Towards Full Automation of Deduction: A Case Study*

Matthias Fuchs
Centre for Learning Systems and Applications (LSA)
Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern
Germany
E-mail: `fuchs@informatik.uni-kl.de`

October 4, 1996

Abstract

We present first steps towards fully automated deduction that merely requires the user to submit proof problems and pick up results. Essentially, this necessitates the automation of the crucial step in the use of a deduction system, namely choosing and configuring an appropriate search-guiding heuristic. Furthermore, we motivate why learning capabilities are pivotal for satisfactory performance. The infrastructure for automating both the selection of a heuristic and integration of learning are provided in form of an environment embedding the "core" deduction system.

We have conducted a case study in connection with a deduction system based on condensed detachment. Our experiments with a fully automated deduction system 'AutoCoDe' have produced remarkable results. We substantiate AutoCoDe's encouraging achievements with a comparison with the renowned theorem prover Otter. AutoCoDe outperforms Otter even when assuming very favorable conditions for Otter.

*This work was supported by the *Deutsche Forschungsgemeinschaft (DFG)*.

1 Introduction

Automated deduction is—at its lowest level—a search problem that spans huge search spaces. The general undecidability of problems connected with (automated) deduction entails an indeterminism that has to and can only be tackled with heuristics. Besides the availability of powerful search-guiding heuristics, also knowledge about when and how to apply a certain heuristic must be accessible. This knowledge is mostly provided by the user.

The ultimate goal of anyone designing automated deduction systems is a system which is *fully* automated in the sense that a user merely needs to specify the proof problems and pick up results, and everything else is taken care of. In particular the crucial choice of the search-guiding heuristic *and* its parameter settings should in the end not be left to the user, because a judicious choice on that score largely depends on intensive experience with the respective system.

In other words, a "user-friendly" deduction system is embedded in an environment which takes over all the steps after being given the problem(s). These steps are only a burden to a user who is not interested in the detailed mechanisms of automated deduction, but merely in its results (i.e., proofs).

Several works regarding learning in connection with automated deduction have demonstrated that—quite expectedly in view of the pivotal role learning plays in human problem solving—substantial improvements can be attained through learning (e.g., [13], [2], [15], [4], [3], [7]). Thus, learning—or exploiting past proof experience—must be considered as an essential part of a powerful deduction system. Therefore, it stands to reason to integrate learning capabilities into the environment, which of course necessitates components and mechanisms for handling proof experience, and thus complicates matters. But we shall see that even under these harder conditions attempts to achieve fully automated and powerful deduction systems are not necessarily doomed to failure.

Most existing systems derive their powerfulness from sophisticated search-guiding heuristics that do not (explicitly) exploit past experience. However, these heuristics and the knowledge when and how to employ them go back to the learning capabilities and experience of their designers. Hence, these systems lack an essential part of automation that puts off potential, but unexperienced users of deduction systems. The well-known theorem prover OTTER alleviates this problem with its 'autonomous mode' (cp. [10]) that attempts to pick an appropriate search-guiding heuristic after analyzing the current problem. But the "selection heuristic" is built-in and hence inflexible in the sense that it encodes the knowledge its designers had at some point in time.

We conducted a case study regarding full automation of deduction in the area of *condensed detachment* (CD), also known as "*substitution and detachment*" (cf. [17], [9]). The main reason for this choice is the fact that there is a large number of such problems within a wide spectrum of difficulty, almost continuously ranging from (nearly) trivial to (very) challenging. This constellation is important if we want to tackle problems with methods that involve previous proof experience. Furthermore, problems in the area of CD are widely acknowledged as prominent test sets for automated deduction systems and their search-guiding heuristics (cp. [18], [11]), and they have received considerable attention, in particular in connection with serious experimental evaluations of existing deduction systems (e.g., [12], [18], [11], [14], [19]).

[7] presented several methods for learning from past experience that were experimentally evaluated in connection with CD. These methods will also be employed here. We shall show that a central problem for utilizing past experience not addressed in [7]—namely which piece(s) of past experience to make use of in order to solve the current problem—could be coped with automatically and satisfactorily in this case study. We substantiate our results by comparing them with the results of the renowned OTTER that were published in [11].

This report essentially aims at demonstrating two things: First, (machine) learning techniques of some form are indispensable to obtain full automation and powerfulness. Second, the additional complications learning causes (in particular the question when and how to exploit past experience) can—at least in this case study—be overcome.

The report is organized as follows. First, sections 2 and 3 introduce the basics concerning CD, the deduction system 'CODE' and its search-guiding heuristics. Then, section 4 outlines the architectural principles of the environment allowing for fully automated deduction, before section 5 presents a concrete realization 'AUTOCODE' of the proposed environment. Experimental results are given in section 6. Finally, a discussion in section 7 concludes the report.

2 Condensed Detachment with CODE

In this section we present the study of logic calculi as a research area that can be tackled with automated deduction systems. (See [17] and [9] for motivation and a detailed theoretical background.) Furthermore, we also introduce such a system named 'CODE'.

The inference rule 'condensed detachment' (CD) is the central part of the different logic calculi we are going to investigate. This inference rule manipulates first-order terms which we shall also call *facts*. The set of terms (facts) $Term(\mathcal{F},\mathcal{V})$ is defined as usual, involving a finite set $\mathcal{F}$ of function symbols and an enumerable set $\mathcal{V}$ of variables.

CD (in its basic form) is defined for a distinguished binary function symbol $f \in \mathcal{F}$, allowing to deduce the fact $\sigma(t)$ from two given facts $f(s,t)$ and s', where σ is the most general unifier of s and s'. (CD can consequently be seen as a generalized version of the well-known modus ponens.) $f(s,t)$ and s' are the *immediate ancestors* of the *descendant* $\sigma(t)$. A proof problem $\mathcal{A} = (Ax, \lambda_G)$ consists in deducing a certain given fact λ_G (the *goal*) from an also given set Ax of facts (the *axioms*) by applying CD.

A very common principle to solve such proof problems algorithmically is employed by most deduction systems based on resolution or the Knuth-Bendix completion procedure. It also constitutes the core of CODE. Essentially, CODE maintains a set F^P of so-called *potential facts* from which it selects and removes one fact λ at a time. λ is put into the set F^A of *activated facts*, or discarded if it is subsumed by an already existing activated fact $\lambda' \in F^A$ (*forward subsumption*, denoted by $\lambda' \lhd \lambda$, which here means that there is a match σ so that $\sigma(\lambda') \equiv \lambda$). Activated facts are, unlike potential facts, allowed to produce new facts via CD, which then are put into F^P. At the beginning, $F^A = \emptyset$ and $F^P = Ax$. The indeterministic selection or *activation step* is realized by heuristic means. To this end, a selection heuristic $\mathcal{H}$ associates a natural number $\mathcal{H}(\lambda) \in \mathbb{N}$ with each $\lambda \in F^P$, which is referred to as "weighting λ with $\mathcal{H}(\lambda)$". Subsequently, that $\lambda \in F^P$ with the smallest weight $\mathcal{H}(\lambda)$ is selected. Ties are broken according to the FIFO-strategy.

The search-guiding heuristic $\mathcal{H}$ is crucial for the efficiency of the proof procedure just described. CODE is in so far a "standard" theorem prover in that it expects the user to select a search-guiding heuristic and to set its parameters, or otherwise uses a default heuristic and default settings regardless of the problem given. The following section concisely describes the heuristics available for CODE, while sections 4 and 5 explain how CODE can be embedded in an environment that takes this crucial and difficult task off the back of the user.

3 Search-guiding Heuristics for CODE

CODE has three heuristics ϖ, ϖ_F, and $\varpi_{FR\&F}$ at its disposal. The "basic" heuristic ϖ does not make use of past experience. It is indispensable in the beginning to establish a basis of proof experiences heuristics ϖ_F and $\varpi_{FR\&F}$ that do exploit past experience can build on. Furthermore, ϖ serves as a foundation for ϖ_F and $\varpi_{FR\&F}$ (see below).

ϖ computes the weight of a fact λ as the weighted sum $c_\delta \cdot \delta(\lambda) + c_w \cdot w(\lambda)$ of λ's *level* $\delta(\lambda)$ and *term weight* $w(\lambda)$. The coefficients c_δ and c_w—often given as a ratio $c_\delta : c_w$—are the parameters of ϖ. The term weight $w(\lambda)$ of λ is two times the number of function symbols plus the number of variables occurring in λ. The level $\delta(\lambda)$ of λ is 0 if λ is an axiom. Otherwise, $\delta(\lambda)$ is the maximum of the levels of its immediate ancestors plus 1. In particular taking into account the level makes ϖ considerably powerful. (See [5] or [6] for more details and experimental evaluation of ϖ.)

