Frode Sørmo

Case-Based Tutoring with Concept Maps

Thesis for the degree doctor scientiarum

Trondheim, January 2006

Norwegian University of Science and Technology
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Computer and Information Science

NTNU
Norwegian University of Science and Technology

Thesis for the degree doctor scientiarum
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Computer and Information Science

ISBN 978-82-471-0444-6 (printed version)
ISBN 978-82-471-0458-3 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2007:21

Printed by NTNU-trykk

Abstract

The goal of this thesis is to investigate methods for computerized tutoring support that is adapted to the individual student. In particular, we are concerned with providing such assistance to students solving exercises in domains where a complete or accurate problem-solving model is infeasible. We propose to do this by using *concept maps* as a means for students to model their own knowledge. Combined with results from earlier exercises, the concept map can form a student model that can be used in exercise selection, conceptualization support, and exercise solving support.

The thesis presents a framework for comparing exercise-oriented intelligent tutoring systems, and uses this framework to describe and analyze earlier systems, as well as our own system, CREEK-ILE. The CREEK-ILE system includes a formalization of a knowledge representation designed to support reasoning with concept maps, and methods for using case-based reasoning with concept maps as student models. A partial implementation of CREEK-ILE designed to support learning of basic Java computer programming is presented. This implementation is used in an experiment to test if students' concept maps, created before an exercise, can be used to predict the students' level of competence on the exercise tasks. The conclusion of these experiments is that although there is a weak correlation, it is not strong enough to serve as a good basis for exercise selection. However, concept maps in student modeling is useful for other tutoring tasks, such as conceptualization support, vocabulary learning and as basis for explanations.

An initial qualitative study on the effect of using inference on concept maps is also performed. This study is done by using the concept maps drawn by students in the experiment, and shows that inference can reveal implicit knowledge in students' concept maps. We demonstrate how this implicit knowledge can be used in various tutoring tasks supported by concept maps, for instance by increasing the quality of concept map similarity measures.

Preface

This doctoral thesis is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree *Doctor Scientiarum*. It is organized as a monography.

The work described in this thesis was performed at the Department of Computer and Information Science (IDI), NTNU, under the supervision of Professor Agnar Aamodt. It was funded through the *IKT og Læring* (Information and Communications Technology and Education) program at NTNU.

Throughout this dissertation, there are many discussions involving students or teachers, as the topic of the thesis is related to tutoring systems. When these discussions concern the general roles of teacher and student, we will use female pronouns to refer to teachers, and the male pronouns to refer to students.

Acknowledgements

I would like to thank Assistant Professor Arvid Holme and University Lecturer Steinar Line for kindly allowing me to perform experiments with the students in their Java programming classes at IDI. Your enthusiasm and assistance were a great help.

My colleges have made the stay here at Department of Computer and Information Science (IDI) a pleasure. I have had many interesting discussions and good times with the people of the Section for Intelligent Systems (DIS), past and present. In particular, I would like to thank Diego Federici, with whom I shared an office, for showing restraint and not bringing the floor down by buying even more monitors, and Jörg Cassens for surviving all those conference trips with me, and rescuing me from that clothing store in Vancouver.

My thanks also go to Peep Küngas, Tore Bruland, Helge Langseth, Anders Kofod-Petersen and Jörg Cassens for proofreading my thesis. The remaining mistakes are all the result of me ignoring their good advice.

Last, my greatest thanks go to my advisor, Agnar Aamodt. Our discussions has been many and interesting, and his encyclopedic and almost uncanny knowledge of research literature has been invaluable. Without his guidance, this thesis would not have been possible.

Trondheim 13.11.06

Frode Sørmo

Contents

List of Figures

Chapter 1

Introduction

In the last decades, we have seen computers emerged to change how we work and play. The ability of computers in a network to provide easy access to information and information processing has left few occupations entirely the same. In education, computers have been introduced as administrative tools, for instance in maintaining student and employee records, and they have slowly been taken up by teachers and students as tools to help find and organize information. Students can, for instance, search for information for a school project on the World Wide Web, or type a report in a word processor. The increasing reliance on computers by society and employers has also led many to believe computer literacy should be a subject for education in school in order to ensure that everyone has the basic knowledge required to participate in tomorrow's work environments and civil society. However, in most educational systems students still get most of their exposure to computers through other venues than schools. The reasons for this are many, among them that schools cannot afford enough computers to give reasonable coverage to the student population, and that many teachers lack the knowledge themselves to use the computers effectively. Even if we imagine that a school had enough computers for everyone, and that the teachers at this school were computer savvy, it still is not clear how computers on every desk can help students to learn and teachers to teach. In language classes, a word processor might serve as a writing tool for reports, but does it help students to read and write better? In a math class, can they ease the task of learning how to solve equations? Is history, geography or science easier to teach somehow, with the help of a computer?

Many teachers are skeptical about the value of a computer on every student's desk, and argue that the computer does not assist in the primary task – learning, and as a tool the computer is more distracting than valuable. We might not entirely agree with this position, but recognize that there is an important point in this argument – in schools, computers are almost never used as tools for learning, but used in lower-level subtasks that are sometimes required in the learning process, such as writing a report. To many teachers, these subtasks are not seen as problematic areas in the first place, and they

thus question the wisdom in investing heavily in computer equipment to solve them . Problematic areas might be such things as classroom discipline, pacing the curriculum and providing assistance to students with a wide range of ability and prior knowledge.

It is certainly possible to create computer software that addresses the learning task directly – most computer software stores have a section dedicated to things such as spelling- and math-games, and interactive environments designed to help you learn a language. Such software is sometimes used in school as well, although it seems it is usually used ad hoc as an addition to the regular curriculum, and viewed almost as a break from the *real* learning. If computers can be integrated in the pedagogical strategy of the teacher, and they usefully address a problem of institutionalized learning, it seems their usefulness would be much greater. It is not apparent to us how computers may help with classroom discipline or social problems, and deciding the overall curriculum is probably best left to the teahcer, but it seems computers may be useful in providing customized support to students. While students differ in their prior knowledge and ability, teaching cannot be tailored to the ability of every single student as there is usually only one teacher per class. Typically, students ahead of the class are bored and the students lagging may be left behind.

The research on *intelligent tutoring systems* (ITS) attempts to make systems that address this problem by having the computer program assist the student in a way tailored to that student's ability. Teachers have always done this, but because they have many students in a class, their time with each student is limited. Intelligent tutoring systems attempt to be the student's own teaching assistant, always there if the student needs help or is stuck. This is typically done by using artificial intelligence techniques to actively model what the student knows so that the system can, to some degree, understand how the student thinks and where he might go wrong. Some ITSs even actively monitor problem-solving by students and pro-actively take the initiative to correct mistakes and clear up misconceptions.

In this research, we address intelligent tutoring systems that are used to aid students in solving exercises. In particular, we would like to examine how this may be accomplished in domains that are not well enough understood to make it possible or feasible to model the problem solving activity completely. This means that while the ITS itself may be unable to solve all problem in this area, it is still expected to provide support customized to the individual student.

A classic approach to exercise support attempts to form a cognitive model on how a student solves (and is supposed to solve) problems by using, for instance, rules in a manner similar to rule-based expert systems. If an accurate model of the student's as well as an expert's reasoning process is available, advice can be formed by looking at how the student's thinking varies from the expert's. This is generally known as *student modeling*. This knowledge about the student allows the system to tailor the tutoring to the individual system – a capability that has generally been seen as what separates ITS systems from other computer-assisted education software.

In order to customize the experience for the individual student, the tutoring

system must have some knowledge about the competence of the student even if it is not complete. It is possible to avoid modeling the complete problems solving process of an individual and still have some knowledge about the competence of a student. Indeed, this is done extensively in education – for instance, most standardized tests administered in schools (e.g. multiple choice tests) do not consider the problem solving process, only the answer given by the student. However, many ITS systems wish to diagnose faults in the student's reasoning process, and for this purpose, answer evaluation is often insufficient. On the other hand, complete student modeling is either impractical or intractable in many domains [71]. Indeed, many domains lack good models of expert behavior, which is a prerequisite.

This thesis will examine if it is possible to support exercise solution without a complete domain model by approaching the task both from above and below in terms of abstraction. This means that instead of modeling accurately how problems are solved, a more abstracted, conceptual model may be used to support high-level tasks such as exercise selection and vocabulary formation. On the lower level tasks, such as assisting a student while solving the exercise, episodic knowledge such as concrete problem-solving traces may be used. While none of these models contain a complete model of how problems are solved, together they may assist the student in forming such a model, and customize the tutoring experience while doing so. In many domains, these models are also much easier to acquire.

On the conceptual level, this thesis suggests using concept maps [56] as an approach to allow the student to model his own knowledge on a topic. Concept maps (sometimes known as mind maps or topic maps) are graph-like structures where concepts are represented by nodes, which are related by labeled links. Concept maps have been shown to correlate highly with multiple choice tests when used as assessment tools, but the use of the approach in ITS has been limited, especially in student modeling. The reason for this may be that the use of assessment tools such as concept maps or multiple choice tests does not focus on the procedural level. This makes it harder to diagnose procedural problems and clearly makes it harder to suggest procedural improvements. On the other hand, the approach does not require a complete model of the domain, and allows the student greater flexibility in constructing his own model. We believe that this means that while concept maps may be useful for higher level tasks such as suggesting exercises or explaining the difference between two concepts, they are likely of limited use in lower level tasks, such as helping a student that is stuck while solving an exercise.

The lower level tasks are generally harder to support without a more complete model of the domain, but we suggest that advice may be provided if the system already knows one or more solutions to the concrete problem faced by the student. If the problem is diagnostic in nature, the system may for instance suggest important features or a set of possible diagnoses. If it is a planning problem, the system may offer partial plans that help the student. These measures of assistance can be offered even if the system does now know how to solve the problem from basic principles, because it can reuse the cases where

the problem has been solved before. This is not an unlikely scenario in tutoring, as many students typically face the same problem, and there may also exist one or more solutions made by teachers or experts. An obvious limitation of this approach is that the system is not able to provide customized explanations for the reasoning behind the advice it offers as it has not done this reasoning itself.

The goal of this thesis is to examine how to support learning tasks where a procedural level cognitive model is unavailable. Specifically, we attempt to model student conceptual knowledge in the form of concept maps, and episodical knowledge in the form of cases, and see how these knowledge sources can form a basis for providing tutoring support that is tailored to the student. This also has the advantage of tying together exercises with the more theoretical conceptual knowledge.

1-1 Background

1-1.1 AI in Education Software

The most straight-forward form of computer-aided instruction (CAI) may be as simple as placing the contents of a textbook in a computer. This gives some benefits in that the computer medium is less restricted in the kind of content it can deliver than a standard book – for instance, a computer textbook may contain sound and video that are not easily delivered by a book. The computer also allows the experience to be more interactive and differently structured, as for instance in hypertext. Quite sophisticated tools have been commercially available for some time to make such "electronic textbooks". These tools assist authors in creating what can be quite complex and impressive content, and they may also assist teachers in selecting and composing smaller instructional units into a whole course.

The "electronic textbook" approach to computer-assisted teaching can be very useful, but these systems are still like textbooks in that they are static tools that deliver a very similar experience to all students. There is usually little differentiation between how material is presented to students with different aptitude or background knowledge.

In contrast, the goal of intelligent tutoring systems (ITS) is to provide tutoring that is tailored to the specific student by specifying what to teach and how to teach [89]. This is typically done by using some kind of model of the domain of discourse (the domain or expert model), a model of the learning process (the pedagogical model) and a model of what the system believes each individual student currently knows (the student model). These models take many forms, and in actual systems may not be so easy to separate or identify, but may be seen as a knowledge level construct [54] of what kind of knowledge the system should have in order to tailor the learning.

The teaching philosophy underlying the intelligent tutoring system approach is in its purest form based on an idea of *knowledge communication* – that is, the process of teaching is that the knowledge of the expert (represented by the

expert model) should be transferred to the student through a communication channel. However, the communication channel is not perfect, and as in all communication, misunderstandings and miscommunication appears on both ends of the channel. Because of this, it is important that the tutoring system attempts to identify these misconceptions and correct them, something that can only be done if the system actively tries to model what the student knows. This view of learning is based on a pedagogical approach often called *instructional design*. The name refers to how school subjects traditionally have been made through explicit design of a curriculum and teaching methodology down to a low level of detail, either by course material such as textbooks and exercise sets or by teachers. This idea that the teacher or some other expert should decide what should be learned is an essential property of this approach. Even the earliest intelligent tutoring system follows this approach implicitly, but it is perhaps clearest formulated in "Artificial Intelligence and Tutoring: Computational and Cognitive Approaches to Knowledge Communication" by E. Wenger [89].

In the last decades, the instructional design approach has been critiqued by proponents of *constructivism*. The constructivists reject the view that learning is communication of knowledge, and claim that a better model of learning is that the learner himself constructs a model of the world that is somehow useful to him, for instance in solving tasks the learner is faced with. The constructivist approach is fundamentally skeptical to the implicit claim of the universal "expert model" that can be learned and usefully applied by anyone. Constructivists often suggest that learning should happen in an *apprentice* setting, where the learner sees not only why something is useful to learn, but learns it in a context of real problem solving. In institutional education, this critique has resulted in less focus on rote learning and more on forms of learning such as project learning that attempt to highlight the real world context and focus on problem solving skills. In computer tutoring systems, this critique has also coincided with a realization that highly accurate student modeling is intractable in many, if not most, domains [71]. There has been some very impressive successes using the traditional ITS approach (e.g. the PACT Cognitive Tutors [13] and ELM [86] systems, both reviewed in Chapter 4), but these are limited to well understood and rather limited domains with formalized languages, such as mathematics or computer programming. Many, if not most tasks, are not naturally limited to formal languages and have complete theories on how problems are solved. For instance, many skills rely on natural language understanding and formalization. This has lead many to a somewhat less ambitious approach to AI in education software, allowing more room for the student to explore and learn in a learning environment while providing some level of support. For instance, the CATO system [7], operates in the domain of law, specifically arguing court cases. This system does not even attempt to model the student's knowledge, but provide a environment where the student can explore, and the system is able to dynamically produce arguments that serves as examples for the student that illustrates the structure of the argument. These systems do use artificial intelligence techniques – CATO uses techniques based on case-based and model-based reasoning to produce legal arguments – but they typically place more emphasis on provid-

ing a learning environment than a knowledge transfer channel. Reflecting this, these systems often do not describe themselves as intelligent tutoring systems (ITSs), but rather as *interactive learning environments* (ILEs). The approach taken in this thesis is between these two. While we wish to address domains where strong domain theories are not available, we believe that there is still something to gain from using some level of student modeling to support learning tasks where possible. In the end, our approach is closer to the ILE appraoch, but with aspirations of adopting some ITS strategies.

In the wake of the constructivist critiques of instructional design, the value of *learning by doing* has been emphasized. This follows from the constructivist belief in apprentice learning, where the student is faced with real problems but assisted by an expert. This is often contrasted with the book learning of instructional design. However, this is not an entirely fair duality. On some level, learning by doing has always been present in school, such as by solving exercises in mathematics and practicing language skills. There is however a fundamental difference between the instructional design learning by doing and constructivist learning by doing. The traditional exercises had as a goal to test and operationalize pre-defined knowledge, while the constructivist learning by doing is about allowing the student to form his own learning goals and motivation by providing tutoring in a real-world context. Constructivists use the term learning by doing strictly in this latter sense. However, the term has been picked up and used by proponents of more traditional methods as well, and in a broad sense, learning by doing can today be said to contain a spectrum from traditional exercises to situated learning. This spectrum of learning by doing approaches is also present in tutoring systems, and in Chapter 2, we survey the tutoring systems that use AI techniques and operate within this broad view of learning by doing.

1-1.2 Concept Maps

Concept maps (sometimes called topic maps or mind maps) were originally designed as an educational aid to assist students in organizing the concepts in a limited domain by connecting them with labeled links [56]. They have since been widely used on many levels of education from elementary school to university studies as an aid to help people conceptualize and organize their knowledge. The concept maps looks similar to semantic networks (see Figure 1.1 for an example), but while the goal of a knowledge representation in the AI sense is to establish a common representation between human and computer [62], the concept map is primarily meant as an aid in human learning, organization and communication. Even when concept maps are represented through a computer tool, the computer is not typically expected to understand the contents any more than a word processor would understand a document. The word processor may provide helpful tips (such as on spell checking in a word processor), but it is not expected to reason over the contents.

Various styles of concept maps have been used and evaluated in a wide variety of settings to present, organize and assess information. The technique

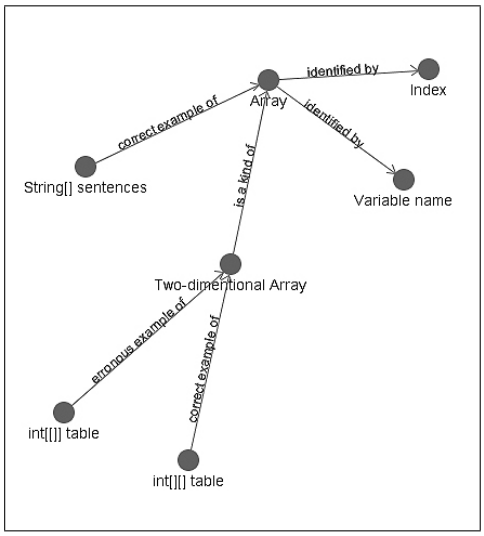


Figure 1.1: A simple concept map: "Arrays in Java"

has also been used to support performance in business and government, for instance in assisting knowledge management. For instance, Leake and Canas et.al. [52, 47] have developed a case-based approach to use concept maps in knowledge management. In this thesis, we will focus on using concept maps in a tutoring context, but recommend [53] for a wider survey.

Concept Maps as Tools of Assessment

Concept maps were introduced by Novak as a way for students to organize their knowledge about a particular topic in a free-form way [56]. Although some simple rules is presented to limit the complexity and to help the student in structuring the concepts, the student is essentially free to form any concepts and links. Because a student's concept map is an expression of the student's knowledge, Novak also suggested that the maps could be used to assess the student's knowledge about a domain. The original proposal from Novak was based on an expert (teacher) examining the concept map and awarding points based on structure, inclusion of relevant concepts, relations and examples. This and later point-based scoring techniques provide guidelines, but they depend to some degree on the judgment of the evaluator and are hard to automate. The completely free-form approach may well go some way towards supporting the constructivist theory of education as it allows the student to form different conceptualizations of the domain, but it also means there are few ways of constraining the student to the domain we wish to evaluate.

In order to guide the students towards modeling the intended topic, it is normal for concept map based assessment tools to use a less free-form approach to mapping. This may range from concept maps that are almost complete where

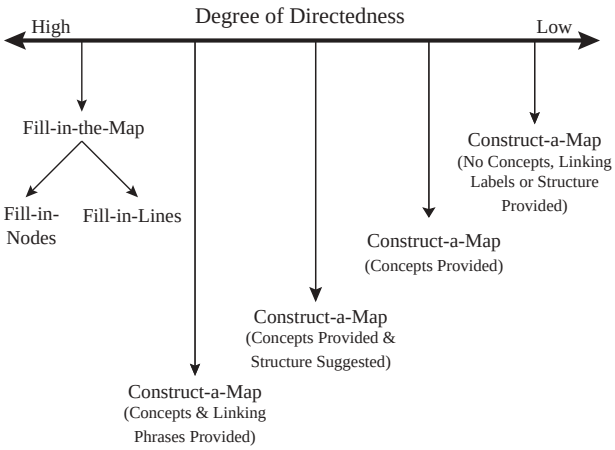


Figure 1.2: Concept map techniques according to directedness of the mapping task (adapted from [65, p.2])

the student's task is to fill in missing links, link names or concept names, to simply providing hints about the central concepts in the domain. Ruiz-Primo [65] identifies a scale from low to high directedness in the approach to concept mapping (Figure 1.2).

The computational approaches to assessing concept maps tend to gravitate towards the high directedness end of this scale. The most obvious reason for this is that it is hard to automate the assessment of the concept map by computer when the student is not constrained in some way. While constrained maps do not necessarily make it possible to computationally "understand" the maps on a semantic level, it makes it easier to compare concept maps syntactically. This allows a computer system to compare concept maps created by students to a teacher or expert map and thus grade it on the similarity to this map. Researchers from CRESST (Center for Research on Evaluation, Standards and Student Testing) have investigated computerized techniques using this approach [58]. In their approach, the teacher will first draw a concept map for a topic, and then the concepts and linking names are extracted from this map. The students are then asked to form a map using the same concepts and link-names, which greatly simplifies computational scoring. This method corresponds to the Construct-a-Map (Concepts & Linking Phrases Provided) on the middle of the degree-of-directedness scale in Figure 1.2.

Concept Maps in ITS

Concept maps have been used and tested for many educational tasks, and there exists quite a few computer tools to assist in building and maintaining concept maps both for the individual and the organization. However, the uses of concept maps in intelligent tutoring systems have been fairly limited. There has been a

trend in ITS towards transparent student modeling, that is making the student model accessible to the student. Some systems have used concept maps or concept map inspired structures to help do this.

The ViSMod [93] system visualizes a student model based on Bayesian belief networks as graph-like structure. However, in this system the student himself does not create the graph, and as it is a visualization of a Bayesian belief net, it has semantics tied to it that a concept map lacks. The same authors have created the ConceptLab system [94], which is a computer-assisted educational system for collaborative creation, browsing and learning concept maps. While ViSMod is a tool for visualizing a student model in a more traditional intelligent tutoring system, ConceptLab is more of a constructivist tool that underlies the educational activity – it does not assist the student or teacher beyond the activity of creating concept maps. In this regard, it is similar to the concept map based knowledge management tool by Leake and Canas et.al. [52, 47], but used in an educational context.

It has been shown that constrained concept maps may be used as assessment tools for conceptual knowledge much in the same way as standard multiple choice tests. This suggests that it may also be possible for intelligent tutoring systems to use student concept maps to assess the level of knowledge of a student. Approaches reminiscent of multiple choice tests have been used for this purpose for in computer-aided learning with the goal of seeding student models, but concept maps have not been similarly used. We believe concept maps should be possible to use to assess a student's conceptual knowledge in an intelligent tutoring context, with the added benefit that the concept maps constrain the student's expression less than a multiple choice test. This has both positive and negative sides. On the positive side is that this holds closer to a constructivist ideal where the student is allowed greater freedom in expressing and modeling his beliefs. On the negative side, this extended freedom of expression clearly makes it harder to interpret information that can be used by the system. In order to use concept maps as student modeling tools, a balance between expression and interpretability must be kept.

1-1.3 Case-Based Reasoning

In case-based reasoning (CBR), concrete, episodic experience from the past is used to solve similar problems in the future. The reasoning process consists of retrieving a set of similar cases to the current problem, reusing the solutions of these cases and revising them to fit the current problem. The new problem can the be retained in the case base as a new case, to be retrieved to solve future problems [4].

A major advantage with CBR is that the method may be used in open domains where problems cannot be solved through a well-described procedure or algorithm in a computer. This may be because the domain theory is not accurate enough for this kind of automation, or because the effort in knowledge acquisition would be prohibitive. While CBR is presented by many as a way to limit the knowledge acquisition effort, it does require a certain effort in this

direction in order to find a good set of initial cases (case acquisition), a way to compute the similarity between cases (similarity measure), how to represent the problem well (case representation) and how to revise retrieved cases to a slightly different situation (case adaptation). However, the episodic nature of cases allows them to capture tacit knowledge that has been harder to acquire and capture in, for instance, rules.

An important distinction among CBR systems is the degree to witch the system is knowledge-light or knowledge-intensive. In this distinction, knowledge is meant to signify generalized knowledge, typically managed and stored by a knowledge representation system such as description logic or a frame-based system. Although the cases, similarity measure and adaptation procedure certainly contain knowledge, the cases are typically limited to episodic knowledge and the knowledge contained in the similarity measure and adaptation procedure are often embedded in program code. On one extreme end of the knowledge-light scale, we find the instance-based methods such as the k-nearest neighbors algorithms [5], and on the knowledge-intensive side we have systems that rely as much on generalized knowledge as cases, such as SaxEx [17] and CREEK [3].

CBR in Tutoring

Case-based reasoning has been used in a wide variety of intelligent reasoning systems. They are typically used as part of the student modeling process and to assist the student in exercising some kind of operational skill. In some domains, such as law and medicine, cases are an important part of the normal problem-solving method. It makes sense to use a case-based approach to teach these subjects, especially when assisting the student in solving exercises. The CATO system [7, 8], helps the student form legal arguments by using a domain model to find appropriate earlier cases to draw analogies from. The ELM system [86] uses general knowledge about the domain (LISP programming) with episodic knowledge about past student programs to identify errors and suggest corrections. These systems are not only able to help the student find a solution to the exercise he is trying to solve, but also help the student to identify particular problems he may have in solving the problem. While a knowledge-light case-based reasoner also may be able to find a good solution to a problem, the addition of a general domain model (e.g. the "Factor Hierarchy" in the CATO system) increases the system's ability to help the learner understand why a past case is relevant. These systems have a knowledge communication strategy that allows the student to learn not only the particular solution to one problem, but is also able to ground it in the theory of the domain.

Case-based reasoning can also be used in the student modeling part of an ITS. The SARA system teaches a case-based reasoning approach by asking the student to find and adapt similar problems, but also uses CBR in the student modeling process. Typically, the approach in case-based student modeling is to attempt to retrieve cases representing students that has already finished the course that are as similar to the current student as possible. This allows the system to reuse experience from the earlier student, for instance by suggesting

exercises the earlier student found challenging but not impossible [70].

Certainly, case-based tutoring seems a good fit to support problem-based learning, where the idea is to expose the student to concrete problems in the domain. Boylan et. al. [23] suggests that case-based tutoring systems also come closer to supporting a more self-directed and constructivist approach to learning than traditional ITS systems. Their BLITS system teaches business letter composition through annotated examples instead of algorithmic rules. While the system does its best to suggest (through examples and annotation) that certain elements (such as a greeting phrase) should be present in such a letter, it is up to the student to compose the letters. Case-based reasoners can also make it possible to build intelligent tutoring systems in domains where it is hard or impossible to build a model that can solve any problem in the domain. This is also illustrated by BLITS, which itself is not able to create a business letter, but its previous cases allows it to help the student by displaying similar examples.

1-2 Research Goals

1-2.1 Student Modeling in Weak Theory Domains

The major motivation of the work in this thesis is to develop methods for exercise support in weak theory domains, where the method does not require a complete cognitive model of how problems in the domain are solved. By complete cognitive model, we mean that the computer can accurately solve new problems within the problem space covered by the system and understand how and why errors in reasoning from a student may arise. Successful systems that rely on procedural-level cognitive modeling have been and are developed (see e.g. the PACT Cognitive Tutors reviwed in chapter 4), but for many domains cognitive models are infeasible or too expensive to create. Systems that rely on a complete procedural-level mode are also typically very inflexible in their view of what to learn – there is really no room for the student to use different strategies or conceptualizations from what are contained in the system. Avoiding the cognitive procedural model, the effort of building a tutoring system can be much lower, and systems can be built for areas that are not covered today.

Our hypothesis is that this can be done by focusing the student modeling effort on the conceptual and episodic levels instead of the procedural level. This also serves the goal of connecting two learning tasks that too often are learned seperatly. Case-based tutoring systems are usually learning by doing systems, which focus on the operationalization of skills in the student. For instance, in teaching programming language, the focus is on writing programs to solve concrete problems instead of learning how a for-loop relates to a while-loop. There seems to be a disconnect between the systems that focus on the operational skill and those that focus on instruction of concepts. We aim to bridge this gap by introducing concept maps as explicitly conceptual models, but used in an exercise-oriented context.

In order to bridge this gap and be a successful student modeling tool, concept mapping must be integrated in the pedagogical strategy and have a format that serves specific goals. It is a goal of this research to develop such a methodology, where concept mapping is a integral part of the system rather than an ad hoc component.

1-2.2 Concept Maps as Student Models

The research in the use of concept maps as assessment tools show that they work well in assessing a student's conceptual knowledge. We suggest that this means that they may also be useful in student modeling in intelligent tutoring systems. Using concept maps in this manner has the advantage that the student would model his own knowledge, and can express his own conceptualizations of the domain. However, if the concept map is to be useful as a student model for the computer, the computer must also have some way of using the information in a map. Traditionally, concept maps have been seen as tools for human expression – even when computer tools are used to create maps, it is not expected that the computer understand the knowledge contents of the map. Some tools offer some utility for comparing maps and introducing concepts from other maps based on syntax and structure. In our use of student maps, we will such syntactical approach, for instance by calculating the similarity between a student's and a teacher's concept maps, and using case-based reasoning or other machine-learning methods to find similar students based on their concept maps.

In particular, we will examine if there are information in students' concept maps that allow us to predict how well the students solve more practical exercises, such as programming tasks. Although concept maps may be used to assess conceptual knowledge, it is currently an open question if concept maps can be used to assess procedural skill. We have examined this by performing an empirical experiment where first year Java-programming students were asked to form concept maps and then solve programming tasks. We will present an analysis with statistical methods, and experiments using machine-learning methods to see how well information from student concept maps can predict competence on the practical tasks.

1-2.3 Concept Maps, Knowledge Representation and Inference

Concept maps are designed first and foremost as mediums of human expression and communication. Knowledge representation languages are designed for this purpose as well, but also seek to establish a common interpretation between computer and human of the knowledge stored in the representation. The cost of doing this is that human expression in a knowledge representation language tends to be more difficult, and require training

We will examine if it is possible and useful to provide some level of machine-interpretable semantics for concept maps so that the system may do some inference over the contents of the maps. In particular, we will examine if path-based

inference (such as inheritance) used in semantic networks may be used without compromising the ease of use and naturalness of expression that is such important properties for concept maps. To do this, we will describe formally the CREEK knowledge representation and inference mechanism, and how it may be used as an inference mechanism for concept maps. The CREEK knowledge representation is then used as the underlaying representation in an experiment. Using the data from this experiment, we will study on the effects of inference and how it can be used.

1-3 Method of Investigation

Several methodical approaches will be used in this thesis. In chapters 2, 3, 4 and 5, an analytical approach will be used in defining a framework for comparing and describing intelligent tutoring systems for exercise support. This framework will be used to analyze existing theories and systems (chapter 4) and describe our own theory (chapter 5).

In chapter 5, we also present the CREEK knowledge representation and inference mechanisms in a formal framework based on set theory, which we use to show how concept maps can be represented in this knowledge representation, and inference used on them.

In chapter 6, we describe a partial implementation of a tutoring system to support solving Java programming exercises for first year computer science and engineering students. Using this system, some quantitative and empirical studies are done on how concept maps may be useful as student models. We also use the data from these experiments to do a qualititative study on the effects of inference on real-world concept maps.

1-4 Organization

In chapter 2, we review computer-assisted approaches to learning-by-doing, as well as critiques of classical approaches to intelligent tutoring, both from members within the research community, and from proponents of constructivist learning. Chapter 3 presents a framework for analyzing and comparing learning-by-doing intelligent tutoring systems. This framework will focus on case-based tutoring (in the sense of exercise-oriented systems). In chapter 4, we use this framework to describe and analyze existing learning-by-doing systems. In chapter 5 we describe our own approach, the CREEK Intelligent Learning Environment (CREEK-ILE), and in chapter 6 a partial implementation of this system as well a few experiments where we attempt to answer the questions posed in the motivation section above. Chapter 7 contains a conclusion and some thoughts of areas of future research.

Chapter 2

Learning by doing in ITS

The learning by doing method has been used in intelligent tutoring systems almost since the inception of the field, at least in the broad meaning of the term encapsulating everything from exercise support to situated learning. In this chapter, we will examine the history of learning by doing in intelligent tutoring systems, and examine in what kinds of systems learning by doing is done today. Last, the constructivist critique of traditional instructional design methods will be reviewed briefly with a view towards what this means for learning by doing, especially in tutoring systems.

2-1 Early Systems

2-1.1 Simulation environments

The first learning by doing systems in ITS were based on *simulation environments*, where the system would provide an setting where the student could manipulate a simulation and get feedback on how the manipulation worked. These systems are perhaps the closest to the constructivist ideal of situated learning, as they typically provide an environment for the student to experiment in as a basis for the tutoring.

The first such system was SOPHIE-I [27, 28], which simulated electric circuits. This system did not use a student model or pedagogical model, but simply provided a simulated lab for the student to try different tasks. In SOPHIE, the student was typically faced with a circuit with some kind of problem, and his task was to diagnose it. SOPHIE-I could answer both factual and hypothetical questions about the circuit, and could help the student evaluate his hypothesis. Later extensions of SOPHIE would also include a demonstrator system in SOPHIE-II [26] and an attempt at modelling the physics of the electronic circuits in mental models closer to what is used in human reasoning in SOPHIE-III [29]. One of the major challenges of this project has been to find the correct level of describing the simulation expertise. The goal has been to find a level where the simulation is accurate, but also remain close to the mental models used

by humans. This is important both for explanation and in order to facilitate student modelling. Another early effort in this direction was the STEAMER system [43], which simulated steam plants on large ships. Unlike the SOPHIE systems, STEAMER used a graphical view of the propulsion system, and allowed the engineer student to manipulate various parameters and experience abnormal situations. Like the SOPHIE-I system, STEAMER did not have a qualitative model of the system it simulated – it was done as a mathematical simulation. In that sense, neither of these two systems had an internal representation of the model it was trying to teach. This means that neither of these systems could provide explanations based on qualitative terms used by people, such as causation. This lead to an effort from researchers working on both these systems to create theories of the qualitative models used by humans to solve such problems. The result of these efforts are the Qualitative Process (QP) theory [37] from the STEAMER group and ENVISION [36] from the SOPHIE effort. These theories attempt to model causality on an abstraction level used by humans in order to facilitate explanations (knowledge communication) and student modelling.

The QUEST system [90] was based on the lessons of the SOPHIE and STEAMER systems, and was based on mental models designed to be similar to those found in people. QUEST was also a system that simulated electric circuits, but did so using a progression of qualitative mental models. QUEST was based on a view that the full expert model should not be introduced to novice students, but that a progression of mental models from basic to advanced should form the basis of the instruction. This meant that the QUEST system started with a very basic model that would only perform well on a subset of the electric circuits, and as the student mastered a level, a more complex model would be introduced. This would be repeated in steps until this model would be expanded until the student had reached the full mental model. The goal of the system was to enable these transitions by giving problems that would lead the student to master the current level. However, since the system contained a complete simulation of circuit physics, the student could also choose to explore on his own by building circuits and receive feedback and explanations. All in all, the qualitative model of QUEST allows it to be very flexible and offer both exercise-oriented learning that is targeted at advancing the student's knowledge and free-form exploration modes, both supported by feedback and explanation capabilities at the causal level.

These early systems have lead to a wide range of simulation systems, using both quantitative and qualitative models. In Section 2-3 simulation environments are listed as on the areas of activity that have reached the level of multi-purpose authoring systems being available.

2-1.2 Exercise-Centric Tutoring

GUIDEON [34] is a tutoring system based on the MYCIN expert system [33], which was designed to diagnose medical problems. In doing this, GUIDEON uses a case method in the sense that the student is presented with a problem case

– a patient with some disease. Once the problem case is described by the system, the student is asked to diagnose the patient. To do this, he can ask questions to the system, which are answered by the MYCIN expert system. The student may also offer hypothesis and receive feedback on these from the GUIDEON. The system may also intervene if it believes the questions asked by the student are suboptimal or irrelevant. This kind of exercise-centric approach was also used in the problem-based mode of the QUEST system, and many other systems that use concrete problems as starting points. These systems are learning by doing systems in the sense that they are bottom up – they start with problems, assume the student has some basic knowledge on how to solve them, and offer assistance if they are not up to the task. This does not mean that the systems see the set of problem cases as the knowledge content that they wish to transfer to the student. Rather, they use the cases as vehicles to communicate more abstract knowledge. In QUEST, this knowledge is the progressively complex mental models, and in GUIDEON there are the (generalized) rules in the MYCIN rule-base.

In many ways, the PACT cognitive tutors follows in the tradition of these systems in that they are exercise-centric and rule-based. However, just as the simulation environment went from quantitative models that were accurate but not good representations of the cognitive processes used by people to representations that are more based on the reasoning in humans, the PACT cognitive tutors use models that are primarily cognitive representations of human reasoning. This contrasts with GUIDEON, which is based on the MYCIN knowledge base, which is primarilly created to solve problems in the domain. This means that case-based reasoning is not necessarily a model for reasoning in this kind of exercise-centric tutoring. The example-centric tutoring approach is using cases as a method of teaching, while case-based reasoning is using it as means to solve the problem.

2-1.3 Student Modelling

Student modelling is not specific to learning by doing – it is used within the full range of intelligent tutoring system, but it is an important component in many learning by doing systems.

The original and still most common form of student model is the overlay model, which for each student has an overlay over the expert model to record what parts of the expert model a particular student is familiar with. This idea was first introduced with the WUSOR-II system [31], although elements of this goes back to the very beginning of intelligent tutoring systems starting with SCHOLAR [30]. The overlay approach has the limitation that it only allows the system to recognize what subset of the expert model a student knows. If the student has misconceptions about the domain – "bugs" in his or her knowledge – this cannot be represented in an overlay model. To account for this, the expert model is often extended with bug libraries of common misconceptions so that the system may identify and record any misconceptions a particular student may have. In this approach, the student overlay model covers both the correct knowledge and the bugs so that the system may record both the units of knowl-

edge the student is familiar with and the misconceptions (bugs) the student has. This approach was introduced by the BUGGY project [25], which attempted to model the task of place-value subtraction so that all misconceptions could be identified as variations of the correct procedure and represented explicitly.

These student models can be used both in learning by instruction and learning by doing systems. In learning by instruction they can for instance be used to customize a lesson plan so that a student is introduced to subjects he is not already familiar with. In learning by doing systems, the goal is often to have an executable student model. This means that if the expert model is a set of rules to perform some task, the student model may be the set of rules known by the student (the overlay model). In addition, erroneous rules believed by the student may be added (the bug model). If the student model is perfect, the ITS system should be able to apply these rules to the problem and go through the same steps to reach the same conclusions as the student modelled.

2-2 Case-Based Cognitive Modeling

In 1977, Schank and Abelson [67] published a theory on how human procedural skill is represented internally in the form of *scripts*. For instance, someone with experience with restaurants would have an internal representation that a visit would involve waiting to be seated, then given a menu, expected to order, eat and at last pay. These scripts allows people to predict and prepare for events in the future, which among other things allow for planning how to reach goals. This theory was later extended in "Dynamic Memory" [69], which breaks scripts into smaller parts and organizes them in conceptual units called *MOPs*. This allows for the reuse of partial scripts in different settings, such as paying for food in a restaurant and for clothes in a mall.

MOPs can represent concepts such as a *restaurant visit*, *fancy restaurant visit* and *fast-food restaurant visit*, which can be organized hierarchically and share some elements (such as eating food) while containing specializations or differences (such as paying before the meal and seating yourself in fast-food restaurants). In addition, specific episodes, or cases, would be stored whenever they deviated unexpectedly from the script. For instance, Schank mentions going to a restaurant that would not fit the category for fast-food restaurants generally, but he was expected to pay before the meal was served. In these cases we tend to remember the exceptions, and Schank suggested this is because episodes are remembered when they do not confirm to expectations. These cases are initially stored as exceptions to well-known MOPs, but if there are repeated, similar exceptions, they are generalized and new MOPs are formed.

In this theory, cases are not merely training examples that allow students to operationalize what they have been taught through other means, but the main vehicle for driving learning, not only of procedural skills, but the formation of conceptual knowledge (MOPs). In this sense, this theory is a strong proponent of learning by doing, because it suggests that learning is bottom up from episodes (cases) to MOPs. One of the early systems based on this theory was Celia

[63], which models case-based apprentice learning. The system takes the role of the student, which attempts to follow the teacher's reasoning in solving a problem, and storing new cases (which may form new MOPs eventually) if the teacher does something unexpected during problem solving. Eventually, the system learns to predict the problem solving steps taken by the teacher, and thus learns to solve problems in the domain. Celia is not a tutoring system as such, but it can (and has, in Ceclia [64]) been used as a component in case-based tutoring systems.

Based on the dynamic memory theory, Schank has more recently developed a theory for education and learning [68, 66]. He points out that in learning by doing it is not always obvious what *doing* means. History, for instance, is taught in school and learned in other ways, but it is not clear what it means to *do* history. Even in subjects that clearly involve doing, such as in mathematics where students routinely train by solving exercises, one cannot really talk of doing mathematics in the general sense. One can do addition or solve algebra equations, and both these activities are contained in the greater field of mathematics, but there is no single activity that can describe the whole field.

If we examine the meaning of the word *skill*, it is fairly vague in day-to-day language. Knowing how to program the VCR can be a skill, but we also talk about skills such as driving a car, human relations and mathematics. While programming the VCR can conceivably be broken down into a set of steps (a mini-script), skills at mathematics, driving and human relations are clearly not of this type. Schank suggest that such skills are packages of related mini-scripts that in themselves are not worth mentioning, but together are important because they serve to achieve some goal. For instance, driving can be seen as a combination of mini-scripts to break, start the engine, look in the mirror before turning and so on. Individually, each of these mini-scripts can be described by a set of concrete steps, but they do not really serve any goals unless they are combined. In other words, mini-scripts (procedural skills) are arranged in skill packages that are interesting because they serve some goal. In learning, the overall goal serves as the motivation to acquire the set of mini-scripts that allows the learner to perform the task. This, Schank argues, is how learning happens in day-to-day situations, and could perhaps be described as *natural* learning. It is acquired through repeated exposure to episodes (cases) that are retained and generalized because they allow the learner to reach some goal.

However, in institutionalized learning, the focus tends to move away from *doing* to a meta-level where the doing is discussed and organized. For instance, physics is often taught by teaching students theories of, for instance, gravity by explaining such concepts like force, mass and acceleration. After this is done, experiments may be conducted, but they are typically described by the curriculum and predetermined to illustrate the truth of the theory. The micro-script level skills associated with conducting physics experiments and learning from them are not the focus of the education – the theories of Newton, Einstein and so forth are. This, claims Schank, robs the student of the motivation for learning and also of the micro-script level skills associated with the subject. In contrast to this, Schank holds how wine classes for adults begin with wine tasting. When

students learn to enjoy particular wines, they naturally develop an interest in identifying these wines, by district, grape, vintage or other properties. A course that began by describing the wine regions and grapes would not similarly engage students, and would not develop the crucial micro-script level skills that allowed the student to practice. A similar approach to the physics education could for instance be to ask students to predict whether a bowling ball would fall to the ground faster than a golf ball dropped from the same height. To answer this, students could design and perform experiments and come up with a theory of how gravity works. At this point, Schank predicts that it is easier to introduce Newtonian physics because the students already have developed an interest through *doing* physics. These micro-script level skills are often hard to measure in standardized tests, though, so what one tends to do is to make tests that tests the measurable – which is the higher level conceptual knowledge. Over time, this has the consequence that the focus of the education shifts from learning the subject matter for practical reasons to learning the subject matter of the test. The test becomes the doing. This results in people that are educated in, for instance psychology, and are expected to be able to handle work in human resources departments. After all, they are supposed to know about human relations and psyche. However, the psychology education may contain a lot of information about history in the field, important findings, methods for doing research and so on, but little of the applied skill required in human resource departments. It is not easy to apply the knowledge from a psychology education when tasked with firing someone, or settling a manager-employee dispute. This requires concrete experience from such tasks on the micro-script level that is not addressed in education.

Schank does acknowledge that there is knowledge that is not tied to micro-script level procedural skills. For instance, many find history interesting, and it is hard to find micro-level scripts associated with the subject. However, these skills can to some degree be traced back to the human curiosity for understanding life and the world. For instance, history may help in explaining why certain events (episodes) came to pass, and how to avoid them in the future. Curiosity may also follow from learning an applied skill, for instance, Schank explains how his interest in the geography and history of wine regions emerged from his interest in the drink.

Unlike the radical constructivist critique, Schank does not discount the idea of having a curriculum. However, he emphasizes that every curriculum should contain packages of micro-script-level skills that serve some goal, and that the focus of the curriculum should be to foster the ability to perform practical tasks in the domain. Students should start out solving problems immediately, and through exposure to problems, their interest in acquiring more theoretical knowledge will emerge. In other words, he does not suggest that theoretical knowledge should be dispensed with, but be available upon request to help students solve their problems. The curriculum could be designed to give students problems that would naturally lead them towards important theories and more advanced internal models, but these models should not be taught directly, but be available upon request. The problem with this approach is that it is not

easily testable, at least not for all domains. The driver's test is an example of a test of procedural skills, but even in this domain where it is possible to arrange, it is far more expensive and subjective than a standardized test.

Schank's theory is a strong proponent for case-based tutoring. It suggests that true learning can in fact only happen through cases, and that procedural and even higher order conceptual knowledge emerge because of experience received through cases. Other theories, like ACT-R reviewed in Chapter 4, also place importance on exercises or cases as a way of operationalizing procedural skill, but suggest that conceptual and procedural instruction should predate exercises, which then serve to encode the explicit knowledge as implicit procedural skill. Schank's case-based approach is more radical because it suggests that conceptual and procedural instruction is not necessary and that the examples, exercises and episodes themselves serve as the primary vehicle of learning, and motivation for learning.

2-3 Intelligent Tutoring System Today

There exist a wide variety of systems that may be called intelligent tutoring systems, and quite a few ways of categorizing these systems. Murray [50] identifies seven categories of ITS where the development has gone to the stage that authoring tools are available:

1. Curriculum Sequencing and Planning. The systems in the first group focus on sequencing pre made instructional units such as text and pictures to suit the individual student. The result may seem like an "electronic textbook" to the student, but in reality it is a textbook that is structured to fit that particular student.

2. Tutoring Strategies. While the systems in the first group focus on the macro level of tutoring strategy the second group primarily deals in the micro level. These systems focus on such issues as how and when to deliver explanations, hints and summaries. Typically, these systems have very well defined explicit pedagogical models that allow a course author to codify different tutoring strategies and meta-strategies that the system may use to tailor the instructions to the individual student.

