferring facts that have often been useful in many proofs. The evaluation may also take the current proof phase and the distance from the goal into account. The second approach, learning by term space mapping, compiles the pattern into a term space map. The terms on a fact to be evaluate are mapped onto this structure, and an evaluation of the terms is computed from the annotations stored at the nodes corresponding to nodes of the term.

While the analogy-based approaches for the application of learned knowledge have their roots in concepts developed by the traditional AI an machine learning communities, the use of learned knowledge in the form of search-control heuristics is a rather new approach. Its importance in automated theorem proving and some other applications has led to a growing interest by the machine learning community.

3.7 How to Detect and Deal with Misleading Knowledge?

Similar to the observation that in a calculus not every applicable inference rule should be applied, one can observe that not all the learned knowledge that can be applied in a given situation really furthers a proof attempt. Naturally, if the application of a piece of learned knowledge always resulted in a limited number of actions that end with either a total fail or a successful proof, the detection of misleading knowledge would be simple and could be dealt with by backtracking and trying the next piece of knowledge. However, in most cases, after applying a piece of knowledge, it is neither immediately clear if it is useful for finding a proof, nor is it possible to determine at which future time such a decision can be made. Moreover, the application of a piece of knowledge may result in a behavior of the prover that can make it impossible (or at least much more difficult) to find a proof, although the prover, without using this special knowledge, would be able to find a proof (rather quickly).

The following concepts for solving this problem can be found in learning provers:

User: Interrupts and backtracking (or new run) forced by the user.

Use of time limits: If such a limit is exceeded without finding a proof another piece of knowledge is tested. Instead of time limits, a limit on the number of inference steps or the depth of deduction can also be used.

Conditions: Parts of the applied knowledge are conditions that should be fulfilled after given time periods/number of steps. If they are not fulfilled another piece of learned knowledge is tested.

Impact: The impact of the applied learned knowledge on the search process is periodically tested and evaluated retrospectively. Knowledge without impact or with negative impact is discarded.

The problem of misleading knowledge is well-known, and has been recognized in several publications. For interactive provers, the problem is less severe. Since they are under constant supervision of the human user, these systems can afford to use the first solution, i.e. letting the user deal with misleading knowledge (see [Me95]). Fully automated proof systems, on the other hand, cannot rely on the user in these situations. However, few systems offer satisfactory solutions. This is in part due to the prototypical character of many systems, and in part due to the relative youth of the field of learning theorem provers. None of the approaches detailed in [SE90], [SF71], [CL73], or [DP97] offer a solution, and many other approaches consider recovery only as a secondary feature.

In [KW96] the problem is acknowledged and dealt with by defining a well-founded ordering on proof goals that prevents an infinite number of steps of the prover (theoretically) after a piece of knowledge is applied. Similarly, in [Fu97a] the second solution, use of a time limit, is employed.

In [Go94] and [Go97], the problem of having wrong ratings (computed by a heuristic evaluation function) for search states of the theorem prover SETHEO is addressed by a mixture of the second and the third concept. Firstly, time limits are applied by combining the heuristic ratings with hard traditional bounds. In this way the set of possible search states is made finite before applying heuristic ratings. Secondly, due to using ratings globally (see also Section 3.6) a postcondition for continuing search after a state with a good rating has been explored is that at least one of its successors also has a good rating. Otherwise, backtracking will take place. However, it has to be emphasized, that in this way only misleading knowledge on the level of single wrong ratings of the applied heuristic evaluation function can be compensated. Means for automatically dealing with totally inappropriate heuristic evaluation functions are not proposed in [Go94] or [Go97].

The third solution concept is suggested by works in the area of proof planning, as for example [Bu88], [Hu+94], [SD93], or [GN97]. Although most proof planning systems do not use knowledge learned automatically, but rather employ knowledge in form of tactics, methods, or meta-methods provided by the user, the detection of misleading knowledge is a crucial problem. In such systems, a tactic consists not only of the actions the system should take, but also of preconditions that determine if a tactic is applicable (we already discussed this concept in Section 3.5) and of postconditions that are expected to hold after the tactic is executed. Provers using proof planning techniques have as central structure a so-called *proof tree* that is the result of employing tactics to the problems represented in the nodes of the tree. In addition to the tactic currently believed to solve a node all tactics currently applicable to a node as well as all tactics already tried out are stored in this tree. Part of the tactics are also conditions that provide the control of the prover with information which of the resulting subproblems have to be tested when. This testing is then done starting with the leaves of the tree. Whenever problems cannot be solved within given time limits, alternatives are tried out. Naturally, in such a context also tactics representing learned knowledge can be included and the approaches of [KW94] and [Me95] are supposed to be integrated into such a proof planning system.

Another model for the use of planning is the core of DISCOUNT's solution to the detection and the handling of misleading knowledge: the TEAMWORK method. The TEAMWORK method provides the ideal environment for the application of learned knowledge.

The TEAMWORK method is a knowledge-based distribution method for certain search processes (see [De95]) that correspond to the search performed by generating provers. In a TEAMWORK-based system there are four different types of agents: experts, specialists, referees, and a supervisor. Experts and specialists are the agents that work on really solving a given problem. *Experts* form the core of a team. They are problem solvers (i.e. theorem provers) that use the same inference mechanism, but different selection strategies (including learned heuristics) for the next inference step to do. *Specialists* can also search for a solution or they can help the supervisor, for example by analyzing and classifying the given problem. An example is the PES specialist described in Section 3.9. Each expert or specialist runs on its own computing node. Therefore, the supervisor determines the subset of experts/specialists that are active during a working period depending on the number of available processors.

The search process is organized in cycles. After each working period, a *team meeting* takes place. In the *judgment phase*, each active expert and specialist is evaluated by a referee. Each referee has two tasks: judging the whole work of the expert/specialist of the last working period, and selecting outstanding results. The first task results in a *measure of success*, an objective measure that allows the supervisor to compare the experts. The second task is responsible for the cooperation of the experts and specialists, since each selected result will be part of the common start search state of the next working period. The referees send the results of their work to the supervisor.