We already emphasized the importance of learning in section 1. Naturally, there are various ways of learning. We chose to design heuristics that exploit past proof experience given by a *source problem* solved previously in order to conduct the search for the proof of the *target problem* at hand more efficiently by activating less facts that do not contribute to the proof eventually found (cp. [7]). Two basic methods are available, namely a *feature-based* approach and *flexible re-enactment*. Both approaches depend on past experience (i.e., solutions of previously solved problems) being represented in form of a *search protocol* $\mathcal{S}$ that was produced when solving (source) problem $\mathcal{A} = (Ax, \lambda_G)$. $\mathcal{S}$ simply records the sequence $\lambda_1; \ldots; \lambda_n$ of facts that were activated when searching for a proof of λ_G using heuristic $\mathcal{H}$. We assume here that $\mathcal{S}$ is the protocol of a successful search yielding a proof, i.e., $\lambda_n \lhd \lambda_G$ (λ_n subsumes λ_G). By tracing back ancestor/descendant relations starting with λ_n, all *positive facts* P can be identified that actually contribute to deducing λ_n. All other facts occurring in $\mathcal{S}$ are referred to as *negative facts* N. (Note that being a negative or positive fact is not a global property, but must be seen in the context of the given particular search protocol $\mathcal{S}$.) Past proof experience $\mathcal{I}$ is represented by triples $\mathcal{I} = (\mathcal{A}, \mathcal{H}, \mathcal{S})$, from which all necessary information—mainly positive and negative facts—can be extracted.

We now briefly sketch the feature-based approach and flexible re-enactment as well as the heuristics ϖ_F and $\varpi_{FR\&F}$ that derive from these approaches.

Features are used to achieve a certain degree of abstraction by representing (mostly syntactic) properties of facts with, say, numbers from the set $\mathbb{Z}$ of integers (cp. [13], [15]). Given $k \geq 1$ features $f_1, \ldots, f_k$, our feature-based approach centers on the k sets $V_1, \ldots, V_k$ of *permissible feature values*, where each $V_i = \{f_i(\lambda^+) \mid \lambda^+ \in P\}$ collects the feature values of positive facts P with respect to feature f_i. P stems from $\mathcal{I}$ associated with a selected source problem. Given a fact λ to be weighted, the *minimal feature-value differences* $\Delta_i(\lambda) = \min(\{|f_i(\lambda) - v_i| \mid v_i \in V_i\})$ are made use of by computing a *weight penalty* $w_F(\lambda) = \sum_{i=1}^{k} c_i \cdot \Delta_i(\lambda)$, where $c_i \geq 0$ for all $1 \leq i \leq k$. (Note that $\Delta_i(\lambda^+) = 0$ for all $\lambda^+ \in P$.) This weight penalty is added to the weight computed by ϖ, i.e., $\varpi_F(\lambda) = \varpi(\lambda) + w_F(\lambda)$. The coefficients $c_1, \ldots, c_k$ are determined based on positive and negative facts obtainable from $\mathcal{I}$. To this end, algorithm CFC is employed that is described in detail in [5] or [7].

Flexible re-enactment centers on the observation that similarity between two proof problems is often reflected by the respective sets of positive facts sharing many facts. Therefore, given a set P of positive facts associated with a selected source problem, flexible re-enactment favors (i.e., associates small weights with) facts λ deduced when searching for a proof of the target problem that are "similar" to a $\lambda^+ \in P$. Such a fact λ is considered similar to a $\lambda^+ \in P$ if $\lambda \lhd \lambda^+$. Besides these *focus facts* λ that subsume a $\lambda^+ \in P$, also the descendants of focus facts are preferred, because they are the most likely to allow for concluding the proof. The preference given to a $\hat{\lambda}$, however, decreases with its distance $d(\hat{\lambda})$ from the closest ancestors that are focus facts, where "distance" essentially corresponds to "the number of inference steps that separate ancestors and descendant".

The details of d are described in [5] and [7]. Basically, $d(\lambda)$ is minimal (i.e., 0) if λ is a focus fact, and increases gradually with the remoteness of the "nearest" ancestor that is a focus fact. Two parameters q_1 and q_2 allow to control the rate of increase. A further parameter $p \geq 0$ controls the impact of $d(\lambda)$ on the final weight of λ, which is—in the case of "pure" flexible re-enactment—computed by $\varpi_{FR}(\lambda) = (d(\lambda) + p) \cdot \varpi(\lambda)$. Experiments documented in [5] and [7] have shown that ϖ_{FR} does not offer a significant advantage compared to a combination of features and flexible re-enactment $\varpi_{FR\&F}(\lambda) = (d(\lambda) + p) \cdot \varpi_F(\lambda)$. We therefore use here only $\varpi_{FR\&F}$ as a representative of flexible re-enactment.

4 An Environment for Full Automation

Achieving *full* automation of deduction necessitates that *the* crucial step in the use of an automated deduction system, namely choosing a search-guiding heuristic and determining its parameters, is no more the responsibility of the user. Anyone who has been working with an automated deduction system will agree that an appropriate choice on that score cannot be made deterministically. The best one can expect is some form of "intuition" ensuing from experience that eventually allows a user to make a good choice after, say, at most two or three failures in the majority of cases

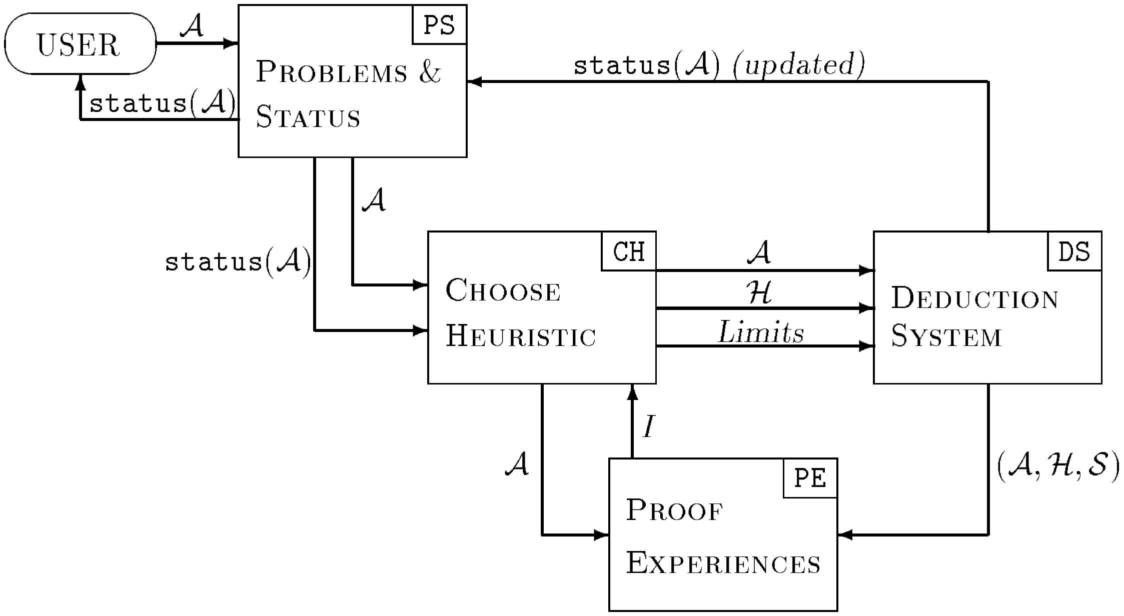

Figure 1: The fundamental architecture of an environment for a fully automated and user-friendly deduction system. The arrows denote the flow of information (i.e., proof problems, status, etc.). See text for a description.

("*trial and error*"). Therefore, it would be unreasonable to assume that this choice can be automated in a way so that the first choice (almost) always leads to success. Consequently, the central idea of a supportive environment for a deduction system (CoDe in this case) is to test a sensible number of alternatives—rather than merely a single search-guiding heuristic—which are deployed in some order. That is, alternative heuristics are "tried out" sequentially.

Apart from a component deciding on (alternative) search-guiding heuristics, the environment must also provide a component that can store and retrieve proof experiences, because we want to apply heuristics centered on these experiences. Moreover, the component for choosing a heuristic may use proof experience to support its decision making. Obviously, the component for handling proof experiences must be connected with the deduction system itself (in order to obtain input) and with the component for choosing heuristics (in order to supply it with the necessary information).

Figure 1 depicts the fundamental architecture of the environment for a user-friendly deduction system. Naturally, the deduction system itself and a component that allows to store proof problems—so to speak the user interface—are two further components.