3. Device Simulation and Equipment Training. Both of the previous groups tend to use instructional strategies where the student is given a text or other material to absorb and think about. In contrast, the third group uses learning by doing by offering a simulated environment to the student that allow him to experiment and see what the consequences of different actions are. A flight simulator is a well-known example of such an environment. In addition to the simulation environment, these systems usually contain tasks, performance monitoring and instructional feedback. Usually, there is no conceptual teaching

in these systems, and the student it is assumed to have some introductory knowledge in the domain.

4. Expert systems and cognitive tutors. These systems are often extensions of rule-based expert systems that contain a complete system that is able to solve problems in the problem domain by itself. The rules of these instructional systems are designed to mimic the cognitive process used by humans to solve problems in the domain, and as such the expert model may be very detailed. The systems may also contain "bug rules" that encode common misconceptions in the domain. This allows the tutor to reproduce mistakes and trace the reasoning process of the student, and even take over and continue the reasoning if the student is stuck.

5. Multiple Knowledge Types. Instructional theories often suggest different methods for learning different kinds of knowledge. For instance, facts may be taught using reading and repetition, concepts by exposure to prototypical examples and procedures by step-wise instruction. The Intelligent Tutoring Systems in this category are often based on a theory like this, with support for course authors in designing instructional units for the different types of knowledge.

6. Special Purpose Systems. There is a wide variety of tutoring systems that are designed with a particular domain in mind, from the pedagogical theory to the system design and implementation. These systems are often quite specialized and build on specific instructional experience in the are for which they are built.

7. Intelligent/Adaptive Hypermedia. Web-based tutors and other adaptive hypermedia systems adapt to each student by selecting the content presented on a topic as well as managing the hyperlinks in this content. This task can be seen as similar to the main objective in the first group of systems in that the hyperlinks provided to the student affect the sequencing of the content. The difference lies in that the student has a greater control over the direction to take, and that the contents are often built dynamically to avoid cognitive overload by presenting the student with too many new concepts and hyperlinks.

One of the dimensions that separate these groups of systems is whether they have a learning by doing approach where the student is exposed to problems, and is asked to solve these problem with the assistance of the tutoring system. In general, groups 3 and 4 in the categorization focus on this kind of tutoring. This approach contrasts with the more theoretical, tutoring- or book-centric tutoring, which we will call learning by instruction although certainly other forms of mediums can be used for this than text. This approach is in general used by systems in groups 1, 2 and 5 and 7.

These learning strategies are useful for different kinds of instruction. The theoretical instructional tutoring targets teaching methods and concepts, while

exercises are used to train the students in methods and to help them operationalize the skills. For instance in primary schools, addition, subtraction and other basic mathematical skills are usually taught by explaining the concept to the pupils and demonstrating the procedure for class before asking the students to solve exercises until they master the method.

Simulation training (group 3) requires that the topic matter can be simulated accurately in a computer. In order to do this, an accurate model must exist. In many areas there are models that are considered accurate enough – flight simulation has already been mentioned as possibly the best known. As discussed in Section 2-1.1, the simulation may be quantitative or qualitative in the sense that it may or may not attempt to match the mental models of human reasoning. If these systems do not adopt a simulation model that is based on human reasoning, it may be hard to communicate for instance explanations to the student as the model of the system does not match the level expected by the student. On the other hand, the efforts of SOPHIE and later systems illustrate that accurate, qualitative models are by no means trivial to create.

The other group that primarily deals with learning by doing is the "Expert Systems and Cognitive Tutor" group. These tutors attempt to approximate a complete theory of the domain. Systems in this group are in many ways siblings of the GUIDEON effort described in Section 2-1.2, but modern systems in this category typically also reason about strategic knowledge, such as seen in the PACT cognitive tutors [13] described in Chapter 4.

The most common practice in developing cognitive tutors is to use rules. This is very useful in domains with a strong domain theory, such as the matematical cognitive tutors. If the theory is more fragmented, many rules only apply to a very limited area of the problem space, and as such they may require a large number of conditions. As these rules get very specific, they may be better captured by prototypical cases that represent actual problems. Cases are often easier for people to relate to than very specialized rules, as they represent actual problem solving episodes, including the context the problem was solved in. ELM [87] and CATO [7] are tutoring systems that uses cases. These systems will be further reviewed in Chapter 4.

Law and medicine are examples of domains that can be categorized as open or weak theory, which means that there exists no complete theory that describe the problem area so that correct solutions can always be derived or proven. Even most engineering disciplines interact in complex ways with the real world that they are hard to describe accurately in a complete theory. In such situations, a combination of general knowledge and cases are useful as an incomplete model with verified data points (the cases) in the problem space.

The case-based tutoring approach usually combines concrete cases and general knowledge in order to support learning in weak theory domains. The focus of these systems is often on the cases, which are usually carefully selected problems in the domain. These may be presented to the student as exercises, and although the system may not have complete knowledge of how to solve any problem in the domain, the problem cases are typically solved problems for which the system may offer assistance. Although computer-assisted instructional sys-

tems may also contain pre-solved exercises, the case-based tutoring systems are different because they still tailor the problem-solving activity to the student, both in the selection of the exercise and in the offering of support during the problem solving itself. Cases may also be used in other ways, for instance the SARA [75] and Ambre-AWP [40] systems ask the student to retrieve and adapt previous problem solutions (cases) to a similar problem. Although SARA is applied to the domain of number theory, which could possibly be described by a complete deductive model, the system relies on the finding that people often do not try to solve a problem from first principles, but by adapting a previous experience. Still, SARA relies on having an accurate model of what adaptations can be applied to a case as the system must be able to evaluate whether the adaptations fulfill the goals of the problem.

2-4 Critiques of ITS

2-4.1 Constructivist critiques

A fundamental difference in pedagogical philosophy exists between the instructional design and constructivist approaches to learning. The instructional design approach views the process of learning as the transfer of knowledge from a source (teacher, book or other educational aide) to the student through a communications channel, typically written or spoken language assisted by visual aides. The constructivist approach tends to focus on acquiring knowledge through exposure to real-world problem solving, arguing that each learner should be allowed to create his own model of the domain. It should be said that both instructional design and constructivism are terms associated with a great range of ideas and theories and as such are hard to define in detail, but a core difference lays in who decides and constructs the knowledge. In the instructional design approach, an expert or teacher decides on a curriculum and desired knowledge state that is the goal of the learning, while constructivism holds that the student should be free to form the knowledge required, although possibly with the assistance of a teacher or expert.

In the instructional design approach, there are several challenges associated with teaching. One is that the communication channel is not perfect – messages from a teacher are likely to be interpreted differently than expected by the student, so there has to be an constant process of evaluation of the transferred information in order to minimize the errors and misconceptions. Another challenge is that people are not passive machines that simply attempt to receive some model and store it away – learned information are generalized and structured to fit with already existing knowledge, and people tend to hold on to these early structures. This means that the knowledge transfer process should be tailored pedagogically in such a way that early structures formed by the student are as accurate as possible and helps the student in organizing later information. This may mean starting with very typical examples, or introducing simplified (but complete) models first before gradually extending the model. There is also

a challenge in tailoring this process to each individual student, as the students have different abilities and backgrounds. However, the goal of these tasks is to establish a pre-determined model in the mind of the student as accurately as possible.

In the knowledge transfer approach, exercises are seen as a way to test that the knowledge has been received correctly, and assist the student in applying the knowledge to problems in the domain. The assumption is that the student already knows in theory how to solve an exercise before attempting it, but needs to train in order to operationalize the skill and uncover misconceptions that may be present in the student's model.

The constructivist approach is skeptical about the predictability of human behavior and the ability to describe an objective world. These properties are fundamental to the instructional design approach because if people's experience of the world is significantly different from each other, it is dubious that an "expert model" may be created which reflects anything other than the expert's subjective view. Many radical constructivists even deny the existence of an objective world, which makes it hard to claim any special significance for an expert's model over anything a student may form. This means that tutoring systems and teachers can do little to affect student understanding, as no objective knowledge about the world is available. This view invalidates the need for an expert model, and if one also holds, as many constructivists do, that human behavior is fundamentally unpredictable, pedagogical and student models also become useless because it is impossible to model the unpredictable. Although this is a radical position, constructivists' critiques may also be appreciated at a more moderate level. For instance, if a colorblind driver cannot see the difference between green and red traffic lights, the textbook information that "red means stop and green means go" is useless. Most drivers easily overcome this little obstacle, for example by using the position of the lights, but typically do not do so by reading a textbook, but by experience. This difference in how people model and learn about the world extends beyond differences in sensory capabilities. Clearly, intellectual abilities and prior knowledge also plays a role. For instance, when learning a computer programming language, students with a good grasp of algebra have an advantage in dealing with variables.

Winn [91] lists the major constructivist critiques against the assumptions of instructional design:

Reductionism: Instructional design typically assume that the world may be usefully broken down into chunks of knowledge (e.g. represented by the curriculum of a school course) that may be learned separately, and that understanding and mastery of the whole may be achieved by mastering the contents of all the chunks separately. Constructivists claim that much of the difficulty of using knowledge lies not in the individual chunks, but in the interaction and relations between chunks, for instance how to use knowledge from both programming and number theory in creating a program that produces prime numbers. They suggest that knowledge has emergent properties that may not be captured by

dividing it up into small chunks. Spiro [79, 80] claims that the student should be encouraged to construct knowledge by "crisscrossing a landscape" many times and in different ways in order to capture relations and interactions, and present the student with the same content in different contexts.

Determinism: Instructional design assume that how a student learns can be modeled and that this model can be used to predict to some degree how a student will receive information and build knowledge structures. Constructivists challenge this assumption on many fronts. First, instructional theory is incomplete in many (if not most) areas. Second, a complete instructional theory may be impossible because humans learn in idiosyncratic ways. Instructional design approaches may contain efforts to tailor the instruction to each student's idiosyncrasies, but constructivists point out that these efforts tend to focus on one or two factors, while the number of factors that affect learning seems to be much larger. This causes an exponential explosion both in the modeling effort and in the uncertainty of the conclusions. Third, Winn claims people typically do not use the academic logic taught in courses in solving real-world problems. This suggests that the assumption of instructional design that people will use the structures and approaches taught through formal training in solving real world problems do not necessarily hold.

Replicability: When a tutoring system, educational textbook or course has been built and proven itself effective on real students over time, the assumption of instructional design is that this resource may be used again with other students at a similar level with similar results. Constructivists question this as educational contexts such as a different teacher, a different school or students with a different background may find the textbook, course or tutoring system less useful because of assumptions made in its creation. This problem has been reported with some intelligent tutoring systems. Payne and Squibb [60] found there was found little overlap in bug libraries across three different schools, and that bugs were unstable and that the use of them by students is unstable. Constructivists explain this by referring to the fundamental difficulty in predicting human behavior – even small differences in background and ability may interact with the learning behavior in complex ways that is essentially impossible to model.

2-4.2 Modelling Intractability

A central tentant in the knowledge communication theory is that the system must contain a good model of the area to teach – if it does not, there is no knowledge to communiacate. In addition to this, the ITS must have some knowledge about how to teach this knowledge and the kinds of misconceptions that may arise in students.

This means that the modeling requirements of ITS system supersedes the requirements of an AI system that solves the problem because it also needs

to model the space of potential errors students may develop [71]. In addition, the set of errors produced by students does not seem to be easily captured. For instance Payne and Squibb [60] showed that when the same system was applied to three different schools, the five top bug rules at each site overlapped so such a low degree that combined they formed 13 different rules. For error rules to be feasible, a firm idea of what a correct and erroneous rule means, is required. This may be possible in formal domains, such as the geometry or algebra domains, but it may limit the approach to these domains. These and other critiques have opened the study of student modelling to many other approaches, including probabilistic methods (e.g. [85]), fuzzy models (e.g. [41]), case-based approaches (e.g. [75, 40]) and constraint-based modeling (e.g. [57]).

A more fundamental critique of the bug library approach to student modeling has been seen from constructivist theories, as seen in the previous section. If the constructivist critiques of instructional design is accepted, this means that it is extremely hard, if not impossible, to model the knowledge of the student as the system cannot know the experience and motivations that forms the individual student's background. Further, many constructivists hold that it is unethical to attempt to impose the computers conceptualisation on a student – he should be able to form his own on his own terms. This suggests that the very idea of student modelling should be abandoned. While few ITS researchers are willing to go this far, there is a trend toward inspectable student models, where the student himself is allowed to access and possibly change the student model. In addition to the pure ethical view that the student should be allowed to know the beliefs held about him by the system, it also gives the student a means to correct mistakes. Although we can all have misconceptions about ourselves, the idea is that we at least know ourselves better than the tutoring system – intelligent or not.

2-5 Consequences for ITS

The critique of the intractability of modelling, and particularly student modelling was addressed in a seminal paper by Self [71]. He recognizes that complete student models that contain the complete mental state of students are close to impossible. Students have models that operate on different level of abstraction at once – problem-solving strategies, analogies to other domains, analogies to previously solved problems and so on. They will also have different background so that the analogies and strategies they use can not be assumed to be the same. This means that a complete student model would seem to encompass almost all of cognitive science, from plan recogniztion, episodic memories, representation issues, mental models, individual differences and so on. However, Self suggests that complete student models are not necessary, and proceeds to suggest four slogans for constructing tractable student models:

1. Avoid guessing – get the student to tell you what you want to know. By using good problem solving environments, Self suggests that it is

possible to implicitly require the student to give the system the information it requires. Instead of having the system infer intermediate problem solving steps the student may normally do in their head, a good problem solving environment can encourage students to do this explicitly. The environment can also require the student to enter other kinds of information, for instance the goal currently persued or the hypothesis tested.

2. Don't diagnose what you cannot treat. The ideal of creating the ultimate student model that accuratley represents everything relevant that the student knows can sometimes obscure the fact that the student model by itself does not accomplish anything. The knowledge must also be used somehow if there is to be a practical benefit to the exercise. Self suggests that the pedagogical strategy of the system should be considered in tandem with the requirements of the student model, so that only the information that is actually useful in tailoring the tutoring is sought modeled.

3. Empathise with the student's beliefs, don't label them as bugs. The general perception on student models is that they should be used to remedy problems in the student's model. Self suggests that this arrogant "the tutor knows best" style alienates teachers more than any technical shortcommings. In addition, there is the philosophical problem of labeling the students' understading as wrong. The QUEST system illustrates that learning often happens through a sequencce of ever more complex models, and a particular belief that may seem wrong from the expert's standpoint may serve as a useful simplification at the level the student is at. It may also be that the student has a conceptualization that differs somewhat from the expert, which does not necessarily mean that it is incorrect.

4. Don't feign omniscience – adopt a "fallible collaborator" role. This position is not based purely on philosophical grouns, but also on the fact that many, if not most, domains are hard to model completely. If the model is less than guaranteed correct, pretending to be omniscient is risky. However, the ITS can provide assistance where it can, but in a manner that allows the student to ignore the advice. The ideal should be for the student to see the system as a peer or assistant as opposed to a infallible teacher.

The gist of these slogans are that the ITS should have use for the information in the student model, and that it should as directly and naturally as possible ask the student for this information. Further, it should take a humble role in not assuming that the inferences it draws from this model is necessary correct. If the system thinks that the student uses a rule that might cause trouble in solving some problems, the system should present such a problem and ask, as a collaborator might, if this rule might not cause problems here. The role of the system should not be to correct the user's beliefs, but help him elaborate and test them. This role of the student model allows for the possibility of a more limited student model that needs not "simulate" the student's problem solving

capability. This opens up the use of student modelling to less formal domains than the earlier models.

In later papers [72, 6], Self and Akhras address the more fundamental constructivist critique of intelligent tutoring systems. At first it appears that if these critiques are accepted at face value, the enterprise of creating intelligent tutoring systems seems futile, as the learning process would be too unpredictable to be analysed and modeled in advance. Self and Akhras suggest a learning model that replaces the traditional expert/student/pedagogical models that is more compatible with the constructivist critiques, while still maintaining many of the properties of intelligent tutoring systems. This model is based on the modeling of the interaction between the student and the system instead of modelling the student himself. The basic idea is that by modeling the interactions, the system may adapt in order to bring about interactions that are considered better learning opportunities than others.

Another approach towards more constructivist learning is through the creation of learning environments that are closer to the real problem situations. These systems can be seen as advanced simulation environments that attempt to be as close as possible to the real thing in order to capture as much of the context as possible. This can for instance take the form of "'virtual reality"' 3D world such as [22] or [38].

The focus of this work, however, is on learning by doing systems that are somewhat closer to the classical exercise-oriented approach. On this front, more systems have appeared that operate in domains that cannot be completely modeled, as we see for instance in the CATO system described in chapter 4. These systems often do not use the term ITS to describe themselves, but rather *interactive* or *intelligent learning environments*.

2-6 Chapter Summary

In this chapter, we have examined intelligent tutoring systems quite broadly, by briefly reviewing classical systems and a classification of how the ITS lanscape looks today. Important venues of critiques to these approaches have come from the constructivist community, which sees fundamental problems in the knowledge transfer model of tutoring and learning, as well as from the researchers pointing out the intractability of complete student modeling in many domains. In the following chapter, the focus will be on developing a framework that is more specifically tailored to the subtype of intelligent tutoring systems examined in this thesis – exerciser-oriented systems.

Chapter 3

Framework

In this chapter, a framework for intelligent tutoring systems in the area of exercise support is presented. This framework contains a set of dimensions on which exercise support systems can be usefully described and analyzed. In chapter 3 this framework will be used to analyze four existing exercise support systems, and in chapter 4 it will be used to describe the CREEK-Tutor architecture.

Most implemented tutoring systems have a theoretical approach and architecture behind it. In many cases, the implemented system does not include all the features of the underlying theory. Even if the implementation is close to complete, it is usually limited to one or a few test domains where the theory is tested. Although actual tests of the theory are performed through an implemented program, it is also useful to be able to refer to the underlying theory, for instance in gauging the likely limitations to the approach. We will use the word theory to refer to the theoretical background specifically, and the word system to refer to actual implemented computer programs based on the theory.

3-1 Focus and Requirements

The framework described in this chapter is designed as an analytical and descriptive tool for intelligent tutoring systems for exercise support – an approach often called case-based tutoring. This is by no means the only task an ITS can support, but in order to limit the scope and focus the comparison, the framework will be limited to these kinds of systems.

This framework is based on the classic three model of ITS design – the expert, student and pedagogical models. In the previous chapters, this approach has been critiques from several angels, but the appraoch to ITS represented by these models is still the basis for discussion and comparison of intelligent tutoring system today. Some attempts have been made at other kinds of framework (such as the constructivist framework suggested by Akhras and Self [6]), but these approaches are not yet in wide use.

While the framework is based on the classical instructional design philoso-

phy, the critiques of modeling tractability and constructivism are discussed for each system analyzed by the framework. That said, the framework is probably less useful for analysing radically constructivist approaches than systems that operate close to the instructional design philosophy but perhaps adopt some contructivist techniques.

3-1.1 Weak vs. Strong Theory

A stated goal of this research is to examine how exercise support may be done in weak theory domains. This means that it is important to examine to what degree a complete model is required. This may be different between the different types of models in the system – for instance it may require a complete expert model, but use a weak model of the student. This means that the weak vs. strong domain model question will be discussed for each model type, as well as discussed under other relevant dimensions of comparison. Typically, a strictly instructional design approach will require strong models, while approaches using constructivism will be based on weaker models.

3-2 Dimensions of comparison

In order to compare explanation support systems, the framework contains four dimensions of comparison, which are then broken down into subcategories and concrete issues. The dimensions examined are domain, cognitive theory, knowledge models and tutoring capabilities. These dimensions are described in detail in this section.

3-2.1 Domain

The domain is the problem area or the kind of problem area the system has been developed to support. While some degree of generality of the theory underlying a concrete system is typically claimed, the concrete system is only tested on a limited number of problems, such as solving LISP program problems (such as in ELM and LISP Cognitive Tutor) or solving a particular kind of mathematical problems, such as geometry problems in mathematics (as in Geometry Cognitive Tutor). This dimension describes the problem area the system was developed for as well as how well the theory claims to generalize to other domains.

Description of the domain

In this subsection, the problem domain or domains the concrete system is developed for is presented. This includes an analysis of what tasks and properties are important to support for learning an exercise support in this domain, and how the system is presented to the user.

1. Domain
1.1. Brief description of the domain
1.2. Scope
1.3. Generality
2. Cognitive Theory
3. Knowledge Models
3.1. Expert Model
3.2. Student Model
3.3. Pedagogical Model
4. Tutoring Capabilities
4.1. Goals
4.2. Exercise Selection
4.3. In-Exercise Support
4.4. Conceptualization Support
4.5. Explanation Ability
4.6. Learning
5. Evaluation

Table 3.1: An overview of the framework

Key Questions:

- What functionality must exist to support learning in this domain?
- What functionality is less important?

Presentation

The actual user interface is not a focus of this framework, but the degrees of freedom the user has in solving the problem is often indicative of assumptions made by the tutoring system. For instance, some programming tutors use special expression editors that make it impossible for the student to make syntactical errors (such as missing end brackets) in order to be able to parse the expression and evaluate a partial solution.

Key Questions:

- How is the system presented to the user?
- What degrees of freedom does the user have in solving problems?

Scope

Most intelligent tutoring systems have a limited scope, both in the kinds of domains they support but also in the learning activities they support. This should be clarified in this section. Examples of scope is that the system is designed to work only on mathematical problems, or only assisting students that have been given some introduction to the concepts and procedures already, but may not be well trained in applying this knowledge on exercises.

Although specific systems typically cover a limited domain, the underlying theory typically claims some kind of generality. For instance, a system may be developed to support a specific branch of mathematics, such as geometry, but the theory is claimed to be suitable for other branches of mathematics as well.

Key Questions:

- What kinds of problems do the system support?
- What tutoring tasks does the systems support?
- What parts of the system is domain specific?
- What range of domains does the theory claim to cover?

3-2.2 Cognitive Theory

Previously, we have discussed how the pedagogical philosophy, such as the difference between the knowledge transfer and constructivism may impact intelligent tutoring systems. However, many systems are based on a more detailed theory of human cognition. For instance the PACT Cognitive Tutors [13] and ELM [86] systems are based on an explicit theory of human cognition. These theories may have been developed as part of the systems themselves, or the system may be based on a cognitive theory developed in advance. Naturally, these theories of how human cognition works have a great impact on tutoring strategies and as such how intelligent tutoring systems are built.

Two cognitive theories used by intelligent tutoring systems are the ACT theory of Anderson [16, 10] and the Dynamic Memory from Schank [69]. ACT uses an approach similar to rule-based expert systems in modeling human problem solving. This approach has been shown to even repeat some of the mistakes done by humans in domains such as mathematics and computer programming. By viewing human problem solving as a kind of rule-based expert system, teaching becomes the task of helping the student to acquire the correct set of rules to solve the problem. This theory can as such be claimed to lean towards the knowledge transfer pedagogical philosophy, but it should be mentioned that it does not underestimate the value of exercises to train, test and operationalize knowledge.

The Dynamic Memory theory is more constructive in nature, in that the memory structures are assumed to start as very specific episodic cases, and

are formed into more generalized concepts only when many very similar cases have been experienced and can usefully be abstracted to a prototypical case. Further cases confirming the structure are then typically not retained, although cases that do not conform to the generalized structures known to the system are retained and indexed by showing its difference to the closest generalized concept. This approach is clearly more constructivist in nature than ACT-R, and suggest that the conceptualization of a student is heavily influenced not only by what exercises (cases) he experiences, but also the ordering of these exercises and the student's prior expectations. This theory suggests that exercises are not only there to test or operationalize knowledge, but actively play a part in forming the memory structures of the student.

These two theories each have a different idea of why exercises are important, and what they are meant to accomplish. Typically, this influences the capabilities of systems and the weight they place on different tutoring tasks.

Not all systems are based on an extensive theory of human cognition, however, but these systems do tend to place themselves, explicitly or implicitly, on the more fundamental dimensions such as the constructivist vs. knowledge transfer scale. If the system examined is not explicitly based on a theory of human cognition, we will attempt to answer these questions by analyzing the capabilities and methods of the system.

Key Questions:

- Does the system have an explicit theory of human cognition?
- How does the theory place on the instructional design vs. constructivist scale?
- What is the role of exercises (problem cases) in learning according to this theory?
- What is the relationship between conceptual (generalized) knowledge and exercises (episodes, cases)?

3-2.3 Knowledge Models

The defining characteristic of intelligent tutoring systems, separating it from other computer-aided learning systems, is the use of knowledge models to customize the learning experience for the student. In this sense, the knowledge models and representation play the same fundamental role in these systems as they do in knowledge-based expert systems. Intelligent tutoring systems generally have three broad categories of knowledge; expert and domain knowledge, knowledge about each student, and pedagogical knowledge. Within these categories, different kinds of knowledge may exist. For instance, the expert and domain knowledge may contain both general knowledge in the form of rules, and episodic knowledge as pre-solved problem cases.

The knowledge models also bounds the capabilities of the system in the sense that the system cannot teach what it does not know, and it cannot customize the experience for a student it knows nothing about. For instance, if a student is faced with a concept he does not know, the system cannot present the student with a definition unless such knowledge is contained in its models.

The contents and abilities of the knowledge models can be described on the knowledge level [54], where the focus is on the knowledge content of the system and the abilities this knowledge afford. The three main models of the ITS system may usefully be analyzed on this level in order to show what kind of knowledge is contained in the different models and how they affect the performance of the system. An analysis on this level may also cover how the knowledge is acquired.

For this framework, it is also useful to look at the knowledge representation on the symbol level, which concerns itself with how the knowledge is stored, data structures, inferences and algorithms. In particular, it is interesting to see to what degree the representation assumes that a strong domain model is possible and available.

Expert Model

The expert or domain model of an intelligent tutoring system contains the knowledge the system has about the domain of discourse. From the earliest intelligent tutoring systems, the goal has been to transfer the knowledge contained in this model from the system to the student. As such, it also formed the basis for the student model, which was seen as the subset of the expert model currently known by the student. This view of the learning process may be called the knowledge transfer model. An opposing view to this can be found in constructivist theories, which claim that knowledge may only be formed by the learning agent in interaction with the environment.

On the extreme end of the knowledge transfer end are systems that claim to model the cognitive reasoning process of the domain completely. Perhaps the best known of these are the Cognitive Tutors [11], where the expert model is complete in the sense that it is able to solve problems in the domain on its own, and do so in a way that should be similar to how a student does it, when he does it correctly. On the other end of the spectrum, constructivist systems may not have expert models as such at all, but Self [72] suggests that such systems may employ situation models that contain knowledge about situations that should facilitate and trigger learning in the student. These models may take a similar role as expert models in an intelligent tutoring system.

Between these extremes are systems that do not have an executable model or do not view this model as the learning target for each student directly, but may use the expert model to form explanations when necessary, or to otherwise assist the student. For example, an exercise support system may contain a set of pre-solved exercises annotated by an expert, but without the complete knowledge required to solve a new problem. This model certainly contains expert knowledge about the domain and affords the system the ability to assist the student on these pre-solved problems, but it does not contain a complete

executable model of the problem domain. It would not be able to assist a student solving a new problem, and could not assist a student that chooses an unexpected strategy in solving a known problem. A way of generalizing this theory is to add adaptation knowledge to the reasoner so that it may adapt solutions from old problems to similar new problems, but unless the adaptation knowledge was good enough to cover the complete problem space, it would still not be a complete model. This also means that it would not subscribe fully to the knowledge transfer view. Hopefully, the student would learn some more general lessons than just the ability to solve the set of exercises known by the system in this case. In effect the student would be expected to construct his own generalized model. However, the system would still have more direction and expert knowledge than the more extreme constructivist systems.

The example above also illustrates how expert knowledge can take many forms. Rules, cases, adaptation knowledge and even knowledge about learning situations may all serve as expert knowledge. In this section, the forms of expert models used by the theory are described.

Relating to the completeness of the expert model is also the question of how hard the knowledge is to acquire. It is for instance conceivable that a complete executable domain model may be possible to create, but too costly or otherwise too difficult to make. An interesting question here how and how hard it is to acquire the expert model required by the system.

Key Questions:

- In what form is the expert model (cases, rules, etc.)?
- Can the expert model be used directly (executed) to solve new problems in the domain?
- To what degree is the goal of the system to transfer the expert model to the student?
- How is the expert model acquired?

Student Model

Describing a student modeling approach may be broken down into three questions:

- What does the system model about the student?
- How does it acquire the knowledge?
- How is the knowledge used?

For the knowledge model content, Chapter 1 describes the development from overlay models, which models the student as a subset of the expert knowledge, to bug libraries which also models common misconceptions in the domain. Other

approaches, such as case-based student modeling are also briefly discussed. In concrete systems, however, it is important to examine the student model strategy in light of exactly what kind of customization the system provides, and what opportunities for learning about the student it has [71]. We will take such a view when describing student models in this framework, asking how it is formed, what is stored and how it is used to customize the experience.

Another important aspect is the degree to which the student himself can access and inspect the model, or whether the model is internal to the system. This is both a practical and ethical issue. It may be prudent to assume that the student knows himself better than the system does, and as such give the student the ability to correct mistakes in the model. It is also a goal in itself to avoid secrecy about what information the computer has about each student. This does of course mean that the model must not only be open, but also understandable to the student. An accessible model that is incomprehensible does little to alleviate practical or ethical concerns.

Key Questions:

- What is the general approach to student modeling (e.g. overlays, bug libraries, cases)?
- Is the student model limited to a subset of the expert model?
- Does the system attempt to recognize common misconceptions (bugs)
- Can the student inspect and/or change the model?
- What kind of tutoring tasks is the model aimed at supporting?

Pedagogical Model

The pedagogical model is often offered the least attention in intelligent tutoring systems, and in many systems is not explicitly represented, but hard-coded in the program. However, almost all intelligent tutoring systems have some underlying idea about pedagogical philosophy. For instance, the difference of pedagogical philosophy between the knowledge transfer and constructivist camps results in very different kinds of systems. The difference may also be less fundamental but still important. The choice of a rule-based expert and student model may suggest that the pedagogical philosophy of the system is to teach these rules to the student and then check if they are retained correctly by giving him exercises using the rules. The case-based tutoring approach, however, is to give the student problems and hope that in solving them, the student himself will form generalized knowledge that may be used to solve never before seen problems. This means that even though the pedagogical theory may be hard coded in an analyzed system, it is still and important aspect to examine in order to understand it.

Key Questions:

- Does the system have an explicit pedagogical model?
- What kind of pedagogical strategic changes can be done by the system?

3-2.4 Tutoring Capabilities

The third dimension of the framework focuses on the actual tutoring abilities of the system. This ability is related to the knowledge models, but instead of looking at the source models of the system, it focuses on what the system is able to do with these models. As exercise support is of particular interest, the exercise selection and in-exercise support are important support tasks. In addition, we will examine how the exercise support system ties in with more theoretical knowledge in the domain in the form of conceptualization support, and what kind of explanations it is able to give to the student.

Goals

Few intelligent tutoring systems attempt to do everything, and even when limiting the scope to exercise support systems, the system will seldomly cover all the activities associated with exercises. Different systems often have slightly different goals and emphasis, either because of research interest or pedagogical philosophy. Before describing the detailed capabilities, this section will address the goals of the system – what it is designed to teach and the tutoring methods used.

Key Questions:

- What are the tutoring goals of the system?
- What and how does it seek to teach?
- How well may the goals be supported by the knowledge sources?

Exercise Selection

An exercise support system typically has a set of exercise problems that may be given to a student. The simplest way of choosing what exercise to present to a student would perhaps be to order them in rising difficulty and present them one after another to the student. However, there is an opportunity here to select the exercises carefully in order to customize the experience to the current student's ability. This may mean focusing on exercises that cover areas where the student has problems, and customizing the difficulty level to make sure the experience is neither too hard nor too easy.

The task of exercise selection is in many ways similar to the curriculum sequencing and planning, which itself is a well studied area of intelligent tutoring [50]. The similarities is that given a set of units covering different subsets of the area, the task of the system is to select an ordering of the units that provide the student with full coverage of the domain, but does so at the student's pace and taking into account what the student already knows. However, the exercises are often more numerous, and usually there may be many covering the same area. Possibly because of this similarity, not all exercise support systems cover this task.

Key Questions:

- How, and on what principles, are exercises selected?
- Can the student influence exercise selection?

In-Exercise Support

When an exercise is selected, either by the system or the student, the student may need further help in solving the problem represented by the exercise. This may take the form of suggesting a step towards the solution of the problem if the student is stuck, finding and presenting examples on how similar problems are solved, or explaining why a solution step suggested by the student will not work.

For many exercise support systems, this is the main tutoring support activity, and exactly how it is done varies with the knowledge model available and the pedagogical philosophy. For instance, a case-based tutoring system containing a wide array of pre-solved examples may use the strategy of finding similar examples, while a system with a complete executable model of the domain may take the partial solution of the student, plan how it can most easily be extended to solve the exercise problem, and suggest steps in this direction.

Key Questions:

- What kind of in-exercise support is offered by the system?
- To what degree can the system explain the reasoning behind suggestions?

Conceptualization Support

An important aspect of learning is acquiring and understanding the vocabulary of the domain. Some systems attempt to model what concepts the student is familiar with in the student model and attempt to limit its explanations to using these concepts. While this is a useful customization of the dialogue, it does not as such assist in the formation of concepts.

Key Questions:

- Does the system help the student in forming or learning concepts?

Explanation Ability

Intelligent tutoring system can potentially have a very wide range of abilities to explain. In selecting exercises, the system can be seen as acting as a form of expert system, using the student model to choose an appropriate exercise for the student. The student may (unless forced) refuse to accept such suggestions if it lacks a better justification than "because it is good for you," and as such explanations are as important here as in most knowledge-based systems.

In assisting the student in solving the exercise, the opportunities and demands for different kinds of explanations may be quite varied indeed. When a human tutor assists in solving exercises, explanations may range from a simple request to explain an unknown concept in the exercise text ("what is a polynomial function?") to clarifications of the exercise text ("when it says that I must use an iterative approach to solve the problem, does that mean I am not allowed to use recursion?") or even fundamental questions about topics required to solve certain kinds of tasks ("how do I use a for-loop?").

In some ways, the exercise support system may be thought of as a particular kind of knowledge-based expert system, but with the difference that the focus is shifted from providing the answer to the exercise (which it typically knows well) towards explaining and teaching how this answer is found. However, this explanation capability has also been given increased focus in the expert system community, and indeed some of the work on explanation for early rule-based systems such as MYCIN [33] resulted in some the early rule-based tutoring systems like XPLAIN [82].

In this framework, we will categorize the explanation capabilities of the tutoring systems by the goals each explanation capability is designed to address. This is done by adapting a set of abstract explanation goals originally used to describe explanation capabilities of case-based expert systems [78]. These explanation goals are again adaptations of similar explanation categorization efforts from the knowledge-based systems community [39]. These five abstract explanation goals are:

Explain How the System Reached the Answer (Transparency). The goal of an explanation of this kind is to impart an understanding of how the system found an answer. This allows the users to check the system by examining the way it reasons and allows them to look for explanations for why the system has reached a surprising or anomalous result. If transparency is the primary goal, the system should not try to oversell a conclusion it is uncertain of. In other words, fidelity is the primary criterion, even though such explanations may place a heavy cognitive load on the user. The original how and why explanations of the MYCIN system would be good examples.

Explain Why the Answer is a Good Answer (Justification). This is the goal of increasing the confidence in the advice or solution offered by the system by giving some kind of support for the conclusion suggested by the system. This goal allows for a simplification of the explanation compared to the actual process the system goes through to find a solution. Potentially, this kind of explanation can be completely decoupled from the reasoning process, but it may also be achieved by using additional background knowledge (as in XPLAIN) or reformulation and simplification of knowledge that is used in the reasoning process. In expert systems, empirical research suggests that this goal is most prevalent in systems with novice users [48], but this may not be users that are particularly interested in learning how to do this themselves and as such they are (presumably) different than students.

Explain Why a Question Asked is Relevant (Relevance). An explanation of this type would have to justify the problem solving strategy pursued by the system. This is in contrast to the previous two goals that focus on the solution. The reasoning trace type of explanations may display the strategy of the system implicitly, but it does not argue why it is a good strategy. In other words, while the focus of these explanations are on the problem solving procedure itself.

Clarify the Meaning of Concepts (Conceptualization). One of the lessons learned after the first wave of expert systems had been analyzed was that the users did not always understand the terms used by a system. This may be because the user is a novice in the domain, but also because different people can use terms differently or organize the knowledge in different ways. It may not be clear, even to an expert, what the system means when using a specific term, and he may want to get an explanation of what the system means when using it. This requirement for providing explanations for the vocabulary was first identified by Swartout and Smoliar [83].

Teach the User About the Domain (Learning). All the previous explanation goals involve learning – about the problem domain, about the system, about the reasoning process or the vocabulary of the system. Educational systems, however, have learning as the primary goal of the whole system. In these systems, we cannot assume that the user will understand even definitions of terms, and may need to provide explanations at different levels of expertise. The goal of the system is typically not only to find a good solution to a problem, but to explain the solution process to the user in a way that will increase his understanding of the domain.

Of these explanation goals, the justification goal may seem relatively less important, and the learning goal relatively more important than in expert systems. However, some situations may call for explanations of the justification type – for instance in the exercise selection task where the system may be asked to justify the exercise it suggested to the student. In this situation, it is not

necessarily important the student understands the details of how this selection was performed, especially if the student model is very complex. A simplified explanation may be sufficient and even preferred.

Key Questions:

- What kinds of explanations are offered?
- What are the limits of explanation support possible from the knowledge sources of the system?

Learning Capabilities

The learning dimension focus on the learning capabilities of the system itself. A key questions here is whether the system is able to potentially learn something for itself while performing its tutoring tasks. Such learning experiences could for instance be to modify the difficulty level associated with a particular exercise if it proves hard for many students to solve, or in more complex ways learn from successful and not so successful student tutoring sessions.

Usually, tutoring systems rely on gaining their knowledge about the domain and the pedagogical model in advance from experts. However, most if not all systems learn something about the current student. An important question in studying how intelligent tutoring system learn, then, is to study if and how it is able to retain and use the knowledge from the individual student in teaching later students.

Key Questions:

- What can the system learn to improve its performance during operation?

3-2.5 Evaluation

While the theory and capability of the theory is interesting, it is important to examine how the theory has been evaluated. Evaluation can be through quantitative empirical tests that examine the system as a whole, for instance compared to traditional teaching methods, or focus on a particular component of the system. Other forms of evaluation can also have been done, for instance in-depth qualitative analysis or case studies.

Key Questions:

- What forms of evaluation have been performed on the system?
- What were the results of the evaluations?

3-3 Chapter Summary

We have presented a framework for describing and analyzing exercies-oriented tutoring systems, where we focus on the domains the systems have been tested on, the cognitive theory behind the system, the knowledge sources the system draws on to provide tutoring capabilities, and the evaluation methods and results. In the next chapter, we will use this framework to examine five exercise-oriented tutoring systems.

Chapter 4

Theories and Systems

This chapter uses the framework described in Chapter 3 to describe and analyze state of the art intelligent tutoring systems for exercise support. While many tutoring systems support exericses in various ways, the analysis in this chapter is limited to systems with these properties:

Case-Based Tutoring: The systems provide exercise support, that is, systems that primarily present and assist students in solving exercises. This means that they have a case-based approach to teaching.

Knowledge Based: The tutoring systems must, to some degree be intelligent in the classic ITS sense – they must be able to provide some actual assistance in solving problems. This excludes simulation environments that only allow the student to explore without providing any guidance or help, and radical constructivist systems. This does not mean that the system must be able to assist in any situation (i.e. assist the student in solving any problem, or recognizing all possible misconceptions), but that it must have some knowledge about the domain and capability to assist.

Theoretically Based: The system should be clear about its basis in cognitive or pedagogical theory.

Evaluated: A wide variety of intelligent tutoring systems have been described, however not all of these have been evaluated by exposure to students. This requirement is not strict in the sense that there must exist strong quantitative support for the theory, only that there exists some kind of evaluation of the approach.

These criteria has led to a selection of three main systems for analysis – the PACT Cognitive Tutors, the case-based student modeling system ELM and CATO, which teaches case argumentation in law.

In additional, features of the BLITS and Ambre-AWP systems will be briefly examined.

4-1 PACT Cognitive Tutors

At the Pittsburgh Advanced Cognitive Tutor (PACT) Center of Carnegie-Mellon University, a number of tutoring systems have been created. The original goal of this research was to evaluate the ACT-R cognitive theory [16], and its predecessor ACT* [10], by modeling how people use and acquire skills. The systems produced at CMU and other systems inspired by them created elsewhere are commonly called *cognitive tutors* [13]. These tutors all focus on acquiring skills through supporting the student when solving exercises.

Currently, the major cognitive tutor systems used at PACT are the LISP Tutor [14, 12], which teaches basic LISP through a self-paced course given at CMU, and a series of mathematical tutors in the geometry [46, 35] and algebra domains [9]. These tutors now covers the curriculum of high school geometry and algebra courses in the US, respectively. The mathematical tutors have been deployed to about 150 high schools in the USA [35] and are in active use.

A great deal of research has gone into these systems, both in evaluating them, formulating and reformulating particular theories of learning. They have been subjected to a wide range of evaluations, e.g by comparing them to conventional problem solving environments [13], but they have also served as platforms for a wide range of experiments on particulars in tutoring, such as how different types of feedback affects the student's ability to correct a mistake [49]. This extensive work makes the cognitive tutor systems the most studied approach to exercise solution support, and is possibly the most well studied approach in all of intelligent tutoring system. This analysis will not cover these efforts in detail, but focus on the more basic foundations of the cognitive tutor systems.

4-1.1 Domain

Description of the domain

The cognitive tutors developed at PACT have focused on either basic level programming (initially LISP, later other languages such as Pascal) or mathematical proof generation in algebra and geometry. The first versions of cognitive tutors on these topics was introduced in the 1980s (LISP Tutor [14, 12] and Geometry Tutor [15]), and has later been evaluated and revised extensively. Currently, the focus of the mathematically oriented tutors are on adaptation to extensive deployment in high schools, while the programming tutors are more tailored for laboratory experiments.

These domains were chosen because they catch the student at the very beginning of learning a topic. They are also fairly constrained and well known, which makes them easier to model. This does not mean that they are easy to model. The approach rests on an assumption that both the cognitive performance and learning models for the domain are well described. However, the domains chosen

are easier for the approach to handle than domains with weaker domain theories or more advanced classes on the same topics that for instance deals with more complex abstractions. On the other hand, solving problems in these domains require more than mere reproduction – they require genuinely new solutions, although the strategies for producing the solutions may be reused. This is in contrast to domains such as learning the multiplication tables and the spelling of words.

Presentation

The cognitive tutor user interface typically consists of a window defining the problem to be solved, and an area where the student can work on composing a solution to the problem. This is typically not a completely free form solution space. For instance in the LISP tutor, the working area is not a completely free form text editor area, but a lisp expression editor that forces the student to maintain a syntactically correct (although possibly incomplete) LISP expression at all times. This makes it possible for the system to continuously analyze the expression and compare it to expected behavior. In addition to this, some cognitive tutors have hint-windows, general help windows (e.g. a manual of LISP expressions in the LISP Tutor) and some indication on how the system believes the student's skill is on different aspects (see Figure 4.1).

Scope

The systems developed so far have focused on mathematical problems and programming. Although the underlying ACT-R theory claims to be a theory for any problem solving activity, and as such is very general, it is probably not possible to construct cognitive tutors for any problem solving domains. Two issues stand out in this regard.

First, programming and mathematical proofs have in common a very formal representation and a limited set of operations, which make it possible for the cognitive tutor to monitor the student's work. To enforce this, the cognitive tutors use editors that force the student to keep within the boundaries of the formal representation. In the LISP Tutor, this can be done by using custom editors that enforce syntacical correct statement. This is even clearer in the algebra tutor, where the student is restricted to inserting pre-made boxes that represent problem solving steps and then filling out values for these boxes (see Figure 4.2).