In the *cooperation phase* the supervisor has to construct a new starting state for the next working period, select the members of the team for this next period and determine the length of the period. The new start state for the whole team consists of the whole search state of the best expert enriched by the selected results of the other experts and the specialists. The supervisor determines the next team with a reactive planning process involving general information about components and problem domains (*long-term memory*) and actual information about the performance of the components (*short-term memory*). The long-term memory suggests a *plan skeleton* that contains several small teams for different phases of a proof attempt. These suggested teams are reinforced with appropriate experts/specialists (if more computing nodes are available). During each team meeting the plan has to be updated. This means that adjustments are made according to the actual results (see [DK96]).

The solution to the problem of detecting and dealing with misleading knowledge provided by TEAMWORK is the retrospective evaluation of experts and specialists in the judgment phase of a team meeting and the consequences taken by the supervisor in the cooperation phase. Experts or specialists representing or using learned knowledge are compared with each other and with conventional search strategies. If learned knowledge does not help in finding a proof, then the measure of success of the expert/specialist with this knowledge either is bad or will become bad in later team meetings. If the measure is already bad then the referees will also find not many or no facts to select. Firstly this results in forgetting nearly all the consequences of the application of misleading knowledge (remember that referees only select facts that have proven to be good). Secondly, a very bad measure of success forces the supervisor to substitute the expert/specialist by another one (perhaps representing another piece of knowledge).

If, during a team meeting, no clear conclusion about the progress of an expert or specialist can be obtained, then at least some of its facts are good enough to be selected by its referee. This means that its best facts are part of the start state of the next working period so that the expert/specialist can in the next period try to prove that its knowledge furthers the proof attempt. Since this expert/specialist will not be evaluated as particularly bad, the supervisor will not exchange it (unless some specialists provided totally new results that suggest a whole new direction the team should go).

So, TEAMWORK does not only allow the detection of misleading knowledge, but it also provides a way to get rid of those facts generated by such knowledge that are indeed useless. If, by chance, not each result of misleading knowledge is bad, then these facts will not be forgotten (no beginning at point zero as in case of backtracking).

Note that the use of measures of success is in a competitive and therefore relative manner: if there is no good expert then TEAMWORK tries to make the best out of a bad situation. The other solutions to the detection problem lack any possibilities for comparisons and therefore it is difficult to determine whether the prover does behave good or bad at a moment.

The detection of applicable knowledge is usually only a heuristic process. It can neither guarantee that the identified pieces of knowledge are actually useful, nor that all necessary pieces have been identified. Therefore the problem discussed in this section is strongly related to the problems discussed in the next two sections.

3.8 How to Combine Knowledge from Different Sources?

The usage of learned knowledge almost always involves a certain amount of transfer actions to bridge the gap between the source of the learned knowledge and the target problem it should be applied to. This may include rather large series of actions to establish parts missing in order to bridge the gap, and is sometimes referred to as *patching* (see Section 3.6). Similarity concepts (see Section 3.9), application conditions (see Section 3.5) and certain features of the application procedures (see Section 3.6) already take care of small gaps between one source and a target. Gaps can also be closed by a combination of learned knowledge and a standard control strategy (see [Fu97a] for an example).

However, in order to solve more complex and harder problems, the use of knowledge from a single source is often insufficient. As in the case of human beings, several lessons are needed and they all must be understood and combined to tackle interesting problems. Therefore, many of the published approaches to learning in the field of theorem proving try to combine knowledge from multiple sources. Note that *combining* here means the use of knowledge from different sources during a single proof attempt.

It does not mean choosing one piece of knowledge to use out of a data base of several sources (which is covered in Section 3.5) or trying out several pieces one after the other (see Section 3.7). But also note that the problems discussed in Section 3.7 and this section are not clearly separable, and that some concepts provide solutions to both problems.

In general, the following ideas have been introduced to address the problem of combining knowledge from several sources:

- Several recursive calls of the prover.
- Accumulation of knowledge from multiple sources into one large "chunk".
- Cooperation of provers using different pieces of learned knowledge.

The use of recursive calls of the prover in order to solve subproblems generated by learned knowledge is a borderline case with respect to our view on the combination of knowledge from different sources. If there are no dependencies between the subproblems then we try one piece of knowledge after another and there is no real combination. But often, as in the case of proof planning, the subproblems are interconnected and then their solutions cannot always be combined. As already mentioned, [KW96] uses this approach for combining several learned proofs in order to prove a given problem. Recursive calls are also the core idea of [DP97] to deal with the lemmata generated by the partial second-order match.

The accumulation of experiences from several different training problems into one "chunk", i.e. one search control heuristic, is a characteristic of the two approaches described in [DS96b] and also of the approaches of [SF71], [SE90], [Go94], and [Go97]. The two approaches of [DS96b] showed opposing tendencies for a large amount of source proofs. While learning by pattern memorization can handle really large numbers of source proofs and continues to improve the performance of the prover with each new proof added to the data base, experiments with learning by term space mapping showed that after a certain number of source proofs the behavior degrades. This is one of the reasons why, although both approaches have some possibilities for combining knowledge directly, [DS96b] also suggests the third alternative, namely the cooperation of provers using different pieces of learned knowledge.

One cooperation method for theorem provers, already presented in Section 3.7, is the TEAMWORK method. Cooperation and therefore combination of results is achieved by the referees by selecting outstanding results that become part of the new start state for the next working period. Although the TEAMWORK method has proven to be a good cooperation concept (see [DS96b] and [De+97a]), it nevertheless always favors one particular piece of knowledge, namely the knowledge used by the best expert of a cycle. Only for this expert all generated facts are used in the new start state. Since this winner can change from cycle to cycle this is no serious flaw. As a matter of fact, using the search state of the winners as basis for the new start state helps to avoid the case that an ill-suited expert directs the search into areas of the search space which are unprofitable and cannot be escaped from even when employing (originally) well-suited experts.

The TECHS approach (TEams for Cooperative Heterogeneous Search, see [DF99]) has exactly the opposite features, i.e. all experts are treated exactly the same. Consequently, the usage of very bad experts may result in a computing node that permanently explores bad parts of the search space. As in TEAMWORK, the TECHS approach uses experts that employ different search control heuristics. Periodically, these experts exchange outstanding results (facts) that are selected by referees. But the TECHS approach uses two kinds of referees, *send-referees* and *receive-referees*. While send-referees serve in essentially the same role as referees in TEAMWORK, receive-referees make a second selection with regards to the special needs of an expert that receives facts. Since there is no supervisor all experts are in the same role, i.e. the TECHS approach is symmetric. The experimental results in [DF99] demonstrate that the TECHS approach is an interesting cooperation concept (with both advantages and disadvantages compared to TEAMWORK), and allows for the combination of different pieces of learned knowledge (in the form of control strategies).