The interplay of the four components is as follows (cp. figure 1): The user presents proof problems $\mathcal{A} = (Ax, \lambda_G)$ to the component PS for storing proof problems. Component CH is responsible for choosing a heuristic $\mathcal{H}$ so as to solve a proof problem $\mathcal{A}$ received from PS. In order to realize a (sequential) test of alternatives, status information status($\mathcal{A}$) associated and stored along with each proof problem $\mathcal{A}$ records the history of heuristics already applied to solve $\mathcal{A}$ and the respective outcomes. In particular, `status`($\mathcal{A}$) provides information on whether or not problem $\mathcal{A}$ could be solved so far. This information is also fed back to the user in order to brief him regarding the current status (in particular *proved / not proved*) of problems submitted to `PS`. The decisions `CH` has to make are supported by the database of proof experiences `PE`. To this end, `CH` can consult `PE` in order to acquire information I related to the current proof problem $\mathcal{A}$ (which is presented to `PE` as kind of a "search key"). Consultation of `PE` especially concerns heuristics that utilize past proof experience in some form, because the information supplied by `PE` is indispensable for such heuristics.

Once the decision is made which of the alternative heuristics to employ in order to solve $\mathcal{A}$, both the chosen heuristic $\mathcal{H}$ and problem $\mathcal{A}$ are given to the actual deduction system `DS`. Besides supplying $\mathcal{H}$ and $\mathcal{A}$, `CH` restricts the resources of `DS` in a suitable way. This essentially means that a maximal run time (a *time-out*) is imposed on `DS`, which is the most effective way to anticipate failure (i.e., failure to solve a problem in "reasonable" time) in many cases. In case of success or failure (due to time-out), `status`($\mathcal{A}$) is updated correspondingly. If a proof of $\mathcal{A}$ was found, then all relevant information concerning the proof and the search for it (namely $(\mathcal{A}, \mathcal{H}, \mathcal{S})$, cp. section 3) is integrated with the current contents of the database `PE`.

Please note that the alternative heuristics to be used to solve some given problem $\mathcal{A}$ cannot be determined once and for all at a certain point in time. Since the alternatives (in part) depend on `PE`, new alternatives may become possible with every dynamic change of `PE` resulting from proofs that are found gradually.

Having outlined the fundamental architecture, the subsequent section describes an implementation 'AutoCoDe' of an environment embedding CoDe based on this very architecture.

5 AutoCoDe: An Environment for CoDe

The fundamental architecture for an environment allowing for user-friendly, fully automated deduction proposed in the previous section comprises four components, namely

1. `PS`: The user interface that stores proof problems of condensed detachment given in the standard form $\mathcal{A} = (Ax, \lambda_G)$ along with their respective status;

2. `PE`: A database of proof experiences;

3. `CH`: The component that receives an unsolved proof problem $\mathcal{A}$ from `PS`, chooses and configures a heuristic $\mathcal{H}$ (possibly based on information I acquired from `PE`), and supplies the deduction system `DS` with the necessary input $\mathcal{A}$ and $\mathcal{H}$ (plus constraints for resources);

4. `DS`: The actual deduction system;

In our particular case here, CoDe is the deduction system `DS`. CoDe and its search-guiding heuristics are described in sections 2 and 3. We shall therefore concentrate here on the remaining three components `PS`, `CH`, and `PE`. First, we examine components `CH` and `PS` in order to determine the demands on component `PE`.

5.1 Components CH and PS

For reasons explained in section 4, component CH determines a sequence (or list) of alternative heuristics $\mathcal{L}(\mathcal{A}) \equiv \mathcal{H}_1;\ldots;\mathcal{H}_m$ for each problem $\mathcal{A}$ (using information from PE). For each $\mathcal{H}_i$, CH can choose between ϖ, ϖ_F, and $\varpi_{FR\&F}$. In connection with $\varpi_{FR\&F}$, we always set the parameters $q_1 = 0.75$, $q_2 = 0.25$, and $p = 20$ (cp. [5] or [7]). The remaining parameters of these heuristics are set as described below.

The $\mathcal{H}_i \in \mathcal{L}(\mathcal{A})$ are tried out in the order given by $\mathcal{L}(\mathcal{A})$. An attempt to solve $\mathcal{A}$ with $\mathcal{H}_i$ is granted a maximal run time T_i. Note again that the sequence is not static. It can grow or may be modified as proof experience accumulates in PE. At the outset, of course only variations of the basic heuristic ϖ can be employed, because there is no proof experience (yet). The basic heuristic anyway plays an important role as a "starter" providing a basis of proof experiences which is necessary to gradually build up more and more experience by ϖ_F and $\varpi_{FR\&F}$ profiting from each other in a bootstrapping manner. Moreover, the basic heuristic is appropriate to sort out simpler problems for which it does not make sense to apply ϖ_F or $\varpi_{FR\&F}$, and to take the risk of failure because an inappropriate source problem is picked. (Understandably, a comparatively small time-out T_i will be imposed on a basic heuristic.)

For both reasons, basic heuristics are applied *at first* and *exhaustively*. This means that, on the one hand, only the first $j \leq m$ heuristics of $\mathcal{L}(\mathcal{A}) \equiv \mathcal{H}_1;\ldots;\mathcal{H}_m$ are basic heuristics (i.e., ϖ with various parameter settings). On the other hand, a heuristic involving past experience (i.e., ϖ_F or $\varpi_{FR\&F}$) is only applied to a problem $\mathcal{A}$ if for *all* unsolved problems in PS all basic heuristics (listed in the respective list of alternatives) have been tested already.

Component CH determines partially with the help of PE the heuristics constituting $\mathcal{L}(\mathcal{A})$ according to their (estimated) appropriateness for solving $\mathcal{A}$. CH does not consult PE in order to decide on the basic heuristics which are at the head of the list. On the one hand, doing without the information of PE in connection with basic heuristics is possibly unavoidable, because there is not yet any information available. On the other hand, our experiments with ϖ have shown (see [5] or [6]) that there are very few parameter settings for c_δ and c_w—i.e., ratios $c_\delta : c_w$—that work satisfactorily well for a large number of problems, so that the parameters c_δ and c_w of ϖ can be set sensibly without falling back on past experience. We decided on two ratios, namely $2:1$ and $4:1$. Hence, the first two heuristics $\mathcal{H}_1$ and $\mathcal{H}_2$ to be tested are (for all problems) $\varpi[2:1]$ and $\varpi[4:1]$. (Henceforth, we shall use the notation $\mathcal{H}[p]$ to denote parameter settings p of heuristic $\mathcal{H}$ which do not derive from context.)

In connection with heuristics ϖ_F and $\varpi_{FR\&F}$, PE is in part necessary to determine the parameters c_δ and c_w. For reasons explained in [5], $c_\delta : c_w$ is set to $0:1$ regardless of past experience if $\varpi_{FR\&F}$ is to be used. In case ϖ_F utilizes a source problem $\mathcal{A}_i$, the *history* of $\mathcal{A}_i$ determines $c_\delta : c_w$. Each source problem $\mathcal{A}_i$ has a unique history of source problems that contributed to its proof. The history of $\mathcal{A}_i$ is a sequence of source problems $\mathcal{B}_1,\ldots,\mathcal{B}_k$, where $\mathcal{B}_k \equiv \mathcal{A}_i$. $\mathcal{B}_1$ was proved with a basic heuristic, and each $\mathcal{B}_{i+1}$ was proved with either ϖ_F or $\varpi_{FR\&F}$ using $\mathcal{B}_i$ as source. Hence, the source problem $\mathcal{B}_1$ which started the history was proved using $\varpi[c_\delta : c_w]$. This very ratio is also set by ϖ_F.

To put it another way, past experience can be viewed as a collection of trees (because a proof of a problem in PE involves at most one source problem). The nodes of a tree represent proof problems, and the vertices represent source/target dependencies. At the root of each tree (the tree possibly merely consists of the root problem), there is a problem solved with a basic heuristic. All other nodes of the tree represent problems solved either with ϖ_F or with $\varpi_{FR\&F}$ using the problem represented by the ancestor node as source. The history of a problem $\mathcal{A}_i$ hence is the list of nodes (problems) occurring on the branch from the root to $\mathcal{A}_i$, starting with the root. (If we make the reasonable assumption that there is exactly one proof for each problem in PE, then the history of each $\mathcal{A}_i$ is indeed unique.) Whenever ϖ_F is employed with source problem $\mathcal{A}_i$, ϖ_F sets the ratio $c_\delta : c_w$ that was also set by the basic heuristic ϖ when proving the root problem of the tree $\mathcal{A}_i$ belongs to.