Areas that lack this formal representation are harder to teach with a cognitive tutor. For instance, it may be possible to make a cognitive model in the ACT-R style for how to write business letters to a certain level of abstraction. The letter should, for instance, have a sender, recipient, title, body and signature, and the body should contain certain elements depending on the purpose of the letter. This breakdown of elements could go from goal to subgoal much like in computer programming, but the final product must consists of a natural language text, which is very hard to interpret computationally. One can argue

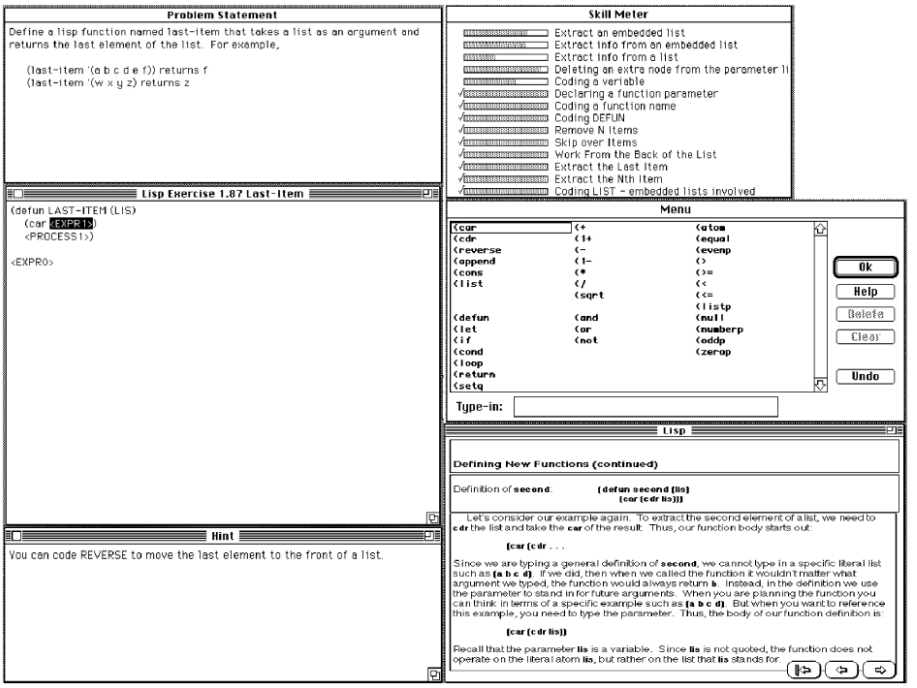


Figure 4.1: The PACT Lisp Tutor Interface (from [35, p.83])

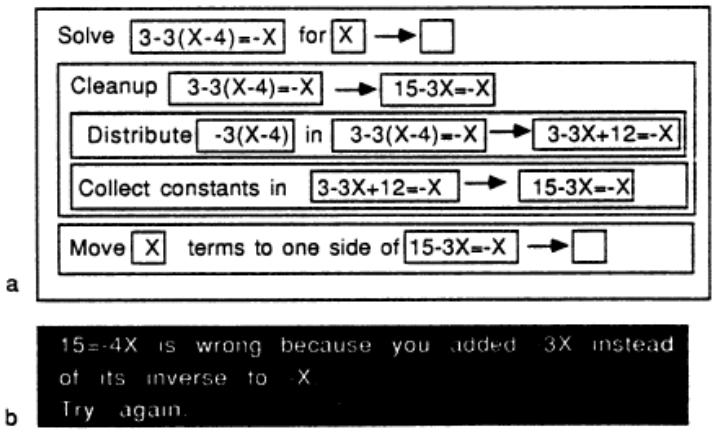


Figure 4.2: Representing the tree-structure of problem solving with a box notation in the PACT Algebra I Tutor, from [13, Figure 3, p.178]

that natural language too, may be a (large) set of production rules on how to form and interpret sentences that is present in everyone's mind, but it is likely that the variations between people in this capability make them hard to model in a computer. This can be seen as a special case of the general problem in AI to operate in domains where generating or interpreting a problem depends on "general" or "common sense" knowledge, so this is not a particular problem associated with the ACT-R theory or cognitive tutor systems, but it does highlight the approach's dependency on quite accurate models of the knowledge required to solve a problem in the domain.

Second, the cognitive tutor approach requires good general models both on performance and learning. In many domains, such models are not available and may be very hard to create. As discussed in Chapter 1, these are weak theory domains, and for instance medical diagnosis as well as many engineering tasks, fall into this category. To this, it can be argued that although it is not currently possible to create a complete model of e.g. medical diagnosis, it is possible to create a cognitive model on how people solve problems, imperfectly. Indeed, this has been done, e.g. in MYCIN and other expert systems. However, creating such models is can be very time-consuming, and experience from expert systems suggests that it is hard to create competent models covering more than a narrow field.

4-1.2 Cognitive Model

The ACT Theories

The cognitive tutor systems are based on the ACT theories of skill acquisition. This was initially known as the ACT* theory [10], while the current revision is known as ACT-R [16], or Atomic Components of Thought. This theory is related to the production rule approach to building expert systems, for instance as used in the MYCIN system [33], but it goes further in that the production system is seen as a cognitive model for how people actually solve problems. It also contains a theory of how learning occurs through the formation of rules. Annderson [13] describes the three central concepts of the theory as:

Procedural-declarative distinction. The ACT theories divide knowledge into declarative knowledge, which is general, conceptual and goal-directed knowledge about the domain, and procedural knowledge, which is to know how to act or solve a type of problem. Declarative knowledge can for instance be to know the side-angle-side theorem in geometry, while procedural knowledge is to know how to apply the theorem in constructing a proof. The assumption is that while declarative knowledge can be learned through observation or direct instruction, procedural skills can only be learned by converting declarative knowledge into applied knowledge such as rules.

Knowledge compilation. Because procedural knowledge cannot be learned directly, there must be a mechanism for converting declarative knowledge into

rules. The theory claims that this can only happen in the context of a problem solving activity. When solving a problem or examining an example, a student can use interpretive strategies such as analogy to generalize from examples by associating declarative knowledge with task goals.

Strengthening. The theory assumes that accurate encoding of knowledge requires repetition and practice, and that practice even after successful encoding leads to faster, smoother and less error-prone reasoning. On the other hand, the use of "weak" knowledge, that is knowledge that is not well practiced, can result in slips and errors.

This theory led to an approach to tutoring where the students are given initial lectures in order to learn the declarative knowledge, before going through guided problem-solving sessions. The goals of these sessions are to allow the students to form the procedural skills required to solve problems, and then foster the practice of the skills until they reach a level where they can be applied without slips and errors.

In a computer-aided tutoring context, the focus of the approach is on the formation and strengthening of procedural knowledge. The assumption is that acquiring declarative knowledge is relatively straightforward, but that declarative knowledge alone is inert and cannot be actively used. The challenge then is to operationalize the knowledge by forming procedural rules. These rules cannot be learned simply by being told what they are; they must be acquired by actively solving problems. The goal of the cognitive tutor systems is to create environments where these skills can be trained while the student is monitored and appropriate feedback is given to check the students' acquisition.

Principles for Cognitive Tutor Design

From the ACT* theory, eight principles for the design of cognitive tutors [13] was formed:

Represent student competence as a production set. The ACT-R theory claims that procedural skills can be represented as production rules, and as an intelligent tutoring system must be able to analyze a student's reasoning process, it must also contain an accurate model of the student.

Communicate the goal structure underlying the problem solving. An assumption of the ACT theories is that problem-solving consists of a process of breaking down goals into subgoals, which can themselves be further broken down until the problem can be solved directly. In order to break down goals to subgoals, the student will need to know the goal structure well, but many students had problems in mastering this structure. It was found that this goal structure was not communicated clearly. This led to the adoption of visualization techniques in the mathematics tutors, whicdisplayed the goal breakdown of particular problems in trees, in order to highlight the goal structure and how

to break down a problem. Figure 4.2 illustrates how this is done by a box notation for the Algebra I tutor, while the Geometry tutor used a graphical tree structure.

Provide instruction in the problem-solving context. Research on situated learning suggests that learning is dependent on context [11]. Although ACT does not provide a detailed theoretical interpretation of this, experimentation has led to the adoption of providing instruction before a new section of the tutor is introduced.

Promote an abstract understanding of the problem-solving knowledge. When observing students, it was found that they tended to retain learning from problem-solving sessions as very specific examples. In the language of ACT, they would form production rules that fit the specific example, but not generalize these productions well. The cognitive tutor should support the formation of correct abstractions.

Minimize working memory load. For a rule to be learned, ACT requires that all information relevant to the goal and action of the rule must be present in working memory. Because of the limited capabilities of working memory, the theory suggest learning can be hindered by placing additional requirements on the working memory. A consequence of this is that the approach advocates learning one or a few things at a time in order to minimize working memory load. They acknowledge that this is at odds with the constructivist "apprentice learning" model, which stresses the importance of learning while being exposed to the complex interactions between different areas.

Provide immediate feedback on errors. The original ACT* theory claimed that new production rules are formed from problem-solving traces, and that the longer one wait before an error was corrected the longer a problem-solving trace must be involved in fixing the error. The conclusion from this was that errors should be repaired as early as possible. In practice interrupting the student at the very moment an error is made. This has been revised somewhat in the ACT-R revision, which claims that learning occurs from the product of problem solving. This means that although the student may have followed incorrect paths for a while when producing a solution, if the correct path was found in the end, the incorrect paths would not be associated with producing the correct solution. This allows for less immediate error correction. In practice, immediate error correction is still often provided in order to cut down the time spent in error states as well as to facilitate interpretation of students' solutions.

Adjust the grain size of instruction while learning. The idea of this principle is that when the student learns, the granularity of problem solving steps will increase. The tutor should be able to follow the development of the

student when these larger steps are formed, and be able to analyze these larger single steps.

Facilitate successive approximations to the target skill. Often, a novice student is not able to perform all the steps required to perform a skill. Instead of trying to force the student to do this, the cognitive tutor approach suggests that the tutor should be able to fill in remaining steps for the student in the beginning, and slowly decrease this support until the student is able to solve the problem completely on his own.

Cognitive Tutors and Constructivism

The ACT view of learning and skill acquisition rests on many of the assumptions of the instructional design approach. The strong dependency, both on problem solving (expert) models and learner models rests on the assumptions that these models are sufficiently universal to be useful across wide populations of students. Further, it is assumed that these models can be accurately used to determine how the students will perform and what "bugs" are present and how to fix them. However, the approach also rests heavily on learning-by-doing, and even goes as far as to claim that procedural knowledge cannot be learned through instruction alone. This is also a constructivist view, although it should be said that the problem solving tasks in the cognitive tutors are very limited in scope compared to what constructivists usually have in mind. For instance, the LISP tutors tend to focus on single-function programs, and the Geometry Tutor limits problems to the mathematical domain without embedding the problem in a larger task where such calculations are needed. As such, the exercises might seem artificial and not teach the student how and when to apply these skills in more realistic contexts. The authors does acknowledge this dependency on reductionism, but also claim that their evaluations show that at least in their domains, this assumption is warranted [13].

The PACT Center has done several evaluation measuring how students do in solving problems outside the cognitive tutor environment. They have done this by administering tests to the students after they have passed through the course, and comparing cognitive tutor-assisted students to control groups. This has shown a significant positive effect of the tutors, as long as the teachers are familiar with the approach and the system [13]. The wide deployment in as many as 150 different high schools also suggest that the cognitive models used have a certain universal applicability, although it may be that the choice of domains help in this regard. The formal nature of mathematics and programming, as well as the relative low level of knowledge students have on these topics in advance, may be helpful in reducing any unforeseen interactions with pre-existing knowledge. All in all, the success of this approach does suggest that the more radical constructivist critiques are less of factor in these domains. This said, the authors do recognize the value of for example "apprentice learning" and actively supports the idea that instruction should happen in the problem-solving context (principle 3, above).

```
IF    the goal is to merge the elements of list1 and list2
      into a list
THEN  APPEND and set as subgoals to code list1 and list2
```

Figure 4.3: Sample rule from the LISP tutor, translated to English [11, p.9]

4-1.3 Knowledge Models

The cognitive tutor approach contains all the three typical models of an intelligent tutoring system. The expert, or target model is the set of production rules that the instructor wishes the student to learn. This is explicitly represented. In addition to this, some of the cognitive tutors have a set of "buggy" rules, which represents misconceptions and faulty rules the students may have formed.

The ACT theories stipulate a difference between procedural and descriptive knowledge. The declarative knowledge contains theoretical and conceptual knowledge, as well as examples and goal structures. The procedural knowledge is tied to problem solving activities and is represented as production rules.

Expert Model

The ACT-R theory claims that skill competence is a product of learning declarative knowledge and applying this knowledge to problems until a set of production rules are learned. After sufficient practice, this can be applied to any problems within the scope of the model. This view of human problem solving is replicated directly in the cognitive tutor expert models in a fashion that is close to rule-based expert systems. Because ACT-R predicts that skill competence is a function of how well the procedural rules have been learned, the focus when creating a cognitive tutor expert model is on designing a set of production rules that covers all the correct ways of solving the problems. The redundancy in problem solving approachs is important because the cognitive tutors analyze students' problem solving by a process of *model tracing*, where the assumption is that any correct step the student takes in solving a problem must be represented as a rule in the expert model. In addition, the expert model contains rules that represent common misconceptions in the domain. If the student performs a problem-solving step that matches one of these misconcepttions, he may be corrected by using specific information associated with this rule.

This makes the expert model in a cognitive tutoring system an extremely important component, and also places restriction on how the rules in order for them to match human problem solving strategies. For instance, the importance of goal structures in the ACT-R theory means that the rules in cognitive tutor systems tend to be organized around the goal-subgoal hierarchy, and the rules often have among the conditions that a certain goal must be active for them to fire (see e.g. table 4.3).

Student Model

The student model in modern cognitive tutors consist of a kind of probabilistic overlay model called *knowledge tracing*. For each student, every rule has a value associated with it between 0 and 1, which represents the system's belief on how well the student knows the rule. When a rule is successfully applied by the student, this value is increased, while the failure to apply a rule in a situation where it should be applied decreases the value. The exact mechanisms for this is based on conditional probability and is presented in [35].

The goal of the student model is to monitor when a student master the full set of rules. At this point, the cognitive tutor will suggest that the student has mastered the part and may move on to other topics. It may also be used to select problems that require rules that the student has not already mastered and as such tailor the experience to the student.

The interface screenshot of the LISP Tutor in Figure 4.1 shows that the information gathered by the knowledge tracing is compiled into more abstract categories, and that the value for these categories are displayed to the student. This provides some level of inspectability of the student model, although on an abstract level.

Pedagogical Model

The whole ACT-R theory is concerned with skill acquisition, and as such it can be called a pedagogical model in itself. The ACT-R theory is used to guide the process of creating the cognitive tutor interface and how to construct rules. The kinds of decisions influenced by this learning theory is quite wide ranging and includes such choices as:

- Limiting the amount of windows in the work area to minimize working memory load.
- Explicitly representing the goal structure, e.g. by including tree graphs of the solutions in the geometry tutor.
- Limiting the freedom for students when solving problems, e.g. by including an expression editor in the LISP tutor.

This is not the same as having an explicit pedagogical model, where the system can reason over tutoring strategies and decide what is best in a given situation. Such a pedagogical model is indeed described as a part of the cognitive tutor architecture (e.g. in [11]), but it has not been a part of the classic implementations. The wide range of evaluations and testing done on the cognitive tutor systems, means that the theoretical fundament behind them has been tuned to include a wealth of information on specifics, such as how explanations should be formulated in order to facilitate understanding beyond the tutoring environment [49] and detecting if students are "gaming" the tutoring system [21].

4-1.4 Tutoring Capabilities

Goals

The main goal of the cognitive tutors is to assist the student in forming operational skills by helping them solve exercises. The ACT-R theory states that the important knowledge structures involved in actual problem solving can only be constructed and reinforced through practice, so the focus is on assisting in this process.

Based on this goal, the cognitive tutors are designed to support the operationalization process, which means that it excludes for instance conceptual learning or exploration of the domain. The goal is to help the student in acquiring the exact rules represented in the expert model as well as possible.

Towards this goal, the model tracing approach effectively limits the "moves" a student can take on a solution path to those represented explicitly by a premade rule in the expert model. Although later revisions of the ACT-R theory places less of an importance of correcting a perceived error immediately as long as the student eventually finds the solution, this "correct early" approach is often preferred for pragmatic reasons. When the student venture into unknown territory, tracking the student becomes much more computationally expensive. This means that it is hard for the student to venture away from a path that leads to the correct solution, but this does not mean that the process is linear for the student. Often, a goal is broken down into several subgoals, and in the newer cognitive tutors, the student is allowed to go back and forth between branches in the tree and expand on them as he sees fit.

Exercise Selection

The knowledge tracing approach of student modeling allows the cognitive tutors to keep a model of how well the student knows each rule in the expert model. When combined with a set of problems and the rules required to solve these problems, this allows the tutor to select problems for the student that requires rules for which the student has not yet demonstrated mastery. This is combined with rules into self-contained sets of increasing difficulty, where the student must demonstrate a sufficient level of mastery on one level in order to advance to the next. This is done to limit the learning load of the student, so that he will not face too many new rules at once.

In-Exercise Support

The main mechanism for in-exercise support in the cognitive tutor systems is the model tracing of the problem solving process. This is not a process that is tailored to the specific student in any way, but interprets each problem solving step and attempt to match it to an existing correct or buggy rule in the expert model. If the input from the student is insufficient to determine what the student attempts to do, the system asks disambiguation questions until it has determined a matching rule. If this rule is correct, the problem solving activity

is allowed to continue uninterrupted, although if it matches a buggy rule, an error message is given along with an explanation as to why this step is not correct.

The model tracing technique allows for a very tight coaching of the problem solving process, but it also constrains it heavily. The student is not allowed to experience on his own why a strategy pursued will fail, but is instead offered an explanation after the very first step down the path. However, this does help the student stay on the correct path to the solution, and given that the ACT-R theory is correct, enforces the rules used in the production of correct solutions.

Conceptual Support

The focus of the cognitive tutors is not on the language, terminology and concepts of the domain. In fact, the use of the tutors assumes that this is taught in an instructional setting in advance. The reason for this is that it is assumed that declarative knowledge can be taught well through textbook reading or classroom instruction, and as such is not the focus of these systems. However, the newer implementations of the LISP Tutor does have a window dedicated to manual-like information on LISP functions, as reference material.

Explanation Ability

The major explanation capability of the cognitive tutors, is the misconception explanations associated with the buggy rules identified during model tracing. When the system finds that a problem-solving step taken by the student matches a misconception rule, the information stored in this rule is presented to the student in an attempt to clear up the misconception. This information may take the form of preformed text, as the misconception rules tend to be quite specific. The goal of these explanations is to improve learning by invalidating buggy rules for the student.

If the student is stuck when solving a problem, the system may also provide hints. This can (initially) take the form of suggesting a goal to pursue, and if that fails, to suggest a problem solving step. This can be done by searching the expert model rule set to find a rule that can be applied to the current problem state. The result of applying this rule can then be presented to the student, along with a premade text associated with the rule, explaining why it is a good choice. This fulfills both the justification and transparency goals.

Other kinds of explanations are not the focus of the cognitive tutor systems, although they do have the knowledge sources to produce relevance explanations in order to justify why the system selects a given problem for the student to solve.

Learning Capabilities

In [11], the authors write that they typically manage to capture about 80% of the buggy rules encountered in the expert model. If a student tries to do

something that may not be captured by any of the existing rules, this action may be the basis of a new buggy rule. However, the explanations associated with these rules cannot be created automatically, so the process of defining these is partially a manual process. Beyond this, the cognitive tutors do not learn or acquire knowledge automatically during operation.

4-1.5 Evaluation

The cognitive tutor systems have been subjected to numerous evaluations not only of the total system, but particular elements and new theories. Only some will be covered here, but see [13, 11] for more extensive overviews.

The first Geometry Tutor was used in a pilot in the school year 1986-1987, and the results showed that access to the tutor program increased the grade significantly, by more than one standard deviation (which works out to more than one letter grade) [13]. A newer iteration of the geometry tutor was also tested in [44], which concluded that the tutor resulted in a significant positive result, but only if the teacher was integrated in the project.

A similar evaluation of the first Algebra Tutor showed no difference between the experiment classes using the algebra tutor and the control classes using traditional methods [13]. The authors suggest that two reasons for this may be that the notation used in the algebra tutor was different from what used in the classroom, and social factors (such as class attendance and motivation) that seemed a large factor in the school tested. A newer iteration of the algebra tutor was tested on over 500 students in three schools, and showed an increase in standardized test scores of 15% compared to control groups. On tests that targeted the objectives of the tutor, the experimental groups performed 100% better [45].

The LISP Tutor has also been evaluated multiple times through its history. For instance, an evaluation on a self-paced LISP course, the students using the tutor solved the exercises 64% faster and scored 30% higher on a post-test than students using a standard LISP environment [13].

4-2 Episodic Learner Model (ELM)

The Episodic Learner Model [86] is an approach to student modeling that uses the concrete previous learning episodes of a student. This was first explored in the ELM-PE system [86], which is an interactive environment where students can solve LISP exercises, and later in the ELM-ART systems [87], which combined exercise solution support with the contents of an adaptive, interactive online textbook[1].

Although there are differences, the exercise support in ELM-PE can be thought of as an extension of the model-tracing approach in the cognitive tutor

[1] ELM-ART can be accessed through the web at http://apsymac33.uni-trier.de:8080/Lisp-Course (last access 25.03.05)

systems. Like these systems, the solution (a LISP program) is diagnosed in order to provide a tree of rules that was used to produce the output. However, the ELM systems attempts to use concerete prior knowledge from problems solved earlier by the same student. The assumption is that the student will attempt to reuse an earlier problem-solving trace as much as possible as much as possible and only adapt the solution where the new problem does not fit the old.

In ELM-ART, the student model is extended to also cover theoretical knowledge and how the student has been exposed to different parts of the curriculum. Combined with a model of dependencies in the curriculum, this allows the system can offer advice on what the student should address next, and anticipate which exercises are appropriate.

Empirical evaluation has been carried out where ELM-PE and ELM-ART is compared to test groups where these systems are not used, and they have also been compared to each other. These studies are described under Section 4-2.5.

4-2.1 Domain

Brief description of the domain

The domain of both the ELM-PE and the ELM-ART systems are introductory programming classes in LISP, with particular focus on recursion. Because both these systems are designed to give in-exercise support on programming tasks, they require a fairly deep understanding of the programming problems to support as well as the programming language itself.

Presentation

The ELM-PE system was designed to work as a standalone application environment on Macintosh computers. As this system only provided exercise support, it's user interface is similar to the LISP Tutors in that it has a text editor area where the student can form the lisp expression, as well as the problem description.

In creating the ELM-ART, the authors decided to create a purely web-based interface that would serve as a complete course. This means that unlike ELM-PE, it contains both the more theoretical background knowledge traditionally found in textbooks and an environment for solving exercises. The course application is organized so that exercises of different kinds (such as multiple-choice or more free-form programming exercises, as for instance illustrated in Figure 4.4) are integrated in the theoretical curriculum.

Although ELM-ART contains a default path through the curriculum for beginners, it also allows the student to access any part of the course at any time. The course is arranged in a tree-structure, with abstract lessons at the top and specific text pages and exercises further down. All of these pages are tied to concepts in a domain model, which also contains information on dependencies between the concepts. Because the system's student model keep tracks of what concepts the current student is familiar with, it can infer what

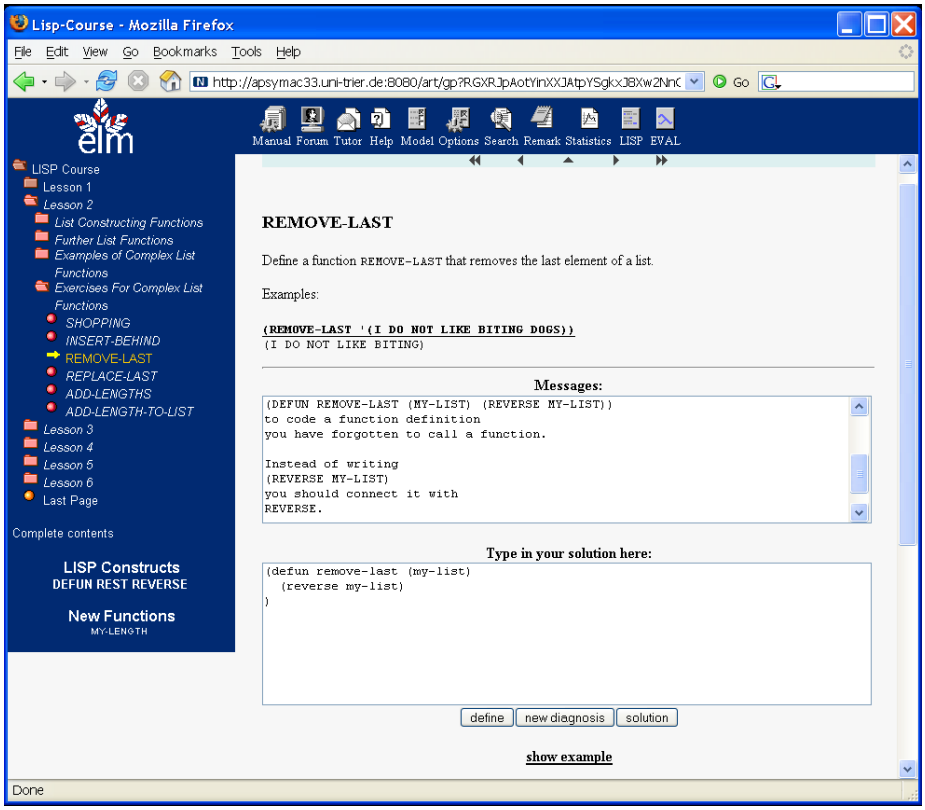


Figure 4.4: Solving a LISP exercise in ELM-ART

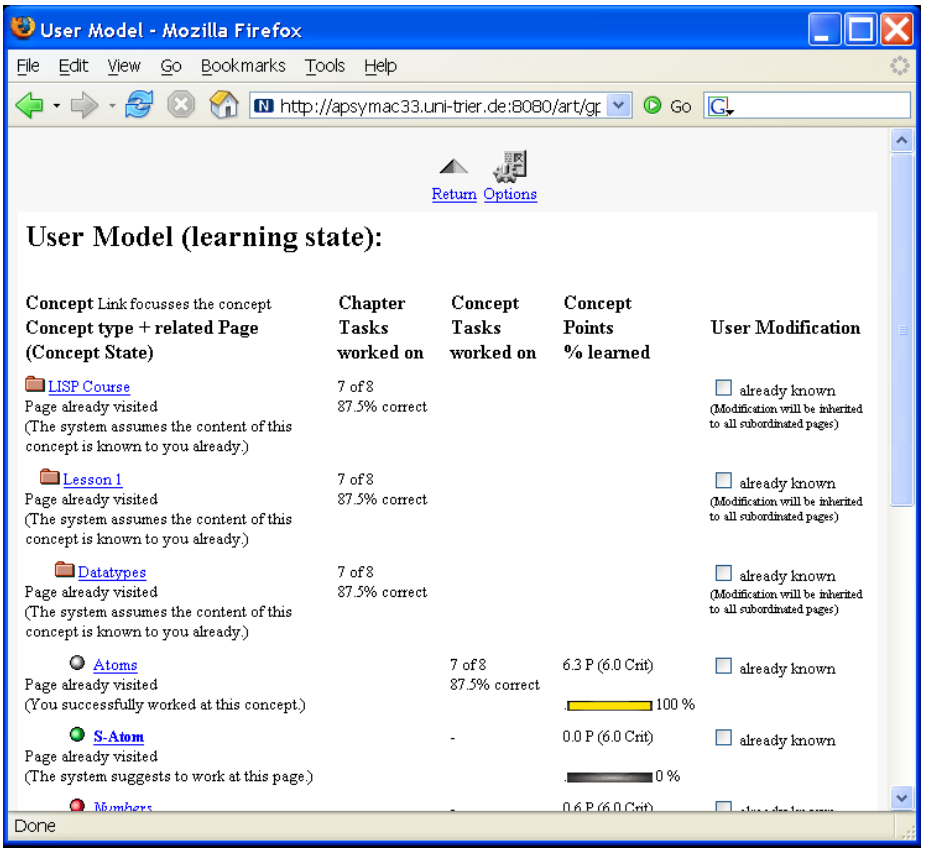


Figure 4.5: Inspecting and modifying the student model in ELM-ART

concepts (and thereby pages) the student would be ready to address. However, the student model may not be accurate, so the student model is only used as advice. Through color codes, the different pages are marked as "assumed to know", "assumed not ready for" and "assumed ready for". The student can also manually inspect and update the student model in order to inform the system about what is already known (see Figure 4.5).

In addition to these core features, ELM-ART also has discussion forums, a chat interface, an online LISP manual and a free-form LISP evaluation engine.

In the ELM-PE system, the student solves LISP exercises in a special expression editor similar to the one used in the PACT LISP Tutor, which hinders the student from making syntactical errors such as missing right or left parenthesis. These types of errors make the program impossible to parse as a tree structure, which the ELM program diagnosis requires. The ELM-ART system does not limit the student in this way, but does a preliminary syntactical check of the expression and will issue an error message if there is a syntactical error.

If it does not find any syntactical errors, it will proceed to diagnose the program using its domain knowledge of LISP programming and the task at hand. If the program fails to solve the goal of the exercise, advice is offered on what the student should do, first through a hint and if that is not enough, by providing an adaptation of the program that solves the problem. A major difference between the PACT LISP Tutor and ELM-ART is that the ELM-ART system does not perform diagnosis continuously, but allows the student to work until diagnosis is requested by the system. This increases the freedom of the user, but also means that the tutor cannot take initiative to correct errors in the way the PACT LISP Tutor does.

Scope

The current ELM systems have only been used on LISP programming tasks. However, to some degree, other courses could be made based on the same system. The part of ELM-ART that relates to course management, textbook knowledge and multiple-choice tests are clearly usable in a wide range of domains, and a commercial course authoring system called NetCoach[2]. NetCoach can be found at [88]. However, the parts of the system concerned with in-exercise support for LISP programming tasks might not be as widely relevant. This does not mean that it is only useable in LISP or other programming domains – like the cognitive tutor systems, the idea behind the LISP programming diagnosis may be applied in other domains. However, as this part shares the goal-oriented rule-based approach with the cognitive tutor systems, it is likely that it is most useful in domains with a formal syntax (i.e. formal languages) and strong domain theory.

4-2.2 Cognitive Theory

Episodic Learner Model

The theoretical basis for skill acquisition and problem solving in the ELM systems are similar to that of the cognitive tutors systems in that both systems distinguish between the declarative and procedural knowledge, and that procedural knowledge can be well represented by production rules. ELM uses a somewhat more elaborate language for storing declarative knowledge, where concepts (such as LISP expressions, problem solving goals and schemas) are stored as frames with slots relating them to other frames (representing other concepts). Initially, when there is no specific knowledge about a student, the in-exercise support offered by ELM is very similar to model tracing in the cognitive tutors. The goal of the exercise is broken down into sub-goals and then programming patterns, which is then matched to the code produced by the student.

A problem in the PACT LISP Tutor was that the model tracing approach found several different paths that could explain a reasoning step taken by the

[2] Available at http://www.net-coach.de (last access 10.03.06)

student. In these situations, the PACT LISP Tutor would ask discrimination questions in order to understand what the student was doing. This was done in order to reduce the number of parallel traces the system would have to track so that the speed of the system would be acceptable. ELM has another solution to this problem, based on the observation that students, when solving exercises, adapt solutions from earlier, similar problems when solving a new problem [86]. This observation means that when faced with two different interpretations of what a student is attempting to do, the path this student has followed when solving another problem, or the student has seen in an example, should be preferred. The ELM system creates a student model based on episodic information – the previous programs created and examined by the student. This is used by ELM as en heuristic in the model trace search, but also in providing hints and tips to a student during exercise solving. By reminding the stuck student of similar examples he is familiar with, or even created himself, the student may see structures that can be useful in solving the current problem.

ELM-ART and Constructivism

The ELM-ART environment is clearly built over an instructional design framework. An expert in the domain predetermines all the goals of the course, the contents and the evaluations. While the system has a default path that is designed for the complete novice, it does have quite sophisticated techniques for allowing more advanced users to enter the course at their skill level. It does this by asking some initial questions to new users (such as computer literacy and pasts programming experience), and by allowing them to inspect and edit parts of the student model. ELM-ART can also infer that a student has knowledge about basic concepts if the student jumps to more advanced sections of the course and successfully solves exercises. This suggests that the ELM-ART system rely less on total replicability than a system where the progression is not as adaptive. However, it still assumes that its expert model is universal, even for students with prior knowledge that is not obtained through ELM-ART. For instance, ELM-ART supposes that knowing about LISP atoms is a prerequisite to solve list-processing problems, such as removing the last element of a list. If a student solves this problem, ELM-ART concludes that the student most likely also knows about LISP atoms. The assumption of universalism also extends to the conceptualization used to describe the course. If the student is to usefully inspect and edit his student model, he must have a shared idea about what the relatively short descriptions of the different knowledge units are.

The ELM system also shares with the cognitive tutor systems a requirement for a strong domain model in order to provide in-exercise support. Although ELM collects concrete episodes (cases) from students, it can only do so if the program traces already matches a model trace through its expert model. This means that it can only interpret student cases through its expert model, and is not able to learn from cases that go beyond its existing model – it can only mark what parts of its existing model was used in this particular case. Since the programming tasks in ELM-ART are fairly limited (single-function without

iteration), it is possible to describe the domain well enough that this may not be a problem here, but it may very well be a problem in larger programming tasks.

The assumption that programming can be learned through very many small task illustrates a dependency on reductionism – the assumption is that once the student learns to create recursive functions, he can create larger programs by putting these together. However, no exercises or tests of this capability are provided by the system.

Conceptual and Procedural Knowledge

The ELM theory separates between conceptual and procedural knowledge in the same way as the cognitive tutor systems. However, the ELM-ART tutor integrates the teaching of conceptual and procedural knowledge much closer than the cognitive tutor systems, where instruction and skill practice are separated into different lessons. The ACT principles for tutor design does state that instruction should happen in the context of procedural training, but in practice it has not been as integrated as what is done in the ELM-ART tutor. However, this tight integration means that an exercise will typically be set to follow a particular instruction page, and after solving this, the student would go on to the next topic even if he was unable to solve it and had to ask the computer for a solution. This is in contrast to the PACT LISP Tutor, which assigns new exercises until it deems the student skilled enough in that area. Dividing the time between instruction and exercise solving allows the system to more dynamically choose appropriate exercises. If the student is used to be provided with exercises that is very tailored to the instructional topic at hand, it might be hard for the system to then introduce exercises that addresses misconceptions from other parts of the curriculum. In other words, while ACT recommends that instruction should happen in a problem-solving context, ELM-ART provides problem-solving in an instructional context.

4-2.3 Knowledge Models

The three main forms of domain knowledge in the ELM system are concepts, rules and task descriptions. The concepts and rules roughly correspond to the declarative and procedural units in the cognitive tutor system, while the task descriptions are machine-interpretable representations of exercise tasks.

The ELM systems also have a conceptual separation between the expert and student models, although they have a common representation.

Expert Model

The expert model in the ELM system may be separated into the online textbook and the diagnostic model. The online textbook knowledge is used to teach the student the concepts and problem-solving strategies theoretically, and helps the system analyze what parts of the curriculum should be presented to the

Name:	**NIL-TEST**
Type:	**static**
Abstractions:	**(Procedure Fn-With-Boole-Res Equality-Rel)**
Specializations:	**NIL**
Parameter:	**(?Arg)**
Sorted-Rules:	**(Negation-NIL-Test-Rule Equal-NIL-Test-Rule Empty-List-NIL-Test-Rule)**
Transformations:	**((NIL-TEST ?EXPR) → (NOT (T-TEST ?EXPR)))**

Figure 4.6: Part of the ELM frame representation for the concept `nil-test`, from [86, Table 1, p.202].

student, while the diagnostic model is used to interpret student program solutions in a way similar to the strategy in the PACT LISP Tutor. However, both these models include references to basic concepts in the domain, such as LISP functions. This means that they are not completely separate models, which is interesting in light of student modeling as it open the potential for the system to know if a concept is just read about or actually actively practiced.

In ELM-PE, which only contains the diagnostic model, the concepts are declarative units of knowledge and in the LISP systems they represent concrete LISP functions (e.g. `first` or `rest`), more generalized semantic concepts (e.g. `List operator`), schemata for common algorithms or problem solving strategies, or diagnostic meta-information. ELM uses a frame-based language, which means that each concept has a number of slots with every concept. All concepts contain slots for the name, type, abstractions and specializations of the concept. The two last slots are used to form a taxonomy of concepts with more general, abstract concepts as superclasses for specific concepts. In addition to these, there are type-specific slots, such as the parameter and sorted-rules slots in the `nil-test` concept shown in Figure 4.6.

With the addition of the online textbook in ELM-ART, the conceptual model has been extended to include information about lessons, sections, subsections and pages. A page can contain information on a subject, a test, example or an exercise. In addition to what is presented to the student, the concept also has information on how it relates to other concepts, which includes dependency information. Combined with a student model, this allows the system to infer if a student is likely to be able to understand a concept based on what other concepts the student is familiar with. The rules represent the procedural knowledge. These rules are designed to be used in diagnosing student programs, where the initial state of the problem (containing the goal) is transformed to a solution by going through iterations where the initial state is broken down into a sub-goal hierarchy, and where the leaf nodes are matched to actual LISP expressions.

Rules are also stored as frames, although with a different set of slots than the concepts. Rules also have an abstraction/specialization hierarchy, but also contain information about the preconditions for them to be applied and the con-

Name:	Equal-NIL-Test-Rule
Type:	static
Abstractions:	(Suboptimal-Rules Not-Semantic-Relation)
Specializations:	NIL
Quality:	suboptimal
Priority:	6
Frequency:	16
Recency:	27116
Activation:	0.495196
Precondition:	P.T
Consequence:	(SOLVE-PLAN (EQUALITY ?ARG (TRUTH-VALUE NIL)))

Figure 4.7: Part of the ELM frame representation for the rule `equal-NIL-test-rule`, from [86, Table 2, p.203].

sequence of applying them. In addition to this, information about the quality, recency, activation and priority is used to prioritize rules (see Figure 4.7. The rules may represent both correct and incorrect transformation.

While the correct rules may be on the path to a correct solution to the problem, incorrect rules represents mistakes students may make when trying to solve the problem. Just like the PACT LISP Tutor, the ELM systems contain both a correct and "buggy" model of the domain. Lik in the PACT LISP Tutor, the buggy rules allows the tutor to recognize errors and present explanations tied to that particular buggy rule. To this end, the rule frames also have various slots for storing such textual explanations.

Last, the actual problems presented to the students are also stored in the frame-based system. The frames representing tasks contain both a textual description that may be presented to the student, a reference solution, several test-cases with expected result, and a high-level plan, which is the initial state to be uses by the diagnostic process when analyzing a student's proposed solution. An example of a task-description frame from ELM-PA can be seen in Figure 4.8.

The diagnostic model in ELM must be strong in the sense that it has to be able to solve all the tasks and do so in all possible ways a student might imagine if it is to succeed in the diagnostic process. This does not mean, however, that it must be a complete model of programming LISP or even of the limited set of LISP programming tasks covered by the curriculum. In principle, the model needs only be able to solve the specific tasks given to the student throughout the course. Because the tasks the student is given is known in advance, the model can be tailored and checked against the actual tasks. This makes the process of forming the expert model considerably easier than the model had to be able to solve any conceivable problem.

```
Name:       Simple-And
Text:       "Define the function SIMPLE-AND. Its argument must be a list of truth
            values (Ts and NILs). SIMPLE-AND tests if the value of all list elements is T
            (or non-NIL). SIMPLE-AND then returns T, else NIL."
I/O-Descr:  (T T T T) --> T, (T NIL T) --> NIL, (NIL NIL NIL) -> NIL, () --> T
Algorithm:  1
            Ref.-Sol.:  (defun simple-and (li)
                          (cond   ((null li) t)
                                  ((not (car li) nil)
                                  (t (simple-and (cdr li)))))
            Type:       DEFINE-PROCEDURE
            Params:     (?LIST)
            Body:       CDR-END-RECURSION

             Rec-Parameter:          ?LIST
             Case 1:   Test:         (NULL-TEST (PARAMETER ?LIST))
                       Consequence:  (TRUTH-VALUE T)
             Case 2:   Test:         (NIL-TEST (FIRST-ELEMENT (PARAMETER ?LIST)))
                       Consequence:  (TRUTH-VALUE NIL)
             Case 3:   Test:         (ELSE-TEST T)
                       Consequence:  (CDR-REC-CLAUSE (PARAMETER ?LIST))
```

Figure 4.8: Part of the ELM frame representation for the `Simple-And` task description, from [86, Table 3, p.204].

Student Model

The student model in the diagnostic model – the Episodic Learner Model – is based on problem solution traces from all the problems seen by the student. These are not stored as a unit, but broken down so that all concepts involved in the trace are described in separate frames. For instance, if the problem trace contains a `nil-test` (Figure 4.7), a specialization of the general `nil-test` is created, and the rule identified as used by the student to implement the `nil-test` is associated with the new frame. This frame is then generalized using a type of explanation-based generalization, where the concrete code is abstracted away, so that for instance a concrete variable name is replaced by a <`variable`> pattern that can be matched to any variable.

There are three goals of student modeling in ELM. First, previous problem solutions can be used as examples to remind students of problem solving strategies when he is solving other problems. Second, it can be used to assist in selecting appropriate exercises for the student. The third goal is to increase the efficiency of the diagnostic process. Because the diagnostic process is searching through a possibly quite large search tree of rule transformations, a good heuristic can be very useful in cutting down on search time. In ELM-PE, the student model is used as a heuristic in that the system tries to match rules that have been used by the student before. An evaluation study of the efficiency gain using this heuristic, shows that the heuristic improved the speed of diagnosis from a median of 7.1 seconds to 2.1 seconds [86].

With the ELM-ART electronic textbook, the student model was extended with information about concepts on a higher level of abstraction. For each concept, ELM-ART records if the student has visited (if he has examined the page

describing the concept), learned (completed tests on this concept successfully), known (manually marked it as known) and inferred (the system has inferred that the student knows it). As such, the ELM-ART student model is a type of overlay model, which contains information not only if the concept is known to the student, but also why the system believes the student knows it. The electronic textbook student model is used to adapt the electronic textbook in various ways, such as marking the part of the course for which the student has all the requisite knowledge. For instance, the user interface of ELM-ART displays this by marking solved exercises and visited text pages with a white ball, pages that the system has inferred that the student knows with an orange ball, pages that the student is not yet ready to address (because of lacking prerequisites) with a red ball, and a green ball means that the student is ready to address that page but has not yet done so. The system also suggests prerequisite topics if a student that attempts to address a "red ball" page despite the advice of the system.

Pedagogical Model

The pedagogical model of the ELM systems is mostly implicit. The ELM-ART system's knowledge about dependencies between the different concepts in the domain may be characterized as a partial pedagogical model. These dependencies are primarily an encoding of the assumptions about prior knowledge made when authoring the pages describing the concept, as opposed to underlying dependencies in the domain itself (although there is clearly an overlap here). T he dependency model affects the order in which lessons are taken, which is clearly a pedagogical decision, and it is a decision the system can take based on the knowledge about the system, the domain and how the domain should be taught. For instance, if a student has told ELM-ART that he knows about Lisp atoms, but not the `rest` and `reverse` list processing functions, he will be advised to read the pages about these functions before attempting to write a function that removes the first element of a list. The system can do this because it knows that the knowledge of Lisp atoms, the `rest` and `reverse` functions are prerequisites to solving that particular problem, and that the student does not know `rest` and `reverse`.

4-2.4 Tutoring Capabilities

Goals

In ELM-PE the goals of the system was to perform efficient in-exercise support, much like the PACT LISP Tutor. This goal was extended in ELM-ART, which attempts to be a complete tutoring environment for distance learning over the web, with instruction, exercises, tests and communication tools such as forums and chat. This difference in emphasis is reflected in the knowledge sources, as the expert model from ELM-PE was extended to include more knowledge required for instruction.

Exercise Selection

The ELM-PA system does not perform exercise selection or advice on that level, although the authors recognize that this could be done in a similar way as done by the PACT LISP Tutor. The ELM-ART system does provide some advice on exercise selection, but does so based on the electronic textbook level of student modeling. This means that ELM-ART will mark a page containing an exercise as green if the student has learned all the required materials to solve that exercise, or red if he has not. However, the finer grained information about what rules a student knows well and less well is not used in here. The focus in ELM-ART may have shifted from exercises to overall course design, and in shifting from a Cognitive Tutor-like model of separated instruction and exercises; the priority is on providing exercises that correspond to exactly what is instructed in that lesson. While this allows the system to provide pinpoint examples and exercises to the topic at hand, it is also more targeted than the PACT LISP Tutor. By suggesting exercises that touch on different parts of the curriculum, the PACT LISP Tutor may provide more opportunity for the student to "criss cross" the different topics of the course so that interactions between them can be exercised. The PACT LISP Tutors can also give further exercises in a specific skill area if the student has problems, whereas the ELM-ART approach seems less able to adapt by giving more exercises in problem areas. Once an exercise has been solved, be it with the computer's help or not, the student is assumed to have learned that topic. This comes at an advantage – the incredibly tight connection between examples, problem solving and instruction. The student gets almost immediate feedback on his understanding of a topic, and misconceptions can be corrected early.

In-Exercise Support

The in-exercise support of the ELM systems is quite sophisticated. Not only can it determine a correct solution from an erroneous one, it can also diagnose incomplete solutions and erroneous solutions and provide assistance in correcting or finishing them. It does this by a mechanism that is in principle similar to the model tracing of the PACT LISP Tutor, although the mechanisms for doing so is different. While the PACT LISP Tutor has a rule-based system inspired by the ACT theories, ELM uses a conceptual network, where the concepts are matched to goals or partial expressions. The rules indexed by these concepts represent different possible correct or buggy solution paths. Although the mechanism is different, both systems can be seen as searching a space of different possible expansions of a code tree, attempting to re-create or trace the code produced by the student. When a match is found, the student's code tree can be compared to a correct solution, and hints provides as to where the student is wrong or missing code. Figure 4.9 contains an example of an ELM code derivation tree.

In an example exercise from the ELM-ART system, the student is asked to create a function that removes the last element of a list. This can be done by using the `rest` function, which removes the first element of a list and the

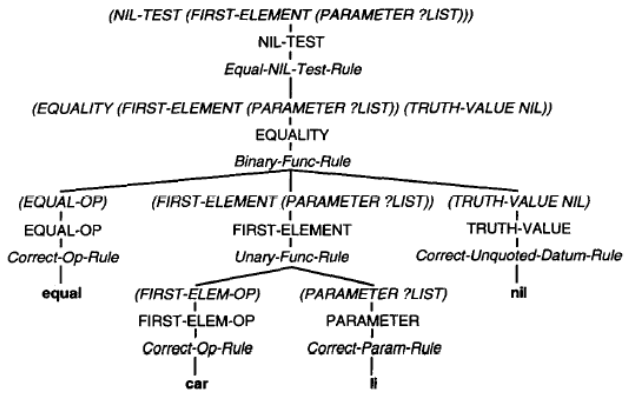


Figure 4.9: The ELM derivation tree for the LISP code `(equal (car li) nil)`, from [86, Figure 2, p.209]. *ITALICS CAPITALS:* plans, CAPITALS: concepts, *italics:* rules and **bold:** LISP code.