It should be noted that there have not been a sufficient number of experiments that deal with proof problems for which a combination of learned knowledge is necessary. Only within the cooperation scenarios decribed above does the potential of combining knowledge become obvious. But the successes of learning provers (see Section 4) suggest that harder problems will come within the reach of provers. In this tase the combination problem will become much more important, and we will certainly see additional solution concepts for it.

3.9 Which Concepts of Similarity are Helpful?

Similarity is a central notion for most forms of learning. Many approaches use similarity implicitly, melted into the processes of learning and application. It is e.g. well-known that standard distance measures applied to the representations generated in hidden layers of multi-layer perceptrons and folding architecture networks reflect a task-specific similarity measure for the input (see also Section 3.3.3). Analogy-driven methods use similarity criteria more explicitly. These criteria have to answer the pivotal questions of analogical reasoning, namely *which* piece of knowledge or *which* analogy relation to use (see Section 3.5) and *how* to use it (see Section 3.6). Therefore, similarity criteria for proof problems and for states of theorem provers are needed.

Similarity criteria are commonly formalized by so-called *similarity measures*. A similarity measure essentially expresses a certain aspect of similarity with a numeric (or Boolean) value. The area of application determines which aspects are useful. As outlined above, similarity measures can be utilized both for detecting applicable knowledge and for applying knowledge. In the following paragraphs, we shall not distinguish between these two uses because the concepts to be presented are so general that they may be employed for both uses. *Distance measures* are complementary to similarity measures. As both kinds of measures can be transformed into each other easily, the choice of which of the two is more natural to use is usually determined by the details of the application.

Basically, two classes of similarity measures are used.

Direct: Similarity measures which operate directly on the syntactic structures that are manipulated by the prover at hand.

Transformational: Similarity measures which first transform the symbolic structures into a different representation (e.g. feature vectors) so that standard distance measures can be used.

Most similarity measures that operate on syntactic structures heavily depend on subsumption as main ingredient. In the simplest case, subsumption based on first-order matching is used. In order to become independent of specific names for symbols, restricted arity-preserving second-order matching, i.e. a renaming of symbols (also called *signature matching*) is often used in addition to first-order matching. Flexible re-enactment ([Fu96], [Fu97b], and [Fu97c]) utilizes this kind of subsumption in order to identify facts which can be re-enacted. However, even in this simple case we already can run into severe complexity problems. This is due to the potentially huge number of renaming possibilities, among which the most suitable one must be found. Finally, full second-order matching, which not only allows plain renaming, but also changing the arity of symbols and thus the syntactic structure, can be used.

A similarity measure which works on syntactic structures can also be used to work on sets of structures. Determining the similarity of sets of syntactic structures is pivotal in order to assess the similarity of proof problems (which are characterized by a set of axioms and a goal). In particular, similarity measures based on subsumption can easily be extended to work with sets of syntactic structures. Given two sets A and B of axioms so that all axioms in A are subsumed by axioms in B, it is clear that every goal which can be derived from A can also be derived from B. Hence a test based on this principle allows for assessing similarity in terms of capability to prove the same or even more goals. A similar test can be applied to goals: if a goal can be derived from given axioms, then also all goals which are subsumed by it can be derived from these axioms. Extensions of similarity measures based on these ideas can be found in [FF97b], [FF97c], and [De+97a]. Proof reuse as described in [KW94] employs second-order matching to sets of axioms and goals in order to find suitable "proof shells".

[DK96] introduces a "domain detection specialist". Its task basically consists in checking whether all logical consequences of a set A of equational axioms are also logical consequences of a set B of (equational) axioms. This is the case if the axioms in A are logical consequences of B. Such a test is in general undecidable. Simply testing $A \subseteq B$ or testing whether for every axiom in A there is an axiom in B subsuming it is not satisfactory due to possible contractions (see Section 2) made with other axioms in B. After the use of a signature match, [DK96] solves this dilemma by computing normal forms of all axioms in A w.r.t. the axioms in B and then checking for syntactic identity or subsumption by an axiom in B. Thus the "ideal" test for logical consequences can be approximated to a certain extent with a rather efficient test.

The second kind of similarity measure centers on a *change of representation* to determine similarity. For this kind of similarity measure, proof structures are represented by feature vectors. Remember that a feature (see Page 19) encodes a certain syntactic property with a numeric value. For instance, the number of function symbols occurring in a syntactic structure is one possible feature of a structure. The number of axioms also is a possible feature of a proof problem. The feature values for a syntactic structure are represented as a feature vector. A distance measure on feature vectors — usually the Euclidean distance or a variant of it — is then employed to determine similarity after the representational change (see Section 3.6). Less distant structures or proof problems ("nearest neighbors") are considered more similar. In [Fu97a] such an approach is used to find similar proof problems.

As already stated in Section 3.3, some of the information contained in the original syntactic structures or proof problems is lost when representing them with feature vectors. But this can be an advantage: The representational change allows to concentrate on those parts which are relevant for assessing similarity. New features can be added if not all relevant parts are covered, or features can be discarded if they represent irrelevant information. Moreover, the use of feature vectors and distance measures opens the door to a well-studied and fruitful area of research, namely instance-based learning (see [Ah+91], also [Ko92]).

[SB99] introduces a different transformation-based similarity measure on sets of facts (more exactly, on unit-equational proof problem specifications, represented as sets of equations). In this approach, each set of terms or equations is compiled into a recursive *term space map* (see Section 3.3.2), and the resulting trees are compared for structural similarity.

[De+97a] describes an approach based on TEAMWORK (see Section 3.7) combining both feature representation and matching. It introduces a specialist named PES (Proof Experience Specialist) which is responsible for detecting applicable knowledge. In addition to the usual signature matching, PES utilizes a variety of similarity measures including equality modulo some theory (usually associativity and commutativity) and homeomorphic embedding (which generalizes subsumption). All these individual or elementary similarity measures (both for individual syntactic structures and sets of structures) are combined resulting in one value that can be used to assess the quality of similarity of sources to a given target (see also Section 3.5).

So, the known approaches to similarity in theorem proving are centered around the use of similarity within the solutions to the other eight fundamental problems of learning provers. Nevertheless, research towards similarity measures usable in theorem proving is interesting even without a connection to learning provers. For example suitable similarity measures can be used in goal-directed search strategies (see [DF94]).