Component PE becomes indispensable as soon as ϖ_F or $\varpi_{FR\&F}$ are to be employed to tackle $\mathcal{A}$. PE supplies the crucial positive facts and feature coefficients $c_1, \ldots, c_k$ required by ϖ_F and $\varpi_{FR\&F}$. To this end, problem $\mathcal{A}$ is presented to PE, whereupon PE selects a set of source problems currently in its database which are similar to $\mathcal{A}$. For each selected source problem, the associated set P of positive facts and the feature coefficients FC form pairs (P, FC). These pairs are collected in the set $R_{\mathtt{PE}}(\mathcal{A})$ and returned.

With the help of $I = R_{\mathtt{PE}}(\mathcal{A})$ retrieved by PE, component CH can set up the heuristics ϖ_F and $\varpi_{FR\&F}$ that (explicitly) exploit past experience. For each pair $(P, FC) \in I$, two heuristics are configured, namely $\varpi_F[P, FC]$ and $\varpi_{FR\&F}[P, FC]$. These heuristics are appended to the current list of heuristics $\mathcal{L}(\mathcal{A})$ in that order unless they are already a member of $\mathcal{L}(\mathcal{A})$. (The ratio $c_\delta : c_w$ is set as described earlier in this section.)

Component CH receives unproved target problems from PS. PS submits unproved problems (one at a time) according to the lexicographic ordering of their names. PS continues to submit problems (upon request from CH) over and over again as long as there are unproved target problems for which not all alternative heuristics have been tried out. In order to ensure that basic heuristics are applied at first and exhaustively, PS does not submit any problem to CH for which all (i.e., both) basic heuristics $\varpi[2:1]$ and $\varpi[4:1]$ have failed, as long as there are other unproved target problems for which attempts with a basic heuristic are still pending.

The status $\mathtt{status}(\mathcal{A})$ associated and stored together with each target problem $\mathcal{A}$ comprises the list of heuristics $\mathcal{L}(\mathcal{A})$. For each novel problem $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$ is the empty list ε. When CH receives a target problem $\mathcal{A}$ from PS with $\mathcal{L}(\mathcal{A}) \equiv \varepsilon$ (i.e., for the first time), CH sets $\mathcal{L}(\mathcal{A}) \equiv \mathcal{H}_1; \mathcal{H}_2$, where $\mathcal{H}_1 \equiv \varpi[2:1]$ and $\mathcal{H}_2 \equiv \varpi[4:1]$. The first heuristic is immediately applied to prove $\mathcal{A}$. $\mathtt{status}(\mathcal{A})$ is updated correspondingly. After that, CH accepts another target problem. When $\mathcal{A}$ is presented to CH for the second time, an attempt to prove $\mathcal{A}$ with $\mathcal{H}_2$ is started.

In case $\mathcal{L}(\mathcal{A}) \equiv \mathcal{H}_1; \ldots; \mathcal{H}_m$, $m \geq 2$, and both $\mathcal{H}_1$ and $\mathcal{H}_2$ have already been tried out, then CH consults PE by presenting $\mathcal{A}$ to PE, and obtains $I = R_{\mathtt{PE}}(\mathcal{A})$ as an answer. $\mathcal{L}(\mathcal{A})$ is then extended as described above, resulting in $\mathcal{L}(\mathcal{A}) \equiv \mathcal{H}_1; \ldots; \mathcal{H}_{m'}$, $m \leq m'$. The "new" $\mathcal{H}_i$ $(m < i \leq m')$ are successively employed to prove $\mathcal{A}$ before accepting the next target problem from PS. (The successive application of new $\mathcal{H}_i$ stops of course as soon as one of these $\mathcal{H}_i$ succeeds.)

By now we have outlined how component CH (gradually) composes and tries out a list of heuristics for each target problem. We have also described how CH interacts with component PS in order to obtain target problems. Furthermore, the role of PE was sketched which consists of supplying CH with information I on past experience. The following subsection explains how component PE determines the information I required by CH.

5.2 Component PE

The database PE stores proof experience in the format $\mathcal{I} = (\mathcal{A}, \mathcal{H}, \mathcal{S})$ advocated in section 3, namely the (source) problem $\mathcal{A}$, the heuristic $\mathcal{H}$ (including its parameter settings) employed by CoDe in order to prove $\mathcal{A}$, thereby producing the search protocol $\mathcal{S}$. For efficiency reasons, in practice the set of positive facts P extricable from $\mathcal{S}$ is also stored along with $\mathcal{A}$, $\mathcal{H}$, and $\mathcal{S}$ so as to avoid computing (extracting) it each time it is required. Furthermore, feature coefficients FC are stored together with each source $\mathcal{I}$. These coefficients are computed according to algorithm CFC presented in [5] and [7]—involving the sets P and N of positive and negative facts extricable from $\mathcal{S}$—*if* heuristic $\mathcal{H}$ is the basic heuristic ϖ (applied with some ratio $c_\delta : c_w$). Otherwise, if $\mathcal{H}$ is one of ϖ_F or $\varpi_{FR\&F}$, then the feature coefficients are simply a copy of the ones associated with the respective source employed by $\mathcal{H}$. Since any source in PE either was solved with the basic heuristic or goes back (possibly over several stages) to a problem solved with the basic heuristic, it is ensured that there are feature coefficients for all sources in PE.

Hence, PE maintains packages $(\mathcal{A}, \mathcal{H}, \mathcal{S}, P, FC)$ of information on previous experience, extending (for efficiency reasons) the "minimal" information $(\mathcal{A}, \mathcal{H}, \mathcal{S})$ with the set P of positive facts (extracted from $\mathcal{S}$) and the feature coefficients FC (inherited from or computed for the "root" problem).

The main task of PE is to find source problems which are similar to a target problem $\mathcal{A}$ presented to PE by CH.[1] In order to accomplish this, a suitable *similarity measure sim* must be available which assesses the similarity $sim(\mathcal{A}, \mathcal{A}_i)$ of target problem $\mathcal{A}$ and source problems $\mathcal{A}_i$ stored as information packages $(\mathcal{A}_i, \mathcal{H}_i, \mathcal{S}_i, P_i, FC_i)$ in PE. Given *sim*, PE returns the set $R_{\mathtt{PE}}(\mathcal{A})$ as information I, where

$$R_{\mathtt{PE}}(\mathcal{A}) = \{(P_i, FC_i) \mid cond(\mathcal{A}, \mathcal{A}_i)\} \ .$$

Condition *cond*—which will be described shortly—determines whether or not the similarity between target problem $\mathcal{A}$ and source problem $\mathcal{A}_i$ is acceptable.

sim computes a similarity measure based on the similarity of target and source axiomatizations as well as on the similarity between target and source goals. Considering that a target problem $\mathcal{A} = (Ax, \lambda_G)$ is proved by virtue of a proof of a source problem $\mathcal{A}_i = (Ax_i, \lambda_i)$ if all axioms of the source problem are subsumed by axioms of the target

[1] Storing source problems is simply done by adding the respective information package to the database. Measures aiming at minimizing the database are possible, but they are currently not applied, because in connection with our experiments the computational effort for retrieving is so low that elaborate methods for reducing the amount of data stored do not pay off.

problem (i.e., $\forall \lambda \in Ax_i \; \exists \lambda' \in Ax : \lambda' \lhd \lambda$), and the target goal is subsumed by the source goal (i.e., $\lambda_i \lhd \lambda_G$), the following realization of *sim* suggests itself.

Definition 5.1 (Similarity Measure) *Let $\mathcal{A} = (Ax, \lambda_G)$ be a target problem, and $\mathcal{A}_i = (Ax_i, \lambda_i)$ a source problem. The similarity of target and source problem is $sim(\mathcal{A}, \mathcal{A}_i) = (s_1, s_2, s_3) \in I\!N^3_{100}$ ($I\!N_{100} = \{0, 1, \ldots, 100\}$), where*

$$\begin{aligned} s_1 &= \frac{|\{\lambda \in Ax_i \mid \exists \lambda' \in Ax : \lambda' \lhd \lambda\}|}{|Ax_i|} \cdot 100 \\ s_2 &= \frac{|\{\lambda \in Ax \mid \exists \lambda' \in Ax_i : \lambda \lhd \lambda'\}|}{|Ax|} \cdot 100 \\ s_3 &= \begin{cases} 100, & \lambda_i \lhd \lambda_G \\ 0, & otherwise. \end{cases} \end{aligned}$$

sim actually computes three measures s_1, s_2 and s_3.[2] s_1 represents the percentage of source axioms that are subsumed by target axioms, so to speak a "coverage". s_2 represents the percentage of target axioms which account for subsuming source axioms. s_3 expresses the fact that the source goal subsumes (or does not subsume) the target goal with 100 (or 0). (A finer grained measure may be desirable for s_3, but we content ourselves with this "Boolean" version for the time being.)