`reverse` function, which reverses the order of the elements in a list. By first reversing the order, removing the first item, and then re-reversing the list, the last element is effectively removed. The LISP code for this can look like this:

```
(defun remove-last (my-list)
       (reverse (rest (reverse my-list)))
)
```

There are numerous mistakes a student can do in this exercise. ELM-ART contains no constraints on the text that can be typed in the area where the function is defined. This means that the student can make syntactical errors, such as missing a right parenthesis. However, in order for the diagnosis to work, the system must be able to parse the program, which requires that the syntax is right. In order to ensure this, ELM-ART first does a syntactical check, which can inform that a parenthesis is missing (for instance), although it does not suggest where it might be missing. Further, a student may make a mistake, for instance by forgetting to re-reverse the list after removing the first element:

```
defun remove-last (my-list)
      (rest (reverse my-list))
)
```

Here, an analysis of the code tree compared with the closes solution will show that the student's solution lacks a `reverse`-step. This will result in ELM-ART suggesting that the student apply the `reverse` function (although in our test it did not suggest where or how). Further help can result in transforming the program into a functioning solution. The authors suggest that this is an

important capability in a distance learning system, as it makes sure that all exercises can be solved even without a human tutor.

A last important capability of ELM is to suggest relevant example the student may wish to look at. In doing this, ELM prefers examples that the student is already familiar with, or solutions created by the student himself in earlier exercises.

Conceptualization Support

While the ELM-PA system does not provide conceptualization support, this is a major focus of the electronic textbook in ELM-ART. However, it is based on a knowledge transfer pedagogical view, where the goal is to transfer the conceptualization of the domain represented in the expert model as accurately as possible to the student. The student is taught (and tested on) understanding of concepts according to the material in the electronic textbook. The assumption is that the conceptualization is universal, which may be a useful assumption in a formal domain like LISP, where functions and syntax are precisely defined. This is particularly visible in the inspectable student model, where the student can mark different concepts as previously known so as to skip over these subjects. These concepts tend to be somewhat abstract so they do not overwhelm the student, but the very abstract nature of these concepts may make the computer's semantic interpretation of them hard to understand.

Explanation Ability

The many knowledge sources in ELM-ART makes it possible for the system to provide explanations on many levels. On the electronic textbook level, some explanations are provided in order to justify why the system does not think the student is ready for a particular subject. This is done by applying the dependency model, and referring to subjects that are prerequisites to the subject at hand but not yet mastered by the student.

On the exercise-support level, explanations are provided in the form of hints about missing structures, explanations about misconceptions (in the form of "canned" text attached to triggered misconception rules), as well as illustration through targeted examples. The examples may not seem like an explanation, but in principle it is similar to what is done in case-based reasoning in general – presenting a similar case with some solution as justification for a adapting that solution to the current problem. The implicit suggestion in presenting similar examples is to adopt the problem solving strategy seen in that example, with the justification that it worked in an earlier case.

However, it seems hard for these systems to provide explanations on a strategic level. For instance, in the "remove last element of list" example, no explanation on the level of "no exact function for this exists in LISP, but there is a function, `rest` that removes the first element of the list, and by reversing the list using `reverse`, this can be solved" seems possible. In an example, the student may produce this solution:

```
(defun remove-last (my-list)
       (rest my-list)
)
```

Given this code, the ELM-ART tutor may suggest that the student take a look at `reverse`, which is a good hint, but it does not suggest why this is necessary. ELM does have some capability to attach strategic explanations to task frames, which addresses this goal, but this means that they are to a degree pre-made hints specific to the task at hand, and not generated dynamically.

Learning Capabilities

The ELM systems are generally not designed to learn about the domain, although if a student does come up with a solution to a programming problem that cannot be (completely) matched to any existing structures, the structure of this solution is stored and can be used in the future.

The major learning capability in ELM is its ability to learn about the student. This is used to give better advice on what parts of the curriculum the student is ready to address. This serves as a heuristic to speed up the program diagnosis and to remind the student of familiar examples that may be relevant to a task.

4-2.5 Evaluation

Studies have been carried out to compare how ELM-PA and ELM-ART performs. In one study, 22 student using ELM-ART was compared to 28 students using ELM-PA [87]. On a final test identical to both groups, both groups did equally well on the two simpler problems, while a more complex problem showed that 54% of the ELM-PA students solved this, compared to 87% for ELM-ART. The study also showed that the students without programming experience benefited significantly more from the ELM-ART system than the ELM-PA system. However, because the ELM-PA system is an on-site system developed for Macintosh computers, the computer resources was limited – each student had two hours a week reserved on the computer. This is in contrast to the ELM-ART system, which is web based and could be accessed from any computer. The authors acknowledge that this difference in accessibility may explain the difference in performance.

4-3 CATO

The CATO system is a tutoring system for teaching law students how to argue court cases. In the United States, the courts use a common law system where earlier cases are binding precedents for similar cases in the future. Lawyers arguing court cases depend on comparing the current case to earlier cases, by arguing how the current situation is similar and different to them. However,

many cases are not clear-cut by being obvious analogies of previous cases. This make it possible to construct arguments for both sides in a conflict, where each try to find positive analogies to cases supporting their side, and negative analogies to cases that do not. This is not a straightforward process – some facets of cases are more important than others, and often, specific sets of conditions must be present. In order to argue a case, a law student must know both the conditions required by law and be familiar with previous case history. Arguing the case draws upon both of these sources of knowledge.

This kind of argumentation is traditionally taught in a Socratic dialogue between the class and the teacher. While the Socratic dialogue has been used as a basis for early systems such as SCHOLAR [30] and WHY [81], the authors of CATO felt that the complexity of legal reasoning is too great to attempt such an approach [7].

Instead, CATO explicitly represents the structure of law arguments and the normative, middle-level knowledge facets, called *factors* identified in and argued about in the cases. It teaches argumentation through generating, displaying and explaining structured arguments, and assisting the students in generating their own.

In some ways, CATO is less ambitious than the PACT cognitive tutors or the ELM systems – it does not attempt to model the student, evaluate solutions or correct mistakes directly. This is at least partially because it more ambitious in that it attempt to tackle a less structured, less formalized natural-language domain. In many ways, it lacks the complete set of capabilities traditionally required by an intelligent tutoring system, and reflecting this, Aleven also refers to CATO as an *intelligent learning environment* – a term that seems to imply a larger degree of initiative and control given to the student.

4-3.1 Domain

Description of the Domain

CATO is designed specifically to teach argumentation skills in law, and the current domain is trade secret law, which aims to protect the owners of commercially valuable information from competitors that have somehow managed to acquire this information through improper means.

Argumentation is not an easily modeled skill. Unlike mathematics and programming, it is not expressed in a formalized language and there is no absolute method of judging the quality of an argument. Instead of attempting to work with English natural-language text, the creators of CATO created a formalization of the law argumentation called the *Factor Hierarchy*. On the bottom of this hierarchy are features used to represent cases, which again are organized into Intermediate Legal Concerns, which again are linked to Legal Issues at the top of the hierarchy. A part of the CATO factor hierarchy can be seen in Figure 4.11. Each high-level factor has conclusions associated with it to support each side. For instance, the "F102 Efforts-to-maintain-secrecy" factor can be concluded for the plaintiff if the plaintiff did take measures to protect its

information, or for the defendant if the plaintiff did not maintain secrecy.

The students work with the CATO system on two tasks; theory testing and argumentation guided by examples. In the theory testing task, students are faced with a general problem and are asked to form a hypothesis. This hypothesis is tested against the cases in the CATO database, and the student is asked to resolve inconsistencies, either by explaining them away or modifying the theory. Although this is not an argumentation task, such generalizations are useful when arguing that a case belong, or does not belong to a certain class, while also illustrating that the generalizations that exists in this domain tend to have exceptions.

The argumentation task is designed to support students in forming written arguments about a situation, and support it with references to cases in the CATO database. To support his process, CATO dynamically generates examples, an example of which can be seen in Figure 4.10.

Presentation

The CATO system consists of six tools that the student can use [7, Figure 18, p. 220]:

- *CATO database.* Contains a set of legal cases. The cases contain both a short textual description (a squib) and a set of factors, which apply to that case. The tool allows the student to search for cases using boolean expressions of these factors. The trade law database has 147 cases.
- *Factor Browser.* Contains the Factor Hierarchy, including a textual description with more detailed information about the meaning of each factor. The trade law database has 26 factors.
- *Case Analyzer.* Allows student to compile a list of factors that apply to a given case, and compares the result with the list of factors stored for that case in the CATO database.
- *Argument Maker.* Shows examples of argumentation that is relevant to the students' present work, and conducts mini-dialogs to help students practice identifying distinctions between cases.
- *Issue-based Argument Window.* Presents arguments, organized over issues, for any case in the CATO database.
- *Squib Reader.* Text display window for the case squibs (short textual descriptions).

The Argument Maker and the Case Database is shown in Figure 4.10, and shows how the structure of the argument is annotated.

In general, CATO does not restrict the freedom of the student in expressing the solution, as the solution is a written argument in natural language English. The knowledge CATO has about argument formation is used primarily to form examples as a form of assistance to the student, and in helping the student by

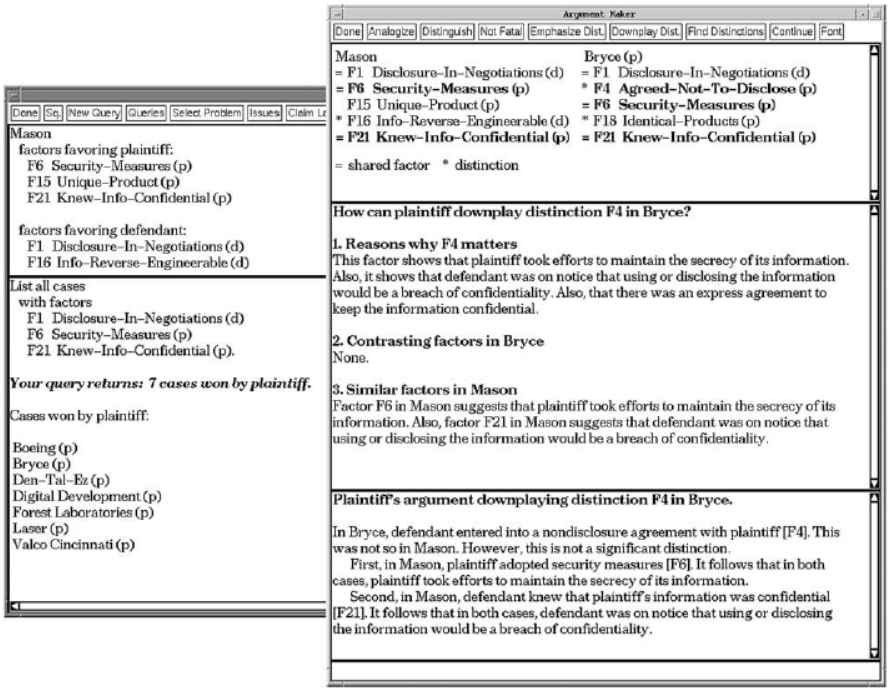


Figure 4.10: The CATO Argument Maker, from [7, figure 19, p. 221]

making the structure of the argument explicit. This means that CATO does not try to evaluate the solution of the student directly, but provide an environment to support the formation of the solution.

Scope

CATO is specifically designed to support tutoring in law, and is limited to the case-based common law approach. The current system contains a model for learning in trade law domains, but it is a stated goal to support other areas of law where cases are an important component of argumentation.

Although CATO is a specialized approach designed to address law argumentation, it can also be seen as a way of creating interactive learning environment from decision support systems based on knowledge-intensive case-based reasoners in general. In many ways, CATO is a decision support system, but a decision support system for students. The goal is for the student to learn the cases and to use the general domain knowledge to qualitatively measure the similarity between cases. Many weak-theory domains have this mix of using cases while still maintaining some rules that structure and affect how the similarity of cases are measured. In this sense, the general approach can be useful in other domains as well.

4-3.2 Cognitive Model

The CATO developers begun by looking at the practical requirements and considerations for tutoring in the law domain, with the goal of providing a low-cost learning environment in a complex domain. This contrast with the PACT cognitive tutors, which was developed from a theoretical point of view as a way of testing the ACT theories. This difference in origin means that the authors of CATO are less concerned with discussing a theory of human cognition, but they do discuss some issues tied to the approach.

A crucial element of CATO is recognizing and defining the structural elements of case law argumentation. This allows the system to approach argumentation as a goal-task hierarchy. However, this is not only important in the sense that it forms a strategy for the system, but it also helps the student to understand the underlying goal structure of the problem, which is a principle of the PACT cognitive tutors as well (principle 2, Section 4-1.2). This argument structure is, according to the authors, not typically explained in law classes and textbooks.

The tutoring capability of CATO is based on providing examples. In the hypothesis testing, examples that do not match the current hypothesis is presented as a form of counter-evidence of the correctness of the current theory, while in the argumentation maker, examples are used both to demonstrate problem solving strategies and as a way for the student to self-evaluate their own solutions.

CATO and Constructivism

The CATO system has many instructional design elements. For instance, the area of study is very clearly defined. The system is used with a workbook, which rigidly defines how the students should use the system. Further, the factor hierarchy is pre-defined and may not be changed by the student.

On the other hand, there are several aspects of the approach that are less stringently designed that for instance the ELM and PACT cognitive tutor systems. While the CATO system has represented each case as a set of factors it believes to apply to that case, the system's approach is that this is just one possible view of what factors should be identified in that case. When students are asked to identify the factors they think apply, the system accepts their judgment, and uses the student-identified factors in the reasoning process. The system also does not attempt to evaluate the final argument, as the student is asked to form the argument in a natural language form. Finally, CATO does not attempt to model the student's knowledge. Any hints are based on examples, and the retrieval of them is based only on the current state of the problem solving. The system is also built as a problem-solving environment first, and a kind of a decision support system second. Although the workbook lies down a very specific path the student should follow, the system is built as an open learning environment that first and foremost provide tools to make the task – learning – easy. In this sense, it is not certain that CATO is an intelligent tutoring system, and the authors prefer the term *intelligent learning environment*. There are certainly AI techniques used, but the system does not attempt to model the student, which is one of the requirements often seen for ITSs. However, the ability CATO has to create examples relevant to the current problem, allows it to be very specific in tailoring the support to the problem at hand. The problem-solving method used in CATO is also designed to mimic the strategy used by the professionals in the domain. The examples produced by CATO not only contain solutions to problems, but problem solving traces in a language that is accessible to the practitioner. At least some of the tasks CATO has been evaluated for, such as predicting the outcome of legal cases, can be performed by standard machine-learning techniques such as Bayesian methods, neural networks or ID3 (see Section 4-3.5), but these techniques lack this inherent ability to explain the reasoning in terms accessible and used by the students. CATO is not simply a decision-support system that focuses on finding correct solutions, but a system built from the start to support domain-specific explanation of its reasoning.

4-3.3 Knowledge Models

The CATO system has two main knowledge sources – the annotated court cases and the factor hierarchy (Figure 4.11). It does not use a student model.

In addition to the system itself, the evaluation of CATO as a pedagogical tool has employed a workbook designed to guide the students in using the system. This workbook contains tasks (exercises) for the student that uses CATO, and

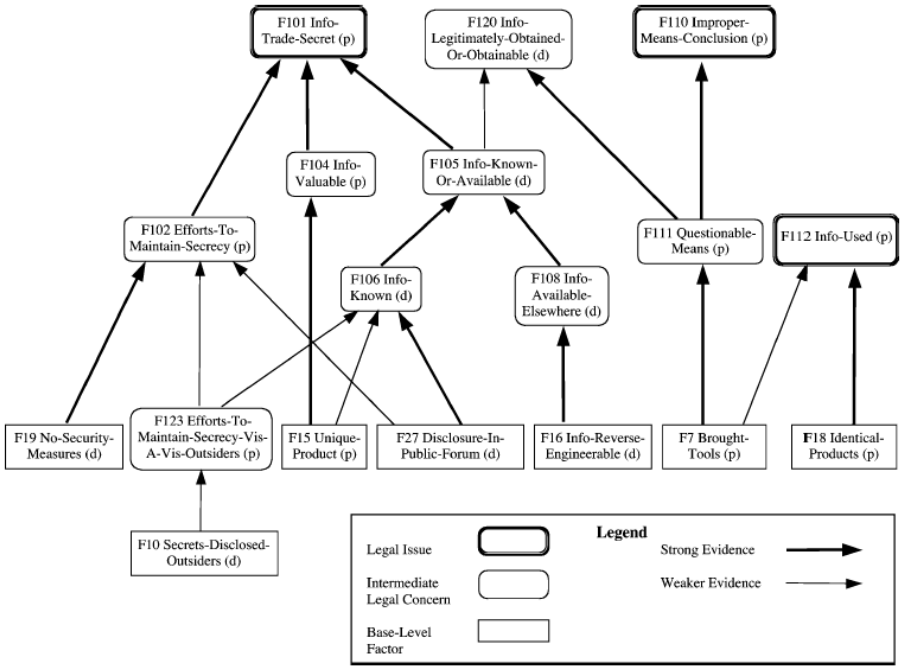


Figure 4.11: Excerpts of the CATO factor hierarchy, from [7, figure 3, p. 192]

explains background knowledge, such as CATO's approach of breaking down the argumentation task in a hierarchical manner. Although external to the computer program, the workbook can be said to be a part of the total system, or at least the evaluation of the system.

Expert Model

In CATO, both the court cases and factor hierarchy can be said to be a part of the expert model that the system wishes the student to learn. In order to solve problems in the domain, both of these knowledge sources are required. In addition to these, CATO contains implicitly the knowledge on how problems (arguments) should be approached in order to be solved. This task-goal hierarchy is chiefly presented to the student through the workbook associated with the system, and examples generated by it. The exercises or tasks given to the students are not stored in the computer system, but presented through the workbook.

The factor hierarchy (Figure 4.11) is a representation of *middle-level normative background knowledge* of the area of law covered by CATO, and is based on earlier work on the HYPO system [18, 19]. In the current study, this is the trade law domain.

The factor hierarchy is divided into three levels of abstraction, where the

lowest level is used to represent features of the legal cases. Each factor in the factor hierarchy is favored by one side in the litigation process (the plaintiff or the defendant). For instance, the *F19 No-Security Measures* is a pro-defendant conclusion (see Figure 4.11). These lower level factors are linked to the mid-level *intermediate legal concerns*, which are again part of *legal issues* on the top of the hierarchy. These two higher-level types of factors are called *abstract factors*, and associated with each is two conclusions – one for the plaintiff and one of the defendant. The links from lower-level factors can support either side, depending on if they are pro-plaintiff or pro-defendant. The strength of this support also varies. For instance, *F19 No Security Measures* is linked strongly to the *F102 Efforts to maintain secrecy*, which means that the former is strong evidence against the presence of the latter (as it a pro-plaintiff conclusion). This means that the support propagates to suggest a pro-defendant conclusion on the *F101 Info-Trade-Secret* issue, which attempts to establish if the information was really a trade secret. However, in many court cases factors are present that can support both side (such as e.g. *F15 Unique Product* and *F19 No-Security-Measures*), which allows for the construction of arguments that support both defendant and plaintiff.

The database of legal cases is useful in these situations where factors point in different directions. If, for instance, all cases that contain both *F15 Unique Product* and *F19 No-Security-Measures*) favored one side, this is a strong precedent to counter an argument based solely on the factor hierarchy. In more complex cases, the cases have different conclusions, and each side's legal arguments are designed to explain how the present case is more similar to the set of cases that favors them.

In CATO, the combination of cases and the factor hierarchy is essential in order to mimic the strategy for argumentation used by real practitioners, and it is able to dynamically generate arguments for any case that can be represented by its factor hierarchy, even cases it has not seen before. These arguments are not definitive for the decision of the case, but the relative strengths of the arguments can give an indication of what the ruling of the case will be (see Section 4-3.5.

The current system is used within trade law, but the system is designed to be used in other areas of law as well. In order to acquire the expert model, the legal issues of the area must be identified and the factor hierarchy defined. When this is done, a set of cases must be identified and annotated with the factors that apply to each case. A domain expert does this manually.

Student Model

The CATO system does not attempt to model the student. One can imagine that the an overlay model over the factor hierarchy and cases may be a possible, and that this knowledge could somehow be used in generating examples that used cases and factors familiar to the student, but this is not currently done.

Pedagogical Model

The CATO software itself is a collection of tools with little guidance on how to use them. It does not even include exercises. The guidance and exercises comes in the form of a workbook associated with the software. This workbook can be said to implicitly represent the pedagogical strategy of CATO. The strategy based on learning-by-doing (constructing arguments), providing examples, and allowing the students to compare their solutions to problem solving traces produced by CATO. Because the pedagogical model is represented externally to the system, CATO cannot change or adapt the strategy, however this loose tie to the pedagogical strategy also means that the system can support the student that decides to forego the workbook and explore on his own.

4-3.4 Tutoring Capabilities

Goals

The goal of CATO is to be an environment where students can learn argumentation skills, by giving easy access to information used in arguments (normative middle-level knowledge and cases), hypothesis testing, and demonstrating how arguments can be formed through dynamically generated examples. The computer system does not seek to enforce a particular learning strategy, or to tailor to the particular student, so it does not include models for these tasks (pedgagogical and student models, respectively).

Exercise Selection

CATO does not perform exercise selection, nor does it actually contain exercises. There does not seem to be a problem in extending the system in this direction – it has simply not been the focus of the system.

In-Exercise Support

Although CATO does not represent exercises explicitly, it is designed to support exercises given through the workbook. Two main types of tasks are given and supported by the system:

Theory Testing The first kind of task is formulated as a kind of hypothesis testing, where the student is given a general statement such as this example from [7, p. 222, fig. 20]

> *Suppose a defendant to whom confidential information was disclosed knew that the information was confidential, but there was no written nondisclosure agreement. Is the defendant under an obligation not to use or disclose the information?*

The task of the student is to translate the text to factors, and use the CATO case database to search for cases that contain the factors. If the search results

is not conclusive in the sense that all cases are in favor of one of the sides, then he must examine the conflicting cases to see if they are outside the scope of the theory or otherwise be explained. This does not look like an argumentation task, but it teaches the student to view the case in terms of abstract factors, and to identify cases that have important similarities.

Argumentation guided by examples. In this second type of task, the student is given a problem situation and asked to write arguments for both the plaintiff and defendant. To do this, the student must use both the factor hierarchy and references to the cases in the CATO database. CATO supports this effort mainly through the issue-based argumentation tool, which produces examples of arguments, including the structure they are built on. This is possible because CATO has explicitly represented the reasoning structure behind the argumentation. Much of the effort of CATO has been in identifying the argumentation structure, and creating algorithms for solving the different argumentation tasks. This allows CATO to visualize and explain the reasoning process well, in a process and language that are known to the students. However, it does not mean that CATO is able to "trace" the problem solving of the student in the same way as the PACT cognitive tutors or ELM. This is partially because the medium for expressing the solution is entirely in natural-language text, but it also seems to be a choice to present CATO as a more hands-off problem-solving tool.

Conceptualization support

A great strength of CATO is in its ability to assist students in acquiring the vocabulary and concepts of the particular domain of law. By explicitly recognizing not only case features, but also more abstract terms and how they interact in a model like the factor hierarchy, students can access an overview they presumably did not have before (as the factor hierarchy was formed through the CATO effort). Since these terms are also directly associated with cases, and one of the tasks CATO supports is recognizing factors in cases, students should also learn how to use these factors on real cases. This ability is limited to teaching students the pre-existing model – it does not allow students to experiment with their own conceptualizations.

Explanation Ability

The major explanation capability of CATO is to provide transparency into a reasoning process (argumentation), with the intention of having the student adopt the technique by observation of enough examples.

On another level, the argumentation itself is a series of explanations, for examples of why a case decided for the opposite side is not relevant. As such, CATO is for instance able to explain why two cases that do not match completely have important or irrelevant differences by using the factor hierarchy (and even cite other cases). In a tutoring context however, this is the subject matter, and

explanation on a meta-level may be expected. For instance, if a student wish to know if a particular argument is good or not, and why, this is not something CATO can answer explicitly. The knowledge may be implicit in the factor hierarchy, case base and argumentation technique, but this cannot be reasoned over or explained on a meta-level.

Learning Capabilities

CATO does not learn through normal operation. Neither is it clear if the system could gain from such learning, since it do not use student models, and the factor hierarchy and case base is fixed in advance.

4-3.5 Evaluation

CATO has been evaluated with regards to its ability to predict the outcomes of court cases, and in a comparative study that sought to compare learning with CATO to traditional learning methods. These evaluations are both described in greater detail in [7].

Predictive accuracy

The first evaluation study was performed to evaluate the predictive accuracy of CATO's legal arguments. The reasoning behind this kind of evaluation is that arguments with good predictive power for the outcomes of the cases would be a good indication of the quality of the arguments. CATO was compared to *k-nearest neighbours (kNN)*, *Naïve Bayes* and *ID3*. It was also a goal to examine if the general background knowledge encoded in the factor hierarchy had an effect on the predictive power.

The results of the evaluation showed that Naïve Bayse predicted 90% of the outcomes correctly; k-NN was 84% correct and ID3 81% correct. CATO had the additional option of abstaining if it was uncertain on the conclusion. The best CATO method using the background knowledge made predictions on 89% of the cases, and was able to predict 88% of the solutions, which was significantly better than the baseline method that did not use background knowledge. This method only classified 62% of the cases, although with a 92% accuracy when it did predict.

The result showed that Naïve Bayes was significantly better than the best CATO method at predicting the outcome of the cases, but also that the best method using background knowledge performed significantly better than any CBR method that did not use background knowledge. The fact that Naïve Bayes performed better than CATO at the classification task is not necessarily a problem for the system, as CATO's primary role is to produce normative arguments for deciding the case – a task that cannot be performed by statistical methods alone.

	Basic argument skills				Memo writing		
	Pre-test		Post-test		Prev	Post-test	
CATO group	60	C−	70	C+	63	70	B−
Control group	55	D	68	C	63	79	B+
CATO's arguments	81	B+	87	A−		62	C

Figure 4.12: Results of the CATO basic argumentation and memo-writing tests, from [7, Table 2, p.227].

Instructional effectiveness

The second evaluation was performed to measure the instructional effectiveness of CATO compared to traditional teaching methods, and was carried out in the context of a second-semester legal writing course at the University of Pittburgh School of Law. The participant volunteered for the experiment, and was divided into a 14 person control group and a 16 person experiment group.

The experiment group recieved basic instruction in the use of CATO and worked in pair using a specially made workbook in nine 50-minutes sessions. The control group was instructed on the same content, but in a traditional setting. This consisted of four classroom sessions where an instructor would use a Socratic method to present the topics and engage the students, and two pretend court sessions led by the instructor (playing "judge" as well as teacher). Students had to prepare for at least 75 minutes before these sessions.

Before and after these sessions, the students were given tests on basic argumentation skills, and in addition they were tested on writing a legal memo. This second test was meant to measure how well the argumentation skills transferred to a related, more complex, task. In addition to the two groups of students, a set of answers from the CATO system itself was also included in the tests, and was graded in a blind test by the legal writing instructor.

The results of the evaluation showed that there was no significant difference between the gain of the control and experiment groups on the basic argumentation test (see Figure 4.12), and in addition that CATO performed very well (it was the third best of all answers even in the post-test). However, on the memo-writing task, the control group scored significantly better than the experiment group, and CATO itself did not receive a high grade.

Matching the result of engaging classroom teaching by experienced instructors is well done by CATO. However, the authors draw the interesting conclusion that the more "holistic" technique used in classroom teaching improved the student's ability to perform a transfer task (legal memo writing) in a way that CATO failed to do. This may imply that the more targeted, task-oriented technique taught by CATO is too specific to induce enough deep learning to be as useful in transfer tasks. The authors specifically mention students running out of time before reaching the more complex parts of the exercises, the reifica-

tion of the argumentation structure not fitting the legal memo-writing task as (partial) other sources.

4-4 Other systems

There exist many other systems that use a case-based tutoring approach, but for various reasons do not fulfill all the criteria set out in the introduction to this chapter. Typically, these systems may have interesting ideas or illustrate some aspect of case-based tutoring, but are works in progress or there is otherwise only one or a few publications about the project. In this section, some such systems are more briefly examined.

4-4.1 Ambre-AWP

The goal of the AMBRE project [40, 55] is to create an intelligent tutoring system that teaches problem solving methods. Specifically, the current system is designed to teach 8-year olds to reformulate a mathematical questions posed in natural language. This project is interesting because its approach to case-based tutoring is to use the steps associated with case-based reasoning as a learning strategy for people. In the AMBRE project, the steps identified are elaborate, retrieve, adapt, revise and store, and these steps, except revise, are explicitly presented as problem solving steps that the student should follow.

Domain

The Ambre-AWP is designed to help 8-year olds understand and reformulate simple arithmetic problems represented as natural language text. The text formulation can for instance look like this [55]:

> *Julia had 18 cookies in her bag. She ate some of them during the break. Now, she has 9 left. How many cookies did Julia eat during the break?*

These questions should be reformulated to an equation, in the above example to *18 - 9 = ?*. Although the actual system is designed towards this domain, the project's aim is to test how the case-based reasoning cycle fares as a tutoring method. The authors suggest that this method is appropriate in many domains where there are few good rules what problem solving method should be used.

Cognitive Theory

The case-based reasoning framework is itself the main cognitive theory of Ambre-AWP. The authors argue that people often solve problems by analogy, and that as such, the approach used by case-based reasoners can reasonably be expected to be a natural model for problem solving.

The adaptation of the CBR cycle for tutoring in Ambre-AWP (summed up in Figure 4.13) is four steps:

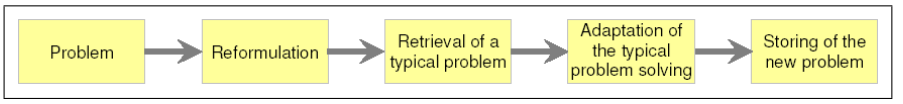


Figure 4.13: The CBR cycle adapted to the AMBRE project, from [55, Figure 2, p. 3]

Reformulation of the problem. The student is asked to identify important characteristics of the problem. In the concrete domain of the current system, the student was asked to identify diagrams representing the class of problems the text was about (subtraction, addition, comparison, etc.).

Retrieval of a typical problem. In the second step, the student is asked to browse a list of similar problems in order to find similar problems. The stored problems are presented with both the original and the reformulated representation.

Adaptation of the typical problem solution to the current problem. After a similar problem is found, the student is asked to adapt it to the new situation by using the strategy of the retrieved problem on the new problem.

Storing the new problem. After solving the problem, it is stored and can be retrieved when solving later problems.

Knowledge Models

The Ambre-AWP system is designed to teach problem-solving methods, and to do this the system requires an explicit taxonomy of the problem classes and the methods used to solve these. This hierarchy is not necessarily presented to the user, but is used internally to help the system to solve problems. The reason for this is that the problem hierarchy is not necessarily explicitly identified when teaching or discussing the domain. This is similar to the situation in CATO system, where the factor hierarchy and the reasoning process were not explicitly taught, but rather identified by the system's creators. However, where the authors of CATO (and the ACT theories) found this explication to be of value in itself, the authors of Ambre-AWP is less certain about the value of presenting this to the user, and argues it can cause confusion. Instead, they argue that the student should naturally build his own internal representation, assisted through prototypical examples presented by the system.

In addition to the problem class taxonomy, the Ambre-AWP has a case base of problems that serves both as exercises and examples for students. Initially, a few examples are presented to the students to "initialize" their case-base before they are asked to solve problems.

The student model of Ambre-AWP can be seen as an overlay over the cases in the problem case base, representing the set of cases known by a particular

student. However, it is not clear if this information is currently used to tailor the tutoring session in any way.

Tutoring Capabilities

The last published implementation of Ambre-AWP was designed to test the case-based methodology for learning, and does not include assistance as such, except to structure the problem-solving session according to the case-based principles already discussed. However, the approach suggests that the capability of their case-based reasoner to solve these problems can offer the student in-exercise assistance. For instance, in the task of reformulation, mistakes can be pointed out and the reasoner can construct explanations using the problem class taxonomy to illustrate why or why not certain features should or should not be present.

Evaluation

Currently, the complete system has not been evaluated, but a study was performed to measure the impact of the case-based tutoring method used in Ambre-AWP [55]. This was performed on 64 students, who were divided into two groups. Both groups used a similar computer-assisted learning environment, but only one used the case-based approach. A post-test of the two groups did not reveal significant differences in problem-solving capability between the groups, however a study of the problem-solving traces of the students using the case-based approach seemed to suggest that many had trouble with the retrieve step. Although many succeeded in the reformulation of the problem, matching this to similar problems proved to be difficult. In response to this, the author's plan to include more support for this step in the future.

4-4.2 BLITS

The BLITS system is designed to help its users write business letters in a workplace setting, for instance by assisting secretaries writing such letters in a foreign language [23, 59]. The method of the BLITS approach is interesting because it is explicitly designed to work as an unobtrusive tool to solve the task at hand (letter composition) as opposed to an active agent in trying to "push" knowledge at the student. This is the result of an explicit design philosophy based on constructivism and self-directed learning that suggests learning should be situated, i.e. in a setting of actual problem solving. The result of this is that the system cannot and will not attempt to set the agenda by suggesting exercises, but instead provides assistance when the user[3] is in the task of composing a real business letter. In many ways, the BLITS system can be perceived as a word processor aid similar to grammar and spell checking.

[3] it makes less sense to call the BLITS user a student than in other systems we have reviewed

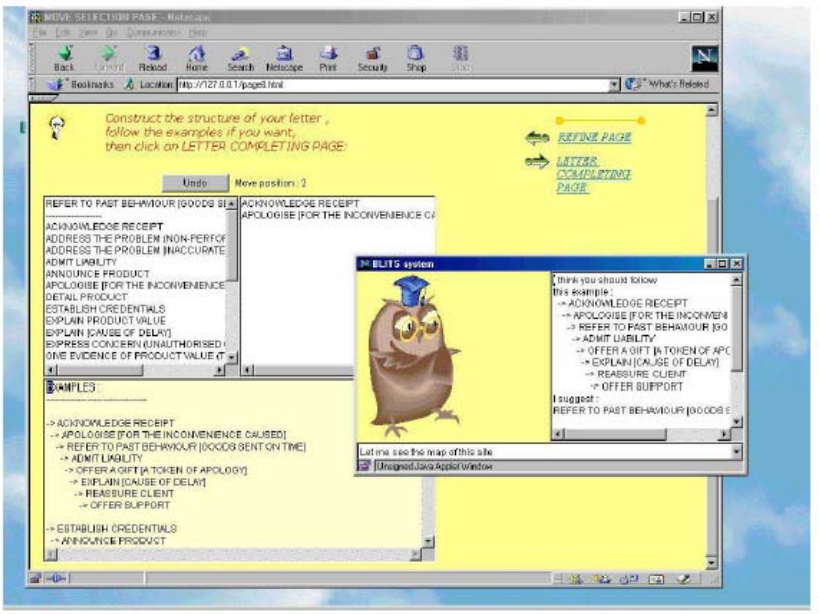


Figure 4.14: Screenshot of the "Move Selection" in BLITS, from [23, figure 6, p.4]

Domain

Although BLITS is designed to help users compose business letters, the authors point out that what constitutes an effective business letter varies. There are for instance different writing styles in the US and in Great Britain, and business English in Asia is different still. There are also differences between industries and companies. In other words, teaching a single correct way of writing such letters is not possible, and that the users must discover what effective business communication means in their particular area.

The application itself is web-based and presented through a browser, where the user goes through steps to choose what kind of letter he is writing, as well as characteristics of the letter (such as the level of familiarity between sender and recipient). Then, the user is presented with a *Move Selection Page* (Figure 4.14), which assists the user in deciding what content elements (such as *Acknowledge Receipt* or *Apologize*) should be included in the letter. The final page is the composition of the letter through realizing these *moves* by writing out each content element. Here, the system can assist by showing examples of such elements from previous letters.

Cognitive Model

The authors of BLITS rely on a constructivist philosophy in their design of BLITS, and indeed the system contains many elements of constructivist. For instance, the system is completely situated in that it does not operate in an artificial educational environment, but as a tool in actual problem-solving. This separates it from the other systems reviewed here. The author's also argue that it allows the user to decide what an efficient writing style is in the given context. This suggests that BLITS it somewhat relativistic and rely less on the universal expert models traditionally found in intelligent tutoring systems. The authors do suggest that they would like the system to be able to evaluate a written letter and compare it to other letters in its case-base to judge the effectiveness. The argument is that because the user have authored the case based and provided feedback on how well these letters were received, it reflects the user's idea of effectiveness, not the expert's. However, the cases are not only represented as text, but in a tree-like structured representation of the contents (*Moves*). No detail on who formulated this structure is present in the description, but this is presumably done by a domain expert of some sort – creating explicit abstractions of letters are likely hard for a novice. BLITS does in other words, have some expert knowledge – the knowledge of how to represent the structure of the document. This kind of knowledge is conceptually very similar to the factor hierarchy in CATO and the ACT theories idea of explicit problem-solving schema – they all seek to break down the problem-solving process into abstract chunks that can be organized in a tree-like structure to form a whole.

Knowledge Models

The two main knowledge models in BLITS are the case-base and the document structure hierarchy (the *Moves*). BLITS does not provide a student model in the classical sense, but since the case-base is user- or context-specific, it serves as a way of tailoring what is considered good and bad writing. The situated nature of BLITS also makes many of the ITS tasks that rely on student modeling impractical or inapplicable. For instance, the documents written in BLITS are suppose to be real business letters, so exercise selection is not a task the system is going to perform.

Tutoring Capabilities

The tutoring capabilities of BLITS are intentionally low-key, in the sense that it does not seek to appear as a tutor at all, but rather a help system that fosters learning. As already mentioned, the situated nature of the system makes exercise selection a non-issue, but it does attempt to give in-exercise support – or rather problem-solving support. In principle, the way it does this is similar to the other case-based tutoring systems reviewed here – it forces the student to analyze the problem through a pre-defined structure (*Moves*), and provides pointers to cases with similar structure. The difference lies in the fact that the case-base is not authored by an expert, but by the user himself.

In providing advice, the system separates the working environment of the system where the user has full control, and the giving of advice, which is done in a separate windows featuring an advice agent similar to the "Office Assistant" found in Microsoft Office products. This agent is visible in the screenshot in Figure 4.14.

Evaluation

There has not yet been published an evaluation study of the BLITS system.

4-5 Chapter Summary

The five systems we have examined in this chapter, can be seen as belonging to two different groups. On the one hand, the PACT cogntitive tutors and ELM goes to great lengths to accurately model a student's procedural skill, including possible misconceptions and "buggy rules" a student may posess. On the other hand, CATO, BLITS and Ambre-AWP focus on providing an environment conductive to learning. In many ways, these groups mirror the divide between instructional design and constructivist approach, where the PACT Cognitive Tutors and ELM represents instructional design, and the others systems a constructivist approach.

In the next chapter, we present our CREEK-ILE theory, but in presenting this theory, we will also draw on and compare it to the theories and systems discussed in this chapter. In particular, Section 5-2, discusses these system's cognitive models and pedagogical theory.

Chapter 5

CREEK-ILE

In this chapter, we present the CREEK Intelligent Learning Environment (CREEK-ILE), using the framework presented in Chapter 3. The implementation and evaluation are separate chapters following this chapter.

5-1 Domain

The CREEK-ILE theory is designed to work across multiple domains. However, the CREEK case-based reasoning engine is primarily designed to work with diagnostic domains, which can be described by a combination of cases (problem solving episodes) and a general domain model. Here, the general domain model is typically in the form of a causal or associative model that links symptoms with underlying problems and diagnoses in a kind of qualitative models (e.g as found in QUEST – see Section 2-1.1). Unlike these system CREEK relies on cases in addition to the model-based reasoning. This allows us to relax the demands on the general domain model, as it is used to support and explain the reasoning process but does not need to be as complete in solving problems.

The tests of the CREEK-ILE approach so far have been carried out in the domain of computer programming (the Java programming language). Computer programming is not a typical diagnostic task. It certainly contains sub-tasks that are diagnostic in nature, such as debugging, but overall it is constructive. However, computer programming has been a favorite topic for learning-by-doing tutoring systems, and as such developing for this domain facilitaes comparison. It is also a practical choice, as it is a class attended by many students and taught in our own department, which makes access to the course as well as students and experts easier. Last, it is a topic we are ourselves familiar with, which reduces the risk of us underestimating the difficulty of teaching the topic. This domain is not chosen because the problem solutions are expressed in a formal language – the CREEK-ILE approach is specifically designed to support domains that lack computational methods of interpreting the data, although this does have trade-off in that CREEK-ILE is unable to provide as much tutoring support.

In this chapter, we will use both a simple computer programming model and a diagnostic model for car start failures to illustrate the CREEK-ILE approach. The car failure diagnostic domain is an artificially constructed domain used to test and demonstrate various features of the CREEK case-based reasoning engine. The reason for including this domain in addition to the computer programming domain is that it allows us to illustrate how various degrees of support for explanation and reasoning by the problem-solving engine affect the approach.

The next chapter (Chapter 6), has more information on how the actual system was designed and implemented. In this chapter, we will focus on the theoretical foundations.

5-1.1 Computer Programming Domain

The computer programming domain is based on our experiments with CREEK-ILE in introductory Java programming classes. The example used in this chapter is for one of the first exercise sets of this class dealing with loops (specifically `while` and `for` statements). The general domain models are in the form of concept map-like structures containing a taxonomi of statements and abstract classes of statements (e.g. `for`-loop is a kind of `loop` that is a kind of `statement`). Specific statements (such as the `for`-loop) also have syntactically correct and incorrect examples that can be associated with them.

Obviously, this general domain model cannot be used to solve programming tasks. It contains some information on the structure of Java programs, but it is too shallow to be used to parse student programs. It is on the conceptual level, and is used only as a basis for student concept maps, and as a guide to how concepts should be organized. At the surface, this can seem like an odd type of model to include in an exercise-oriented system, but conceptual knowledge seems to interact with procedural and episodic knowledge in many ways – it may for instance be useful in interpreting examples and problems, such as how the factor hierarchy is used in CATO.

The exercises associated with this model are simple tasks that ask the student to, for instance, write a program that lists all the numbers from 1 to an integer given by the user, or to take a pre-written program that uses a `for-loop` and transform it so that it does the same with a `while-loop`. The exercises used for illustration in this chapter are shorter and easier than the actual exercises used in the Java programming courses we have done our experiments on, as they are designed primarily to illustrate the approach. The actual exercises used are presented in chapters 6 and 7.

5-1.2 Car Failure Domain

The car failure domain is an artificial diagnostics domain. It is in principle similar to many of the machine learning classification problems in that it contains a set of instances. These cases each have a set of known features (we call them *findings*), and the task is to identify the problem class. In addition to the

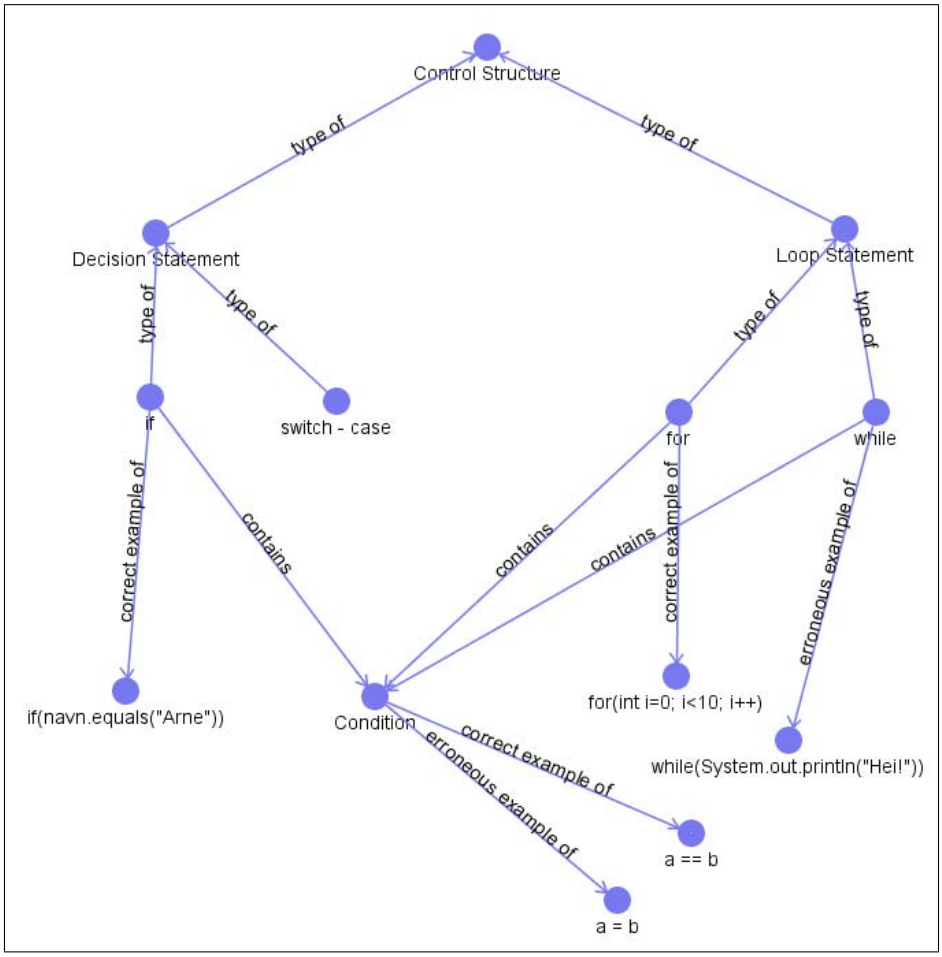


Figure 5.1: Example expert concept map for the computer programming domain.