4 Evaluation and Discussion

The goal of including learning capabilities into a system is *"the improvement of performance in some environment through acquisition of knowledge resulting from experience in that environment"* (according to [La96]). In automated theorem proving the quality of performance of a prover can be measured rather differently. For example, for fully automated provers the run-time is usually used as measure while for interactive provers often the number of user interactions necessary to find a proof is used to indicate the quality of performance of the prover. But the number of parameters that have to be set by the user (and the number of possible values for these parameters) is an important criterion of performance as well. Additionally, a totally different measure of performance can often be observed when asking mathematicians, namely understandability and clarity of the generated proofs, with additional criteria like proof length or quality of the lemmata generated.

So, although the amount of time (either of a single proof run or spend by the user in a session with the theorem prover) it takes to solve a given proof problem is in most cases the main criterion when determining the performance of a prover, there are other criteria that may be targeted by learning theorem provers. Since the provers also may use different proof paradigms, different logics, and different calculi, it is nearly impossible to compare the various solution combinations to our nine fundamental problems of Section 3 on a general level. Only those solution combinations that can be applied by a particular prover may be compared and that only for a particular set of proof problems. For this set of problems and this prover we may then be able (if the problems are not too different) to determine the solution combination that achieves the most improvement in performance (according to a concrete definition of what performance is). However, due to the usually very different nature of other provers, and due to the fact that at most very few learning approaches have been integrated into any one prover, a direct comparison of the impact of different learning approaches is hardly possible, and the results are not usually interesting.

In the following, we will first report on some successes achieved by learning theorem provers in order to show some of the potential that including learning techniques into provers has. After that we will elaborate a little more on criteria for evaluating learning theorem provers and present a spectrum for classifying the uses of the known learning provers. Finally, despite the difficulty of comparing different solution combinations of the fundamental problems, we will give some recommendations for building learning theorem provers.

If we take a look at the performance of existing learning provers reported in literature, we find that most of them have demonstrated the high potential of learning in automated deduction. If we look at interactive provers, then ABALONE (see [MW97]) was able to prove several examples fully automatically that its "parent" prover CLAM could only prove with the help of the user. [KW94] also reports an increase of automatically proven examples by PLAGIATOR versus the version of their prover without PLAGIATOR.

For fully automated provers based on generating calculi, faster run times and an increase in the number of solved problems could be achieved. In [DS96b], 10 percent more examples of a domain could be proven in half the time than with the best non-learning strategy. While the improvement in the times used in the proof search is certainly due to the simple reproduction of input proofs, previously unprovable new examples could be proven as well. In [De+97a], flexible re-enactment and PES were used in a bootstrapping process that enabled the system to use easy examples to solve harder problems, which in turn were used to solve even harder ones, and so on. As a result, in one domain 16 percent more problems could be solved, in the other 21 percent more (compared to OTTER in auto mode, which already performs slightly better than the best non-learning strategy of DISCOUNT). It should be noted that the approaches of [DS96b] and [De+97a] resulted in some different examples that could be solved with the different learning or application concepts.

In [Go97], heuristic evaluation functions for solving word problems in group theory were learned for the theorem prover SETHEO. Of course word problems in group theory are trivial for systems using completion techniques. However, SETHEO is a prover for pure first-order logic, without special treatment for the equality relation. A standard axiomatization of group theory with axiomatic equality was used for the experiment. It is known that under these conditions word problems in group theory can be extremely hard to prove. The experimental results obtained are very good. The system was trained on simple problems that could be solved by the conventional prover, yielding an heuristic evaluation function for proof states. Controlled by this heuristic, the system was able to solve nearly all of the tested new problems (all of which had been beyond reach of the conventional SETHEO).

As stated at the beginning of this section, the goal when building a learning prover (or including learning into an existing one) is an improvement in performance when compared to not-learning variants of the prover. Depending on the definition of performance, one also has to define criteria for evaluating success. In the area of machine learning such evaluations traditionally consist of two kinds of experiments, namely experiments with the examples one has learned from (measuring the *recall* capability of the particular learning concept) and experiments with new examples (measuring the *transfer* capabilities of the concept). Naturally, all the examples belong to the class of problems one is interested in (or one wants to improve upon).

Although in some applications of automated deduction it may be already useful to improve on the performance of a prover with respect to already solved proof problems (as in the CASC scenario, see later), in most cases the interest is focused on the transfer capabilities of the learning concept. If one is, for example, interested in the performance of fully automated provers with respect to run-time, then improvements with respect to already solved proof problems can be best achieved by simply memorizing the proofs of these problems and applying an efficient method for detecting applicable source proofs (see also Section 3.5). In fact, all the learning approaches described in this paper lead to an improvement with respect to their training problems. Unfortunately, it is much more difficult to achieve improvements with respect to new, so far unsolved proof problems.

It should be noted that it is not sufficient to demonstrate success of learning on a single new test problem. Improvement on a single test problem may e.g. indicate that very specific knowledge has been learned that may even decrease performance of the system with respect to other problems of the respective class considerably (this is known as *over-fitting*). Unfortunately, in literature it is typical for systems using variants of memorization (see Sections 3.3.1 and 3.4.1) that knowledge bases consisting only of very few source problems are generated and only very few test problems are used for evaluation. Consequently, it is often impossible to get an impression of the costs implied by search in big knowledge bases and by the redundancy caused by having many potentially applicable source problems (see also Section 3.6). We claim that the performance of learning theorem provers has to be tested w.r.t. sets of problems which are as "representative" as possible for the respective problem class.

An example of the problems, like over-fitting, occurring when defining the performance only on the training examples surfaced in the 1998 CADE ATP System competition (CASC-15, `http://www.cs.jcu.edu.au/~tptp/CASC-15/`). We know from personal communications with several of the participants of CASC-15 that they used the set of so called "eligible" problems for tuning their systems [10]. Tuning was realized by either hand-crafted or automated methods which analyze proof problems and determine system parameters and search strategies. These tuning methods are very closely related to learning as described in [Fu97a]. Some of them even involve feature vectors for determining similarity of proof problems (see also Section 3.9). The actual competition and evaluation of the systems was done by selecting test problems from this previously determined set of eligible problems. So in the terminology of learning, the final evaluation actually was done on a subset of the training problems. In personal communications we have been told about strong over-fitting effects. The systems tuned w.r.t. the eligible problems often turned out to be weaker than the respective untuned base provers when they were evaluated on the whole TPTP problem library (see [Su+94]).