A perfect similarity is there if $s_1 = s_2 = s_3 = 100$. In this case, all source axioms are subsumed by target axioms, and the source goal subsumes the target goal. As a consequence, the target is proved simply because of the existence of a proof of the source problem. $s_2 = 100$ merely states that there are no additional (unnecessary) target axioms. This is anyway the sole purpose of s_2: Identifying target axioms which do not have an obvious correspondence with source axioms. Such target axioms are likely to be superfluous (in particular if $s_1 = 100$), and hence might unnecessarily complicate the search by bloating up the search space.

The similarity measure *sim* is utilized to determine $R_{\mathtt{PE}}(\mathcal{A})$. For a sensible use of the information provided by $R_{\mathtt{PE}}(\mathcal{A})$, this information should be *reliable* and *sparse*, i.e., only "sufficiently similar" and few source problems should qualify. In order to accomplish this, we have to formalize notions like "minimal similarity" and "more similar than".

First of all recall that two problems are considered the more similar the higher the measures s_1, s_2 and s_3 are. Thus, a *minimal similarity ms* can be defined by

$$ms(sim(\mathcal{A}, \mathcal{A}_i)) = a_1 \cdot s_1 + a_2 \cdot s_2 + a_3 \cdot s_3 \geq l\,,$$

[2] The similarity between two problems is centered on subsumption and consequently on syntactic structure. (Recall that "λ_1 subsumes λ_2", i.e., $\lambda_1 \lhd \lambda_2$, here simply means that there is a match σ so that $\sigma(\lambda_1) \equiv \lambda_2$.) Therefore, '$\lhd$' is computed modulo renaming of function symbols (and variables), i.e., modulo an arity-preserving signature match. Since the source problem is to be employed to solve the target problem, the target should not be modified. For this reason, a signature match is applied to the source. The simplicity of problems here entails that there is *at most one* signature match from source to target, and we therefore refrain from explicitly integrating signature matching with our notation. (If no signature match from source to target exists, then we obtain $s_1 = s_2 = s_3 = 0$.)

where $a_1, a_2, a_3 \in \mathbb{N}$, and $l \in \mathbb{N}$ is the *threshold*. Considering that the coverage of source axioms is the most important criterion for the similarity of proof problems, and the existence of "superfluous" target axioms is comparatively insignificant, we set $a_1 = 3$, $a_2 = 1$, and $a_3 = 2$. Using a threshold $l = 100$, a coverage of one third, or no superfluous target axioms, or the "same" goal each suffice alone to reach the threshold. So, a necessary criterion for a source problem $\mathcal{A}_i$ to qualify for $R_{\text{PE}}(\mathcal{A})$ is $3 \cdot s_1 + 1 \cdot s_2 + 2 \cdot s_3 \geq 100$.

Naturally, only the "most similar" source problems should contribute to $R_{\text{PE}}(\mathcal{A})$. For this purpose, a comparison of, or a "better-than" relation $\succ$ on similarity measures must be defined. In the sequel, this relation $\succ$ is developed gradually with the help of orderings $\gg$, $>_D$ and $>_P$.

Given two source problems $\mathcal{A}_i$ and $\mathcal{A}_j$, it certainly makes sense to consider target problem $\mathcal{A}$ more similar to $\mathcal{A}_i$ than to $\mathcal{A}_j$ if $sim(\mathcal{A}, \mathcal{A}_i)$ beats $sim(\mathcal{A}, \mathcal{A}_j)$ in every respect:

Definition 5.2 (Ordering $\gg$)
Let $sim(\mathcal{A}, \mathcal{A}_i) = (s_1, s_2, s_3)$ and $sim(\mathcal{A}, \mathcal{A}_j) = (s'_1, s'_2, s'_3)$.

$$sim(\mathcal{A}, \mathcal{A}_i) \gg sim(\mathcal{A}, \mathcal{A}_j) \quad \textit{if and only if} \quad s_1 > s'_1 \;\wedge\; s_2 > s'_2 \;\wedge\; s_3 > s'_3 \,.$$

The second ordering $>_D$ centers on the "difficulty" of the source problem measured in terms of the length $|\mathcal{S}|$ of the associated search protocol $\mathcal{S}$. The longer the search protocol is, the more difficult a problem is considered to be. Since more can be learned from more difficult problems, more difficult problems supersede less difficult ones.

Definition 5.3 (Ordering $>_D$) *Let $\mathcal{S}_i$ and $\mathcal{S}_j$ be the search protocols associated with source problems $\mathcal{A}_i$ and $\mathcal{A}_j$, respectively.*

$$sim(\mathcal{A}, \mathcal{A}_i) >_D sim(\mathcal{A}, \mathcal{A}_j) \quad \textit{if and only if} \quad |\mathcal{S}_i| > |\mathcal{S}_j| \,.$$

Ordering $>_P$ takes into account whether or not $\mathcal{A}_i$ was proved with the help of $\mathcal{A}_j$, i.e., whether or not $\mathcal{A}_j$ was used as a source problem in the history of $\mathcal{A}_i$, i.e., a sequence of source problems $\mathcal{B}_1, \ldots, \mathcal{B}_k$ with properties as outlined above. $\mathcal{A}_i$ is said to be proved with the help of $\mathcal{A}_j$—denoted by $PW(\mathcal{A}_i, \mathcal{A}_j)$—if $\mathcal{A}_j \in \{\mathcal{B}_1, \ldots, \mathcal{B}_{k-1}\}$. If $\mathcal{A}_i$ actually was proved with the help of $\mathcal{A}_j$, then it is reasonable to assume that at least as much can be learned from a proof of $\mathcal{A}_i$ as can be learned from a proof of $\mathcal{A}_j$. (Recall that feature coefficients are passed on anyway.)

Definition 5.4 (Ordering $>_P$, $\approx_P$) *Let $PW(\mathcal{A}_i, \mathcal{A}_j)$ denote that $\mathcal{A}_i$ was proved with the help of $\mathcal{A}_j$.*

$$sim(\mathcal{A}, \mathcal{A}_i) >_P sim(\mathcal{A}, \mathcal{A}_j) \quad \textit{if and only if} \quad PW(\mathcal{A}_i, \mathcal{A}_j) \,.$$

$sim(\mathcal{A}, \mathcal{A}_i) \approx_P sim(\mathcal{A}, \mathcal{A}_j)$ denotes that neither $PW(\mathcal{A}_i, \mathcal{A}_j)$ nor $PW(\mathcal{A}_j, \mathcal{A}_i)$, i.e., $sim(\mathcal{A}, \mathcal{A}_i)$ and $sim(\mathcal{A}, \mathcal{A}_j)$ cannot be compared with $>_P$.

Evidently, ordering $\gg$ is the most important for deciding whether one source problem is more similar to the target problem than another source problem. Orderings $>_D$ and $>_P$ play a minor role in that they are consulted only if both similarity measures are *equal*. If this is the case, ordering $>_P$ should be given priority over $>_D$, because a problem $\mathcal{B}$ that can be proved with the help of some source problem $\mathcal{B}'$ often has a shorter search protocol. Consequently, problem $\mathcal{B}'$ would be preferred if $>_D$ were given priority over $>_P$. Nonetheless, problem $\mathcal{B}$ must be considered as more difficult, because a proof of $\mathcal{B}'$ was needed to prove $\mathcal{B}$. The difficulty is simply not reflected by the length of the search protocol, because the search protocol could be kept short on account of the experiences made in connection with $\mathcal{B}'$.

Relation $\succ$ therefore is a lexicographic combination of $\gg$, $>_P$ and $>_D$ that is defined as follows.

Definition 5.5 ("Better-than" Relation $\succ$ on Similarity Measures) *Let $\mathcal{A}$ be a target problem, and $\mathcal{A}_i$ and $\mathcal{A}_j$ two source problems. $\mathcal{A}$ is more similar to $\mathcal{A}_i$ than to $\mathcal{A}_j$ if and only if $sim(\mathcal{A},\mathcal{A}_i) \succ sim(\mathcal{A},\mathcal{A}_j)$, where $sim(\mathcal{A},\mathcal{A}_i) \succ sim(\mathcal{A},\mathcal{A}_j)$ if and only if*

$$\begin{array}{l} \quad sim(\mathcal{A},\mathcal{A}_i) \gg sim(\mathcal{A},\mathcal{A}_j) \\ \vee\ sim(\mathcal{A},\mathcal{A}_i) = sim(\mathcal{A},\mathcal{A}_j) \wedge sim(\mathcal{A},\mathcal{A}_i) >_P sim(\mathcal{A},\mathcal{A}_j) \\ \vee\ sim(\mathcal{A},\mathcal{A}_i) = sim(\mathcal{A},\mathcal{A}_j) \wedge sim(\mathcal{A},\mathcal{A}_i) \approx_P sim(\mathcal{A},\mathcal{A}_j) \wedge sim(\mathcal{A},\mathcal{A}_i) >_D sim(\mathcal{A},\mathcal{A}_j) \end{array}$$

(Note that $\succ$ is not an ordering, because it is not transitive due to $\approx_P$.)