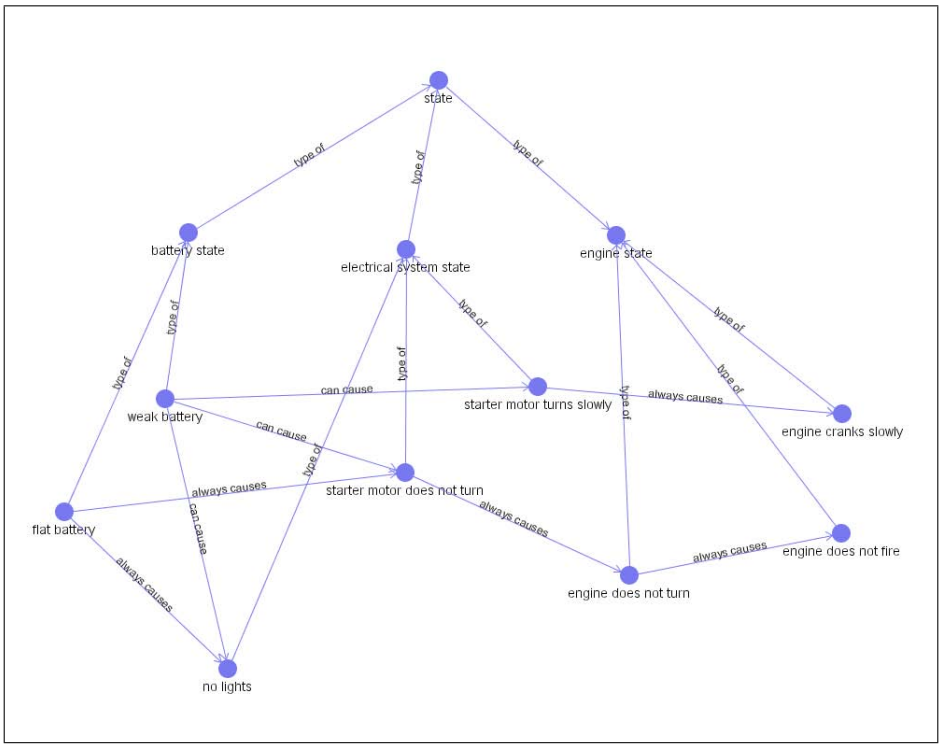


Figure 5.2: Example expert concept map for the car failure domain.

instances and problem classes, the CREEK model also has a high-level, loose causal model of how some findings can cause or be caused by various problem states. These again can be tied to problem classes. Taken alone, the set of cases or the causal model can to some degree solve problems in the domain. The cases can use an instance-based method to identify problem classes by matching a new problem to similar cases in the case base, and the causal model can chain causal relationships from symptoms to underlying causes (problem classes) and by this solve problems. The CREEK knowledge-intensive case-based reasoner uses both these knowledge sources to improve what either of them can do on its own.

In the car failure domain, the general domain model is similar to the computer programming domain in that it is conceptual in nature. It deals with car parts and problem states and how they are related. However, in addition to the taxonomical relations, the general domain model in the car failure domain contains causal relations that make it possible to link surface findings with problem classes. As in the computer programming domain, the expert conceptual models are used as a basis for student concept maps, but they may also be used to interpret problem cases, if well designed. The "if well designed" caveat is important here. Typically, concept maps are not well designed knowledge

models. The method is designed to be easy to access and use even for young school-children, and lacks the computational semantics required by knowledge representations. The car failure domain highlights this, as two types of models (student concept map and expert causal model) on essentially the same level have different requirements.

5-2 Cognitive Model

The previous chapters briefly introduced the ACT and Dynamic Memory theories of procedural skill acquisition, as well as critiques of intelligent tutoring systems from constructivist philosophers. Before discussing these in relation to our own approach, it may be useful to examine the commonalities these approaches have.

5-2.1 Kinds of Memory

A useful abstraction is the separation of human memory into three different types of knowledge.

Conceptual Knowledge is general knowledge about things and ideas. Often, the knowledge is hierarchically arranged such as in an animal taxonomy, but it may also have other kinds of relations. An example is the dictionary, where each word is defined in terms of other words. Definitions and symbols are conceptual knowledge; the factor hierarchy of CATO and the document structure information in BLITS are other examples. This kind of knowledge is consciously available to people, and we can talk about it, explain it and learn it through studying.

Procedural Knowledge is knowledge on how to do things in practice. For instance, the various abilities involved in driving a car, how to order food in a restaurant and how to program the VCR. This kind of knowledge contains Schank's micro-scripts and the rules in the ACT theories. Much of the procedural knowledge is tacit – that is, not available to conscious thought the way conceptual knowledge is. For instance, it is hard to explain to oneself or others the processes involved in catching a ball. It is certainly possible to discuss how to be a good driver, but there are elements of it that can much easier be shown than told. Even in purely intellectual skills, research has shown that experts often have trouble explaining how they solved a problem, and offer explanations that are formed after the fact that does not necessarily reflect the reasoning process [51]. This does not mean that the expert does not know what she is doing, but rather that there are parts of the expert's experience and knowledge that cannot easily be communicated through language.

Episodic Knowledge is the memory of concrete episodes. While procedural and conceptual knowledge are generalizations, it is obvious that people also

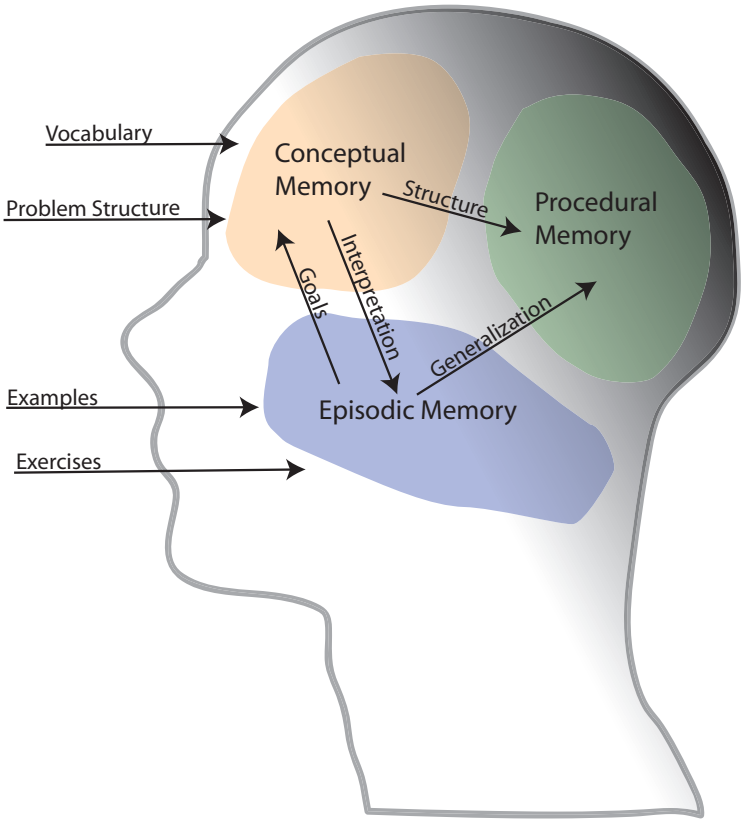


Figure 5.3: Abstracted cognitive model

remember specific information. For instance, someone might remember the day they graduated from high school, or the time a particular train was missed or where they had dinner last Monday. In a tutoring perspective, specific examples and previously solved or unsolved problems are episodic knowledge, and in case-based reasoning, the case represents an episode.

The difference between these types of memory is not absolute – clearly it is possible to have conceptual knowledge of procedural approaches, indeed this is the foundation for much of engineering. Schank [69] also suggests that a lot of episodic knowledge is remembered as generalizations when nothing exceptional happens ("*Last Tuesday at 7.45 AM? I suppose I was on the train to work as always,*") and we only remember the unexpected explicitly ("*Last Tuesday I missed my morning train because the ticket dispenser was out of order.*") However, the distinction is useful when discussing learning. The goal of learning by

doing systems in the wide sense (from exercise support to constructivist learning) is to assist the student in acquiring procedural knowledge in the form of applied skill, and there is generally an agreement that such knowledge is not fully available to conscious thought, and may not be taught through instruction alone. However, conceptual and episodic knowledge may be communicated and experienced, and through these forms of knowledge, procedural knowledge is formed (Figure 5.3).

5-2.2 Conceptual Knowledge and Interpretation

In the systems that has been examined, conceptual knowledge have been seen in the form of *vocabulary* (*"What is a trade secret?"*), *structure* (*"A business letter contain a greeting, the body proper and a signature"*) and problem-solving strategy (*"To solve a recursion problem, test on the first element and call the same function on the rest of the list if it is not empty."*) All the systems examined in the previous chapter have some kind of conceptual knowledge, and it is typically stored in a class-subclass hierarchy, as a task-goal-hierarchy or both.

Representing the procedural level is essential for the systems based on the ACT theories. The goal of these systems are typically to help construct specific procedural knowledge in the learner by following the reasoning and intervening when evidence suggests that the student is missing a rule or has a buggy rule. However, many case-based tutoring systems do not try to represent procedural knowledge. With the BLITS system, the authors do not wish to be normative in what procedural knowledge is considered correct, and thereby have no need to represent it. AMBRE-AWP and CATO are to various degrees able to reason about their domains in the sense that they can construct solutions, but the authors do not claim to do so in an entirely cognitively plausible way. The arguments produced by CATO are designed to be similar in structure and form to those made by experts, but they do not claim that their program create these arguments in the same way as people, and they do not use the program's problem solving strategy as a mold for human reasoning. Rather, these systems leave it to the natural processes of students to form procedural structures as they are given examples, exercises and conceptual information.

The episodic knowledge is in the form of exercises and examples, represented internally in the systems as cases. However, for this representation to be useful in the sense that the program can reason symbolically over them, they must be represented at a higher level of interpretation than raw data. For instance, the CATO system is unable to deal with the raw text of law cases, and BLITS cannot interpret business letters. The ELM system and the PACT cognitive tutors are able to deal with the low level representation in the form of program code because it follows a strict syntax that can be computationally parsed. At first glance, it is remarkable that all these different systems contain knowledge on the conceptual level, even though they stress their case-oriented approach, and in the case of BLITS, claim to follow constructivist approaches to learning where the system should avoid normative expert models. However, it is not so strange if we recognize that in order to allow these higher-level representations of

interpreted cases, the systems must define vocabularies to represent them. This illustrates a use of conceptual knowledge also found in humans – we use it to form higher-level representations, to interpret data and to recognize important and spurious distinctions between episodes.

In many domains, we are currently unable to produce systems that genuinely interpret data (e.g. natural language text) into structures (e.g. cases), and when designing a case-based tutoring system, it is tempting to pre-define the conceptual knowledge required to encode the cases. Using this vocabulary, the cases can be interpreted and encoded in advance using the vocabulary and structure already decided. However, this means that the student is not allowed to form his own conceptual knowledge, and often leads to the temptation of teaching the conceptual knowledge first, because that is required to understand the case representation and thereby the examples and problems of the system. This is acceptable in the ACT theory, but it is strongly criticized by those inspired by even moderate constructivism, and by learning by doing proponents such as Schank. As we have seen, Schank argues that when learning conceptual knowledge first, the value of the knowledge is not clear to the student, and it becomes detached from goals and micro-script level skills.

The CREEK-ILE approach to case-based tutoring is closer to Schank's idea of learning by doing and case-based tutoring than the ACT theories. We agree with Self [71] and others that the complete cognitive model of procedural knowledge is intractable in many domains, and while we find it very interesting to study and form such cognitive models, we think that good learning environments can be created even where there are no such complete cognitive models. Instead, the CREEK-ILE system focuses on episodic and conceptual knowledge models, and as the BLITS, AMBRE-AWP and CATO systems depend on the natural ability of people to form procedural skills through exposure to primarily episodic, but also conceptual knowledge. This has some disadvantages, however. If the system is not able to simulate the cognitive abilities of a learner, it limits the ability to explain and diagnose. The PACT cognitive tutor is able to diagnose reasoning errors even if the solution is correct, and to point out exactly what the errors were. This level of diagnostics may only be possible in a complete cognitive simulation of the human problem solving process – assuming that there is even a canonical model of this for the domain at hand. However, a learning environment can offer support on various tasks without such a capability. For instance, it may offer examples similar to the problem at hand (as BLITS and CATO does in various forms). It may assist in identifying discriminating features (AMBRE-AWP) and it may suggest interesting problems by providing exercise selection support.

CREEK-ILE supports these tasks by relying on the conceptual and episodic knowledge. Acquiring this kind of knowledge from an expert is much easier than procedural knowledge, because it is consciously available, and it can be communicated much more easily both by students and teachers. In this sense, CREEK-ILE is a system in the same class as CATO, BLITS and Ambre-AWP. These systems also primarily rely on the conceptual and episodic knowledge to help students to acquire procedural skill. However, CREEK-ILE is different

from these systems in that we wish to allow the student some ability to form his own conceptual knowledge.

5-2.3 Modelling Conceptual Knowledge in Concept Maps

A practical property of conceptual knowledge is that it is generally available to conscious thought, and can often be communicated fairly easily, as opposed to procedural knowledge. This means that if we wish to model the student, it is possible to simply ask the student to tell the system what he thinks. This follows the advice of Self reviewed in Chapter 2, but also a trend in intelligent tutoring systems towards inspectable student models in general. The conceptual knowledge representations in the tutoring systems we have reviewed has either been very complex (in the case of the rule-based cognitive tutors and ELM) or domain-specific (in the case of CATO). These representation languages are not made to be easily accessible to students, but for experts to accurately express their knowledge in a way that can be used by the computer. In Chapter 1, concept maps were introduced as an accessible way for students to represent conceptual knowledge. There, we reviewed how certain forms of concept maps can even be used to evaluate the student, with results that correlate highly with more classical evaluation tools such as multiple-choice tests. This suggests that using concept maps is a natural solution to the problem of allowing students to form and explicitly represent their conceptual knowledge in a learning-by-doing system. However, this does not solve the practical problem that there must be a vocabulary to represent the cases in. To solve this problem, the CREEK-ILE system does not use completely free-form concept maps, but rather a variant where the concepts (nodes) and relation-names (link labels) are pre-defined, but the student is given complete freedom in how they should be related and linked, as well as how many of them to use. One can argue that this is not really the freedom to define the vocabulary, as all the words are already defined. On the other hand, if meaning of symbols is defined through how they are related, the student has complete freedom of expression. We think that our approach is a reasonable and practical compromise that allows the student some freedom of expression while still ensuring that he uses common terms and stay within the boundaries of the system's competence. It also allows us to form higher-level representation of cases. CREEK-ILE is tested in the domain of computer programming, so concepts such as `for`, `while` and `loop` may exist in the word list. This makes it possible to pre-tag program cases as containing `for` and `loop`.

Allowing the student to form his own conceptual knowledge is a goal in itself, but when the student does this explicitly through a representation in the system, it can is also used to tailor the learning experience. Concept maps have been used in tools that seek to support collaboration by helping people represent and assimilate different models to form a common vocabulary, and this approach could also be taken with a group of students. Closer to the traditional intelligent tutoring approach, concept maps can also serve as representation of the student knowledge and be treated as part of the student model. By com-

paring it to an expert's representation (the expert model), it may be possible to evaluate for which areas of the curriculum the student has a correct or incorrect conceptualization (according to the expert's representation). This use may on the surface appear to be similar to using concept maps as evaluation tools in the same vein as traditional multiple choice tests. If one, for instance, define that when the student has produced the same conceptualization as the teacher, he was ready to advance to the next subject, then it would have the same problems Schank criticizes traditional institutional learning for having. The goal for the student would no longer be the procedural skills, but rather to form the right conceptualization as soon as possible in order to be done with the task, thus making reproducing the conceptual knowledge the primary focus. However, the conceptual knowledge can be used in other ways that support procedural skill acquisition. First, encouraging the student to explicitly represent conceptual knowledge is likely to force some reflection on abstraction, which is good in itself. Second, Schank suggests that a student that has formed his own theories may be more receptive to improvements of the theory in the form of expert knowledge. This may take the form of suggestions if the student is unable to reconcile seemingly conflicting partial models, or simply displaying how the expert represented the part of the model the student is currently involved in.

5-2.4 Uses of Student Concept Maps in Exercise-Oriented ILEs

Our first idea for using the conceptual knowledge in a concept map was to use it in support of exercise selection. While episodic knowledge of the student (what examples he has reviewed, what former problems he has solved) can be helpful in identifying what parts of the curriculum he has covered (e.g. the student has solved problems involving the `for-loop` but not problems with the `while-loop`), it may be hard to identify structural misconceptions (e.g. the student may believe the `while-loop` to be a test statement similar to `if`). Such information allows the learning environment to select or recommend exercises that challenges the student's preconceptions and forces him to reconsider his current model. This is an ability that is not easily formed by analyzing the episodic knowledge alone, as the conceptual knowledge is an indicator on how the student has generalized his knowledge. Simply knowing that the student has solved some exercises using `for-loop`, does not tell the system how well the student knows loops in general, for instance. The PACT cognitive tutors uses procedural reasoning traces to identify such problems, but in CREEK-ILE the choice not to model procedural reasoning limits this capability, but some of the same information can be found through analysis of the conceptual structures.

In the initial evaluations we did on this, the students were asked to make a concept map for an area covering a set of exercises (e.g. `for-` and `while-loops`), before the students were asked to solve these exercises. The, perhaps somewhat naïve, hypothesis was that students showing a greater knowledge of the conceptual knowledge would also fare better on the exercises. This hypothesis did not

prove to be entirely correct. Although many students had studied the subject before starting the exercise, there were also a number of students that apparently had not, but still did well on the exercises. It may be that these students have a learning strategy closer to what Schank describes – they start with the exercise problems and only then go looking for the knowledge they need. This would mean that at the time they were asked to form the concept map, they had not yet studied the topic enough to do the task well. However, not all students seemed to have followed that approach – some had clearly studied in advance. Our results also do not contradict the insight from ACT that conceptual knowledge is in fact required to solve problems, because this knowledge was actively sought once an exercise was given.

The result of these experiments was a revision to the approach of when and how students should be asked to create concept maps. It is possible to give this task after the exercise set is finished, but this would mean that the knowledge from the concept maps could not be used to support exercise selection. Instead, we suggest attaching a small concept map to each exercise, either right after presenting the problem (assisting the student in interpreting the problem description) or after the task is finished. This also allows us to support the student in generalizing the knowledge gained from each task, taking it from the task-specific map to a generalized map. As knowledge is gathered after each task, the student model can be used during the exercise session.

5-3 Knowledge Models

The CREEK-ILE system is designed to work with the CREEK knowledge representation, which is in the tradition of frame-based and semantic network representations. These types of representations use abductive and inductive reasoning, which contrasts with deductive-based representations such as KL-ONE [24], Description Logic [20].

First, the CREEK representation will be presented and formally defined and the extensions made to the representation in order to support CREEK-ILE will be presented. The expert, student and pedagogical models of CREEK-ILE then follow.

5-3.1 Representation Language

The CREEK representation has gone through several iterations, and was originally known as the CreekL representation [1, 2]. The current version is developed on the basis of the representation presented in [76], which defines it in a formalization inspired by the semantic network formalizations defined by Shastri [74] and Touretzky [84]. The basic representation formalism including concepts, relationships (except relationship strengths), relation-types and inverse relationships was developed in [76], while the extensions relating to relationship strength, partial preference ordering of the relationships, defaults and submodels are developed as a part of this work.

Concepts and Relationships

The CREEK representation is syntactically a graph-based representation and can be presented as a directed, labeled graph. In this graph, the nodes represents concepts and the edges relationships between the conepts. CREEK does not distinguish syntactically between different types of concepts, and they may represent general, abstract classes as well as individuals. When referring to a variable representing a single concept, we will use the letter c, while the capital letter C refers to a set of concepts.

The *relationship* is the CREEK equivalent to graph edges – it ties together two concepts with an arc of a particular type. The variables representing single relationships are denoted by the letter r, while sets of relationships use the capital letter R. Formally, a relationship is an ordered set of triplets of concepts and a relationship strength between 0 and 1 $(c_o, t, c_v, x) \in (C \times T \times C \times [0..1])$, where c_o represents the origin of the relationship (the concept it is "from"), c_v represents the value (the concept it is "to") and t represents the type of the relationship. The value x is the strength of the relationship. The relation-type fills the role of the relationships label, but is more than a text string – they are actually represented as concepts of their own. This provides a meta-level that allows the representation to contain information about the labels themselves. In CREEK, the concepts representing relation labels are known as *relation-types*, with the letter t representing a single relation-type and the capital letter T representing a set of relation-types. Because all relation-types are also concepts, T is a subset of C. The strength of a relationship can be interpreted as a measure of how frequently the relationship is expected to hold. For instance, in a basic weather prediction domain, a causal relationship between HURRICANE and RAIN can be very strong (i.e. close to 1.0), while a causal relationship between CLOUDS and RAIN would be less strong (perhaps 0.5), as observing clouds need not necessarily mean it will be rain.

Definition 5-3.1 (CREEK Semantic Network) *A CREEK semantic network is an ordered set* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$*, where*

C *is a finite set of concepts,*

$R \subseteq (C \times T \times C \times [0..1])$ *is the set of the relationships in the network.*

$T \subseteq C$ *is the set of relation-types,*

$\lambda : T \rightarrow T$ *is a symetric function giving the inverse relation-type for each relation-type in* T*,*

$\psi : R \rightarrow R$ *is a symetric function representing the inverse relationship for each relationship in*

R*, so that for all* $r = (c_o, t, c_v, x_1) \in R$*, it holds that* $\psi(r) = (c_v, \lambda(t), c_o, x_2)$*, where* x_1 *and* x_2 *can be any strength value, and need not be the same,*

$\gg \subseteq (R \times R)$ *is a partial ordering of relationships that represents explicitly which relationships override other relationships,*

$S \subseteq S_\eta$ is the set of submodels of this model, where $S_\eta = (\mathcal{P}(C) \times \mathcal{P}(R))$ is the set of all possible submodels of η, and

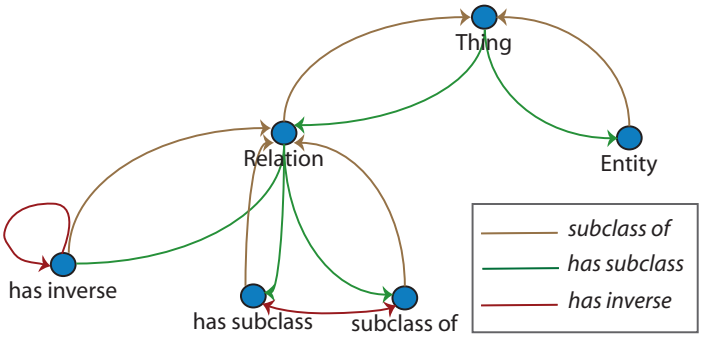
$\rho : C \rightarrow S$ is a function associating a submodel with a concept.

We use $\mathcal{P}(A)$ to denote the power-set of A, i.e. the set of all subsets of A.

The CREEK representation enforces the constraint on the representation that every relationship must have an inverse relationship. The reason for this is the idea that if the concept c_o has a relationship with c_v, then there is also some kind of relationship between c_v and c_o. This needs not be the same kind of relationship, for instance in our previous example, if HURRICANE *causes* RAIN, then this implies an inverse relationship of a different kind between RAIN and CAUSES, specifically that RAIN is *caused by* HURRICANE. This is in practice a constraint that is automatically upheld by the implementation, as the inverse relationship (except the strength) is implied by the original relationship. Basically, the inverse relationship has just inversed the origin and target concepts, and the relation type is given by the λ function. However, the relationship strengths may be (and often are) different between original and inverse relationships. In the weather example, it is certainly the case that most hurricanes are caused by rain, but although rain may be caused by hurricanes, it is normally the result of more mundane weather. For an example of a basic CREEK model, presented both visually and in the formal syntax, see Figure 5.4

A CREEK model also contains a partial ordering of the relationships, $\gg$. We will say that if relationship $(r_1, r_2) \in \gg$, then r_1 *overrides* r_2, and we will denote this $r_1 \gg r_2$. This feature is required when applying inference to the representation, and is specifically designed to resolve multiple inheritance conflicts. The basic idea is that if two relationships r_1 and r_2 are both inferred to be present at the same concept, but $r_1 \gg r_2$, r_2 should be removed from the list of inferred relationships. This is important because CREEK does not assume that there can be only one of each relation-type from a particular concept. In classical discussions of inheritance in semantic networks, it is often assumed that for instance ROYAL ELEPHANT *has color* WHITE implicitly overrides ELEPHANT *has color* GREY when ROYAL ELEPHANT is a subclass of ELEPHANT. The assumption here is that there can only be one HAS COLOR value for a given concept. In this example, it may seem natural that an elephant should only have one color, but this is not generally the case. When modeling cause and effect, for instance, there may be many possible causes for a single phenomenon. This means that there must be some mechanism for identifying when a relationship is meant to override a default as opposed to creating an additional relationship.

Last, a CREEK model may contain a number of submodels (S). These contain subsets of the concepts and relationships in the total model. This allows the representation to model some second-order properties, such as the beliefs of a person, and separate those from the beliefs of the system. Each submodel must also be associated with a concept that represents the submodel. This allows for representing meta-information about the submodel itself, such as whose beliefs it represents. This is particularly important for student modeling and representing



BasicModel = $(C, R, T, \lambda, \psi, \gg, S, \rho)$, where

$C =$ {THING,ENTITY,RELATION,HAS SUBCLASS,SUBCLASS OF,HAS INVERSE}

$R =$ { $r_1, r_2, ..., r_{13}$ }, where

r_1 = (THING,*has subclass*,RELATION,1.0)

r_2 = (THING,*has subclass*,ENTITY,1.0)

r_3 = (ENTITY,*subclass of*,THING,1.0)

r_4 = (RELATION,*subclass of*,THING,1.0)

r_5 = (RELATION,*has subclass*,SUBCLASS of,1.0)

r_6 =(RELATION,*has subclass*,HAS SUBCLASS,1.0)

r_7 =(RELATION,*has subclass*,INVERSE OF,1.0)

r_8 = (SUBCLASS OF,*subclass of*,RELATION,1.0)

r_9 = (SUBCLASS OF,*inverse of*,HAS SUBCLASS,1.0),

r_{10} = (HAS SUBCLASS,*subclass of*,RELATION,1.0)

r_{11} = (HAS SUBCLASS,*inverse of*,SUBCLASS OF,1.0),

r_{12} = (INVERSE OF,*subclass of*,RELATION,1.0)

r_{13} = (INVERSE OF,*inverse of*,INVERSE OF,1.0)

$T =$ {*subclass of,has subclass,inverse of*}

$\lambda =$ { *subclass of* $\rightarrow$ *has subclass*,
has subclass $\rightarrow$ *subclass of*,
has inverse $\rightarrow$ *has inverse* }

$\psi =$ { $r_1 \rightarrow r_4$, $r_2 \rightarrow r_3$,
$r_3 \rightarrow r_2$, $r_4 \rightarrow r_1$,
$r_5 \rightarrow r_8$, $r_6 \rightarrow r_{10}$,
$r_7 \rightarrow r_{12}$, $r_8 \rightarrow r_5$,
$r_9 \rightarrow r_{11}$, $r_{10} \rightarrow r_6$,
$r_{11} \rightarrow r_9$, $r_{12} \rightarrow r_7$,
$r_{13} \rightarrow r_{13}$ }

$\gg =$ $\emptyset$

$S =$ $\emptyset$

$\rho =$ $\emptyset$

Figure 5.4: A basic CREEK model

concept maps. While this is a new feature of the CREEK representation, it was introduced by Hendrix [42] as Partioned Networks.

Frames

The definition of a CREEK model is centered on nodes (concepts) and edges (relations), but we often refer to *frames*, which we take to be the concept and all the relationships in the model from that concept. A formal definition of the *frame function* for a CREEK model is given by Definition 5-3.2.

Definition 5-3.2 (Frame Function) *A function* $F : C \rightarrow \mathcal{P}(R)$ *is the* frame function *for a CREEK model* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$ *iff for all* $c \in C$*, the relationship* $r \in F(c)$ *iff the origin of* r *is* c*.*

The basic inference mechanisms of CREEK are centered on the frame. The first of these are inferring additional relationships to the frame, forming what we call the *extended frame*. The second mechanism is frame matching, that is, searching for frames that are similar to the current frame. CREEK does not implement general frame matching, but its case-based reasoning engine performs a particular type of frame matching.

Subclass Inheritance

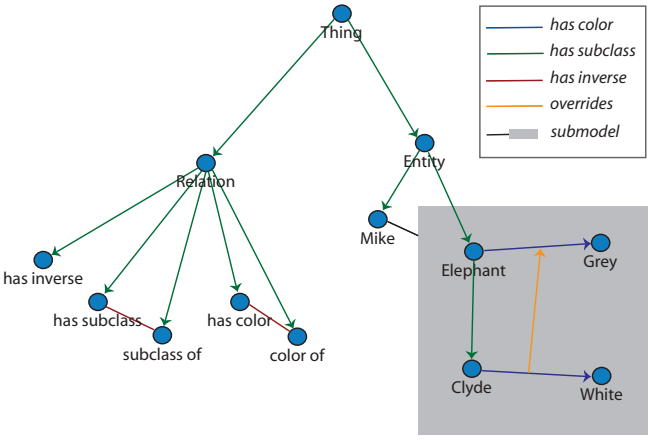
The first of these mechanisms, inferring additional relationships, is an extension of the traditional subclass inheritance mechanism in the tradition of *path-based inference*, as defined in [84]. An important feature of path-based inference is that each relationship $r_i = (c_o, t, c_v, x)$ that is inferred to be a member of an extended frame for the concept c_o must be justified by a path from c_o to c_v.

Definition 5-3.3 (Paths) *A path* p *from* c_1 *to* c_n *is an ordered, loop-free set of chained relationships, i.e. so that*
$p = ((c_1, t_1, c_2, x_1), (c_2, t_2, c_3, x_2), (c_3, t_3, c_4, x_3), ..., (c_{n-1}, t_{n-1}, c_n, x_{x-1})$.

Definition 5-3.4 (Path Strength) *We define the strength of a path* $p = ((c_1, t_1, c_2, x_1), (c_2, t_2, c_3, x_2)..., (c_{n-1}, t_{n-1}, c_n, x_{x-1})$ *as* $x_p = x_1 * x_2 * ... * x_{n-1}$.

Definition 5-3.5 (Path-Set) $P_\eta \subseteq \mathcal{P}(R)$ *is the path-set for* η *iff it contains all possible paths in* η*, i.e.* $p \in P_\eta$ *iff* p *is a path, and for all* $r \in p$, $r \in R$.

If we assume that each path can at most justify a single inferred relationship, and the set of possible paths P_η for any CREEK model η is finite, the set of possible inferred relationships for any model is also finite.



Model = $(C, R, T, \lambda, \psi, \gg, S, \rho)$, where

$C =$ { THING,ENTITY,RELATION,HAS SUBCLASS,SUBCLASS OF, HAS INVERSE,HAS COLOR,COLOR OF,MIKE,ELEPHANT,CLYDE,WHITE,GREY}

$R =$ $\{r_1, r_2, ..., r_{23}\}$, where

r_1 = (THING,*has subclass*,RELATION,1.0)
r_2 = (THING,*has subclass*,ENTITY,1.0)
r_3 = (ENTITY,*subclass of*,THING,1.0)
r_4 = (RELATION,*subclass of*,THING,1.0)
r_5 = (RELATION,*has subclass*,SUBCLASS OF,1.0)
r_6 =(RELATION,*has subclass*,HAS SUBCLASS,1.0)
r_7 =(RELATION,*has subclass*,INVERSE OF,1.0)
r_8 = (SUBCLASS OF,*subclass of*,RELATION,1.0)
r_9 = (SUBCLASS OF,*inverse of*,HAS SUBCLASS,1.0)
r_{10} = (HAS SUBCLASS,*subclass of*,RELATION,1.0)
r_{11} = (HAS SUBCLASS,*inverse of*,SUBCLASS OF,1.0)
r_{12} = (INVERSE OF,*subclass of*,RELATION,1.0)
r_{13} = (INVERSE OF,*inverse of*,INVERSE OF,1.0)
r_{14} = (ENTITY,*has subclass*,MIKE,1.0)
r_{15} = (MIKE,*subclass of*,ENTITY,1.0)
r_{16} = (ENTITY,*has subclass*,ELEPHANt,1.0)
r_{17} = (ELEPHANT,*subclass of*,ENTITY,1.0)
r_{18} = (ELEPHANT,*has color*,GREY,0.7)
r_{19} = (GREY,*color of*,ELEPHANT,0.7)
r_{20} = (ELEPHANT,*has subclass*,CLYDE,1.0)
r_{21} = (CLYDE,*subclass of*,ELEPHANT,1.0)
r_{22} = (CLYDE,*has color*,WHITE,1.0)
r_{23} = (WHITE,*color of*,CLYDE,1.0)

$T =$ {*subclass of*,*has subclass*,*inverse of*,*has color*,*color of*}

$\lambda =$ { *subclass of* $\rightarrow$ *has subclass*, *has subclass* $\rightarrow$ *subclass of*, *has color* $\rightarrow$ *color of*, *color of* $\rightarrow$ *has color*, *has inverse* $\rightarrow$ *has inverse* }

ψ : For all $r = (c_o, t, c_v, x) \in R$, $\psi(r) = (c_o, \lambda(t), c_v, x)$.

$\gg =$ { (r_{22}, r_{18}),(r_{23}, r_{19}) }

$S =$ { s_1 }, where

s_1 = {ELEPHANT,GREY,CLYDE,WHITE,r_{18}, r_{20}, r_{22} }

$\rho =$ { *Mike* $\rightarrow s_1$ }

Figure 5.5: A CREEK model with overrides and a submodel. Inverse relations are not drawn.

Definition 5-3.6 (Justified Inferred Relationships) *A justified relationship j for the CREEK model $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$ is an ordered tuple $(r, p) \in (R_\eta \times P_\eta)$ where*

$R_\eta = (C \times T \times C \times [0..1])$

$r = (c_o, t, c_v, x)$ *is a relationship,*

p *is a path from* c_o *to* c_t*, and*

the strength of p *equals the the strength of* r *(i.e.* x*).*

A set of justified relationships is denoted J, and the set of all possible justified relationships in η is denoted J_η.

This means that a path-based inference method can be defined as a function that given a CREEK model η will identify the subset of all possible justified relationships J_η for this model supported by that method. However, in practical use it is more common to extend the frame of a single concept at a time, finding all justified relationships originating from a given concept in the model. We will use the form given in Definition 5-3.7 when defining path-based inference functions.

Definition 5-3.7 *(Path-Based Inference Function) The function $I : C \rightarrow \mathcal{P}(J_\eta)$ is a path-based inference function for η iff for all $(c \rightarrow J) \in I$, all justified relationships $(r, p) \in J$ has c as the origin concept for r.*

This definition of a path-based inference function allows for a multitude of different inference mechanism, one of which is the classic subclass inheritance method. The inference function for the classic inheritance method would require that all relation-types in the path that justifies an inherited relationship must be of the type *subclass of*, and that the type of the inferred relationship equals the last relationship in the path:

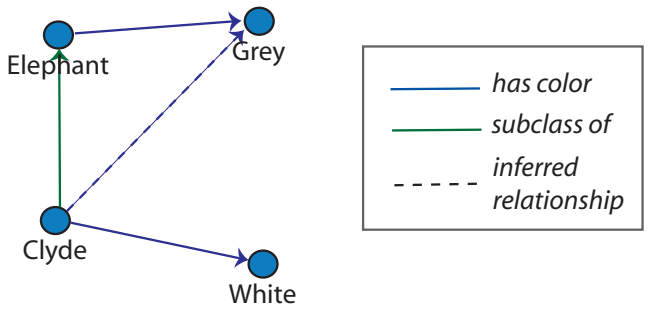
Definition 5-3.8 (Subclass-Inheritance Function) *The subclass inheritance function for CREEK model $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$ is a path-based inference function $I_{subclass} : C \rightarrow \mathcal{P}(J_\eta)$ where*

for all $c \in C$, $I_{subclass}(c)$ contains (r, p) iff,

given $p = (r_1, r_2, ..., r_{n-1}, r_n)$, the relation-type of all relationships $r_1 ... r_{n-1}$ is "subclass of", and

$r = (c, t, c_v, x)$, where t is the relation-type of r_n, c_v is the value concept of r_n and x is the strength of path p.

An example of the subclass inheritance function applied to the CLYDE concept from the extended CREEK model (Figure 5.5) is found in Figure 5.6.



The frame function applied to CLYDE:
F(CLYDE) → {(CLYDE,*subclass of*,ELEPHANT,0.95),
(CLYDE,*has color*,WHITE,0.7) }

The subclass inheritance function applied to CLYDE:
$I_{subclass}$(CLYDE) → {
((CLYDE,*has color*,WHITE,0.63),
((CLYDE,*subclass of*,ELEPHANT,0.9),(ELEPHANT,*has color*,WHITE,0.7))),
((CLYDE,*subclass of*,ELEPHANT,0.95),
((CLYDE,*subclass of*,ELEPHANT,0.95))),
((CLYDE,*has color*,WHITE,0.7),
((CLYDE,*has color*,WHITE,0.7))) }
}

In this example, the relationship (CLYDE,*has color*,WHITE,0.63) was inferred by the subclass inheritance function, justified by the path ((CLYDE,*subclass of*,ELEPHANT,0.9),(ELEPHANT,*has color*,WHITE,0.7)) . Note that the strength of the inherited relationship (0.63) is the product of the strengths of the relationships in the path (0.7×0.9). Note that the inheritance function will also give the local relationships, which are trivially justified by a one-step path containing only the relationship itself.

Figure 5.6: The subclass inheritance function applied to CLYDE.

Plausible Inheritance

In CREEK, the default path-based inference mechanism is an extension of basic inheritance we call *plausible inheritance* [76]. This mechanism allows for inheritance over any relation-type, as dictated by a set of rules saying for instance that *causes* and *has color* may be inherited over *subclass of*, while *causes* may also be inherited over other *causes* relationships. This last rule, for instance, makes the *causes* relation-type transitive. Each CREEK model may have a set of such plausible inheritance rules.

Definition 5-3.9 (Plausible Inheritance Rule-Set) *A plausible inheritance rule-set for a CREEK model* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$ *is a mapping* $\delta \subseteq (T \times T)$ *from relation-types to relation-types. If* $(t_1, t_2) \in \delta$*, this means that all relationships of the type* t_2 *may be transferred over any relationships of type* t_1*. Here,* t_1 *is said to* transfer t_2*. We can also say that if relationship* r_1 *is of type* t_1 *and* r_2 *is of type* t_2*,* r_1 *transfers* r_2*.*

This allows us to formulate a more general inheritance mechanism than subclass inheritance – plausible inheritance. In plausible inheritance, each inferred relationship r must be justified by a path of the same origin and value as r, and in this path, all relationships except the last must transfer r. The last relationship in the path must be of the same type as r.

Definition 5-3.10 (Plausible Inheritance Function) *The plausible inheritance function for CREEK model* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$ *is a path-based inference function* $I_{plausible} : C \rightarrow \mathcal{P}(J_\eta)$ *where*

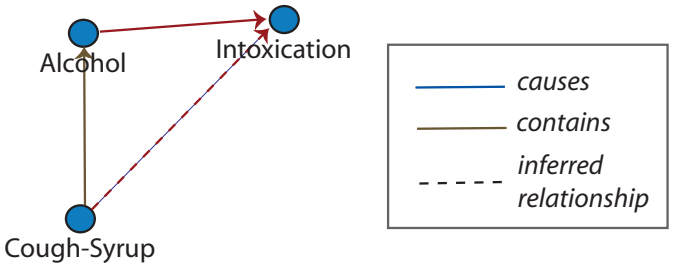
for all $c \in C$*,* $I_{plausible}(c)$ *contains* (r, p) *iff,*

given $p = (r_1, r_2, .., r_{n-1}, r_n)$*, it holds that all relationships* $r_1 ... r_{n-1}$ *transfers* r_n*, and*

$r = (c, t, c_v, x)$*, where* t *is the relation-type of* r_n*,* c_v *is the value concept of* r_n *and* x *is the strength of path* p*.*

These rules allow the modeler to create domain-specific inference rules. For example, the simple example in Figure 5.7 has a rule saying that *contains* transfers *causes*. This may make sense in a domain reasoning about liquids, but makes less sense in a car domain, for instance. Just because a car contains fuel does not necessarily make it a fire bomb, at least outside Hollywood movies.

The algorithm implementing this mechanism is slightly more computationally complex than classical subclass inheritance, although still in polynomial time. It is based on a basic spreading activation scheme, where the set of relation-types that can be inherited to the origin concept (c) is propagated through the concepts of the model. The initial transfer set of types is the complete set of relation-types, or the set of relation-types we are interested in. Then, relationships are traversed in a breadth-first search, where the target concept of each relationship is assigned the transfer set of relation-types at the origin intersected with the set of relation-types the relationship transfers. If there are



The frame function applied to COUGH-SYRUP:
F(COUGH-SYRUP) $\rightarrow$ {(COUGH-SYRUP, *contains*, ALCOHOL, 0.4),

The plausible inheritance rule set is δ = { (*contains*, *causes*) }.
The plausible inheritance function applied to COUGH-SYRUP:
$I_{plausible}$(COUGH-SYRUP) $\rightarrow$ {
((COUGH-SYRUP, *causes*, INTOXICATION, 0.2),
((COUGH-SYRUP, *contains*, ALCOHOL, 0.4), (ALCOHOL, *causes*, INTOXICATION, 0.5))),
((COUGH-SYRUP, *contains*, ALCOHOL, 0.4),
((COUGH-SYRUP, *contains*, ALCOHOL, 0.4))) }

This CREEK model fragment illustrates how plausible inheritance can be used to infer the relationship COUGH-SYRUP *causes* INTOXICACTION. This requires that the rule (CONTAINS, CAUSES) is in the plausible inheritance rule set.

Figure 5.7: A simple example of plausible inheritance.

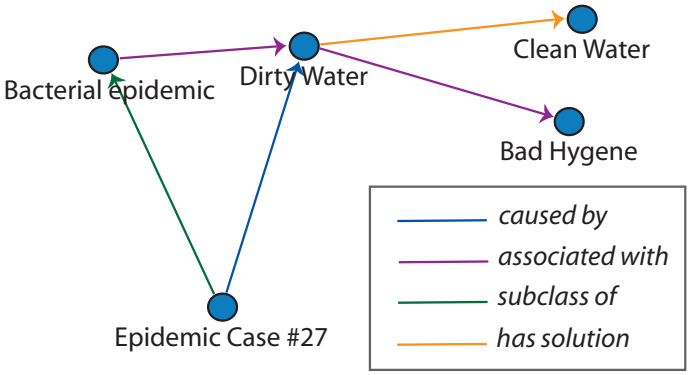
more than one path between any given concept and the origin, the concept may first be assigned a transfer set from path one, and then another from the second path. Since the transfer set represents the set of relation-types that can be transferred from this concept to the origin, these sets are then combined, and spreading activation from this concept must be redone. This re-spreading is the cause of the increased complexity, but it is bounded by the number of relation-types in the model. Since the transfer set at a concept can only increase, and at worst by one at a time, the complexity of the plausible inheritance is $O(|T|*|R|)$. For a more in-debt presentation of the algorithm, see [76]. An example of the plausible inheritance with several types of relationships inherited along different paths are shown in Figure 5.8.

Default Inheritance

One problem with the above methods of inference is that they do not address default reasoning. The definition of the CREEK model contains a partial preference ordering of the relationships $\gg$. The semantics of this partial ordering is that no inference should result in an extended frame that contain any relationships justified by conflicting paths. By conflicting paths, we mean two paths p_1 and p_2 that contain respectively relationships r_1 and r_2 where r_1 overrides r_2 ($r_1 \gg r_2$). For example, the model in Figure 5.9 may on the surface look quite simple. ROYAL ELPHANT is a kind of ELEPHANT that is WHITE instead of the normal GREY. CLYDE is a ROYAL ELEPHANT, but is not WHITE. In determining the color of Clyde, the subclass inheritance function $I_{subclass}$ described earlier would infer that Clyde is both white and grey, as it does not consider overrides at all. A version of this function that considered overrides according to the semantics of $\gg$ would not be able to include the relationship saying that Clyde is white, as the path supporting this conclution contains a relationship (ROYAL ELEPHANT *has color* WHITE) that is overriden by another relationship (CLYDE *subclass of* ROYAL ELEPHANT). Consider then if the default subclass inheritance function may infer that Claude is grey instead. This is compatible with the definition of $\gg$, because the relationship stating that elephants are grey was only overridden by a relationship that was itself overriden and removed. This means that when the ROYAL ELEPHANT *has color* WHITE was removed from the extended frame, there is no longer any relationship used locally that overrides the ELEPHANT *has color* GREY relationship. The conclusion here is that we can infer that Clyde is grey because the more specific relationship (that he is white) is overriden by an even more specific relationship.