If we concentrate on run-time as the measure of performance then obviously one is interested in fast but not specialized (i.e. flexible) provers. But unfortunately, there is a certain trade-off between efficiency w.r.t a given problem class and generality. A prover that solves only one problem by immediately returning a memorized proof is extremely efficient for this problem. It is, however, not useful for other problems. On the other hand, a prover with a high degree of generality, cannot be as efficient as the hypothetical one-problem-only prover since it is in general impossible to memorize proofs for an infinite number of problems. Note that specialization is not "bad" per se. If only problems in one certain area of interest are to be solved then specialization on this specific class of problems is sufficient for a user.

We can place the known learning provers into a spectrum ranging from very specialized to more flexible provers. Naturally, on one end of this spectrum we have the hypothetical one-problem-only prover that has memorized one problem and the best solution to it. The other end is a general purpose prover using its standard (usually non-learning)

[10]This is not against the competition rules and shows again, that learning by developers and users of theorem provers is already very important. Note however that it is planned not to make the set of eligible problems known in future competitions.

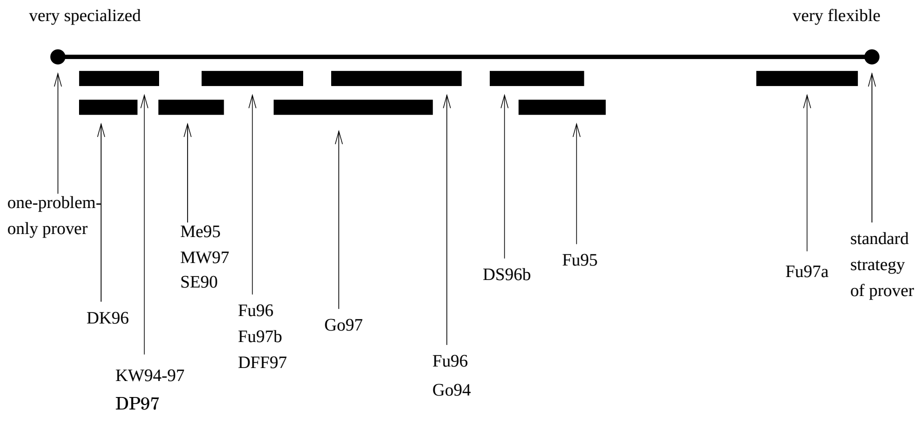

Figure 1: Spectrum of known learning provers

control strategy that is intended to have the least (in most cases no) specialization and some basic efficiency. Note that the spectrum corresponds to the degree of generality of learned knowledge introduced in Section 3.2. The part between the two extremes represents provers which have specialized on specific domains or problem classes.

In Figure 1 we (qualitatively) depict this spectrum for the known learning provers. The positions of individual approaches are determined by the way in which the approaches are applied in the respective papers. This means that we actually show the positions of the respective provers after learning. We do not depict and we do not analyze in detail the potential of individual learning provers for more or less specialization. However, we would like to give the user some ideas for such considerations. The learning method (see Section 3.4) and the representation scheme (see Section 3.3) are the core of a learning prover. They can severely limit the degree of specialization that is possible. If e.g. the representation scheme does not allow to uniquely represent training examples, exact memorization of proofs is impossible and only more general knowledge can be learned. On the other hand, if a representation scheme with a high representational power is used, the learning prover can in principle cover the whole spectrum. The position for a trained prover then only depends on the selection of training problems. If they are selected from a very specific problem class, the prover will be very specialized while it will be quite general if they are selected from a wider class. Some of the methods for detecting applicable knowledge (see Section 3.5), for handling misleading knowledge (see Section 3.7), and for combining different sources of (maybe very special) knowledge (see Section 3.8) are so general that they can be used for all learning approaches in order to increase the range of specialization. Additional organization of learned knowledge or the cooperation/competition approaches for example allow a prover to have less specialization (for the whole system, not for individual pieces of knowledge or for components of the prover) while still being quite effective as demonstrated in [De+97a].

After this rather abstract classification we will look at the different proof philosophies and give recommendations which learning approaches can be used for provers employing these philosophies.

In generating provers, the use of similarity measures (see Section 3.9) for the identification of analogical situations on various levels of abstraction (specifications, goals, individual facts) both for the selection of suitable training (source) problems and in derivational analogy (see Section 3.5) is particularly promising. In general, all techniques which compute (even approximate) evaluations for generated facts or potential inferences can be used. On the other hand, transformational analogy (see Section 3.5) is rather difficult to integrate, as the cost of the test for the applicability of a given piece of knowledge ranges from very expensive to undecidable.

For analytical provers, learning on the calculus level (see Section 3.2) such as lemma mechanisms or macro- and meta-inference rules and proof schemata applied via transformational analogy are very suitable, since they introduce a bottom-up element into the top-down search. Thus the effective depth of proofs can be reduced and longer, more complex proofs can be found. As this is achieved at the cost of increasing the branching factor during the search, careful supervision is necessary to find the optimal balance. Since backward chaining normally means a goal-directed search, there are also more possibilities for explicitly applying planning methods than in case of generating calculi. Thus derivational analogy is also very suitable for analytical provers. Since analytical provers work by enumerating proof attempts instead of enumerating consequences of axioms, the search state consists of a proof structure and evaluation functions for whole proof structures or parts of them can be learned. Such evaluation functions have a more global view than evaluation functions for facts in generating calculi. It is for example possible to specifically evaluate combinations of proof steps. Numeric and hybrid learning algorithms (see Section 3.4) seem particularly suitable for learning such evaluation functions.

For interactive provers, the goal of learning is in most cases to reduce the interaction with the user, as already stated. Typical application fields such as software verification produce several very similar proof problems. As an example, consider the case that a verified program is slightly changed. Most of the proof obligations remain the same and are likely to have the same or very similar proofs. The re-enactment of user interactions by derivational analogy provides a good basis for learning in interactive provers. As in the case of other provers, similarity criteria have to reflect the inference rules used, or better those inferences that are controlled by the user.

Interactive provers that employ proof planning offer an excellent environment for the combination of different kinds of learned knowledge and for tactics that employ derivational analogy. The detection of applicable knowledge and of misleading knowledge can be easily integrated into such an environment. The proof trees that are used by such planning provers can offer additional information which on the other side allows for more possibilities for abstraction.