Condition *cond* which determines if a source problem is acceptable for $R_{\mathtt{PE}}(\mathcal{A})$ is now given by

$$cond(\mathcal{A},\mathcal{A}_i) \quad \textit{if and only if} \quad ms(sim(\mathcal{A},\mathcal{A}_i)) \geq l \wedge \neg\exists \mathcal{A}_j : sim(\mathcal{A},\mathcal{A}_j) \succ sim(\mathcal{A},\mathcal{A}_i)\,.$$

Note that there can be several source problems that qualify for $R_{\mathtt{PE}}(\mathcal{A})$ (which is reasonable considering that the parameter 'source problem' is—as all other parameters—mostly determined by mere intuition and expertise that may be treacherous). But our experiments have shown that $\succ$ allows us to restrict the number of source problems to very few (mostly one or two).

Readers who are familiar with *case-based reasoning* (CBR) will have noticed the close relation of PE and (problem-solving) CBR (cp. [8]). The information packages $(\mathcal{A},\mathcal{H},\mathcal{S})$ (or their extended counterparts) correspond to cases, and PE hence represents a case base. The major processes of CBR, namely *case storage* and *case retrieval*, are also present. Case retrieval is accomplished by computing an explicit similarity measure which is deployed to select the cases that appear to be most similar and hence most appropriate to tackle a current problem. Solutions to "old" problems of the case base are utilized as a heuristic guideline as opposed to being explicitly reused. Therefore, *adaptation*—the process of adjusting an old solution to fit a new situation—is not necessary (except for a signature match). Also, *criticism* and *evaluation* of solutions (proofs) are merely intimated by $>_D$ and $>_P$.

We have now outlined how past experience is made available by PE and used by CH in order to create, update, and apply a list of heuristics in order to prove the target problems submitted by PS. The next section presents experimental results obtained with AutoCoDe which demonstrate the feasibility and excellent performance of fully automated deduction with learning capabilities in the area of condensed detachment.

6 Experimental Results

We experimented with AutoCoDe in the light of problems from the MV and CN calculus (cp. [11]). All problems considered here are taken from [11]. The name of a problem is composed of the abbreviation of the calculus it belongs to and of the continuous numbering used in [11]. More specifically, we consider problems cn1,...,cn33 and problems mv55,...,mv62, which correspond to problems LCL040-1,...,LCL072-1 and LCL109-1,...,LCL116-1 in the TPTP problem library ([16]) version 1.2.0, respectively.

First, AutoCoDe (i.e., component PS) was given the eight problems mv55,...,mv62 of the MV calculus. Second, 33 problems of the CN calculus, namely cn1,...,cn33, had to be tackled. In both cases, AutoCoDe started with an empty database PE of proof experiences. The maximal run time granted to basic heuristics was 20 seconds. For ϖ_F and $\varpi_{FR\&F}$, the run time was restricted to 60 and 240 seconds, respectively.

We compare our results obtained with AutoCoDe with the results of Otter published in [11] in order to substantiate AutoCoDe's achievements. It must be emphasized that CoDe (i.e., the "core deduction machine") is a purely experimental program (implemented in C). Its core was developed in a couple of weeks as opposed to the well-renowned Otter which has been improved over several years. CoDe does not use sophisticated indexing techniques. These are crucial for efficient (forward) subsumption which is exhaustively needed in connection with CD. CoDe might be faster at very early stages of the search (if at all) because of a specialized implementation of CD, which Otter "simulates" with hyper-resolution. But efficiency increasing techniques like flat terms and indexing cause Otter to surpass CoDe (in terms of inferences per second) after these early stages. Consequently, faster run times of AutoCoDe (or CoDe) can only stem from a suitable use of powerful heuristics.

Table 1 summarizes the experiment conducted with problems mv55,...,mv62. The first and second column display the target and source problem, respectively. The target problems are listed in the order in which they were proved by AutoCoDe.[3] The entry '$\star$' in the column 'Source Problem' signifies that no source problem was made use of, because a basic heuristic was applied. The heuristic which finally allowed to prove the target problem is specified in column 3 ($\mathcal{H}[\ldots]$). (Recall that $\varpi_{FR\&F}$ always sets the ratio $0:1$.) The fourth column lists the number of failed attempts to prove the target (due to a time-out) in the form 'B / L', where B and L correspond to the number of failures of a basic heuristic (ϖ) and a learning heuristic (ϖ_F or $\varpi_{FR\&F}$), respectively, before finally proving the target. The fifth column displays the time spent on failed attempts which is the sum of the respective time-outs. The sixth column shows the run time of the (final) successful attempt. Run times are CPU time in seconds, obtained on

[3] This order in parts depends on the lexicographic order of problem names and of course on the difficulty of the problems, in particular on source/target dependencies.

Table 1: AutoCoDe and the MV Calculus

Target Problem	Source Problem	$\mathcal{H}[\ldots]$	Failures B / L	Run Time Failures	Run Time Success	Total Time	Otter (best)
`mv56`	$\star$	$\varpi[2:1]$	0 / 0	0s	0.7s	0.7s	2s
`mv57`	$\star$	$\varpi[2:1]$	0 / 0	0s	1.1s	1.1s	5s
`mv58`	$\star$	$\varpi[2:1]$	0 / 0	0s	0.7s	0.7s	2s
`mv59`	$\star$	$\varpi[2:1]$	0 / 0	0s	13s	13s	1468s
`mv61`	$\star$	$\varpi[2:1]$	0 / 0	0s	18s	18s	7s
`mv60`	`mv61`	$\varpi_F[2:1]$	2 / 0	40s	50s	108s	2035s
`mv62`	`mv60`	$\varpi_{FR\&F}$	2 / 1	100s	17s	225s	2041s
`mv55`	—	—	2 / 4	640s	∞	∞	∞

a SPARCstation ELC. An entry '∞' denotes that no proof was found. (According to [11] Otter operated with a time-out of 4 hours.) The seventh column shows the total run time spent on solving the target problem which is composed of the time spent on failed attempts (column 5) and the time for the successful attempt. For target problems proved with ϖ_F or $\varpi_{FR\&F}$, the *total* time needed to prove the respective source problem is also included. The last column lists the *best* results of Otter (obtained on a SPARCstation 1+ that is comparable to a SPARCstation ELC).

An example should clarify how to interpret table 1. Consider target problem `mv62`. Problem `mv62` was proved with $\varpi_{FR\&F}$ using source problem `mv60`. The two attempts with basic heuristics (namely $\varpi[2:1]$ and $\varpi[4:1]$) and one attempt with ϖ_F (using source problem `mv60`) failed before (hence the entry '2 / 1' in column 'Failures B / L'). These three failures account for 100 seconds CPU time ($2 \cdot 20s + 1 \cdot 60s$). The attempt with $\varpi_{FR\&F}$ succeeded after 17 seconds. Taking into account the total time of 108 seconds needed to prove the source problem `mv60`, the total time for `mv62` is $100s + 17s + 108s = 225s$.

At this point we would like to point out that the run times given are only the time spent by CoDe itself, i.e., component `DS`. The effort for retrieving/storing proof experience (component `PE`), setting up a list of alternative heuristics (component `CH`) etc. is negligible, although brute-force algorithms are employed for the respective components, which are in parts implemented as UNIX shell scripts (e.g., [1]).