This can be further complicated, because it is possible to model situations where two paths that override each other can be used to justify relationships inherited to the same concept. In this situation, there is not necessarily enough information to know which path to prefer. For instance, Figure 5.10 illustrates such a situation, where Clyde can not both be white and grey[1]. In this example,

[1] This example is not strictly speaking entirely syntactically correct as partial orderings may not contain loops. Currently, the implementation of this is a basic mapping that does not enforce this constraint. It is possible to generate syntactically correct examples with the



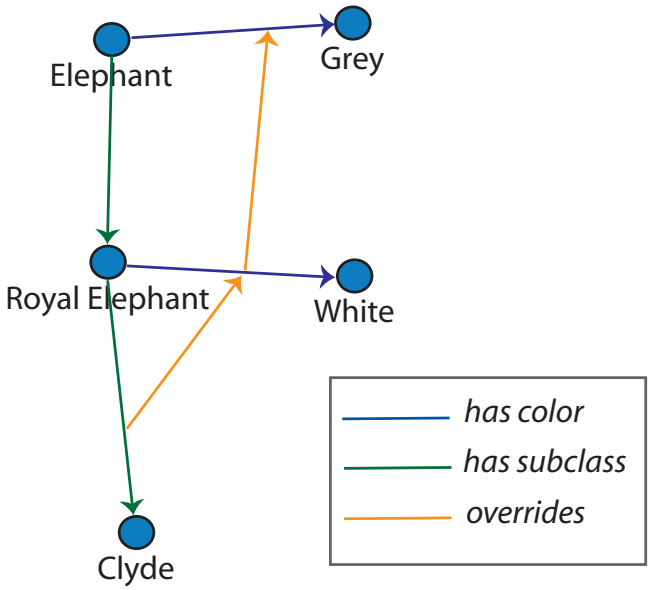
PlausibleEx = $(C, R, T, \lambda, \psi, \gg, S, \rho)$, where
$C \supset$ { Epidemic Case #27,Bacterial Epidemic,Dirty Water,
Clean Water,Bad Hygiene }
$T \supset$ { *caused by,associated with,subclass of,has solution* }
$R \supset$ { $r_1, r_2, ..., r_5$ }
r_1 = (Epidemic Case #27,*subclass of*,Bacterial Epidemic,0.8)
r_2= (Epidemic Case #27,*caused by*,Dirty Water,0.9)
r_3= (Bacterial Epidemic,*associated with*,Dirty Water,0.6)
r_4= (Dirty Water,*has solution*,Clean Water,0.9)
r_5= (Dirty Water,*associated with*,Bad Hygiene,0.7)

The plausible inheritance rule-set δ =
{ (*subclass of,associated with*),
(*subclass of,has solution*),
(*subclass of,caused by*),
(*associated with,associated with*),
(*caused by,caused by*),
(*caused by,has solution*) }

The plausible inheritance function applied to Epidemic Case #27:
$I_{plausible}$(Epidemic Case #27) →{
((Epidemic Case #27,*has solution*,Clean Water Supply,0.81),
(r_2,r_4)),
((Epidemic Case #27,*associated with*,Dirty Water,0.48),
(r_1,r_3)),
((Epidemic Case #27,*associated with*,Bad Hygiene,0.336),
(r_1,r_3,r_5)),
((Epidemic Case #27,*caused by*,Dirty Water,0.9),
(r_2)),
((Epidemic Case #27,*subclass of*,Bacterial Epidemic,0.8),
(r_1)) }

This plausible inheritance example illustrates how one relationship (the *has solution*) is inherited through the *caused by* relationship, and two others (the *associated with* relationships) are inherited through the *subclass of* relationship. In the interest of brevity, the model described above is a partial CREEK model, with only subsets of R, C and T defined. This figure is adapted from [76, fig. 4.11,p 58].

Figure 5.8: A larger example of the plausible inheritance method.



DefaultEx = $(C, R, T, \lambda, \psi, \gg, S, \rho)$, where
$C \supset$ { CLYDE,ROYAL ELEPHANT,ELEPHANT,GREY,WHITE }
$T \supset$ { *subclass of,has color* }
$R \supset$ { r_1 =(CLYDE,*subclass of*,ROYAL ELEPHANT,0.9),
r_2 =(ROYAL ELEPHANT,*subclass of*,ELEPHANT,0.9),
r_3 =(ELEPHANT,*has color*,GREY,0.7),
r_4 =(ROYAL ELEPHANT,*has color*,WHITE,0.7) }
$\gg\supset$ { (r_4, r_3),
(r_1, r_4) }

Figure 5.9: An example of default reasoning in the CREEK representation.

there can be said to be two conflicting internally consistent models that can be inferred. The first is that Clyde is a grey, royal, Indian elephant, and the second that he is a white, royal, Indian elephant. Touretzky suggests a *skeptical* approach to path-based inference, which allows the inference of relationship if and only if it is contained in all models. Touretzky suggests that a relationship should not be inferred if there is active evidence against it.

A study of the complexity of this problem by Selman [73] shows that there are variations of this task that is possible to do in polynomial time, although Touretzky's formulation is unfortunately NP-hard. Currently, the CREEK system does not incorporate these results completely. The current CREEK approach is to remove any path from the set of inferred paths that has a relationship that is overridden by a relationship that is in any other path in the set. This approach only infers relationships that are *correct* with regards to Touretzky's skeptical inference, but it is not *complete* – it may omit relationships that are actually consistent with all models. For example, in Figure 5.9, CREEK is unable to conclude that Clyde is grey (although it will not conclude Clyde is white either). In practical modeling, this has not been a major problem, although it certainly is a topic for further research. The definition of CREEK's default-aware inheritance functions can be formally defined as:

Definition 5-3.11 (Default-Aware Subclass Inheritance Function) *The default-aware subclass inheritance function for the CREEK model* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$ *is a path-based inference function* $I_{da-subclass} : C \rightarrow \mathcal{P}(J_\eta)$ *where* $(r, p) \in I_{da-subclass}$ *iff*

$(r, p) \in I_{subclass}(c)$, *and*

there is no $(r', p') \in I_{subclass}(c)$ *where*

there is a $r_p \in p$ *and* $r_{p'} \in p'$ *so that* $r_{p'} \gg r_p$.

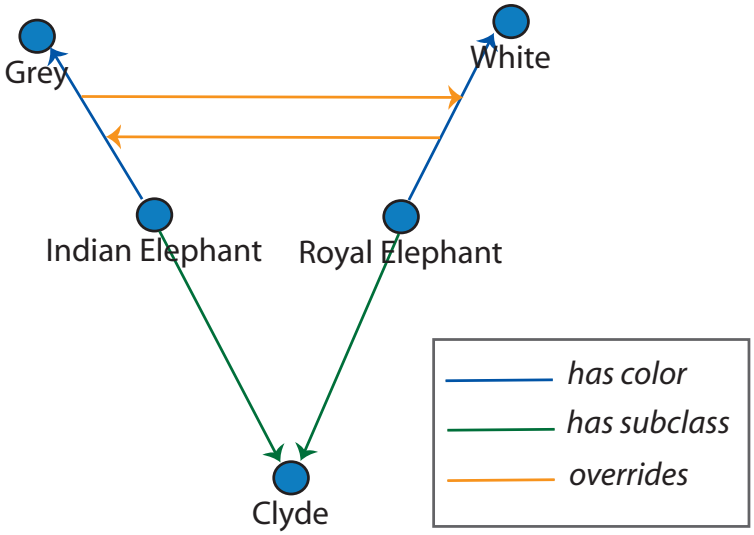
Definition 5-3.12 (Default-Aware Plausible Inheritance Function) *The default-aware plausible inheritance function for the CREEK model* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$ *is a path-based inference function* $I_{da-plausible} : C \rightarrow \mathcal{P}(J_\eta)$ *where* $(r, p) \in I_{da-plausible}$ *iff*

$(r, p) \in I_{plausible}(c)$, *and*

there is no $(r', p') \in I_{plausible}(c)$ *where,*

there is a $r_p \in p$ *and* $r_{p'} \in p'$ *so that* $r_{p'} \gg r_p$.

same problem, for instance by adding another step in each path and having the middle step of each path overriding the last step of the other path.



DefaultEx = $(C, R, T, \lambda, \psi, \gg, S, \rho)$, where
$C \supset$ { Clyde,Royal Elephant,African Elephant,Elephant }
$T \supset$ { *subclass of*,*has color* }
$R \supset$ { $r_1, r_2, ..., r_5$ }, where
r_1 =(Clyde,*subclass of*,Royal Elephant,0.9)
r_2 =(Royal Elephant,*subclass of*,Elephant,0.9)
r_3 =(Clyde,*subclass of*,African Elephant,0.9)
r_4 =(Royal Elephant,*has color*,White,0.9)
r_5 =(African Elephant,*has color*,Grey,0.9)
$\gg \supset$ { (r_4, r_5),
(r_5, r_4) }

Figure 5.10: A problematic default-reasoning model.

Semantics in Path-Based Inference

The semantics of path-based inference is markedly different from that of deductive systems, such as first-order predicate calculus. The most obvious difference is that path-based inference is not sound. This is consistent with the view of knowledge representation often represented by frame-based systems that models should represent prototypicality. In this view, a frame is not a hard definition of a concept, but a collection of properties that *typically* hold for a concept or type of object. This means that if a given instance is recognized as being of a particular frame, it is not necessarily true that all the properties of the identified frame hold for the instance. This suggests that inference should also be geared towards providing conclusions that are *typically* true, as opposed to conclusions that are *always* true. The central tenant of the CREEK knowledge representation is that only what is observed directly is taken to be true and represented explicitly, while inferences are treated as uncertain knowledge.

It is interesting to contrast the CREEK knowledge representation with description logics [20], which is similar in that both representations are created to support representations of taxonomic information, inference along subclass/superclass relations and recognition of instances. In description logics, knowledge is definitional in nature, and it uses sound inference. This means that it is able to reach stronger conclusions than CREEK, but only at the expense of requiring a priori correct and consistent taxonomic knowledge. Description logics also have a separate instance level, which CREEK lacks.

Although CREEK and path-based inference systems in general are not sound, model-based semantics can be useful. Touretzky adopts such an approach in formulating his goal for default path-based inference, for instance by requiring that a *skeptical* method should only accept the inference of relations that exists in all models. In this case, this does not mean that the relation must always be true, but rather that there is no opposing evidence represented. Because CREEK uses a measure of strength on each relationship to represent the degree of belief, we could in theory extend this approach to calculate model strength and accept the stronger model, but at this time the details and computational complexity of such an approach has not been investigated.

At this time, the submodel feature is only used to limit inference, for instance by using a special version of the plausible inheritance method that only considers the concepts and relationships in a submodel. Beyond this, the submodel feature has no machine-interpretable meaning.

Concept and Epistemological Level Semantics

It has been a goal for our representation to allow the modeler to work with a visual 2-dimensional graph as much as possible. Clearly, the set of concepts, C, in a CREEK model may be represented as nodes, the set of relationships (R) as labeled edges (with weights), but a CREEK model also contains other elements that have no obvious analogue in a graph. In order to solve this, we use a variant of the CREEK representation where special meaning has been assigned by the

system to a set of required elements so that these constructs can be identified when using a visual graph tool.

Already, the *subclass of* relation-type has been identified as having special meaning for the subclass inheritance method, and in the first example of a CREEK model, the *has inverse* relation-type was used to illustrate the λ function in Figure 5.4. This relation-type can actually be used to represent the λ function, as long as any relation-type has only one of them and the relation-type is its own inverse (the inverse relationship requirement forces a relationship of the same type in the opposite direction in this case, enforcing the symetricity of λ).

The definition of the CREEK model requires that the set of relation-types, T, is a subset of the concepts. In these CREEK models, the presence of a concept with the name RELATION is required, and all concepts that have a direct or indirect *subclass of* relationship from this concept is a member of the set of relation-types R.

Neither the set of submodels (S), the function associating each submodel with a concept (ρ) nor the partial ordering of the relationships ($\gg$) can be dealt with by defining them in terms of other constructs, as there is no way to store relationships from or to other relationships. These constructs must for that reason remain outside graph form. However, it seems a natural extension to a graph manipulation tool to select parts of the graphs as members of submodels, and the relationship overrides may also be illustrated as special relationship-to-relationship arcs in a tool.

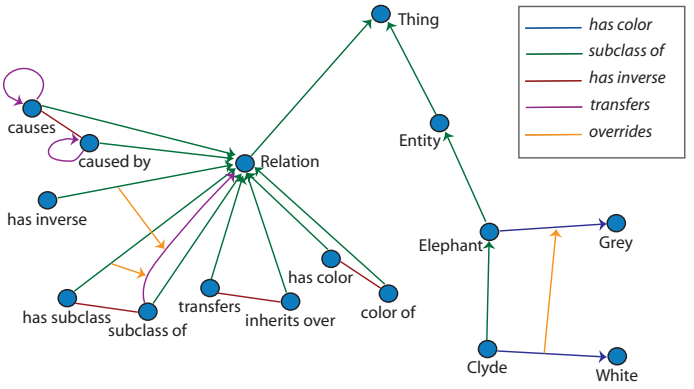
A particularly useful property of these CREEK models is that is possible to represent the plausible inheritance rules directly as part of the model. Since relationships are from concepts to concepts, and all relation-types are stored as concepts in the model, the mapping may be stored as a subset of the set of relationships, using a particular kind of relation-type called *transfers*. Using this relation, we can say that:

Definition 5-3.13 *Plausible Inheritace Rule-Set] Given a CREEK model* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$*, the default plausible inheritance rule-set* δ_η *contains* (t_1, t_2) *iff,*

$$t_1, t_2 \in C,$$

$$(t_2, \lambda(\text{"}transfers\text{"}), t_1, x) \in I_{da-subclass}(t_2).$$

In order to allow for instance the *subclass of* relation-type to inherit all other relation-types without making a rule for all relation-types, we use the subclass inheritance method to inherit rules along the *subclass of* relation-type. An example of a CREEK model where this is done is found in Figure 5.8. Obviously, the plausible inheritance method itself cannot be used for this, as the result of the operation (the plausible inheritance rule set) is required to run the plausible inheritance method. This "bootstrapping" of inheritance methods may seem cumbersome in formalization, but it presents a fairly unified method of expression for the knowledge modeler. In practice, modelers working with CREEK have used only *subclass of* relationships to form the relation-type hierarchy.



This example illustrates how relation-types (T), inverse relation-types (λ) and the default plausible inheritance rules (δ) are represented as part of the model. The plausible inheritance rules may not be obvious at first. The *transfer* relationship from SUBCLASS OF to RELATION means that the inverse relationship is inherited to all subclasses of RELATION. In effect, this means that SUBCLASS OF transfers all relation-types, except HAS SUBCLASS and HAS INVERSE, which has relationships overriding the transfer relationship.

In this model, T, λ and δ have the following values:

$$
\begin{aligned}
T = \{ & \textit{ has subclass,subclass of,has inverse,has color,} \\
& \textit{color of,causes,caused by,transfers, inherited over } \} \\
\lambda = \{ & \textit{ has subclass} \rightarrow \textit{subclass of}, \textit{ subclass of} \rightarrow \textit{has subclass}, \\
& \textit{has inverse} \rightarrow \textit{has inverse}, \\
& \textit{transfers} \rightarrow \textit{inherited over}, \textit{ inherited over} \rightarrow \textit{transfers}, \\
& \textit{has color} \rightarrow \textit{color of}, \textit{ color of} \rightarrow \textit{has color}, \\
& \textit{causes} \rightarrow \textit{caused by}, \textit{ caused by} \rightarrow \textit{causes} \ \} \\
\delta = \{ & (\textit{subclass of, subclass of}), \\
& (\textit{subclass of, has color}), \\
& (\textit{subclass of, color of}), \\
& (\textit{subclass of, transfers}), \\
& (\textit{subclass of, inherits over}), \\
& (\textit{subclass of, causes}), \\
& (\textit{subclass of, caused by}), \\
& (\textit{causes, causes}), \\
& (\textit{caused by, caused by}) \ \}
\end{aligned}
$$

Figure 5.11: A basic CREEK graph model.

In this section, only a few concepts and relation-types have been given specific semantics by the system. These identified here are used to re-represent formal constructs, but in implementation, the CREEK representation contains more such special concepts and relation-types. The latter do not directly affect the formalization here, but are used for conveniences such as providing default relationship strength values. For this reason, there is a common base for new CREEK models containing all the concepts and relationships imbued with special semantics by the system. This common base, along with some illustrative concepts, is presented in Figure 5.11.

5-3.2 Expert Model

The CREEK-ILE approach uses two major sources of expert knowledge – *teacher concept maps* and *problem cases*.

Teacher Concept Maps

In CREEK-ILE, students are not given complete freedom to form their own maps, but are limited to using concepts and link labels defined by the teacher as appropriate for the area at hand. In practice, this task is performed by the teacher or expert creating a map from which the concept and link labels are extracted. It is also possible to use the teacher map in other ways, which will be discussed in Section 5-4.

Concept maps follow the basic syntactical structure of labeled graphs. While there are typically also other considerations when creating concept maps, these are usually not absolute rules and thus harder to encode explicitly in the syntax. For instance, many mapping methods require that the map should be hierarchical with general concepts generally positioned higher on the surface used to draw the map. This means that when storing conceptual maps, it is important to store the position of nodes on the surface in order to be able to reproduce them exactly as drawn. However, when multiple persons are asked to draw the same map, the positions of the nodes will vary between different maps even if the graphs are the same. Because of this we do not currently use the node's position when computationally comparing concept maps and below we use definitions that do not include positional information. This allows us to represent a single map using a directed, labeled graph.

Definition 5-3.14 (Concept Map) *A concept map is an ordered set* $M = (V, L, A)$*, where*

V *is a finite set of vertices (concepts),*

L *is a finite set of link labels,*

$A \subseteq V \times L \times V$ *defines the labeled arcs (relations).*

A mapping between this representation and the CREEK representation is fairly straightforward. The idea is to use a submodel to store the vertices

(concepts) and arcs (relationships) of the map, and create each labeled arc as a relation-type.

Definition 5-3.15 (Concept Maps in a CREEK Model) *The concept map* $M = (V, L, A)$ *can be said to be represented as submodel* $s = (C_s, R_s)$ *in the CREEK Model* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$*, iff*

$s \in S$,

$L \subseteq T$,

$C_s = V$, *and*

there is a relation $(c_o, t, c_t, 1.0) \in R_s$*, iff there is an arc* $(c_o, t, c_t) \in A$.

The CREEK representation forces each relationship to have an inverse, and all labels to be represented as concepts, which are not normal requirements of concept maps. However, these constraints need not be upheld internally in a submodel – it can contain a relationship without including the inverse, and use relation-types defined outside the submodel.

In Section 5-1, two example domains were introduced, and the general domain models introduced in figures 5.1 and 5.2 also represent the teacher's concept maps for these two domains.

Problem cases

Exercises represent the problem cases that can be presented to the student. It is important that these exercises also have solutions attached to them, since CREEK-ILE generally will not have the capability of solving the cases (except in certain diagnostic domains, see Section 5-4.2). The solution is important, not necessarily as a way of testing the student's solution, but in the case the student is unable to solve the problem. However, it may be possible to test the student's solution without comparing it directly to a canned solution. A good test of an axe is if it can be used to cut down a tree. In our Java programming domain, we used test against the text of the code as well as a set of unit tests that simulated input to the program and compared output with the expected result in a set of scenarios. For instance, if a student was asked to write a program that used the `for`-loop to count from 1 to a number given by the user, the tests on the student's code would check that there was a for-loop in there, and try a handful of scenarios – for instance inputting 7, and checking that the output matched "1,2,3,4,5,6,7".

The requirements for exercise representation and solution testing obviously vary widely from domain to domain. If the task is to solve equations, such as in the math cognitive tutors from Chapter 4, the answer to the problem is enough to evaluate the student's solution. There may be several ways of reaching an answer to the equation $2x = x + 9$, but in the end, the only correct answer is $x = 9$. However, in many domains, such as the business letter writing in BLITS, there is no single correct answer. Even in the simple programming

example above, students produce a surprisingly large variety of solutions. Thus, in some of these domains, it may be easier to produce tests than an exhaustive list of accepted solutions.
A problem case contains:

- Textual problem description. The presentation of the exercise to the student. This is the only information actually presented to the student.
- Identified features. A subset of the concepts in the teacher's concept map covering this area.
- Solution. A solution to the problem.
- Tests. Tests to check if the student's solution is correct (if possible).

5-3.3 Student Model

The student model in CREEK-ILE is divided into *student concept map* and *solved cases*. The solved cases part is simply an overlay model of the cases already solved by the student. The student concept maps, however, are less straightforward.

In chapter 1, we briefly reviewed how concept maps can be used with different degree of directedness (Figure 1.2). The approach to concept mapping in CREEK-ILE is similar to that of the CRESST research, where the student is presented with a pre-made list of concepts and link labels. This is primarily motivated by pragmatic concerns in allowing the comparisons of maps to be automated, but it also serves the goal of ensuring that the maps produced by students are constrained within the topic decided by the teacher, while also allowing the student some degree of freedom of expression.

Definition 5-3.16 (Student Concept Map) *A concept map* $M_s = (V_s, L_s, A_s)$ *is a student concept map for the teacher concept map* $M_t = (V_t, L_t, A_t)$ *iff*

$V_s \subseteq V_t$ *and*

$L_s \subseteq L_t$.

At the onset, it may seem like this approach limits the student too much, but if we examine a given teacher concept map $M_t = (V_t, L_t, A_t)$, there is vast, although finite, number of student maps that can be created over it. The greatest degree of freedom is granted in the arc – the student may form arcs that do not exist in the teacher map, as long as they are between two existing concepts and uses existing labels. This means that there are $|V_t| * (|V_t| - 1) * |L_t|$ possible arcs that can be drawn, forming $2^{|V_t|*(|V_t|-1)*|L_t|}$ possible student concept maps. The relatively small computer programming teacher map in Figure 5.1, has 13 concepts and 4 arc labels, which means there are 624 possible arcs to draw in a student map, and over 10^{187} possible combination of arcs. Arguably, most of

Description	Write a program that uses a `for`-loop to print all numbers from 1 to 10, except the number 4.
Features	for if

Solution	`public class ForTask`
	`{`
	`  public static void main(String argv[])`
	`  {`
	`   for(int i=1;i<=10;i++)`
	`   {`
	`   if(i != 4)`
	`   {`
	`   System.out.prinlnt("Count is "+i);`
	`   }`
	`   }`
	`  }`
	`}`

Tests	Program source pattern:
	`.*for[\s\n]*\([\s\n]*int[\s\n]+([\w\d]+);.*?\).*println[\s\n]\(.*?)\1(.*?)\).*`
	Program output pattern:
	`.*1.*2.*3.*5.*6.*7.*8.*9.*10.*`

The task above is an example of a simple Java programming task. The tests are regular expressions that are tested against the program source and output to see if the student has successfully solved the problem. The student need not reproduce the suggested solution – it is there for reference if the student fails to find a solution on his own.

Figure 5.12: Example of a simple Java programming exercise.

Description	My car will not start! When I turn the key, there is no sound from the starter motor, and the main engine certainly will not turn. Even the headlights will not come on. I've just filled it up with gas, so that cannot be the problem.
Features	engine does not turn no lights starter motor does not turn
Solution	empty battery
Tests	Check if solution equals "'empty battery"'.

In this task, the student is asked to find the cause of the car problem. The task contains a set of findings from the expert. These are not presented to the student, at least initially, but should be identified from the text. In this domain, the task is to identify the root problem from a given set of possible classes, thus no advanced tests are required.

Figure 5.13: Example of a car problem exercise.

the possible arcs make little sense and can be immediately dismissed, but the possibility space here is so large that it seems to suggest that the student needs to have a clear idea of how concepts relate to each other in order to form a map.

Similarity of Concept Maps

Comparing concept maps is important in many of the CREEK-ILE tutoring tasks. For instance, concept maps created by diffrent students can be compared in order to see if their conceptualization is similar, or a student's map can be compared to the teacher's in an attempt to judge the quality of the student's map.

In order to measure the similarity of two graphs with different sets of vertices, it is necessary to create a mapping function between the sets of vertices and then measure the similarity between each different possible mapping. In general, searching this space for the maximum similarity of the two graphs is combinatorial – Champin and Solnon [32] point out, the problem is more general than the graph isomorphism problem. If the task is to compare the similarity of any two concept maps, a greedy approach such as that proposed by Champin and Solnon may be required. Fortunately, the constraints we have introduced on the concept maps simplify the comparison. Although we allow the teacher to first model the domain freely by creating any labeled graph, the student may use only the concepts and arc labels used in the teacher's map. This means that given a teacher map $M_t = (V_t, L_t, A_t)$, we can compare two student maps $M' = (V', L', A')$, and $M'' = (V'', L'', A'')$. Because both V' and V'' are subsets of V_t, and L' and L'' are subsets of L_t, the A', A'' and A_t relations are all defined over the same set of vertices and labels. In essence, the teacher defines the mapping ahead of time, and guarantees a one-to-one correspondence between

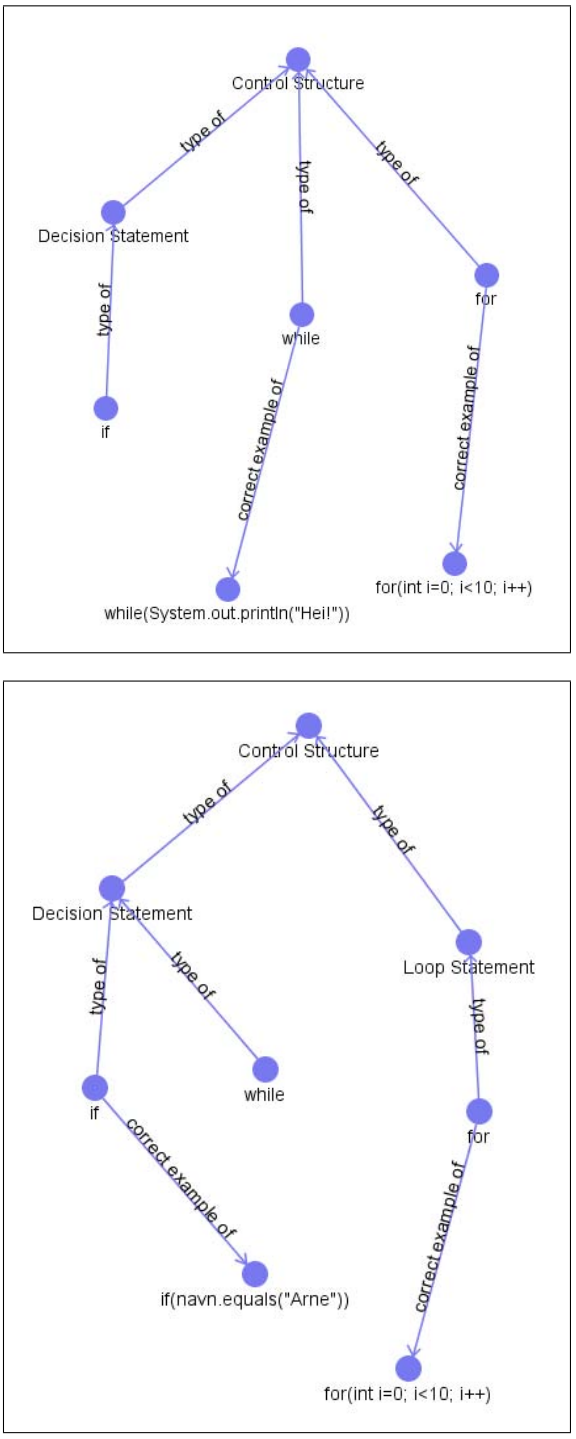


Figure 5.14: Student concept maps from the computer programming domain

vertices and labels used in the student maps. This means that the degree of overlap between the graphs can be measured by measuring the intersection on the vertices and edges. Because the major computational complexity associated with graph comparisons is finding this mapping, computing the similarity of this kind of concept maps becomes trivial.

Sometimes, it is useful to quantify the similarity of two student maps, for instance if they are used in student modeling, and the tutoring system needs to identify other students similar to the current student. In these situations, we use a similarity measure that is an adaptation of the Jaccard Coefficient (also used in [32]), which measure the difference between the union and intersection of the two graphs.

Through testing we have found that some students like to place all the available concepts on the drawing surface before drawing relations between them. This may leave them with several concepts that are not connected to the graph, and if they are included in the similarity measure they may introduce inaccuracies when compared to another student that places concepts on the drawing surface on demand. The presence or non-presence of concepts in the graphs is thus not really indicative of similarity. Because of this, we only use the relations to measure the similarity.

The similarity between two student concept maps $M' = (V', L', A')$, and $M'' = (V'', L'', A'')$, over the same teacher concept map $M_t = (V_t, L_t, A_t)$ is calculated with equation 5.1.

$$sim(M', M'') = \frac{|A' \cap A''|}{|A' \cup A''|} \tag{5.1}$$

If we look at the student concept maps in Figure 5.14, we see that the two student maps have three arcs in common (FOR *correct example* FOR(INT I=0;I<10;I++), IF *type of* DECISION STATEMENT and DECISION STATEMENT *type of* CONTROL STRUCTURE), and there is a total of 11 different arcs between the two maps combined. This means that the similarity of the two maps is $3/10 = 0.3$ or 30%. Teacher maps can also be compared to student maps in the same way, by treating the teacher map as student map of itself.

Inference on Concept Maps

Because concept maps in CREEK-ILE are actually stored as submodels in the CREEK representation, it is possible to apply CREEKs inference to the concept maps. This may be interesting if it is possible to fairly accurately describe some of the semantics of the relation-types in for instance plausible inheritance rules. In the example of the two student maps from Figure 5.14, the relation-type TYPE OF is naturally interpreted as transitive – if a is a type of b, and b is a type of c, it seems fair to describe a as a type of c. These semantics seem implicit in the name of the relation-type to people, but obviously do not for computers. However, this particular property of the TYPE OF relation-type may be described as a plausible inheritance rule in CREEK. Applying this rule to the two maps infers the existence of an additional relationship (IF *type of*

CONTROL STRUCTURE) in the left map, and three additional relationships (IF *type of* CONTROL STRUCTURE, WHILE *type of* CONTROL STRUCTURE, FOR *type of* CONTROL STRUCTURE). If these relationships are included in the similarity measure of the graphs, we find that they now share 6 out of a total of 12 relationships, for a total similarity of 50%.

To our knowledge, inference has not been applied to concept maps in this manner before, and perhaps for good reason. As has been previously stated, concept maps are primarily a medium for human communication, and it is not a requirement that the computer should be able to understand the contents. However, in the CREEK-ILE use of concept maps, terms are defined in advance, and this also allows us to define some semantics for these terms. Second, the path-based inference system does not place difficult constraints on the user as for instance deductive systems often do. This means that this particular use of inference need not change the modelling task from the perspective of the user, but may allow the system to perform some reasoning over the contents.

5-3.4 Pedagogical Model

There is no explicit pedagogical model (yet) in CREEK-ILE. As in many of the other approaches we have reviewed, the pedagogical approach is implicit in the theory, and is not explicitly modeled. It is likely that pedagogical modeling becomes more relevant once the approach has been tested further and meaningful differences in strategies have been identified.

5-4 Tutoring Capabilities

5-4.1 Exercise Selection

It is possible to provide advice on exercise selection with criteria based purely on the record of what exercises and examples the student has experienced. For instance, the system could encourage the student to try exercises that deals with new topics, so that the total set of exercises covers all the topics in the curriculum. This would be analogous to providing good case coverage in case-based reasoning systems. In the CREEK-ILE representation of exercises, this can be done by suggesting exercises that introduce new features (concepts from the teacher map). Because the student model contains the set of exercises already solved by the student, and each problem has a set of features associated with it, it is fairly trivial to combine the set of features from all solved exercises in the student model. From this combined set the system can recommend exercises that introduce new concepts which have not yet been encountered. Variants of this approach are used by many intelligent tutoring systems, and may also be combined with pre-assigned difficulty ratings and other measures.

This approach can be viewed as a kind of knowledge transfer, where the system attempts to give knowledge to the student in the form of exercises. The assumption here is that any exercise that exposes the student to a concept will

leave the student with an accurate understanding of it. There is no feedback mechanism to allow the system to learn about the student's beliefs, beyond recognizing whether the student was able to solve the exercise or not. The PACT cognitive tutors and the ELM systems solves this by using a more fine-grained student model that attempts to model the procedural steps taken by the student in solving the problem. This makes it much easier to pin-point problem areas ("buggy rules"). By keeping the student model on the case level, where each case may cover several concepts, it is not easy to pin-point what the source of the problem is even if the system is able to detect that a student finds a particular exercise hard. For instance, if the student is unable to solve the `for`-to-`while` loop problem in Figure 5.12, is this because the student does not know the `for`-loop, the `while`-loop or a fundamental issue with the understanding of looping statements? It may also be that a student has solved an exercise successfully, but over-generalized or developed some misconception. This would be impossible to track on a purely episodic level of student modeling. A possible solution might be to expose the student to so many exercises covering the same area that one can reasonably hope that the correct lesson is learned. Presumably, this is idea behind the pre-printed sets of similar exercises in traditional school instruction.

In Section 5-2, we suggest that there is an interaction effect between conceptual, procedural and episodic knowledge. In particular, conceptual models may be used to get some information on how students generalize the knowledge gained from solving exercises, which can be very valuable in exercise selection. Rhere is a number of ways on how concept maps may be used in this regard.

Concept Maps as Gatekeepers. A simple and tempting method for using concept maps in exercise selection support, is to view the teacher's map as the correct solution, and ask the student to solve exercises covering areas of the student's map that does not yet match what the teacher modeled in that area. The assumption is that as the student increases in procedural capability, he will refine his conceptual model until it matches the teacher's, at which point the student is considered to have mastered the area. There are two major problems with this approach. First, it requires that the teacher's conceptualization is the only correct one. This seems unlikely. For instance, the teacher's map in Figure 5.1 says that the FOR and WHILE concepts *contain*s CONDITION. It would be entirely reasonable to generalize this property to the LOOP statement and it is not reasonable to expect a student to know the preference of the teacher here. This can possibly be alleviated to some degree by applying inference to the concept maps, but even then it is likely that conceptualizations that can be considered correct are recognized. The second problem is that using the concept map as a gatekeeper changes the goal from learning the subject matter to reproducing the model of the teacher. While learning the subject matter and the procedural skill associated with this may be a way of achieving this, it is unlikely to be the most efficient. If the conceptual knowledge serves as the gatekeeper, it is likely more efficient for the student to study conceptual knowledge exclusively, for instance by careful reading of a textbook. Obviously,

this will have a rather limited effect on the student's procedural skill. For this reason, we suggest that although concept maps may be used to evaluate conceptual skill in ways similar to multiple choice tests, they should not be used as gatekeepers in systems focusing on teaching procedural skill.

Conceptual Models' Relation to Procedural Skill. Related to the above issue is how well any conceptual model relates to procedural skill. As already mentioned, it is possible to acquire conceptual knowledge about procedural skill without practicing the skill. On the other hand both more traditional instruction by design approaches (for instance the ACT-R theory) and learning-by-doing approaches closer to constructivism (for instance Schank's theories), agree that learning conceptual knowledge is tied to learning procedural skill, at least for tasks that are cognitive in nature. The strategy in the PACT cognitive tutors is to teach conceptual knowledge and problem solving strategy knowledge before exercises are given, as the view is that these are required to solve the problems. Schank would suggest that this is exactly backwards, because the exercises provide the goals for acquiring this knowledge. In both cases, however, conceptual understanding is related to procedural skill, either as a prerequisite or an indication of interest in that area. To our knowledge, research on using concept maps as evaluation tools have focused on conceptual knowledge and has not addressed how they may correlate with procedural skill. In Chapter 7, we describe empirical experiments within the CREEK-ILE framework that examines this question. The idea here is that if concept maps created within the context of a problem-solving environment do not correlate with problem solving skill measures, they are likely difficult to use in exercise selection – although they may still be useful in conceptualization and explanation support.

Methods for using Concept Maps in Exercise Selection

Although student concept maps should not be used as gatekeepers, they may still be useful in exercise selection. If the system requires a test to see if the student has mastered an area, this can be in the form of a particularly hard task or set of tasks that must be solved. The system can then present itself as an accomplice to the student in learning the skill required to solve the gatekeeper task, and help him to assess his own knowledge to determine when he is ready to face it. This has the advantage of defining a "real" goal for the exercise session, providing motivation and goal for the student, and removing the incentive to game the tutoring system.

The two first methods described here are partially tested through empirical experiments described in Chapter 7, and the latter methods take on refinements of our approach taking into account the results of these experiments.

Method 1: Model First, Compare to Teacher. This method asks the student to form a concept map of the problem area before solving any problems. This map is then compared to the teacher's map. In its most basic form, a higher degree of similarity to the teacher map can be interpreted as a higher degree

of competence. If there is a strong link between conceptual and procedural skill, a student with a map that is very similar to the teacher's can be assumed to also be good at solving procedual problems. In Section 7-3, we test this hypothesis by measuring the correlation between the similarity of student maps to the teacher map and various measures of competence.

A more fine-grained approach is to assign a value representing the assumed level of competence for each concept to each student model. This can be calculated by comparing the similarity of the submodel-specific frame from the teacher's map to the submodel-specific frame from the student's map. The similarity of submodel-specific frames is defined in the same manner as the similarity of concept maps – by comparing the relative size of the union and intersection of the sets:

Definition 5-4.1 (Submodel-Specific Frame Function) *A function* $F_s : C \rightarrow \mathcal{P}(R)$ *is the* submodel-specific frame function *for the submodel* $s = (C_s, R_s)$ *of the CREEK Model* $\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$ *iff for all* $c \in C$*, the relationship* $r \in F_s(c)$ *iff*

$r = (c, t, c_v, x)$*, i.e. the origin of* r *is* c*, and*

$r \in R_s$*, i.e. the relationship* r *is a member of the submodel* s*.*

The similarity of two submodel-specific frames f' *and* f' *is defined by equation 5.2*

$$sim(f', f'') = \frac{|f' \cap f''|}{|f' \cup f''|} \tag{5.2}$$

For instance, the submodel-specific frame for the FOR concept would be different in the teacher's map in Figure 5.1 and the bottom student's map in Figure 5.14. If one imagines that the teacher's map here is stored in the submodel TEACHER and the student's in the submodel STUDENT, the similarity of the two concepts would be as calculated in Figure 5-4.1.

This method can be extended to use inference to extend the frames of both the student and teacher models by using inference methods like plausible inheritance to infer additional relationships for each concept in the submodel. As an example, we can use a similar variation to the plausible inheritance function which only considers entities and relations that are members of a given submodel. We call this function $I_{submodel-plausible}$. In Figure 5-4.1 we apply this inheritance to the FOR concept in the teacher map from Figure 5.1 and one of the student map in Figure 5.14.

The similarity between a concept as expressed by the teacher and the student may be interpreted as the system's belief in how well the student knows that concept. If the similarity is close to 100%, the student's view of that concept is similar to the teacher and can be assumed to be valid. Because the expert model representation of a problem case contains a list of features addressed by that problem, the system can prioritize those exercises that address concepts that have a low similarity score.

$F_{teacher}(for)$ = { (for,type of,Loop Statement,1),
(for,contains,Condition,1),
(for,correct example,for(int i=0;i<10;i++),1)
}

$F_{student}(for)$ = { (for,type of,Control Structure,1),
(for,correct example,for(int i=0;i<10;i++),1)
}

$sim(F_{student}(for), F_{teacher}(for)) = 1/3$

Figure 5.15: The similarity of the FOR concept, as modeled by teacher and student.

The plausible inheritance rule-set is $\delta = (typeof, typeof)$.

$I_{submodel-plausible}(for, teacher)$ =
{ ((for,type of,Loop Statement,1),
((for,type of,Loop Statement,1))),
((for,contains,Condition,1),
((for,contains,Condition,1))),
((for,correct example,for(int i=0;i¡10;i++)),
((for,correct example,for(int i=0;i¡10;i++)))),
(((for,type of,Control Structure,1))),
((for,type of,Loop Statement,1),
(Loop Statement,type of,Control Structure,1)))
}

$I_{submodel-plausible}(for, student)$ =
{ ((for,type of,Control Structure,1),
((for,type of,Control Structure,1))),
((for,correct example,for(int i=0;i¡10;i++)),
((for,correct example,for(int i=0;i¡10;i++))))
}

$sim(I_{submodel-plausible}(for, student), I_{submodel-plausible}(for, teacher)) = 2/4$

Figure 5.16: The similarity of the FOR concept, as modeled by teacher and student, using plausible inheritance.

This approach is conceptually similar to seeding a conceptual student model with information gained through a pre-test, which is a method that has been used for a long time in intelligent tutoring system. Using this approach in giving advice is less problematic than in evaluation, since the student may choose to ignore advice and overrule the exercise selections suggested by the system, but it is obviously still a problem if the system gives bad advice.

Another potential problem is that this method initially uses a purely conceptual model to assign beliefs about student competence in procedural skill, and this means that it is dependent on conceptual knowledge actually reflecting procedural skill. This may be partially alleviated by combining the approach with updates of the belief-value for each concept as exercises are solved, for instance by increasing the value of concepts that are used in successfully solved tasks and decreasing the values of concepts featured in exercises that the student failed to solve.

Method 2: Model First, Compare to Students Method 1 relies on explicitly modeled expert knowledge in the form of a teacher's map and a list of features attached to each exercise. However, it is also possible to use experience from students that have previously gone through the exercise set. Using machine learning methods, it may be possible to predict the difficulty of the different tasks. A simple way of doing this based on case-based reasoning would be to first ask each student to create a concept map, and then use this map to find students with similar maps that have already completed the set. The assumption is that students with similar maps would have similar competence in solving problems. For instance, a student with misconceptions about the `while`-loop, may be matched to a previous student with the same misconceptions that also had trouble solving `while`-tasks. Again, this is a controversial assumption and is unlikely to hold in all cases. For instance, some students may have little or no conceptual knowledge before starting the exercise set, and learn it during problem solving. Such students might form random or very sparse concept maps that say very little about their actual ability once they start solving problems. On the other hand, students that have studied in advance may form concept maps that are in line with what they have studied, and may as such be useful.

This approach has the advantage that it does not rely on the teacher's map as the only known correct solution. The system can retrieve different conceptual maps that indicate proficiency, instead of the single correct solution, and it can continuously learn more.

The machine learning approach can be used to predict how the student will do on exercises directly. For instance, a case-based reasoning method can find the most similar student that has already gone through the exercise by comparing the new student's concept map to the set of previous students' maps. If the system keeps track of how previous students do in solving procedural tasks, the system could simply adopt the previous student's procedural competence measures for the new student. For instance, if the previous student had trouble with two tasks, but solved the others well, the system may adopt the hypothesis

that this new student, too, will have trouble on those two tasks. This hypothesis – that it is possible to predict student competence based on earlier student's competence measures, is tested in Section 7-4.

As in method 1, it is possible to assign a value about the degree of knowledge on each concept. Exercise proficiency may be predicted by treating each student as an instance or case, using the concept maps as the features. Using the similarity measure described in Section 5-3.3, the students that have completed the exercise set with the most similar concept map can be identified, and the system can adopt the results of this student on the different problems as the default assumption for how the new student will perform. A problem here is that the former student may not have attempted to solve all exercises (if this is not required), and as such the system will have no information about these problems. It is also not clear how information about a student's assumed ability at different exercises can be used. If the system thinks that a student will have trouble with a certain exercise, should it avoid suggesting this task to the student? It might perhaps be better to suggest an easier problem in the same area, but this requires that the system has knowledge about difficulty and features of each exercise. If knowledge about features present in each exercise exists, it is also possible to use the most similar student to gain a measure of the assumed competence on each concept, as in the teacher's map method, but without using the teacher map as the only correct solution.

First, we define an exercise e as the set of concepts associated with it, and a student model s as the concept map of the student and the set of solved and unsuccessfully solved exercises:

Definition 5-4.2 (Exercise Set) *Given the teacher map $M_t = (V_t, R_t, L_t)$, an exercise $e \subseteq \mathcal{P}(V_t)$ contains the set of concepts associated with it. A set of exercises is denoted by E.*

Definition 5-4.3 (Student Model) *A student model is an ordered set $s = (M_s, E_s, E_u)$ for an exercise-set E, where*

M_s is the student concept map,

$E_s \subseteq E$ is the set of exercises solved by the student,

$E_u \subseteq E$ is the set of exercises unsuccessfully attempted by the student, and

E_u and E_s are disjunct.

By calculating the ratio of solved and unsolved exercises that address each concept, the system can calculate a normalized value that represents the system's belief in how well the student knows this concept.

Definition 5-4.4 (Relevant Exercise Function) *Given the set of exercises E and the set of teacher concepts V_t, the function $RE : (\mathcal{P}(E) \times V_t) \rightarrow \mathcal{P}(E)$ is the relevant exercise function if it identifies the subset of the input exercise set that contains the given concept, i.e iff for all $RE(E', c) \rightarrow E''$,*

$E'' \subseteq E'$,

for all $e \in E''$, $c \in e$,

there is no $e \in (E' \cap E'')$ *so that* $c \in e$.

Definition 5-4.5 (Assumed Conceptual Competence) *For the student* $s_r = (M_s, E_s, E_u)$*, the assumed conceptual competence* B *of the student* s_r *on the concept* c *is the ratio between the solved and the unsolved exercises using this concept:*

$$B(s_r, c) = \frac{|RE(E_s, c)| + |RE(E_u, c)|}{|RE(E_s, c)|} \tag{5.3}$$

Now, if a particular new student s_n creates a concept map, the previous student (s_r) with the most similar concept map may be retrieved, and the system's belief about the new system can be seeded with the values of the previous student.

A problem with this approach is that the student model does not represent a snapshot of the student's competence at a particular point in time, but aggregates over the entire set of exercises. It is probably not reasonable to adopt the conceptual competence values of a student that has already worked through the exercise set to the new student. The approach ignores that the student has learned during the exercise session, and that while the student was able to solve an exercise at the end of the session, he may not have been able to do it at the beginning. It is not reasonable to expect a new student to, at once, be able to solve the last exercise solved by a previous student, just because they initially formed similar concept maps.

Method 3: Model Repeatedly The two previous methods have two major problems in that they assume students had studied the topic of the exercise before it was given, and that a concept map formed before problem solving had begun would be an indicatiion of problem solving capability. Even before empirical tests, there were doubts about this assumption, and informal observation of students during the empirical tests seemed to confirm that many (although far from all) arrived at exercise session with little pre-existing knowledge on the topic. These students relied on looking up conceptual knowledge and examples when faced with a problem. This means that the initial concept map did not capture their conceptual knowledge at all. There are likely to be refinements of this knowledge during the exercise even for those students that had studied in advance. One way to capture this, is to ask the student to re-do or refine the concept map between each task. This is a particularly useful refinement of the strategy in method 2, as this allows us to break up each student's exercise session into multiple cases, so that a case is formed, for instance, for each concept map created. Each case would then contain the exercise immediately before and after the map was formed, as the map would presumably be fairly accurate in representing the student's conceptual knowledge at the time of solving these

exercises. Although each case would contain information on only two exercises, many more cases would exist as each student would produce as many cases as they produced concept maps. This would make it more reasonable to retrieve multiple cases and form a combined conceptual belief value.