Finally, we briefly discuss preconditions provers have to fulfill in order to make learning applicable. Of course, an approach as described in [Fu97a] (selection of one of the prover's standard strategies based on learning) can be applied to all provers. For a lot of learning approaches, however, proofs are the basic form of experience. For applying them the respective prover has to be able to produce proofs. Simply producing "success" or "failure" as output is not sufficient. Consider further that for a lot of learning approaches negative experience is needed. Producing negative experience is sometimes tricky, especially if provers throw away parts of their results during the search. So far we discussed aspects which are important for learning. There are also aspects which are important when it comes to the application of learned knowledge. For derivational analogy and heuristic evaluation functions it is e.g. necessary that the prover offers easy access to proof states during search.

While learning theorem provers may conceal weaknesses in the calculus, control, and implementation of a prover, it must be mentioned that a strong base prover is always preferable. Learning can only improve on the abilities of a prover, it cannot generate abilities out of thin air. The more proof problems a prover can solve without learning, the higher is the probability that one (or several) of these problems are appropriate to help solving a given new proof problem. Moreover, longer, more difficult proof problems are, in our experience, more likely to reveal important concepts than trivial ones.

5 Future Developments

Learning theorem provers offer substantial improvements compared to today's state-of-the-art conventional provers, with respect to several measures of performance. In particular, improvements have been reported with respect to the time needed to prove a given problem and the grade of automation, i.e. the amount of user interaction and the number of attempts necessary until a problem is solved. For application areas with problems in various degrees of difficulty, learning can allow a prover to automatically adapt, resulting in more difficult problems solved, as demonstrated in [De+97a] or [Go97].

In order to enhance a prover by adding learning capabilities, a number of problems (given by our nine fundamental questions) have to be solved, some of which are obviously interconnected. Although for each type of prover there is already at least one first solution to each of these problems, much more work has to be done to find additional (maybe better) solutions and to determine which combinations of solutions offer the highest gains for a certain type of prover and certain types and classes of proof problems.

While the basic techniques for enhancing conventional control strategies by learned knowledge are quite well known, similarity measures for different calculi used for selecting the right pieces of knowledge require both new ideas and more theoretical work. The levels and aspects on which proof problems are similar cannot only help a certain prover, but may also help to suggest what type of prover with what proof philosophy should be chosen for a given proof attempt in the first place. So similarity measures may also offer insight into the classes of problems a particular prover or a particular calculus can solve (in practice).

Most of the presented learning provers are rather experimental. Therefore it is an important goal to include the necessary concepts into production-quality provers and then test them on real applications, as for example verification or mathematics. Since the (basic) training of learning provers often can be separated from the application of learned knowledge, the performance of learning provers adapted to an application may be able to reach the level that is required by the users for many of these applications.

Finally, with the development of more concepts and solutions, appropriate criteria and measures are needed to compare learning theorem provers (see also Section 4). The TPTP library [Su+94] already includes a few domains that provide a wide range of examples suitable for learning, however, more such domains are needed. In particular, there is no standard benchmark set for provers using other logics than first order (clausal) logic or unit equality logic. As stated above, the success of learning provers cannot be adequately measured on single examples, but only on whole domains. Important criteria for measuring the success of a learning approach then have to be coverage of such a domain (how many of the examples can be proven), flexibility (how many different domains are well covered), degrees of automation and other criteria in addition to the criteria already used, for example, in CADE's theorem prover competition (see [SS97a] and [SS97b]).

References

[Ah+91] Aha, D.W.; Kibler, D.; Albert, M.K. (1991). Instance-Based Learning Algorithms. *Machine Learning* **6**, 37–66.

[Ba88] Bachmair, L. (1988). Proof by Consistency in Equational Theories. Proc. 3rd LICS, LNAI, pp. 228–233.

[Ba+89] Bachmair, L.; Derschowitz, N.; Plaisted, D.A. (1989). Completion Without Failure. Coll. on the Resolution of Equations in Algebraic=20 Structures, Austin, 1987.

[BG94] Bachmair, L.; Ganzinger, H. (1994). Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation* **4(3)**, 217–247.

[Br+88] Brock, B.; Cooper, S.; Pierce, W. (1988). Analogical Reasoning and Proof Discovery. Proc. CADE-9, Argonne, LNCS 310, pp. 454–468.

[Bu88] Bundy, A. (1988). The use of explicit plans to guide inductive proofs. Proc. CADE-9, Argonne, LNCS 310, pp. 111–120.

[Ca86] Carbonell, J. G. (1986). Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. Machine Learning: An Artificial Intelligence Approach, Morgan Kaufmann.

[CL73] Chang, C.; Lee, R.C. (1973). Symbolic Logic and Mechanical Theorem Proving. Academic Press.

[CV88] Carbonell, J. G.; Veloso, M. (1988). Integrating Derivational Analogy into a General Problem Solving Architecture. Proc. of the 1988 DARPA Workshop on Case-Based Reasoning, Clearwater Beach.

[Cy89] Cybenko, G. (1989). Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems* **2**, 303-314.

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

[De90] Dershowitz, N. (1990). A maximal-Literal Unit Strategy for Horn Clauses. Proc. 2nd CTRS, Montreal, LNCS 516, pp. 14–25.

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

[De+97a] Denzinger, J.; Fuchs, M.; Fuchs, Marc (1997a). High Performance ATP Systems by Combining Several AI Methods. Proc. IJCAI-97, Nagoya, Morgan Kaufmann, pp. 102–107.

[De+97b] Denzinger, J.; Kronenburg, M.; Schulz, S. (1997b). DISCOUNT. A Distributed and Learning Equational Prover. *Journal of Automated Reasoning* **18(2)**, 189–198.

[DF94] Denzinger, J.; Fuchs, M. (1994). Goal oriented equational theorem proving using team work. Proc. KI-94, Saarbrücken, LNAI 861, pp. 343–354.

[DF96] Denzinger, J.; Fuchs, D. (1996). Referees for Teamwork. Proc. FLAIRS '96, Key West, ISBN 0-9620-1738-8, pp. 454–458.

[DF99] Denzinger, J.; Fuchs, D. (1999). Cooperation of Heterogeneous Provers. Proc. IJCAI-99, Stockholm, Morgan Kaufmann, to appear.