Table 1 reveals that AutoCoDe performs significantly better than Otter even when the total time—which includes failed attempts and the total time needed to prove source problems—is taken into account. Note that AutoCoDe achieved this without any interaction on the parts of the user. A comparison of AutoCoDe and Otter is also given by figure 2. This figure displays the percentage of problems `mv55`,...,`mv62` solved depending on the time spent by AutoCoDe and Otter. In case of AutoCoDe, this time is the accumulated run time of all failed and successful attempts undertaken by AutoCoDe. While the graph related to AutoCoDe depicts the actual performance of AutoCoDe as observed during our experiment, we made the following

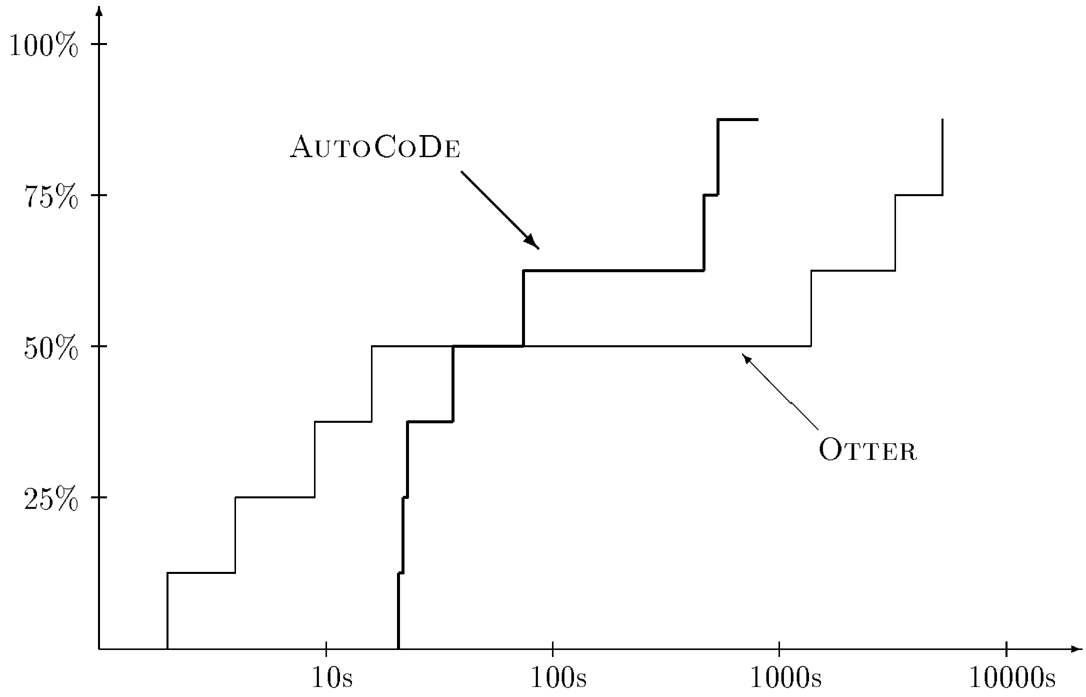

Figure 2: Performance of AutoCoDe and Otter for problems of the MV calculus. Performance is measured in percentage of problems solved (y-axis) vs. accumulated run time (x-axis, logarithmic scale). AutoCoDe solves 87.5% of the problems (seven out of eight) within 881 seconds. Otter solves (the same) 87.5% of the problems within 5560 seconds if very generous assumptions are made in favor of Otter.

very optimistic and favorable assumptions regarding Otter: Otter always picks the most suitable heuristic. Moreover, Otter does not spend any time on problems which it cannot solve (here only `mv55`). The graph related to Otter hence depicts accumulated run times of the *best* successful runs for each problem. Problems are considered according to the ascending order of their run times.

Despite these (unrealistic and very generous) assumptions, Otter spends significantly more time—namely 5560 seconds—on solving seven out of the eight problems than AutoCoDe which solves the same seven problems within 881 seconds. Note that these 881 seconds include 640 seconds that are spent on six futile attempts to solve `mv55`.[4] (Recall that basic heuristics are applied first and exhaustively. To be more exact, the first basic heuristic $\varpi[2:1]$ is applied to all target problems, and then the second basic heuristic $\varpi[4:1]$ is applied to all remaining unproved target problems. Considering that problem `mv55` is the first target problem since problems are ordered lexicographically according to their names, AutoCoDe spends the first 20 seconds on a failed attempt to prove `mv55` with $\varpi[2:1]$, and the last 300 seconds on failed

[4] To our knowledge, problem `mv55` has not (yet) been proved by an (unassisted) automated deduction system.

Table 2: AutoCoDe and the CN Calculus

Target Problem	Source Problem	$\mathcal{H}[\ldots]$	Failures B / L	Run Time Failures	Run Time Success	Total Time	Otter (best)
cn12	$\star$	$\varpi[2:1]$	0 / 0	0s	10s	10s	74s
cn25	$\star$	$\varpi[2:1]$	0 / 0	0s	19s	19s	89s
cn26	$\star$	$\varpi[2:1]$	0 / 0	0s	2s	2s	1s
cn6	$\star$	$\varpi[4:1]$	1 / 0	20s	11s	31s	1467s
cn1	cn25	$\varpi_F[2:1]$	2 / 2	340s	8s	367s	16s
cn19	cn12	$\varpi_{FR\&F}$	2 / 1	100s	33s	143s	423s
cn21	cn19	$\varpi_F[2:1]$	2 / 0	40s	41s	224s	447s
cn22	cn21	$\varpi_{FR\&F}$	2 / 1	100s	194s	518s	∞
cn23	cn22	$\varpi_{FR\&F}$	2 / 1	100s	1s	619s	∞
cn29	cn26	$\varpi_F[2:1]$	2 / 0	40s	25s	67s	257s
cn3	cn6	$\varpi_F[4:1]$	2 / 0	40s	20s	91s	3657s
cn32	cn29	$\varpi_F[2:1]$	2 / 0	40s	48s	155s	511s
cn15	cn23	$\varpi_{FR\&F}$	2 / 3	400s	14s	1033s	∞
cn24	—	—	2 / 4	640s	∞	∞	∞

attempts using ϖ_F and $\varpi_{FR\&F}$; cp. figure 2.)

On the one hand, this first experiment with the MV calculus is encouraging and promising. On the other hand, it did not really put to a test the components which are to replace user interaction, i.e., components PE and CH (and PS). (But it did re-confirm the powerfulness of the heuristics ϖ_F and $\varpi_{FR\&F}$.) All problems mv55,. . . ,mv62 have the same axiomatization (and different goals). Consequently, $\gg$ does not play a role for $\succ$. Therefore, similarity is decided with the help of $>_P$ and $>_D$ only, which both have a minor potential of being misleading compared to $\gg$. The following experiment with the CN calculus is more challenging. First, there are much more problems, namely cn1,. . . ,cn33. Second, there are several groups of problems with differing axiomatizations (cp. [11] or [5]), possibly having the same goal. Consequently, finding a sufficiently similar source problem becomes much more difficult, this time definitely involving $\gg$.

Table 2 lists the results of the experiment with the CN calculus. Table 2 is organized like table 1. In order to keep the size of the table manageable, we omitted all those target problems which could be proved with the first basic heuristic $\varpi[2:1]$ and did not play a role as a source problem. (Those problems could all be proved within 15 seconds. See [5] for details.)

Once again table 2 demonstrates that AutoCoDe outperforms Otter even with respect to the *total* time required to solve a target problem.[5] The only exception is problem cn1. The main reason are two failed attempts with ϖ_F and $\varpi_{FR\&F}$—accounting for 300 seconds spent in vain—when using the evidently inappropriate source problem

[5] In particular, AutoCoDe solves problems that Otter cannot solve.

cn11. (Recall that each source problem selected by PE entails two heuristics—namely ϖ_F and $\varpi_{FR\&F}$ using the respective source problem—set up by CH.) Problems cn1 and cn11 have the same goal, but do not share any axioms, whereas cn25 and cn1 do share some axioms (but have different goals). Both problems qualify as source problems for cn1. Source problem cn11 is made use of first simply because it occurs before cn25 in the lexicographic order of names. This example suggests possible refinements of AutoCoDe, e.g., determining the order in which source problems come into play with the help of more elaborate methods (for instance, "sharing axioms" should be preferred to "having the same goal, but sharing less (no) axioms").

Similar to cn1, the first source problem PE selected in connection with cn15—namely cn12—did not prove useful. But this time, PE cannot be blamed for the failure, because at the time a source problem for cn15 was requested for the first time, the appropriate source cn23 was not yet proven. Due to the lexicographic order in which target problems are processed, cn15 was dealt with before cn19. Problem cn19 could be proved with the help of source problem cn12, and cn19 was useful to prove cn21, which in turn allowed to prove cn22, and cn22 made it possible to prove cn23. (This chain of source/target dependencies is a good example for the bootstrapping qualities of learning heuristics.) So, when a source problem was requested for cn15 (for the second time) *after* cn23 had been proved, the "right" source cn23 was picked. (If cn15 had not been processed before cn19, cn21, cn22 and cn23, then the right choice would have been made by PE, and the unsuccessful attempts involving cn12 would have been avoided.)

But these little "flaws" are acceptable in view of the overall performance of AutoCoDe depicted by figure 3. (Its interpretation is analogous to the interpretation of figure 2.) AutoCoDe proves 32 out of the 33 problems (ca. 97%) within 2358 seconds—three more than Otter can solve (ca. 88%) within 8098 seconds. Again, figure 3 displays AutoCoDe's actual performance *including* failed attempts,[6] whereas only the *best* results *excluding* time spent on failed attempts are considered in connection with Otter. (Similar to the MV calculus, AutoCoDe gets off to a bad start with a failed attempt to prove cn1. Furthermore, AutoCoDe "wastes" 300 seconds on futile attempts to prove cn24 right before terminating.[7])

As a general result of the experiment with the CN calculus, components PE and CH have demonstrated satisfactory performance. The excellent (overall) performance of AutoCoDe illustrated by figure 3 must, however, also be accredited in parts to the powerfulness of our basic and learning heuristics which allowed us to operate with comparatively small time-outs, and hence allowed to keep the time spent on failures down at an acceptable level.