From a student modeling viewpoint, there are strong arguments for this method over the previous two, but from a practical point of view, it is infeasible to ask the students to perform what is essentially the same task between each problem solving exercise. If they are given their own concept map from the previous iteration to refine, they may perhaps find something to improve the first few times, but if facing eight or ten problem solving tasks in an exercise set, connecting the same concepts with the same relation-types would soon become very repetitive, and not necessarily invite reflection. Although the idea is that students should incorporate new knowledge gained through solving exercises, it is unlikely to appear convincing to students.

This may perhaps be solved by triggering the student through challenging some part of their concept map. For instance, after solving an exercise involving the `for`-loop, the system may highlight the relationships from the FOR concept in the student's map and point out parts that may be wrong, or suggest changes. For instance, this same area of the teacher's map may be revealed, and the student asked to compare his own map to the teacher's in light of the exercise just solved. This is not technically difficult if the exercise is annotated with the concepts it contains, and these same concepts are used both in the teacher and student maps.

Method 4: Exercise-Specific Models, Explicit Generalization A more ambitious approach to the problem of concept model refinements is to attach a smaller, exercise-specific concept modeling task to each problem. This task could be presented as a tool to assist the student in identifying features of the presented problem, similar to how BLITS and CATO have feature identification steps in their problem solving. This could even be used as in BLITS and ELM to help the student to find relevant examples containing similar features, and as such it may provide a more immediate assistance to the student. A problem here is that it is not immediately clear how these maps can be useful in student modeling for exercise selection. Because these mapping tasks are attached to exercises, the exercise must be selected before the task has begun. Potentially, maps from earlier exercises may help the system in forming beliefs about the student's conceptual knowledge, but the case-based approach here is more limited because it can only match maps of the same type.

It might be possible to include an explicit generalization step after the problem solving session, where the student is asked to take parts of the exercise-specific concept map and generalize them into a kind of master map that is a higher level representation of his conceptual knowledge. At the present, it is not clear how the system may assist in such a task, but it would provide a concept map that would be more useful for exercise selection advice, and also provide explicit support for generalization from individual exercises.

5-4.2 In-Exercise Support

While the exercise selection task may be described on a generalized level, assistance within the context of the exercise tends to be more domain-specific, or tied to classes of domains such as diagnostics. The tutoring systems reviewed earlier revealed some shared elements of strategy, such as breaking down the goal of the exercise into sub-goals and in this way dividing the problem into smaller sub-problems. In BLITS, the system provided the structure and only offered support in the form of examples in solving the leaf-nodes of the goal-subgoal tree, while ELM and the PACT cognitive tutors could use their procedural knowledge to provide support in more detail.

In CREEK-ILE, there is no procedural model of how the student is assumed to solve the problem, and the conceptual model we have used have not contained information on problem-solving strategies or problem structuring. While it may be possible to use an approach such as concept maps to express goal-subgoal trees or similar structures, concept maps usually do not enforce constraints associated with such structures (such as a lack of cycles in goal-subgoal trees). This means that CREEK-ILE has neither the cognitive procedural model of ELM and the PACT cognitive tutors, nor the structural model used in BLITS. We do not discount these approaches, and structural model that contain the goal-subgoal hierarchy of the problem are likely good extensions to the approach. However, the different structural constraints of such models mean that it likely requires a different approach than how CREEK-ILE uses concept maps.

The question then remain – how can a system with no model on how to break down the problem, and no model to simulate the student's problem solving procedure assist in solving the exercise? The approach taken in CREEK-ILE is to include a limited ability by the system to solve the problem in whole or in part without regard to its cognitive plausibility. For instance, in the car diagnostics domain introduced earlier, the case-based reasoning engine in CREEK can be used to find the most likely cause of a problem. Usually, case-based reasoning employs a strategy of finding the n most similar earlier cases matching a new case, and then adapting or adopting the solutions from these cases as its own. This strategy has some cognitive plausibility when the number of earlier experiences is low, but as cases are accrued, people naturally generalize the individual experiences to generalized concepts and procedures. This step is not usually taken in modern case-based reasoning systems[2]. With this approach, CREEK-ILE is able to provide some support for the student in that it can provide whole or partial solutions, but since the mechanism for finding these solutions is likely very different from the student's, the reasoning trace of the system is unlikely to provide a good source of explanation as to how the (partial) solution was found. If the student himself fails to understand how the solution was found, the system is unable to back it up with explanations referring the problem-solving strategy (*relevance explanations*) or explanations that use the problem-solving trace of the system (*transparency explanations*). For some domains, CREEK

[2]In this regard, earlier systems based on Schank's Dynamic Memory did generalize individual episodes and formed conceptual structures.

is able to provide *justification explanations*. These are explanations that support the solution but is not grounded in the problem-solving strategy. The car failure diagnostics domain is such a domain where CREEK can provide justification explanations, while the computer programming domain is not. Below, we will describe how in-exercise support can be provided in CREEK-ILE for these domains.

Exercise Support in Computer Programming

The ELM and PACT LISP tutors both provide very good and detailed capabilities for basic computer programming. On the surface, it may seem like adopting this approach is a good solution for all computer programming-related tutoring systems. However, in addition to our expressed wish in not using a procedural model, there are several ways the tasks in our system differ from the ELM and PACT LISP exercises. First, these systems were created to support fairly small exercises. Usually, ELM and PACT LISP exercises are solved by a single function. Where possible, we have tried to adopt the exercises already in use in the programming courses, which tends to have fewer and larger (although not necessarily more difficult) tasks. Second, ELM and PACT LISP were constructed to support a limited type of problems. This is tied to the procedural model, which requires different set of rules to break down, e.g. recursion tasks than iterative tasks. CREEK-ILE has been tested with exercises that range from simple `if-then` problems to more advanced issues spanning multiple classes, such as constructing data-structures (e.g. linked lists). Third, the Java programming language has a syntactical structure that is harder to parse, and seems to offer the programmer more degrees of freedom in structuring the program, e.g in ordering expressions. The increased size of the programs increase the depth of the branching tree, the syntactical differences increase the number of possible branches at each level, and the increases scope means that to follow a model-tracing approach, the procedural model must be larger. These factors taken together, suggests that it may be infeasible, or at least very labor and computationally intensive, to use the model tracing approach.

At this time, we do not have a complete solution to provide in-exercise support for the programming domain. One possible solution that has been examined, is to continuously store all changes done to a program by a student, so that each student's program begins at a state s_1, which is either an empty program, or the program given as a starting point by CREEK-ILE. Each time a change is done to the program, the state is recorded, until the student has reached a state s_n that satisfies the tests for the program. A similar trace would be recorded of the teacher's proposed solution. This would in theory allow the system to help a student in trouble by matching this student's state s' to the set of states by earlier students, and if these are sufficiently similar to state s_m, present steps s_{m+1} to s_{m+k} as an aid to the stuck student. In educational settings, there are many students attempting to solve the exact same problem, so the chance that there exists a similar state is much greater than in "free-form" programming.

There are several challenges associated with this approach. First, a sensible representation of states must be identified. For instance, an ideal representation of a series of states may be going from a syntactically correct state to another syntactically correct state, with a minimal change, such as the addition of a single statement. This is of course not the situation in reality. Often, syntactical errors such as missing end-brackets signaling the end of a block persist all through the initial creation of the program, and are only identified when it is compiled.

Second, a similarity measure must be found that is not confused by trivialities, e.g. differences in variable, function or class names. This can be done by using a Java parser that interprets the raw text of the source code into a tree structure, where different types of elements are identified. This would allow the similarity measure to ignore names and such, and focus on the structural elements. It would also allow the formation of rules about which types of statements can be replaced by others. However, the only obvious similarity measures for two such program-trees are based on the edit-distance between them, and this suffers from the same computational problems as the model-tracing approach – the task branches exponentially.

It is possible that these challenges may be overcome and a solution close to the sketched above may be found. Similar strategies have been used in other tutoring systems. The JV^{2}M [38] system stores Java programs in tree structures that represents a subset of the Java syntax, and is able to measure similarity and even adapt these programs. However, all of their programs are complete and syntactically correct in advance (they are used as examples, as opposed to attempting to parse student's programs), and they are much smaller than the exercises in CREEK-ILE.

It may be interesting to note that even if the above solution was implemented and found to work well, it would still suffer from the inability to explain any suggestions. If a student is stuck in a state, the system would be able to suggest what to do next, but if this suggestion failed to communicate the idea behind the move, the system would be unable to explain *why*. It could only say that another student did this, and he was able to solve the exercise. For instance, if the task was to change a `for` loop to a `while` loop, the system may suggest that the student declares an integer variable before the loop, but cannot explain the reason for doing so – that the while loop does not contain a declaration of a counter, so this must be done in advanced if required by the loop.

In practical testing, it was found that the program tests were surprisingly popular, and could also be used as a kind of in-exercise support. For the computer programming domain, three kinds of tests were designed:

- Source tests. A regular expression that checked if the source code matched some kind of pattern. For instance, made sure the code contained a `while`-loop, if the task was to change a `for`-loop to a `while`-loop.

- Output tests. For programs that have no input from the user, test that the output produced by the program is correct. For instance, a program that

counts from 1 to 10 must have all these integers in the correct sequence. This is also expressed as a regular expression.

- Input Scenarios. Programs that require user input could have tests associated with them that included different sets of inputs and expected outputs for those inputs. For instance, a program that should count from a to b, where a and b is read from the user, can have these scenarios:
  a:5 – b:8 – output:5,6,7,8
  a:2 – b:12 – output:2,3,4,5,6,7,8,9,10,11,12
  a:-5 – b:0 – output:-5,-4,-3,-2,-1,0

Each test contains canned text associated with it which was displayed if the program failed the test, and the scenario tests typically go from simple tests to tests that has more pathological cases. In these tests, the scenario is also revealed to the student if the test fails.

Tests such as these can not by themselves be considered as par of an *intelligent* tutoring system. They consist solely of pre-defined structures that are applied without discretion or understanding by the system. This, however, does not make them any less useful, and they are also potentially useful indicators in student modeling. Although each exercise is associated with a fragment of the expert model, it may be possible to associate each test for that exercise again with a smaller subset. Depending on how a student's program performs on the different tasks, and how well the student is able to correct it afterwards, knowledge about the student can be gained.

Exercise Support in Car Failure Diagnostics

The car failure domain is created specifically as a demonstration domain for the CREEK case-based reasoning engine. This means that in contrast to the computer programming domain, CREEK-ILE can use the existing case-base reasoning engine to solve problems in the domain, and to provide explanations for these solutions. However, while the CREEK case-based reasoning engine uses both general domain knowledge in the form of an expert model as seen in Figure 5.1, and episodic knowledge in the form of cases, it does not claim to be an accurate cognitive model of human problem solving. The question is, then, how this capability can be useful in a tutoring context, where the goal is to help students develop problem solving skills.

To illustrate this and demonstrate how the explanation capability is useful in tutoring, we will demonstrate how the CREEK case-based reasoning engine solves a problem in the car-failure domain. Imagine that CREEK is asked to solve the problem of the car failure in Figure 5.13. Here we have a case with the findings ENGINE DOES NOT TURN, NO LIGHTS and STARTER ENGINE DOES NOT TURN. In CREEK, cases are stored as frames oriented around a concept representing the case, for instance CASE 3. The findings are associated with this concept through a set of *has finding* relationships from the concept representing the case to the concept representing the finding.

Conceptually, the CREEK principle of case-based reasoning is to form a submodel for each case that contains i) all the findings of that case, ii) all the relationships in the extended frame for all findings of the case, and iii) all relationships in all paths justifying the relationships in the finding's extended frames:

Figure 5.17 contains an example of two extended cases. An interesting property of these extended cases, is that they contain paths that connects symptoms and causes. For instance, the problem in case 1 in Figure 5.17 is that a weak battery caused the starter engine to turn slowly, which caused the engine to crank too slow to fire. In case 1, the root cause of the problem is not established, but both weak and empty battery can explain the observed findings.

The similarity of cases can then be calculated by measuring the degree of overlap between the two extended cases[3]. For instance, case 1 and case 3 from Figure 5.17 share no findings directly, but they do share parts of the extended case. This is illustrated in Figure 5.18. These cases are not very similar, but if the case base is small, one can imagine that CASE 1 in the most similar case for the new CASE 3. In this situation, CREEK adopts the solution of CASE 1 (WEAK BATTERY). In order to justify this solution, CREEK creates a new extended case which only contains those paths inferred by plausible inheritance that connects a finding with the proposed solution. In this situation, that would be:

NO LIGHTS *can be caused by* WEAK BATTERY,
STARTER MOTOR DOES NOT TURN *can be caused by* WEAK BATTERY,
ENGINE DOES NOT TURN *can be caused by* STARTER MOTOR DOES NOT TURN *can be caused by* WEAK BATTERY,

The question remains how this is useful in tutoring. First, CREEK-ILE can assist in identifying findings of the problem case. This can also be done for the computer programming domain, as findings are stored with the exercise. However, in the car failure domain, CREEK-ILE may also assist during problem solving, for instance by retrieving examples that are similar to this problem, if the student is stuck. This includes the ability to find cases that are not similar on surface, such as CASE 1 and CASE 3 above. However, this is dependent on an expert model that explains the similarity between the cases (by extending them). If the student does not have the same understanding of the domain as the expert, he may not see how the retrieved examples are similar to the current situation. A particularly interesting ability here is that CREEK-ILE may use the *students* concept map instead of the expert model to extend the case, and thus retrieve cases that the student would agree is similar to the current case.

[3]This is a conceptual view of the case-based reasoning matching in CREEK – the actual algorithm in implementation does not operate on extended cases directly, but compares the extended frames of the findings. This is necessary because feature weighting in CREEK is tied to the individual findings, and as such the similarity must be calculated independently for each finding. The algorithm also considers the strength of the inferred relationship so that inferred matches are weaker than direct matches. For more details on the actual implementation, see [3].

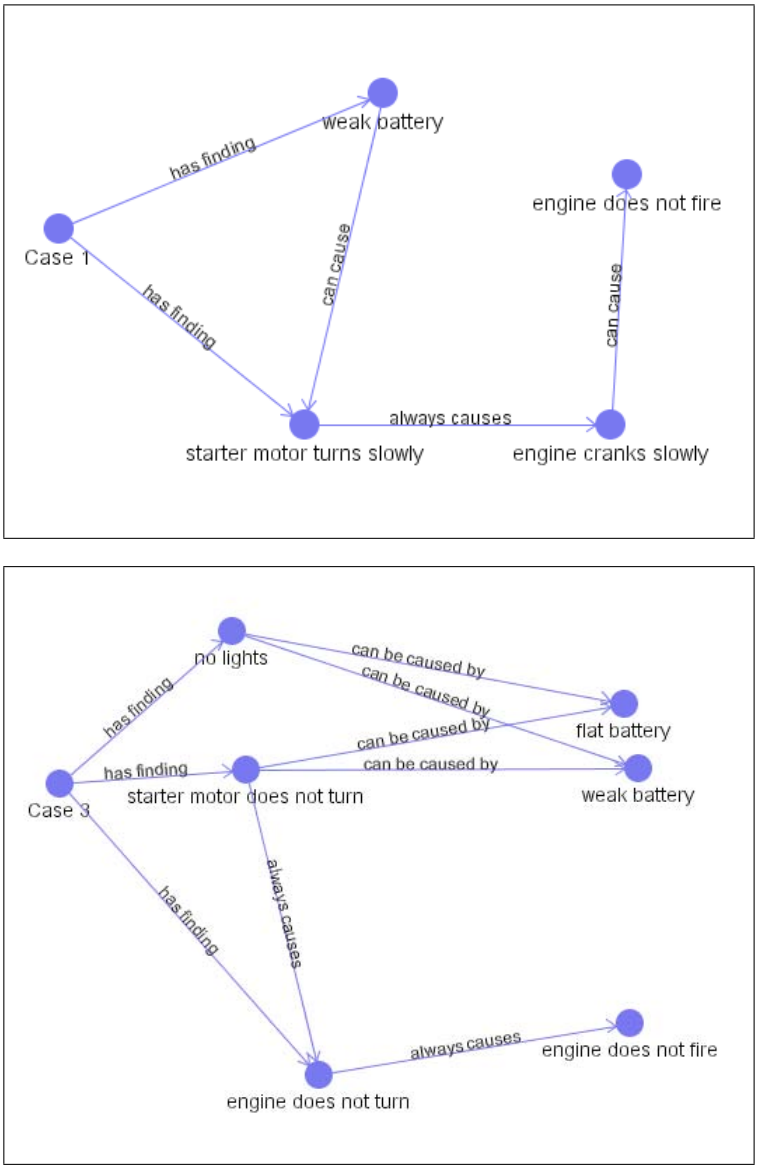


Figure 5.17: The extended cases for two car failure diagnostics cases.

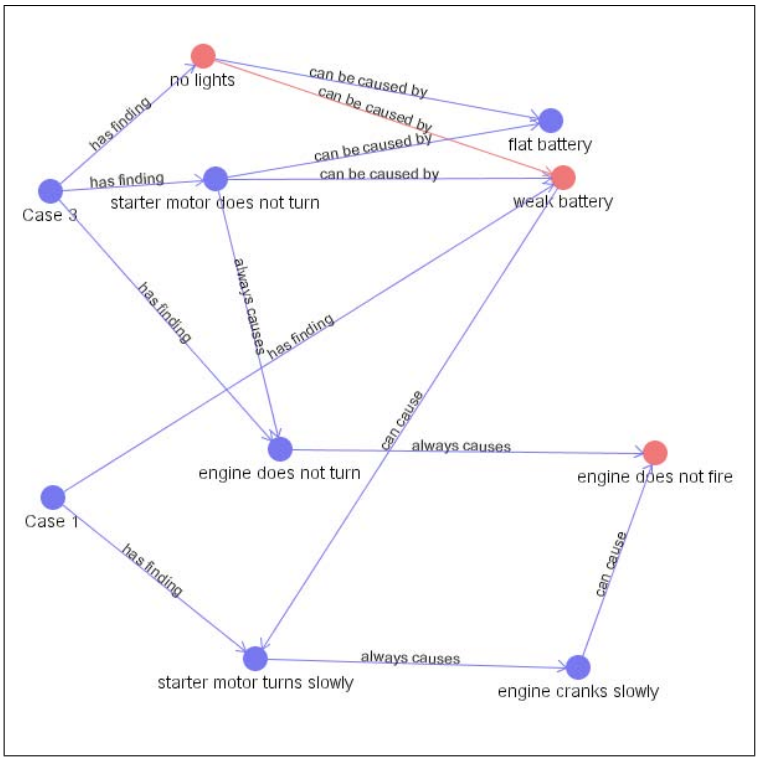


Figure 5.18: Comparison of the extended case for car case 1 and 3. The shared parts of the cases are marked in red.

This offers some unique opportunities:

- *Student's view* – suggest cases that are similar to the current problem according to the student's model, and offer explanations from the student's own model.
- *Teacher's view* – suggest cases that are similar to the current problem according to the teacher's model, and offer explanations for this from the teacher's model (or challenge the student to extend his own model).
- *Other student's view* – suggest cases that another student finds to be similar to the current problem, and offer explanations from this student's model (or challenge the student to extend his own model).

The major benefit here is to tie the conceptual model tightly with the episodic knowledge. An obvious problem is that while this is possible in the quite simple car failure diagnostic domain on the level described here, it is not feasible, for instance, in the computer programming domain. The reason for this is not only that the computer programming domain is not a diagnostic domain – if we imagine a computer program debugging domain, where the task is to find bugs in programs, it still is not possible to represent the problem completely as a set of findings, and connect these with root causes in a simple model. The simplicity of the representation language in concept maps limits us to domains where a limited, unordered set of symbolic features can characterize the problems, and where fairly small models are required to connect symptoms and end causes. Although many classical machine-learning domains may be in this space, we believe that few real-world domains for which tutoring support is relevant can be represented within these constraints. Because of this, we have not attempted to do any empirical tests of this approach, although as a proof of concept it is easy to demonstrate with the existing case-based reasoning capabilities of CREEK.

5-4.3 Conceptualization Support

A great strength in having explicit conceptual models is that it is possible for the system to provide assistance in the conceptualization process. A student may start off with fairly simple conceptualizations, and while we have argued that one should not view it as a goal to morph the student's map into the teacher's map, it is possible to challenge the contents of the student's concept maps. For instance, in the car failure domain, a student may be challenged by the system if he solves an exercise, correctly identifies the root cause, but his model fails to explain how the root cause follows from the symptoms. CREEK-ILE then has the opportunity to inform the student of this. CREEK-ILE can also identify the part of the teacher's map that connects these two concepts, and display that to the student if the student requires a hint. He can then choose to adopt the teacher's map, or model it differently on his own.

In the programming domain, the concept maps do not contain explanations of the same type as the car failure diagnostics, but it is still possible to provide

conceptualization support. The mere presence of the concept mapping task itself forces consideration on this issue, but more can also be done to challenge the student's conceptualization. If CREEK-ILE finds that the student's use of these concepts in his concept map is different from the teacher's, it can present a small part of the teacher's map, and ask the student to reflect on this in light of the exercise just solved. For instance, if the student with bottom concept map from Figure 5.14 solved the exercise in Figure 5.12, a question such as:

> In the previous exercise, you dealt with the `for`-statement. The teacher's thinks that "FOR *type of* LOOP STATEMENT", while your concept map has "FOR *type of* CONTROL STRUCTURE". Would you like to review your concept map?

Technically, this can be done by comparing the relationships from the concepts in the exercise in the teacher's and student's model, choosing one or a few relationships that differ on random, and fill in a pre-generated text form (as the one above). This can be visually represented, by cutting out the part of the student's and teacher's maps that contain these relations and displaying them.

This approach has two main advantages. First, it actively challenges the student to think about conceptualization. Second, it may solve the problem identified in Section 5-4.1, in that a student is likely to find repeated request to solve the same conceptualization task repetitive and boring. This kind of challenge after each exercise gives the concept mapping task a slightly different flavor, as the "challenges" are different each time, and related to the just-solved exercise.

5-4.4 Explanation Support

Explanation has been discussed in the previous sections in conjunction with the different tutoring capabilities. One persistent limitation of CREEK-ILEs explanation capability is that the lack of a cognitive procedural model makes transparency explanations (explanations based on the reasoning trace of the system) of little use. The closest CREEK-ILE gets to these explanations is the in-exercise support of simple domains such as the car failure diagnostic domains. Here, CREEK uses knowledge that is also used in the case-based reasoning process in producing explanations in support of the solution, but the reasoning leans heavier on the nearest neighbor approach of case-based reasoning than the explanations suggest. These explanations fulfill the justification role – they support the solution, but are only partly based on how the system found the solution. For instance, in Section 5-4.2, explanations are provided in support of the WEAK BATTERY solution for a particular case. However, CREEK could also supply equally convincing explanations to support the EMPTY BATTERY solution if that solution was identified by the case-based reasoning process. These were ignored because the result of the case-based reasoning process did not arrive at that conclusion. The role of these explanations, then, is not to impart a solid understanding of the reasoning process, but rather to *tell the story of the case* –

how the different findings in the case relate to each other, and concept implied by those findings. These explanations may help the student's conceptual model, but is unlikely to directly influence the procedural skill. This way of forming explanations are also common in human communication. Because procedural skill is often tacit, explanations are often formed after the solution is found and does not take account of the complete reasoning process.

CREEK-ILE also has very limited ability to provide strategic explanation, because of the lack of a model for how a problem can be broken down into sub-problems. This is likely both more serious and easier to rectify than the lack of procedural explanations. A separate conceptual level model for problem structure (such as the one used by BLITS) would likely provide a good knowledge source for such explanations, and would be very useful in assisting the student in structuring the problem solving.

5-4.5 Learning Capability

The methods that rely on comparisons to other students instead of the teacher's map, can be seen as a way of learning. For instance, methods 3 and 4 in Section 5-4.1, suggests comparing the current student's map to previous students, and use the most similar matches as predictions for how the student would proceed with different exercises, or what he knows about different concepts. As more students solve the exercise, the system will increase the case-based and hopefully achieve better modeling quality.

Both the approach to in-exercise support for the computer programming domain and the car failure diagnostics domain use case-based reasoning, including comparisons to earlier student's solutions. If implemented, these too would have the capability to learn as more cases is retained in the case base.

5-5 Evaluation

The partial implementation and evaluation of CREEK-ILE is discussed in Chapter 7.

5-6 Chapter Summary

In this chapter, we have presented the theory behind the CREEK-ILE system. The approach is based on a combation of case-based reasoning and concept maps to provide tutoring support in case-based tutoring, including exercise selection, explanation and conceptualization. There is also some suggestions on how case-based methods can be used to provide some in-exercise assistance. We have presented the CREEK knowledge representation, and a method for using it to represent concept maps. This knowledge representation can also be used to infer implicit relationships in concept maps, which may be useful in tutoring tasks. In the next chapter, we will examine implementation of a subset of this theory, with the goal of supporting an evaluation study described in Chapter 7.

Chapter 6

Implementation

The implementation of the CREEK Intelligent Learning Environment (CREEK-ILE) as presented in this work is not designed to work as a fully functional intelligent tutoring system, but as a basis for experiments. The system has been designed to work as a useful environment for students to solve exercises in Java programming, but does not contain any online capabilities for providing tutoring support. However, concept maps has been integrated as parts of the exercises, and data is collected through the students' use of the system so that data analysis can be performed offline to see, for instance, how knowledge expressed in concept maps correlates with how the students solve programming exercises. These tests are easier to perform than full blown comparative studies of students using the CREEK-ILE system and students using traditional exercise strategies. Such comparative studies are very good for evaluating the overall effectiveness of an approach once it has been well established, but it can often be hard to pinpoint what elements of the approach contribute to the end result. Before this step is taken, we felt that it was necessary to develop strategies and evaluate how cognitive mapping could be usefully employed in an exercise environment. Indeed, the experiments below have resulted in revisions to our methodology.

The Java programming domain was chosen because earlier exercise support systems (such as ELM and the PACT Lisp Tutor described in Chapter 4) operates in programming domains, which makes it easier to compare the approaches. In addition, practical concerns make programming classes attractive, as we have ready access both to enough students to perform empirical tests, and domain experts. In the previous chapter, we discussed how it in Java programming is hard to give in-exercise assistance to particular questions that may come up while solving an exercise, at least without modeling procedural knowledge to the same level of detail as ELM and the PACT Lisp Tutor. Although we suggested some means to provide some level of assistance there, the focus of our implementation and experiments have been on how cognitive maps can be used as part of the student model. In particular, the ACT theories and classic instructional design principles suggest that conceptual knowledge is required before solving exercises, while approaches leaning more on learning-by-doing suggest

that conceptual knowledge is formed by the student as problems are solved. The instructional design approaches on the other hand, suggest that concept maps can be used as a kind of pre-test to show which exercises the student has the prior knowledge required to solve. This has important implications for using concept maps in student modeling.

In addition to the exercise environment presented to the students, the CREEK-ILE system is based on the CREEK knowledge representation and case-based reasoning engine, with extensions developed for this work. The CREEK knowledge representation is used in the CREEK-ILE exercise environment whenever the student forms concept maps, but the exercise environment currently do not do on-line case-based reasoning on this data. The reason for this is not primarily technological – the off-line tests performed are efficient enough that to adapt them to online use is unlikely to pose a problem. However, the datasets collected allowed us to test and compare the effectiveness of different machine-learning methods and methodologies.

6-1 Exercise Environment

The CREEK-ILE exercise environment consists of a client program written in Java that may in principle be installed on any computer supporting the Java Virtual Machine. In our first experiment, students were allowed to download and install the client on their own computers, as well as use it on the university computer labs, although the client must have internet access as it communicates with a server when active. The server module is used to store and track the state of the exercise, including the students' concept maps, program source code, et cetra. The server also tracks how long the student works on each exercise and to a certain extent the development of the source code for each programming exercise, by saving the state at regular intervals.

After an initial login screen (see Figure 6.1), the student is presented with a screen listing the tasks of the exercise on the left-hand side, and a pane dedicated to the currently chosen task on the right. In the exercise environment, we call these tasks panes *pages*. These pages can be navigated from beginning to end using buttons on each page, but the student may also jump forward to a later task.

There are currently four types of task pages:

- *The information page.* A webpage with general information or an introduction to the exercise.
- *The concept mapping page.* Asks the student to form a concept map over a particular topic.
- *The programming task page.* Gives the student a Java programming task. This page contains a simple program development environment.
- *The question task page.* Asks a question of the student that must be answered in a text box. In this context, this usually means that the

Figure 6.1: The CREEK-ILE login screen.

student is asked to interpret what a given Java program would print to screen.

An exercise may contain any number of these pages, in any order, as determined by the teacher or expert when designing the exercises. However, in practice the exercises have typically been similarly structured in the experiments so far. After the initial login screen, the first page has been a general information page about the exercise environment, followed by a concept mapping page for the topic covered by the exercise. After this, a series of programming and question pages have followed.

The examples in the figures below are screenshots from exercises used in the experiments, except that user interface and exercises have been translated from Norwegian to English.

6-1.1 Information Page

The information page is the simplest form of page. It does not contain any interactive elements, but is simply a web page with some kind of information. The other pages also include smaller information pages to describe the particular task they cover, but occasionally it is useful to have a separate page for things such as general help or reference information. Figure 6.2 shows the initial information page used to inform students about the exercise environment.

6-1.2 Concept Mapping Page

The concept mapping page consists of a central area for drawing the concept map, and lists of relation labels and concepts available on the right-hand side. The student can drag and drop concepts from the concept list to the central drawing area. Concepts may only be used once, so those that are already in use are shown with a grey background in the list. The concepts may be placed or moved by holding down the left mouse button while the cursor is over a blue dot, and dragging it. Relations are drawn by holding down the left mouse button while move the mouse cursor from one blue dot to another. At any given time, relations drawn in this way are given the label of the relation label marked with a pink background in the list to the right. Concepts and relations may be deleted by clicking on them, which marks them in pink, and then clicking the *Delete* button. Deleted concepts are returned to the list and may be used again later, while deleted relations may be re-drawn if the student wishes. An example of the use of the concept mapping page can be seen in Figure 6.3.

On the top, a small information page area explains the task to the student. There is also a *Help* button if the student requires further assistance in using the tool. When the student feel his concept map is done, he may click the *Next*-button to go to the next task. This freezes the concept map so that the student may no longer edit it, but he may go back and refer to it at any time by clicking on the page name in the list to the left. The freezing of the concept map is done primarily for methodological reasons in order to make sure that all

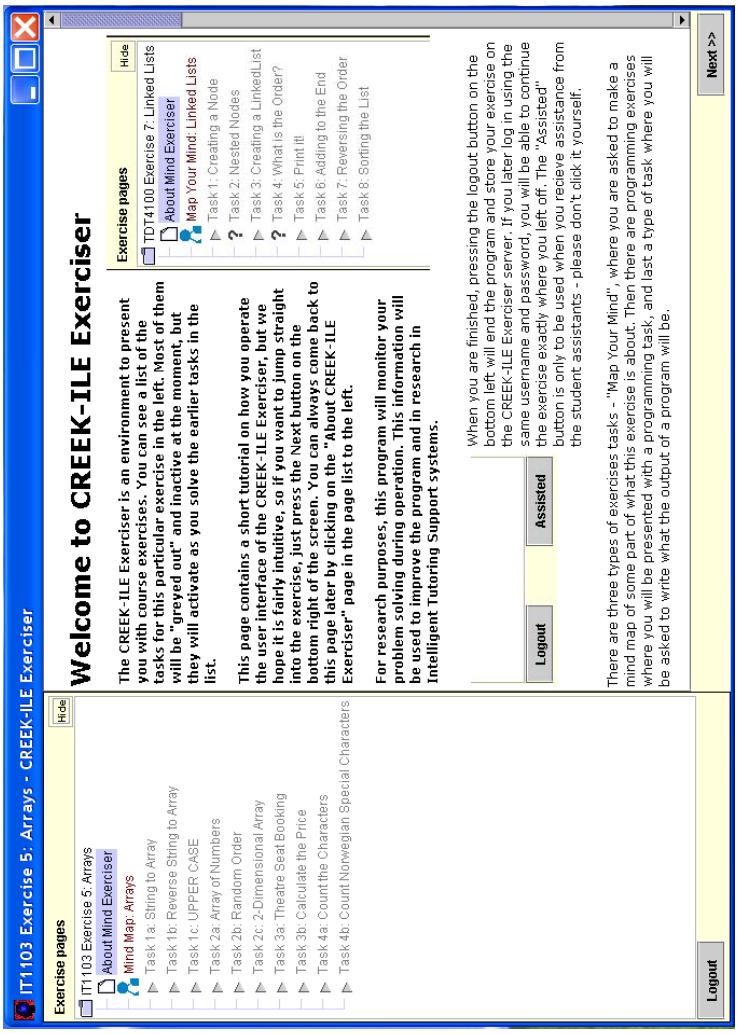
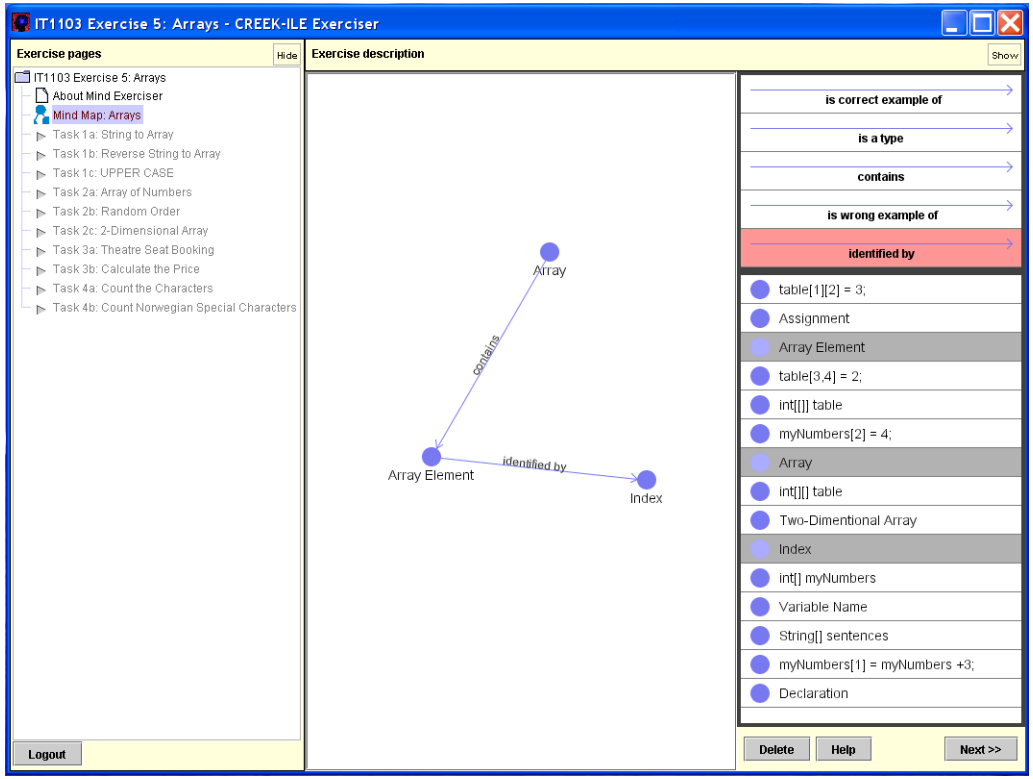


Figure 6.2: An information page giving an introduction to the CREEK-ILE Exercise environment.



In this concept map, the student has hidden the information page on top to get more space in the central concept mapping area, and has dragged the ARRAY, ARRAY ELEMENT and INDEX concepts onto the mapping area. He has then connected them using *contains* and *identified by* relations. The currently chosen relation name is *identified by*, and is identified by the pink background color.

Figure 6.3: A concept mapping page in CREEK-ILE.

the student finish the concept map before working on the programming tasks and is not a technological constraint.

6-1.3 Programming Page

The programming page is a minimal Java development environment with a syntax-highlighting text editor[1]. The editor may contain several files, which initially may be empty or contain code that should be extended or modified by the student. In Figure 6.4, there are two files, only one of which may be edited by the student. The student may not create or remove files, and may not change which file the systems has designated the main program (i.e. the class to execute).

The student may compile the program by clicking the *Compile* button, and run it by clicking *Run*. Error messages and the output of the program are placed in a text box below the text editor, which may be minimized when not in use (it is minimized in Figure 6.4, but visible in Figure 6.5).

In addition to the standard *Compile* and *Run* buttons, the student must test his program against the goals of the task before CREEK-ILE accepts the task as solved. When clicking on the *Test*-button, CREEK-ILE uses a set of automated tests specified for the task, as described in Section 5-4.2.

These tests may be:

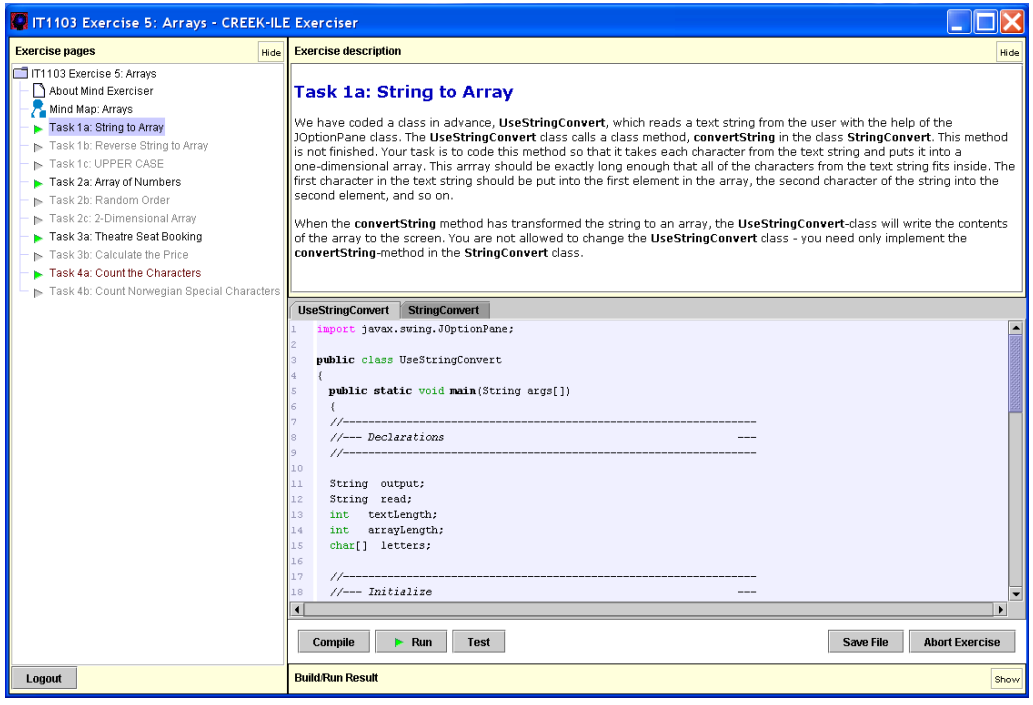
- Source tests. Attempts to match the source code the student has produced against a regular expression pattern. If this fails, the test fails and the student is given an error message. We may for instance test if the student has used a for loop by using a source test like: `.*for(.*).*`.

- Output tests. Similar to the source code test, but tests a regular expression against the output of the program. This is mostly useful if the student is asked to write a simple program that does not require any input from the user. For instance, if the student is asked to write a program that counts from 1 to 10, an output test pattern can look like this: `.*1.*2.*3.*4.*5.*6.*7.*8.*9.*10.*` . If it fails to match the pattern, an error message is given.

- Input Scenarios. This is an advanced output test that also provides input scenarios, which means that the CREEK-ILE system is in the role of the user providing input for the program and then comparing the output of the program with what is expected. In Figure 6.5, such a test is used to identify that the student has not successfully solved the task yet.

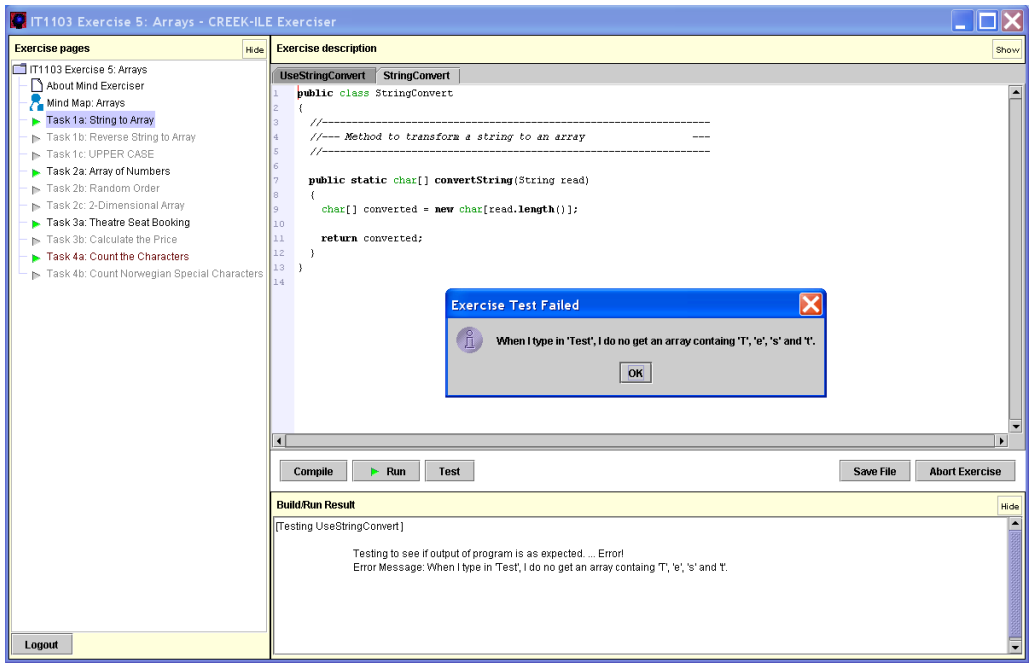
These tests are not "bullet proof" in the sense that they guarantee that the student has solved the task properly. The most obvious ways of cheating are removed – the source code is stripped for comments before a source test is applied, for instance, but it is seldom possible to design tests that do not contain

[1] The text editor used in CREEK-ILE is a slightly modified version of jEdit, an open source Java editor available from http://www.jedit.org/.



In this task, the student is asked to implement a method that converts a string to an array of characters. The task is described in a small information page on the top of the page (which may be hidden when read). Below this information area is a text editor with two files – UseStringConvert.java and StringConvert.java. Only the StringConvert.java file may be edited by the student in this task.

Figure 6.4: A programming page for a simple *array* task.



In this screenshot, the student has created a syntactically correct program that compiles and runs, but it only initializes an array of the same length as the string – it does not copy the contents like the task requires. The test applied here simulates a user entering 'Test' and then compares the output of the program with a regular expression (in this test, the regular expression `.*T.*e.*s.*t`).

Figure 6.5: Testing a solution in the programming page.

loopholes. The source test example given above, for instance, would match successfully against a program for instance containing the method invocation `isfor()` even if it did not contain a for loop. More stringent expressions are certainly possible, but as they become more specific, the chance increases that there is some legitimate way of solving the task that fails the test. Similarly, the input scenario tests can all be "solved" by creating special cases for them, as the input and expected output are given to the student whenever a test is failed. We do not view this as problematic because the tests are primarily designed to be a self-evaluation aid to the student, and not meant to play a serious role, for instance in grading. The tests can also be useful in illustrating difficult special cases the program is expected to handle. For instance, can the program handle a situation where the user enters a string when a number is expected? What if the number is larger than what can be represented in a standard 32 bit integer? By designing input scenario tests for such hard cases, students are forced to think about situations they may otherwise not think about.

When all tests are successful, the student is congratulated, the task is marked as successfully solved and the student is asked if he would like to go on to the next task.

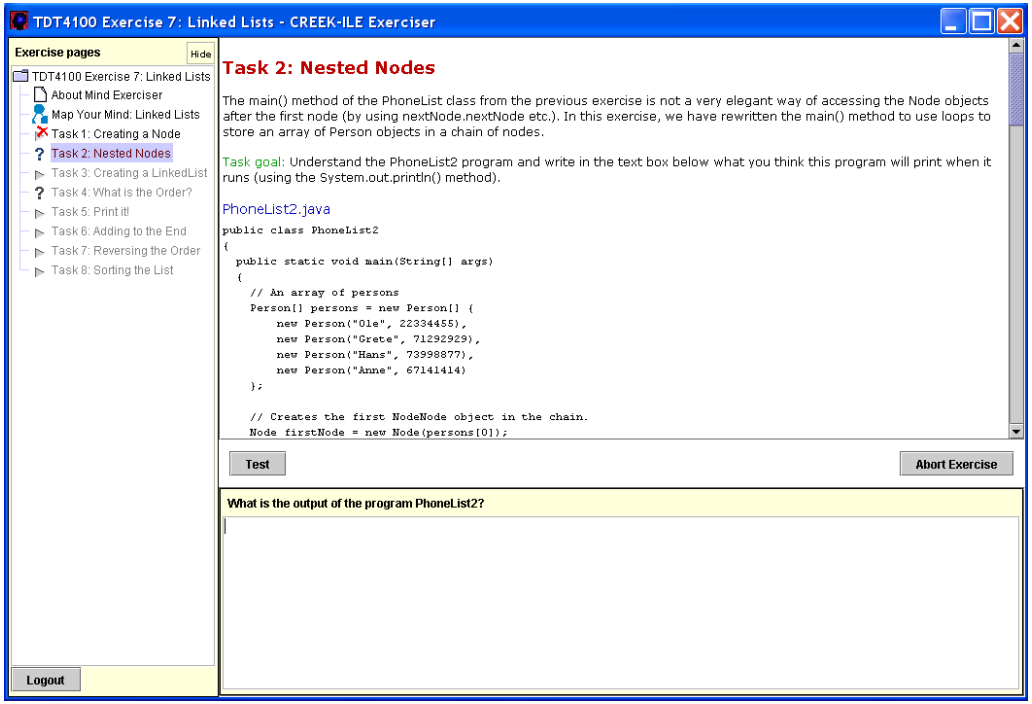
If all else fails, the student may give up a programming task by clicking the "Abort Exercise" button. This takes him to the next task. If this task builds on the previous task, the student will be presented with a working solution to the previous task so that the student need not give up this task as well. This solution is also always available to students that successfully solved the previous task, but by default their own solution from the previous task is open in the editor.