[DK96] Denzinger, J.; Kronenburg, M. (1996). Planning for Distributed Theorem Proving: The Teamwork Approach. Proc. KI-96, Dresden, LNAI 1137, pp. 43–56.

[DM86] DeJong, G.; Mooney, R. (1986). Explanation-Based Learning: An Alternative View. *Machine Learning* **1**, 145–176.

[DP97] Defourneaux, G.; Peltier, N. (1997). Analogy and Abduction in Automated Deduction. Proc. IJCAI-97, Nagoya, Morgan Kaufmann, pp. 216–221.

[Dr98a] Draeger, J. (1998). Acquisition of Useful Lemma Knowledge in Automated Reasoning. Proc. AIMSA-98, LNCS 1480, pp. 230–239.

[Dr98b] Draeger, J. (1998). Modularisierte Suche in Theorembeweisern Ph.D. Thesis, Technische Universität München, Fakultät für Informatik.

[DS96a] Denzinger, J.; Schulz, S. (1996a). Recording and Analyzing Knowledge-Based Distributed Deduction Processes. *Journal of Symbolic Computation* **21**, 523–541.

[DS96b] Denzinger, J.; Schulz, S. (1996b). Learning Domain Knowledge to Improve Theorem Proving. Proc. CADE-13, New Brunswick, LNAI 1104, pp. 62–76.

[DW96] Dahn, B.I.; Wolf, A. (1996). Natural Language Presentation and Combination of Automatically Generated Proofs. Proc. FroCoS'96, München, pp. 175–192.

[Et93] Etzioni, O. (1993). A Structural Theory of Explanation-Based Learning. *Artificial Intelligence* **60(1)**, 93–139.

[FF97a] Fuchs, D.; Fuchs, Marc (1997a). Self-Modifying Theorem Provers. Proc. FLAIRS-97, Daytona Beach, ISBN 0-9620-1739-6, pp. 176–180.

[FF97b] Fuchs, M.; Fuchs, Marc (1997b). Case-Based Reasoning for Automated Deduction. Proc. FLAIRS-97, Daytona Beach, ISBN 0-9620-1739-6, pp. 6–10.

[FF97c] Fuchs, M.; Fuchs, Marc (1997c). Applying Case-based Reasoning to Automated Deduction. Proc. 2nd International Conference on Case-based Reasoning (ICCBR-97), Providence, LNAI.

[Fu95] Fuchs, M. (1995). Learning Proof Heuristics by Adapting Parameters. Proc. 12th Machine Learning, San Francisco, Morgan Kaufmann, pp. 235–243.

[Fu96] Fuchs, M. (1996). Experiments in the Heuristic Use of Past Proof Experience. Proc. CADE-13, New Brunswick, LNAI 1104, pp. 523–537.

[Fu97a] Fuchs, M. (1997a). Automatic Selection of Search-Guiding Heuristics. Proc. FLAIRS-97, Daytona Beach, ISBN 0-9620-1739-6, pp. 1–5.

[Fu97b] Fuchs, M. (1997b). Flexible Re-Enactment of Proofs. Proc. EPIA-97, Coimbra, LNAI 1323, pp. 13–24.

[Fu97c] Fuchs, M. (1997c). Learning Search Heuristics for Automated Deduction. Ph.D. Thesis, University of Kaiserslautern, Verlag Dr. Kovač, ISBN 3-86064-623-0.

[Fu97d] Fuchs, Marc (1997). Similarity-Based Lemma Generation for Model Elimination. Proc. FTP-97 Workshop, RISC-Linz Report Series No. 97-50, pp. 63-67.

[GK96] Goller, C.; Küchler, A. (1996). Learning Task-dependent Distributed Representations by Backpropagation Through Structure. Proc. ICNN-96, vol. 1, IEEE, pp. 347–352.

[GN97] Gerberding, S.; Noltemeier, A. (1997). Incremental Proof Planning by Meta-Rules. Proc. FLAIRS-97, Daytona Beach, ISBN 0-9620-1739-6, pp. 171–175.

[Go69] Goldberg, D.E. (1989). Genetic Algorithms in Search, Optimization & Machine Learning. Addison Wesley.

[Go94] Goller, C. (1994). A Connectionist Control Component for the Theorem Prover SETHEO. Proc. of the ECAI'94 Workshop: Combining Symbolic and Connectionist Processing.

[Go97] Goller, C. (1997). A Connectionist Approach for Learning Search-Control Heuristics for Automated Deduction Systems. Ph.D. Thesis, Technische Universität München, Fakultät für Informatik.

[Ha96] Hammer, B. (1996). Universal Approximation of Mappings on Structured Objects using the Folding Architecture. Osnabrücker Schriften zur Mathematik, Reihe P, Heft 183, Fachbereich Mathematik/Informatik, Universität Osnabrück.

[Hi+97] Hillenbrand, T.; Buch, A.; Vogt, R.; Löchner, B. (1997). WALDMEISTER. High Performance Equational Deduction. *Journal of Automated Reasoning* **18(2)**, pp. 265–270.

[Hi+99] Hillenbrand, T.; Jaeger, A.; Löchner, B. (1999). System Abstract: Waldmeister – Improvements in Performance and Ease of Use. *Proc. of the 16th CADE, Trento*, LNAI 1632, pp. 232–236.

[Ho+89] Hornik, K. and Stinchcombe, M. and White, H. (1989). Multilayer Feedforward Networks are Universal Approximators. *Neural Networks* **2**, pp. 359–366.

[HS97] Hammer, B.; Sperschneider, V. (1997). Neural Networks can approximate Mappings on Structured Objects. Proc. 2nd International Conference on Computational Intelligence and Neuroscience, VOL. 2, pp. 211-214

[Hu96] Hutter, D. (1996). Using Rippling for Equational Reasoning. Proc. KI-96, Dresden, LNAI 1137, pp. 121–134.

[Hu+94] Huang, X.; Kerber, M.; Richts, J.; Sehn, A. (1994). Planning mathematical proofs with methods. *Journal of Information Processing and Cybernetics*, **30(5-6)**, pp. 277–291.

[KG96] Küchler, A.; Goller, C. (1996). Inductive Learning in Symbolic Domains Using Structure-Driven Recurrent Neural Networks. Proc. KI-96, Dresden, LNAI 1137, pp. 183-198.