7 Discussion

We have presented first steps towards fully automated and powerful deduction systems. We consider a deduction system to be fully automated if a user merely has to submit

[6] The overhead due to component PE and CH (and PS) is again negligible, and is therefore ignored.

[7] To our knowledge, problem cn24 has not (yet) been proved by an (unassisted) automated deduction system.

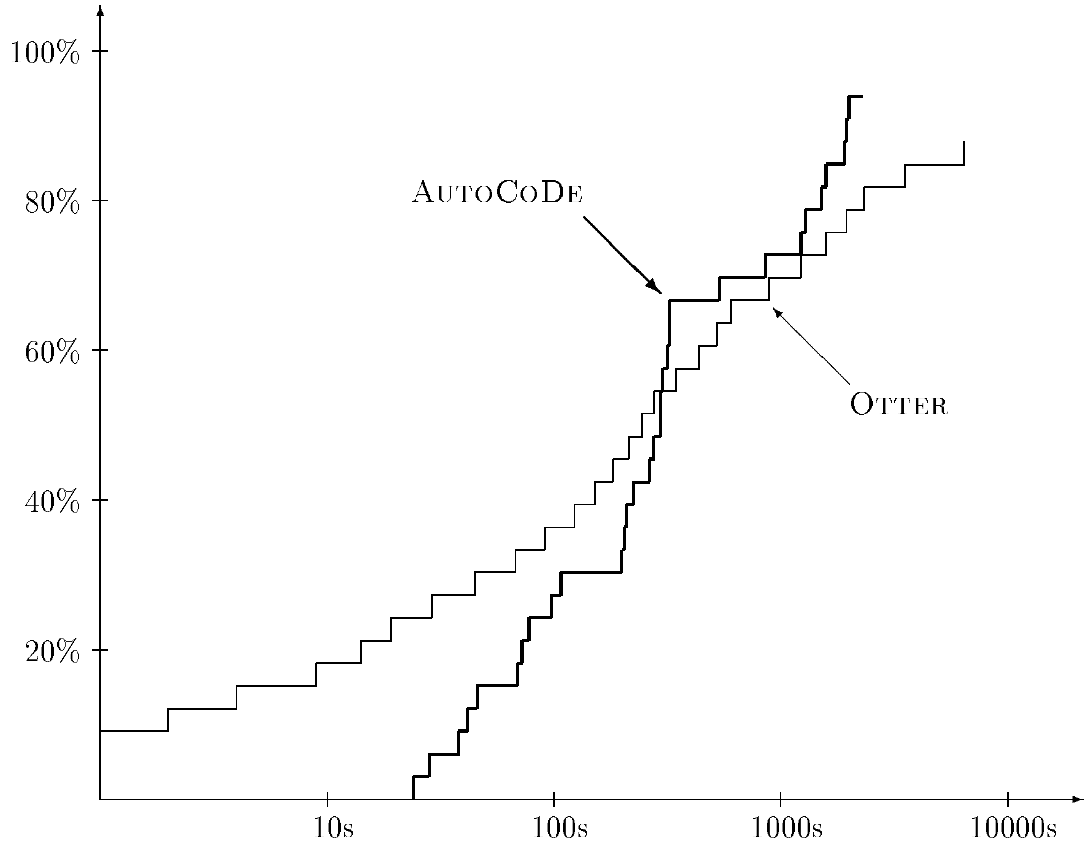

Figure 3: Performance of AutoCoDe and Otter for 33 problems of the CN calculus. Performance is measured in percentage of problems solved (y-axis) vs. accumulated run time (x-axis, logarithmic scale). AutoCoDe solves ca. 97% of the problems (32 out of 33) within 2358 seconds. Otter solves ca. 88% of the problems (29 out of 33—a subset of the problems solved by AutoCoDe) within 8098 seconds, again assuming very favorable conditions for Otter.

proof problems and pick up results. This essentially means that the crucial step in the use of a deduction system, namely choosing an appropriate search-guiding heuristic, must be automated.

We have also motivated why learning capabilities are indispensable to attain satisfactory performance. The necessary infrastructure for both automatically selecting a search-guiding heuristic and learning is provided in form of an environment embedding the "core" deduction system.

We have conducted a case study in the area of condensed detachment. A fully automated deduction system 'AutoCoDe' for problems of condensed detachment was created by integrating the core system 'CoDe' into the proposed environment. Our experiments with AutoCoDe have shown that *fully* automated deduction is not necessarily science fiction. Furthermore, the success and design of AutoCoDe underline that a "little pragmatism" can be beneficial and can make a difference. Our results demonstrate that fully automated deduction can be a serious competitor for automated deduction with user interaction, in particular when learning capabilities are present as it is the case with AutoCoDe. Naturally, certain (obvious) rules must be obeyed. For instance, problems of increasing difficulty should be posed. To put it another way, a learning (fully) automated system should not be confronted with hard problems without having had the chance to gain experience with simpler problems. But this probably goes without saying, since it is the general and reasonable way in which human researchers proceed.

References

[1] **Bourne, S.R.:** *The UNIX System*, Bell Telephone Laboratories, Addison-Wesley, 1982.

[2] **Brock, B.; Cooper, S.; Pierce, W.:** *Analogical reasoning and proof discovery*, Proc. CADE-9, Argonne, IL, USA, 1988, LNCS 310, pp. 454–468.

[3] **Denzinger J.; Schulz, S.:** *Learning Domain Knowledge to Improve Theorem Proving*, Proc. CADE-13, New Brunswick, NJ, USA, 1996, LNAI 1104, pp. 62–76.

[4] **Fuchs, M.:** *Learning proof heuristics by adapting parameters*, Proc. 12th ICML, Tahoe City, CA, USA, 1995, pp. 235–243.

[5] **Fuchs, M.:** *Experiments in the Heuristic Use of Past Proof Experience*, SEKI-Report SR-95-10, University of Kaiserslautern, 1995, obtainable via WWW at the URL `http://www.uni-kl.de/AG-AvenhausMadlener/fuchs.html`.

[6] **Fuchs, M.:** *Powerful Search Heuristics Based on Weighted Symbols, Level and Features*, Proc. FLAIRS '96, Key West, FL, USA, 1996, pp. 449–453.

[7] **Fuchs, M.:** *Experiments in the Heuristic Use of Past Proof Experience*, Proc. CADE-13, New Brunswick, NJ, USA, 1996, LNAI 1104, pp. 523–537.

[8] **Kolodner, J.L.:** *An Introduction to Case-Based Reasoning*, Artificial Intelligence Review **6**, pp. 3–34, 1992.

[9] **Łukasiewicz, J.:** *Selected Works*, L. Borkowski (ed.), North-Holland, 1970.

[10] **McCune, W.W.:** *OTTER 3.0 reference manual and guide*, Techn. report ANL-94/6, Argonne Natl. Laboratory, 1994.

[11] **McCune, W.; Wos, L.:** *Experiments in Automated Deduction with Condensed Detachment*, Proc. CADE-11, Saratoga Springs, NY, USA, 1992, LNAI 607, pp. 209–223.

[12] **Peterson, G.J.:** *An automatic theorem prover for substitution and detachment systems*, Notre Dame Journal of Formal Logic, Vol. 19, Number 1, January 1976, pp. 119–122.

[13] **Slagle, J.R.; Farrell, C.D.:** *Experiments in automatic learning for a multipurpose heuristic program*, Communications of the ACM, Vol. 14, Nr. 2, 1971, pp. 91–99.

[14] **Slaney, J.:** *SCOTT: A Model-Guided Theorem Prover*, Proc. IJCAI '93, Chambery, FRA, 1993, pp. 109–114.

[15] **Suttner, C.; Ertel, W.:** *Automatic acquisition of search-guiding heuristics*, Proc. CADE-10, Kaiserslautern, FRG, 1990, LNAI 449, pp. 470–484.

[16] **Sutcliffe, G.; Suttner, C.; Yemenis, T.:** *The TPTP Problem Library*, Proc. CADE-12, Nancy, FRA, 1994, LNAI 814, pp. 252–266.

[17] **Tarski, A.:** *Logic, Semantics, Metamathematics*, Oxford University Press, 1956.

[18] **Wos, L.:** *Meeting the Challenge of Fifty Years of Logic*, JAR 6, 1990, pp. 213-232.

[19] **Wos, L.:** *Searching for Circles of Pure Proofs*, JAR 15, 1995, pp. 279–315.