[Kl71] Kling, R.E. (1971). A paradigm for reasoning by analogy. *Artificial Intelligence* **2** 147–178.

[Ko92] Kolodner, J.L. (1992). An Introduction to Case-Based Reasoning. *Artificial Intelligence Review* **6**, 3–34.

[Kü98] Küchler, A. (1998). On the Correspondence between Neural Folding Architectures and Tree Automata. Technical Report, Computer Science, University of Ulm.

[KV96] Kapur, D.; Vandevoorde, M.T. (1996). Distributed Larch Prover (DLP): An Experiment in Parallelizing a Rewrite-Rule Based Prover. Proc. RTA-96, New Brunswick, LNCS 1103, pp. 420–423.

[KW94] Kolbe, T.; Walther, C. (1994). Reusing Proofs. Proc. ECAI '94, Amsterdam, pp. 80–84.

[KW95] Kolbe, T.; Walther, C. (1995). Proof Management and Retrieval. Proc. IJCAI '95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs, pp. 16-20.

[KW96] Kolbe, T.; Walther, C. (1996). Termination of Theorem Proving by Reuse. Proc. CADE-13, New Brunswick, LNAI 1104, pp. 106–120.

[La96] Langley, P. (1996). Elements of Machine Learning. Morgan Kaufmann.

[Le+92] Letz, R.; Schumann, J.; Bayerl, S.; Bibel, W. (1992). SETHEO: A High-Performance Theorem Prover. *Journal of Automated Reasoning* **1(8)**, 183–212.

[Ll69] Lloyd, J.W. (1984). Foundations of Logic Programming. Symbolic Computation, Springer.

[Lo69] Loveland, D.W. (1969). A Simplified Format for the Model Elimination Procedure. *JACM* **16(3)**, 233–248.

[Mc94] McCune, W.W. (1994). OTTER 3.0 Reference manual and Guide. Tech. rep. ANL-94/6, Argonne National Laboratory.

[Mc97] McCune, W.W. (1997). Solution of the Robbins Problem. *Journal of Automated Reasoning* **19(3)**, 263-276.

[Me95] Melis, E. (1995). A model of analogy-driven proof-plan construction. Proc. 14th IJCAI, Montreal, pp. 182–189.

[Mi90] Minton, S. (1990). Quantitative Results Concerning the Utility of Explanation-Based Learning. *Artificial Intelligence* **42**, 363–391.

[Mi+86] Mitchell, T. M.; Keller, R. M.; Kedar-Cabellis, T. (1986). Explanation-Based Generalization: A Unifying View. *Machine Learning* **1**, 47–80.

[MW97] Melis, E.; Whittle, J. (1997). Analogy as a Control Strategy in Theorem Proving. Proc. FLAIRS-97, Daytona Beach, ISBN 0-9620-1739-6, pp. 367–371.

[MR94] Muggleton, S.; Raedt, L. (1994). Inductive Logic Programming: Theory and Methods. *Journal of Logic Programming* **19,20**, 629-679.

[PN90] Paulson, L.C.; Nipkow, T. (1990). Isabelle tutorial and user's manual. Tech. Report 189, University of Cambridge Computer Laboratory.

[Re+97] Reif, W.; Schellhorn, G.; Stenzel, K. (1997). Proving System Correctness with KIV 3.0. Proc. CADE-14, Townsville, LNAI 1249, pp. 69–72.

[Ro65] Robinson, J.A. (1965). A Machine Oriented Logic based on the Resolution Principle. *JACM* **12(1)**, 23–41.

[Ru+86] Rumelhart, D. E.; McClelland, J. L.; Hinton, G. E. (1986). Parallel Distributed Processing: Explorations in the Microstructure of Cognition. MIT Press, Cambridge/London.

[Sc97] Schmitt, T. (1997). Evaluation of the Neural Folding Architecture for Inductive Learning Tasks concerning Logical Terms and Chemical Structures. Masters-thesis, Technical University of Munich, Computer Science.

[SB99] Schulz, S.; Brandt, F. (1999). Using Term Space Maps to Capture Search Control Knowledge in Equational Theorem Proving. Proc. FLAIRS-99, Orlando, AAAI Press, pp. 244–248.

[Sc+97] Schulz, S.; Küchler, A.; Goller, C. (1997). Some Experiments on the Applicability of Folding Architecture Networks to Guide Theorem Proving. Proc. FLAIRS-97, Daytona Beach, ISBN 0-9620-1739-6, pp. 377–381.

[SD93] Sonntag, I.; Denzinger, J. (1993). Extending automated theorem proving by planning. SEKI-Report SR-93-02, University of Kaiserslautern.

[SE90] Suttner, C.; Ertel, W. (1990). Automatic Acquisition of Search Guiding Heuristics. Proc. CADE-10, Kaiserslautern, LNAI 449, pp. 470–484.

[SF71] Slagle, J.R.; Farrell, C.D. (1971). Experiments in automatic learning for a multipurpose heuristic program. *Communications of the ACM* **14(2)**, 91–99.

[SG98] Schmitt, T.; Goller, C. (1998). Relating Chemical Structure to Activity with the Structure Processing Neural Folding Architecture. Proc. EANN 98, Gibraltar, June, 1998.

[Si86] Silver, B. (1986). Meta-Level Inference. Studies in Computer Science and Artificial Intelligence, Elsevier Science Publishers B.V.

[Sm68] Smullyan, R.M. (1968). First Order Logic. Springer.

[SS97a] Sutcliffe, G.; Suttner, C.B. (1997a). Special Issue: The CADE-13 ATP System Competition. *JAR* **18(2)**.

[SS97b] Sutcliffe, G.; Suttner, C.B. (1997b). The CADE-14 ATP System Competition. Technical Report 98/01, Department of Computer Science, James Cook University.

[Su+94] Sutcliffe, G.; Suttner, C.B.; Yemenis, T. (1994). The TPTP Problem Library. Proc. CADE-12, Nancy, LNAI 814, pp. 252–266.

[Va95] Vapnik, V.N. (1995). The Nature of Statistical Learning Theory. Springer, New York.

[Ve96] Veroff, R. (1996). Using Hints to Increase the Effectiveness of an Automated Reasoning Program: Case Studies. *JAR* **16:**, 223–239.

[We74] Werbos, P. J. (1974). Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science. PhD-thesis, Harvard University.

[Wo96] Wos, L. (1996). The Automation of Reasoning. Academic Press.