The automated tests also have a methodological advantage, in that they are entirely mechanical and judge all students on exactly the same criteria. This means that the subjectivity of teaching assistants or teachers in deciding if a task is solved or not, is avoided. It is possible for the student to cheat these tests on some of the tasks, but we have not observed obvious attempts at this in the experiments so far.

6-1.4 Question pages

The question page is simply an information page posing some kind of questions to the student, which must be answered by typing text in a text area below (see Figure 6.6). The student's answer is matched to a regular expression to see if it is correct. This means that this test is currently only useful when the answer can be identified in this way. The question page has specifically been created for the programming domain, and is used to ask the students to evaluate the output of a program. The program source is contained in the information page, and the student must then type the output of the program into the text area.

This particular task has been used in early evaluations of the user interface, but not in the data collection for the concept map experiments.



The information page on top of the screen asks the student to interpret the output of a Java program, and type it in the text box below.

Figure 6.6: A CREEK-ILE question page.

6-2 Designing an Exercise

The CREEK-ILE system in its current incarnation is designed to support exercises that are fairly close to the exercises already given through traditional means in our computer programming classes. There are several reasons for this. First, it is practical in the sense that it allows greater reuse of existing exercises. Second, it is advantageous to keep as many factors constant as possible if we'd like to do comparative studies between CREEK-ILE and the traditional non-computer aided method. Last, it has helped in building confidence in the course teachers that these alternative methods will not leave the students that participate with a lower quality educational experience.

In practical terms, this means that the exercises currently offered through CREEK-ILE has had fewer but larger tasks, and the tasks are often tied together, so that a subsequent task builds on and extends code developed in an earlier task. Often, students can spend many hours on the same program source. These few, large programs contrast with the more numerous but smaller programs typically seen in the ELM and PACT Lisp Tutor systems. The advantages of these longer programs are that they tend to be closer to real-world applications, and often reuse previously taught course material. This allows the exercise to be closer to constructivist ideals of real-world relevance and teaching the complexity in combining different elements from the course material.

For instance, in the exercise on arrays in Java programming, the last set of tasks asks the student to create a theater booking system. The theatre seats are represented as a two-dimensional array, so it is related to arrays, but it also requires file access operations, integer calculation and interaction with the user. It is obviously a task numerous real-world compute systems are designed for, and as such relevant. The disadvantage of longer programs is that it is hard to use methods such as model-tracing, as the complexity of the task increases exponentially with the length of the program, as the number of possible branches multiplies at each step. These tasks also do not invite customization to the individual student. When they are designed to follow each other, rearranging and omitting some of them are not really possible. This is not really a practical problem for the experiments in this work – our goals has been to see how conceptual knowledge expressed in concept maps relate to procedural skill and episodic knowledge. It does however question our motivation for studying this question. We are interested in using the concept maps in the student model specifically for supporting task order customization. If student-specific task ordering is not beneficial, this places in doubt the benefit of concept maps in student models.

These exercise sets often havea a few fairly large tasks that may be broken down into subtasks. Typically, a student may be asked to create a simple program first, for instance a program that asks the user for a number, and then use a loop to count from 1 up to that number. The second task then often builds on this program and asks the student to extend or change the program in some way, for instance by asking the user for two numbers and then writing a program that counts from the lower number to the higher number. This means that the result of the previous task must be carried over to the next, and that

it makes little sense to jump ahead before the first task is solved.

The teacher specifies an exercise by creating a specially formatted text file, which specify general information about the exercise (such as the name of the exercise), and the number, ordering and type of pages it contains. For each page, a set of properties is specified. A partial exercise definition file is found in Figure 6.7. This file in this figure defines the pages displayed in figures 6.2 through Figure 6.5. Some additional resources are referred through URLs in the definition file. The information texts are stored as regular HTML-web pages, while the teacher cognitive maps are CREEK .km files. Java source files are plain text files.

The definition files are currently created by us, using the traditional exercises as a basis. It is certainly possible for a teacher to design by herself by hand, if some effort is spent in learning the formats, but with the limited trials so far this has not been pursued.

6-3 Knowledge Representation

The concepts maps created in the CREEK-ILE Exercise environment is based on a teacher map, represented in the CREEK Knowledge Representation. When creating a concept map page, the teacher must first use the TrollCREEK Knowledge Editor to create a teacher's map (see Figure 6.8). This editor is similar in kind to the tool the student uses when creating concept maps, but it is designed to manipulate the full range of knowledge representation capabilities, including creating new concepts and relation-types. In order to make a teacher's map, the teacher must create a new submodel (called *Map View* in the editor), and build the model there. Once finished, a name is given to the submodel, and the model is saved as a *.km* file. This file is what is referenced in the exercise definition file. For instance, in Figure 6.7, `map1.model` refers to the CREEK knowledge model that contains the teacher map for this concept map page, and `map1.partition` refers to the name of the submodel that contains the teacher's map. The concept map page then extracts the list of concepts and relation-types used in that submodel, and these are then placed in the list of concepts and relation-types available to the student. Once the student has finished his concept map, that concept map is also stored as a submodel (with the name specified in `map1.solutionParititionName`). The model is then saved, and sent to the server along with the other data. Currently, this means that there is a separate knowledge model for each student and for each concept map page, but technically, they are not hard to combine to a common knowledge model that contains all the student maps in separate submodels as well as the teacher's.

The implementation of the CREEK Knowledge Representation is based on the formalization from Chapter 5, but uses the Graph Model approach where identifying the relation-type (T) subset of the concepts (C) is done by checking if the concept is a subclass of the special RELATION concept, and the inverse type (λ) function is done by following *has inverse* relationships between concepts. There are also semantics attached to various other concepts in the model. All

```
## CREEK-ILE Exerciser Property File

name=IT1103 Exercise 5: Arrays
pages=about,map1,task1a,task1b,task1c,task2a,task2b,task2c,
task3a,task3b,task4a,task4b
finalPages=task1c,task2c,task3b,task4b

mindMapHelp.URL=http://www.idi.ntnu.no/~frodeso/
me/it1103/Hjelp_Tankekart.html

about.name=About Mind Exerciser
about.type=webpage
about.URL= http://www.idi.ntnu.no/~frodeso/me/
it1103/About_Exerciser.html
about.activates=map1

map1.name=Mind Map: Arrays
map1.type=concept map
map1.description=http://www.idi.ntnu.no/~frodeso/me/
it1103/Oving5/map1.html
map1.model= http://www.idi.ntnu.no/~frodeso/me/
it1103/Oving5/
OriginalOving5-english.km
map1.partition=it1103-oving5
map1.solutionPartitionName=it1103-oving5-solution
map1.active=yes
map1.activates=task3a,task4a,task2a,task1a

task1a.name=Task 1a: String to Array
task1a.type=programming exercise
task1a.description=file:///C:/projects/
CreekITS-it1103/Oving5/task1a-eng.html
task1a.files=UseStringConvert,StringConvert
task1a.UseStringConvert.url= http://www.idi.ntnu.no/~frodeso/me/
it1103/Oving5/task1a/UseStringConvert.java
task1a.UseStringConvert.editable=false
task1a.StringConvert.url=file:///C:/projects/CreekITS-it1103/
Oving5/task1a/StringConvert.java
task1a.StringConvert.editable=true
task1a.tests=s1
task1a.test.s1.type=inputScenario
task1a.test.s1.inputScenario=Test
task1a.test.s1.outputPattern=.*0:T.*1:e.*2:s.*3:t.*
task1a.test.s1.outputFailed=When I type in 'Test',
I do no get an array containg 'T', 'e', 's' and 't'.
task1a.active=no
task1a.activates=task1b
```

Figure 6.7: A partial CREEK-ILE exercise definition file.

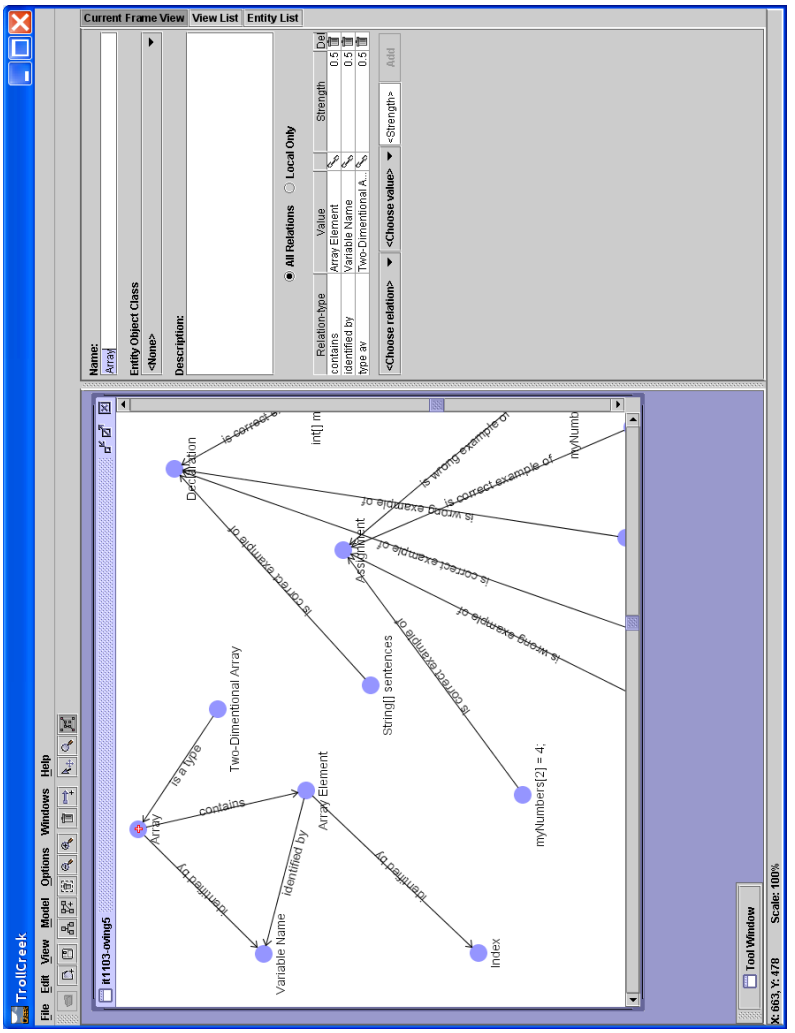


Figure 6.8: The TrollCREEK Knowledge Editor.

In this screenshot, the teacher is creating the teacher's map used for the concept map page in Figure 6.3.

instances of CASE is assumed to be cases by the reasoning engine, and there are also different similarity measures associated with the concepts SYMBOL, NUMBER, STRING, and so on. This means that when creating a new model in CREEK, these concepts must already be present. In Figure 6.9, the *Top Level Model* view shows a part of the model that is created[2].

The knowledge representation is implemented in Java using a set of classes in the `jcreek.representation`and `jcreek.representation.cbr` packages. The implementation of the CREEK knowledge representation predates this work, but has been further developed, restructured and extended as part of this work. The major classes of these packages are:

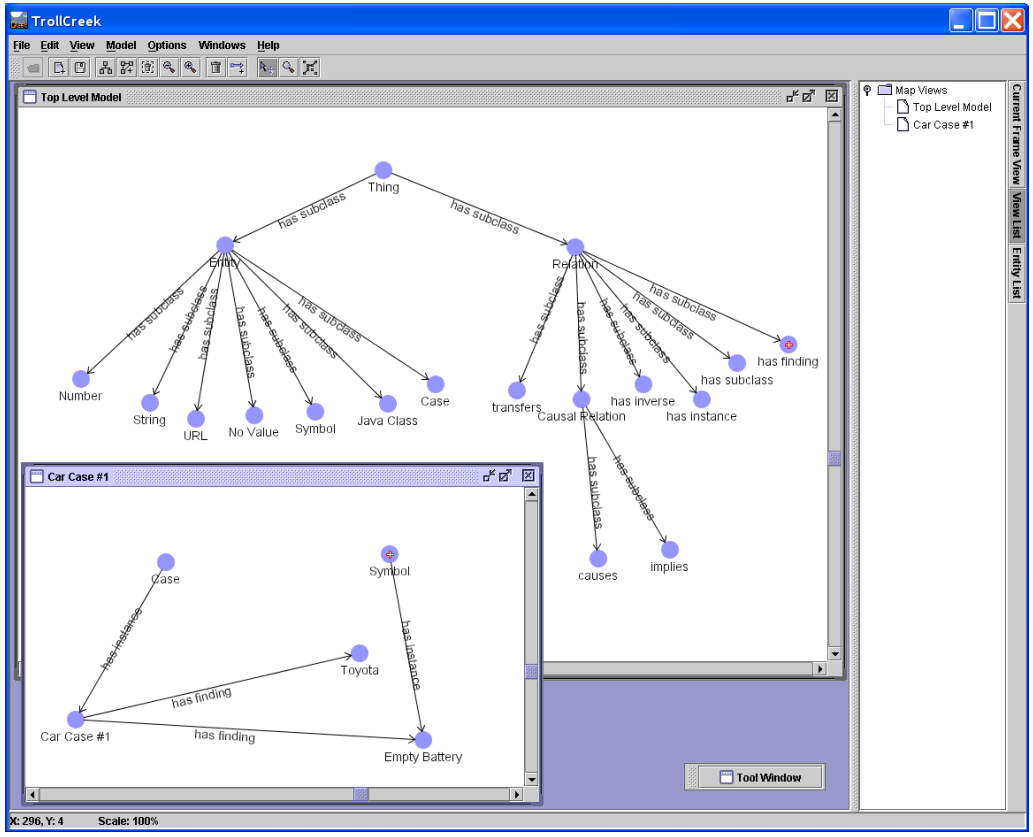
KnowledgeModel. The `KnowledgeModel` interface represents a complete model. It is the implementation equivalent of the CREEK model ($\eta = (C, R, T, \lambda, \psi, \gg, S, \rho)$) in Section 5-3.1. This is actually an interface in order to allow different implementations to store and access the underlying model in different ways. This interface contains methods for accessing the concepts (`Entity` instances), relationships (`Relation` instances), submodels (`Partition` instances) and particular types of concepts, such as cases and relation-types.

LocalKnowledgeModel. The standard implementation of the KnowledgeModel interface, which keeps the model in memory and saves the model to disk in a binary format when asked.

Entity. For historical reason, the class representing the concept (a member of C) is called `Entity`. Any concept in the model can be accessed as an `Entity` object, but the class does not contain persistent data – it is merely an interface to access it. Access methods include getting and setting the name of the concept and accessing and adding relationships to the concept. There is a choice in getting only the local relationships, or also including relationships found through an inheritance method. In this implementation, every concept may also be associated with a single Java object, which allows concept nodes to encapsulate images, URLs, numbers and even references to Java classes. These are called encapsulated entity objects.

Relation. An instance of the `Relation` class represents an individual relationship in the model (a member of R). Each relationship has an origin `Entity` instance, a value `Entity`, a relation-type represented by an instance of the `RelationType` class and a strength value from 0 to 1. Because each relationship in a CREEK model has an inverse, the `Relation` instance also has a reference to its inverse.

[2]This is part of a minimal model that contains the most necessary concepts to do case-based reasoning. There is ongoing work to define an ontology for top level models.



The two inner windows represent two different submodels in the same CREEK model. Notice that the CASE and SYMBOL concepts are used in both submodels. They are the same concepts in the underlying model.

Figure 6.9: Submodels and Top-Level Model in TrollCREEK.

Partition. The `Partition` class represents a submodel in the CREEK model (a member of the S set). Each submodel has a name, and a list of concepts (represented by `Entity` instances) and relationships (`Relation` instances) it contains. These may be accessed and changed through an instance of the class.

EntityType. The `EntityType` class is an abstract subclass of `Entity` that allows subclasses to specify criteria a concept must match in order to be represented by that class. For instance, the `Case` class is an `EntityType` that specifies that in order for a concept to be represented by it, the concept must be a direct or indirect subclass of the CASE concept in the model. These concepts could also be accessed through an `Entity` instance, but the more specific class contains methods that are useful when we know what type of concept it is. For instance, the `Case` class has methods to access all findings. This does mean that the same concept can be represented by multiple instances (e.g. both by an `Entity` and a `Case`), but since the data is not stored in the class itself, this is not a problem, and changes made in one instance is instantly updated in the other.

RelationType. The `RelationType` class is a subclass of `EntityType`, which requires that the concepts represented by it must be a direct or indirect subclass of the RELATION concept. This means that the `RelationType` class represents the relation-types (T) in the CREEK Model, as all the relation-types are represented by concepts. In addition to the normal `Entity` methods, this class contain methods for accessing and changing the inverse of the relation-type. The inverse of the relation-type cannot be stored directly in the `RelationType` object as subclasses of `EntityType` may not store persistent information directly. However, a pointer to the inverse relation-type is stored in the model by *has inverse* relationships. A similar technique is used to store a default strength for the relation-type. Using a `RelationType` object, the default strength can be set and read through normal Java methods, but is stored in the underlying model through a *has default explanation strength* relationship. This means the default strength can be changed by accessing this relationship through an `Entity` object as well, but the `RelationType` object makes it easier to do from Java code, and also makes it apparent what kind of concept this is.

Case. An `EntityType` subclass that represents a Case. In order for a concept to be used as a case (and be a `Case` instance), the concept must be a direct or indirect subclass of the CASE concept. This class provides convenience methods for accessing and changing a cases solution and status (although this information is stored in the model through special relationships, as with the `RelationType` class). In order for a concept to be submitted the case-based reasoning process, it must be represented as a `Case` instance.

NumberEntity. An `EntityType` subclass that represents a concept encapsulating a number. This means that in order to be represented by a `Number`,

the concept must have an encapsulated entity object that is a subclass of the `java.lang.Number` class. This includes all object-encapsulated numeric data types in the Java language (both integers and floating points values). The concept must also be a direct or indirect instance of the NUMBER concept. This class has convenience methods for accessing and setting the numbers directly.

The packages also contain other classes, among them `EntityType` subclasses for representing internet addresses in the form of Uniform Resource Locators (`URLEntity`), strings (`StringEntity`), specific types of numbers, (`DoubleEntity`) and so on. In addition there are classes that define various levels of top-level models. The `BasicModel` class creates a new model with only the bare minimum to support the representation, while the `CBRModel` also contain concepts such as CASE and associates specific similarity measures with concepts such as SYMBOL, NUMBER and PARTITION.

The inference mechanisms are implemented in a separate package (`jcreek.representation.inference`). Currently, the available inference methods are represented by the `SubclassInheritanceMethod` class, which implements the CREEK default-aware subclass inheritance method defined as $I_{da-subclass}$ in Section 5-3.1, and the CREEK default-aware plausible inheritance method defined as $I_{da-plausible}$ in Section 5-3.1. Appendix A contains an example of a Java program creating a small CREEK model. For more details on algorithms and details for implementing these, see [76].

6-4 Case-Based Reasoning

The case-based reasoning engine in CREEK is designed to operate on information stored in the CREEK knowledge representation. The knowledge representation allows us to associate similarity measures (in the form of implementations of a specific Java interface) to specific classes. This similarity measure is then inherited to subclasses of the concept. For instance, the CASE concept has a similarity measure associated with it, which given two cases measures their similarity examining the findings for each case. A case's findings are found by following the *has finding* relationship from the case. All the concepts that are in such a relationship with the case are considered findings. For instance, in Figure 6.9, the TOYOTA and EMPTY BATTERY concepts are findings of CAR CASE #1. Given these sets of findings, the case similarity measure tries to match the findings from the input case to the target case by trying all combinations of findings and using the best match. The similarity measure used for the findings depends on the similarity measure associated with that particular finding. For instance, the TOYOTA concept is an instance of SYMBOL, which has a similarity measure using plausible inheritance associated with it. Other findings may be instances of NUMBER, STRING, or other concepts associated with specific similarity measures. In this work, a new similarity measure was introduced for comparing submodels, which is used to compare concept maps. This similarity measure uses the similarity measure defined in Section 5-3.3. An example of a CREEK model for case-based reasoning using submodels is found

in Figure 6.10.

The case-based reasoning engine in CREEK can be invoked from the Troll-CREEK editor directly (Figure 6.11).

The CREEK case-based reasoning engine is implemented in the `jcreek.reasoning` package, where the most important classes are:

EntityComparison. Abstract class that represents a similarity measure between two entities. In CREEK, similarity assessment is done in stages with *activate* generally using a computationally cheap method, and *explain* using more in-debt strategies to refine the similarity. However, the explain step may only increase the matching strength – the activate similarity is taken as a lower bound. This class also has class methods for identifying if two entities are comparable (i.e. they have the same similarity measure represented by a subclass of `EntityComparison`), and creating an instance of the proper subclass of `EntityComparison` to perform the similarity measure.

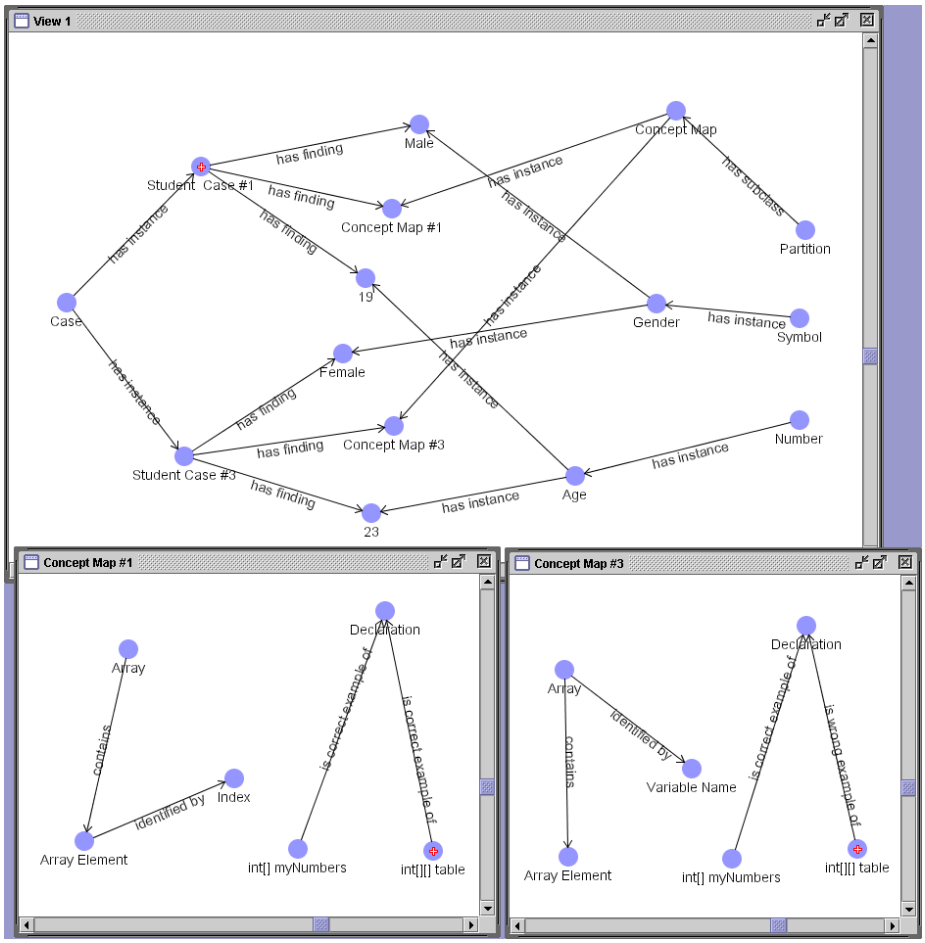
CaseComparison. Represents the similarity measure used to compare two cases. This is done by identifying the set of findings for each case (concept to which the case has a *has finding* relationship). Once the sets of findings for each case is identified, each finding from the input case is paired with each finding from the target case, and if they are comparable, a comparison object is created. The similarity between the input and target case is based on an aggregation of the similarity scores of the findings.

SymbolComparison. Similarity measure for symbols. In activate, this checks if the two symbols are the same (they represent the same concept), but in explain, this method uses plausible inheritance to measure the degree of overlap in the symbols' causal models (for details see [3]).

NumberComparison. Similarity measure for numbers. This measure examines the attribute the number is an instance of (e.g. 19 is an instance of AGE), finds the highest and lowest value of this attribute and uses this to create a linear, normalized scale from 0 to 1.

PartitionComparison. Similarity measure for submodels. This method compares the overlap of relationships in the partitions to the size of the union of the relationships in the submodels, as defined in Section 5-3.3.

RetrieveResult. Takes an (unsolved) input case and creates one `CaseComparison` instance for each solved case in the model, comparing it to the input case. This class has various parameters for threshold values for when the explain step should be run and so on. When reasoning is done, the `CaseComparison` instances are sorted according to similarity score.



In this artificial model, two cases (Student Case #1 and Student Case #2) has three findings each. One is the age of the student (a number), the gender (a symbol), and a concept map (a submodel). The concept maps are shown in the separate windows titled Concept Map #1 and Concept Map #2. The Case, Number, Symbol and Partition concepts have special semantics in that similarity measures are associated with them that is inherited down to subclasses and instances.

Figure 6.10: A model for case-based reasoning in CREEK.

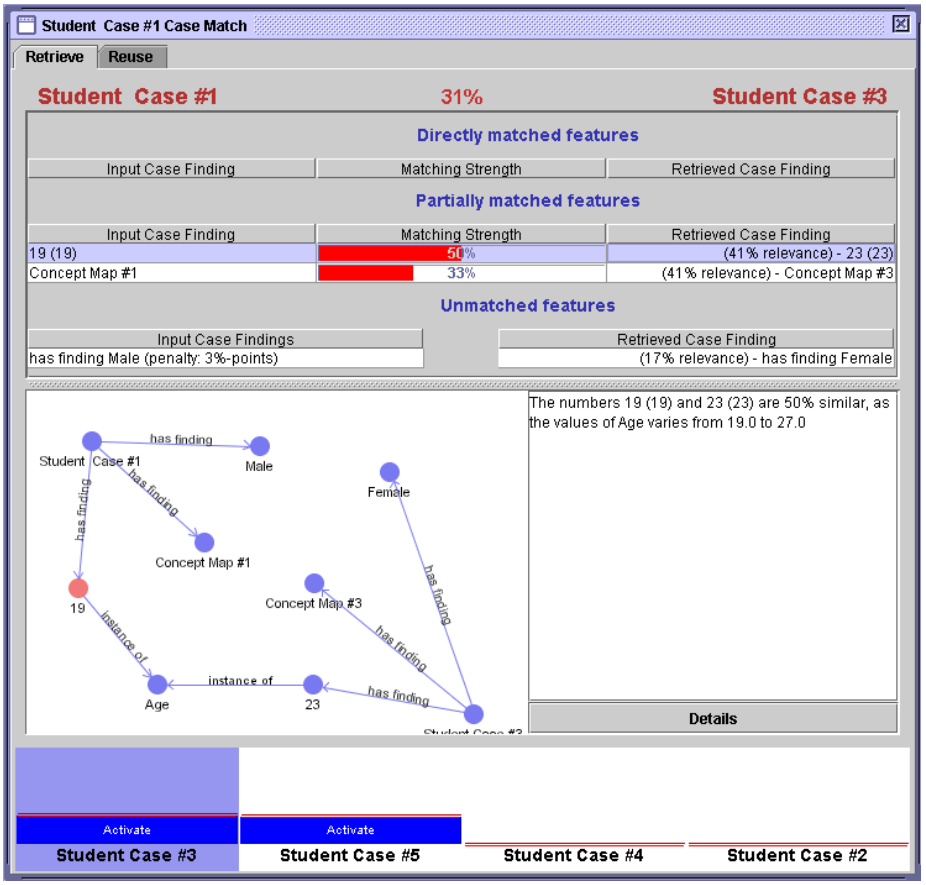


Figure 6.11: Case-based reasoning in CREEK.

ReuseResult. Runs the reuse step, currently by doing a simple k-Nearest Neighbor vote among the top k cases.

These classes have graphical user interface (GUI) classes that allow the user to explore their contents in detail. For instance, the window in Figure 6.11 is the GUI class for the `RetrieveResult` class. Here, the user may browse the individual `CaseComparison` instances by clicking on a case on the bottom bar graph to get detail information about that comparison (shown above). Appendix B contains a sample Java program that executes the case-based reasoning on the model creating in the program in Appendix A.

6-5 Chapter Summary

The partial implementation of the CREEK-ILE system contains a full implementation of the CREEK knowledge representation language, as well as a case-based reasoner that includes the capability of retriving cases based on student concept maps. With the exception of using the knowledge representation as a storage medium for concept maps, these capabilities were not used in the online learning environment, which was primarilly designed to collect data for use in offline analysis. In the next chapter, we describe the data collection methodology, and evaluation of the datasets collected.

Chapter 7

Evaluation

In evaluating the CREEK-ILE approach, we have focused on examining how concept maps can be used as part of the student model. While it is clear that they express a different facet of knowledge than what is directly used to solve exercises, there are theoretical differences in how and if the conceptual knowledge represented in concept maps affect procedural skill. First, that conceptual knowledge is required to solve problems, and as a prerequisite can be expected to correlate with measures of procedural skill. This can be seen as an extreme version of what is practiced by the ACT-based Cognitive Tutor courses, where theoretical classroom sessions are followed by exercise sessions. Second, conceptual knowledge can be seen as completely different from procedural skill, which would mean that measures of conceptual knowledge would not correlate with measures of procedural skill. Third, learning conceptual knowledge and procedural skill can be seen as an intertwined process, for instance as Schank describes it, where exercises motivate theoretical learning.

In Chapter 5, we suggest that concept maps may contain information that will help the system to select exercises appropriate for the individual student. This rests on the assumption that it is possible to predict approximately how hard each individual task is for a given student, using the information in the student's concept map. In other words, this theory suggests a correlation between the conceptual knowledge measure (the concept map) and the procedural skill measure (competence at solving problem tasks). The first goal of our evaluation has been to test this hypothesis, to see if and how concept map correlate with procedural skill measures such as how many tasks a student can solve in a particular exercise, or how long he takes to solve it. To do this, we collected data from two exercises in an introductory Java class. This was done by adopting the standard exercise sets given in the class to the CREEK-ILE exercise environment. In addition to the standard programming tasks, the students were asked to form concept maps, and the resulting maps were stored along with various measures of problem solving competence. After the second exercise, we also asked the students to fill in a small survey with some background data (age, gender, previous experience) and to respond to some questions about the

experience. The information was collected into two datasets, one for each exercise. Once collected, they were subjected to statistical analysis, and various machine-learning methods were used to see if it was possible to predict the problem solving competence based on the information from the concept maps.

7-1 Data Collection

The data collection was done during two exercises (number 3 and 5) of the introductory course in Java course give to first-year students in the NTNU open computer science program. This program is open in the sense that anyone enrolled at the university may take the course, and it does not require previous knowledge of computer science, nor any high school science specialization. This means that there is a wide range of students attending the class, and it is normal to see a fairly high drop-out rate. A consequence of this is that it is hard to know exactly how many are following the course actively, but exercise approval lists suggests that approximately 130 students were active in the course at the time of the experiments.

The first data collection were done during exercise 3, which had tasks that combined basic class structures (such as methods) and control structures (such as `if`-statements and `for`-loops). Participation in the experiment was on a volunteer basis, and students had to sign up to be a part of it. Before the exercise, the CREEK-ILE environment was installed in the computer rooms available to the students, and they could also download and install the client software on their own computers, but with the additional requirement that they had to stay connected to the internet while working on exercises. This effectively limited participation to those that worked at school or with a broadband connection at home. Although participation was encouraged by the course lecturer and our research group, no further incentive to participate was offered. This does mean that our group has a self-selection bias. Because the students had already done exercises 1 and 2 using another programming environment (a text editor called TextPad), it is likely that this self-selection bias prefers students that are willing to try new things. This may mean that the group had a more positive outlook on solving the concept map tasks, which for them is a new kind of task, than a random sampling. Of the approximately 130 students, 28 signed up for and completed the exercise using the CREEK-ILE environment. Of these, 4 skipped the concept map exercises (defined as creating two or fewer relationships), and for this reason were excluded from the dataset. The exercise contained two concept map tasks, given at the very beginning of the exercise. One of these were on the topic of *Control Structures* (the teacher map is shown in Figure 7.1), and the other on *Classes and Methods* (teacher map in Figure 7.2). These had to be done in turn, and the student could not go back and alter the concept maps when they were finished, but they were allowed to go back and examine them. This was done to control that the maps represented the knowledge of the student before the exercise, as opposed to a mix of before, during and after.

The initial examination of the data from the first experiment showed that

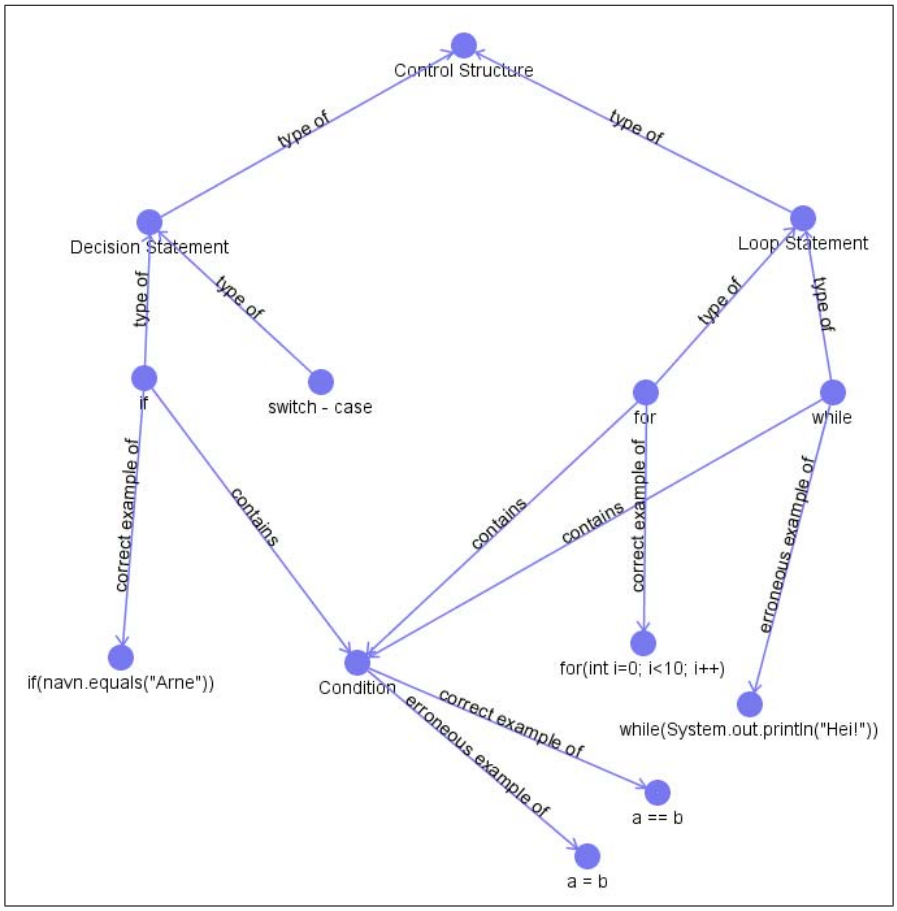


Figure 7.1: Teacher map for the *Control Structures* topic.

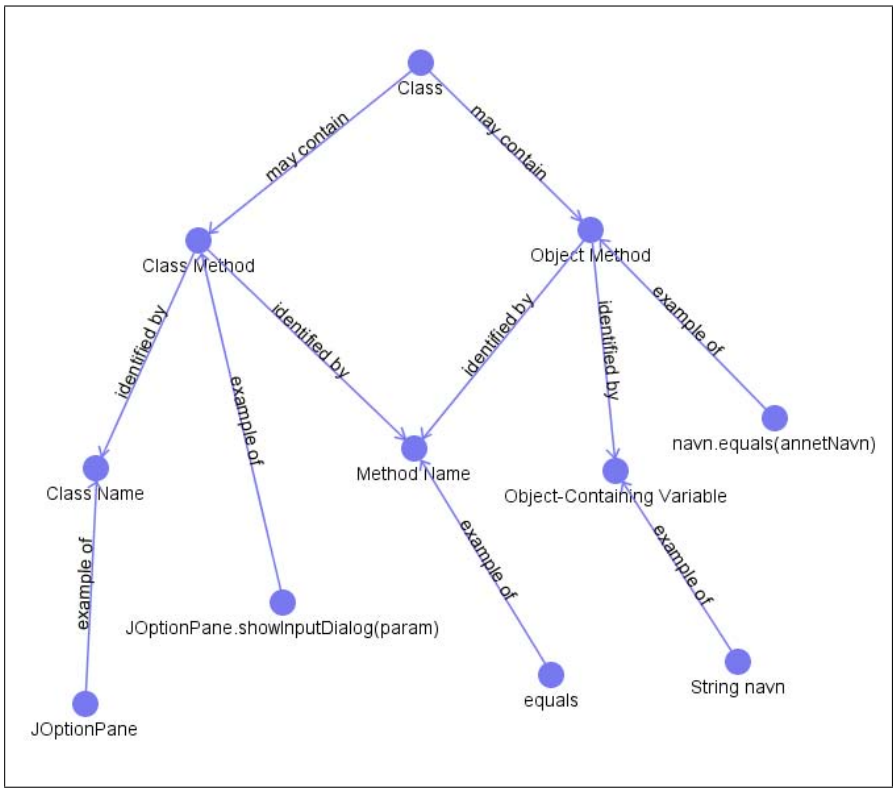


Figure 7.2: Teacher map for the *Classes and Methods* topic.

19 of the 24 had managed to solve at least 7 out of the 9 programming tasks in that exercise. If a task was successfully solved or not were decided by using the automated tests attached to each task. These tests were fairly stringent, including input scenarios to handle atypical situations. In theory, students are required to solve all the tasks to pass an exercise, but in practice, any real effort spent towards this goal is accepted. That so many students managed to fulfill so many of the tasks also suggested that the group may have been more skilled than a random group.

In the second round of data collection, we decided to use a more controlled environment, in that all that participated had to be at a computer lab during two three-hour sessions, and that the experiment would be time-limited to these sessions. The students were divided into two approximately equally sized groups with sessions at different times. I was present at all times during both of these groups, and served as their teaching assistant. The students were told before signing up that anyone that was not finished with the exercise after these six hours would be approved as long as they showed up and worked at the exercise the whole time. This was designed to equalize the time spent on the exercise so that the number of tasks the students would have solved within the time limit would be more indicative of their skill level. It also gave weaker students an incentive to participate, and likely put a bias on the participating group towards these students. In this second round, 48 people signed up, of whom 4 were excluded from the dataset because they did not fill in the concept map (again, defined as creating two or fewer relationships). This exercise set contained one concept map task, on the topic of *Arrays* (teacher map is in Figure 7.3). This concept map was designed with fewer abstract concepts, and more code examples than in the previous round. Again, the students were not allowed to change the concept map once they deemed in finished, but could go back and examine it. There were 9 programming tasks in this exercise, of which the first 5 were available in the first session, and the 4 last tasks were added for the second session.

After the second round of data collection, a small survey was sent to the participants. This survey contained a number of background variables (gender, age and previous experience in programming), as well as twelve statements the students were asked to state if they agreed with or not, and a free-form feedback textbox.

7-1.1 Error Sources

There are several possible errors sources in the data collection method described above. Participation in both rounds was voluntarily, which introduces a possible selection bias. This could mean that conclusions drawn from the data may only be applicable to a subset of the student population. In the main question of study in this experiment – how knowledge expressed in concept maps relate to exercise competency, there may be an effect where students volunteering for experiments are more accepting of non-traditional exercise tasks (such as forming concept maps). However, we have no reason to believe that the difference

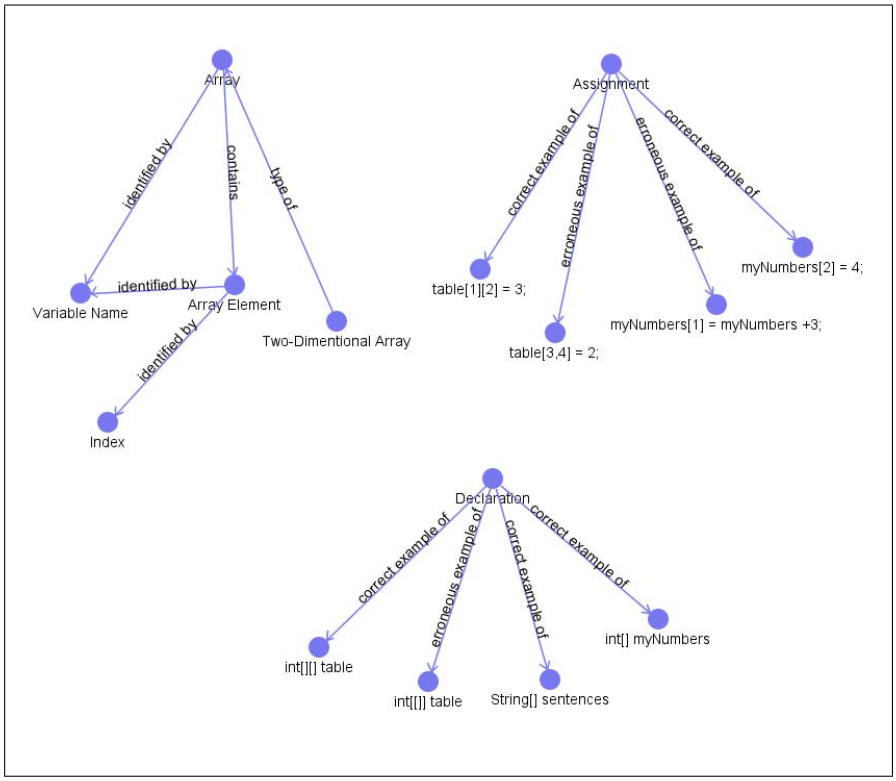


Figure 7.3: Teacher map for the *Arrays* topic.

between the experiment and the total population on this is a major factor.

There are various spurious variables that can affect the similarity of concept maps and exercise competency. Most of these not problematic for what we wish to study. For instance, students working together in groups before the experiment may have formed similar conceptualizations, and also have more similar exercise competency. Students attending lectures may likewise be more skilled in programming tasks than the group of students not attending lectures, and the students attending lectures may also have adopted similar conceptualizations based on the teacher's. These spurious relationships are not problematic because they still allow us to use the correlation in a student model. Although it would be interesting to determine causation in how conceptual and episodic knowledge is formed, for use in for instance determining individual difficulty of exercise tasks, this is not required. However, there are some spurious relationships that are problematic. In particular, we have identified *cheating* and *cooperation*.

Cheating is when a student does not solve a task himself. It may argued that a cheating student would copy the concept map tasks as well as the programming tasks and as such simply represent another (presumably more skilled) student. However, since there is no evaluation criteria for the concept map tasks, the incentive for copying it seems much less than copying the programming tasks. There is also the factor that some of the programming tasks were quite hard to solve for many, and it is fairly common for students to become time constrained towards the end. This suggests that cheating would be far more common on the harder tasks towards the end of the exercise than the easier tasks. We believe the first dataset is more likely to be affected by cheating than the second. In the second collection, students were approved based on the time put into the exercise and not on actually completing all the tasks. This reduces the incentive to cheat. They also did all their work in a computer lab attended by a teaching assistant at all times, which would make cheating more difficult.

Cooperation is the most serious error source we have identified. This occurs when two or more students work together to solve the exercise, and for this reason may have very similar concept maps as well as exercise competency. Depending on the degree of cooperation, this could range from some occasional similarities to representing the same data instance twice. This is not a great problem when searching for statistical correlations between variables, but it is a serious problem when using machine-learning methods to predict student behavior or competency. If the dataset contains an instance twice, a case-based reasoner used in an offline leave-one-out cross validation would identify the twin instance as the most similar case and use its (correct) solution as its prediction. However, in an online system, cooperating students would have to work during the same time periods, and as such they would not be available as previous cases.

Cooperation was observed and allowed during the second data collection, although the analysis suggested that there was little total cooperation. The degree of cooperation in the first collection was not observed, so it is unknown. We have considered several ideas for measuring the cooperation between stu-

dents based on the tracking data collected. For instance, it might be possible to judge the level of cooperation based on the similarity of programs submitted, or the time spent on each task. However, such measures can be formed in many ways, and the likelihood that one or more exists that would agree with the hypothesis through chance alone is great, so in the end this approach was abandoned. Instead, the fact that the second data collection was conducted using two independent groups in two different timeslots was used to divide the second data set into two groups corresponding to these timeslot groups. Because these groups operated during different times, we know that there was no cooperation between members of these groups, and by using one as training data and one as testing data, the influence of cooperation could be removed in the analysis of this dataset. The cost of doing this is that the size of the training data is almost halved, affecting the precision of the prediction.

7-2 Dataset Variables

The data collected in the data collection phase was compiled into two datasets, one for each round. These datasets are in WEKA [92][1] and comma-separated file format designed to be imported to such tools as SPSS or Excel, and is available at http://www.idi.ntnu.no/~frodeso/creek-ile/datasets/.

The dataset contains data that are collected directly from the students as well as some recoded and composite variables. This section contains a description of all the variables in the datasets.

7-2.1 Programming task variables

For each programming task in the exercise, there are three variables. The names of these variables contains the name of the task, for instance the name of the variable representing the time a student spent on the task *Oppgave1a* is called *eOppgave1aTime*. Figure 7.4 contains a summary of the percentage of students that completed each task as well as the average competency measure for each task in the second dataset.

e<Taskname>Solved. This is a binary variable that represents if the task was successfully solved or not. The 0 value means that the task was not solved, the 1 value means that it was solved. A task is defined as solved if the program passed all the automated tests for that task.

e<Taskname>Time. The time, in seconds, required by the student to solve the task. Only time actively spent on that task is counted – if there was no change in the program state for 60 seconds, the student was assumed to be idle (not active) until the next change.

[1] WEKA is an open-source machine learning and data mining suite available from available from the University of Waikato at http://www.cs.waikato.ac.nz/ml/weka/

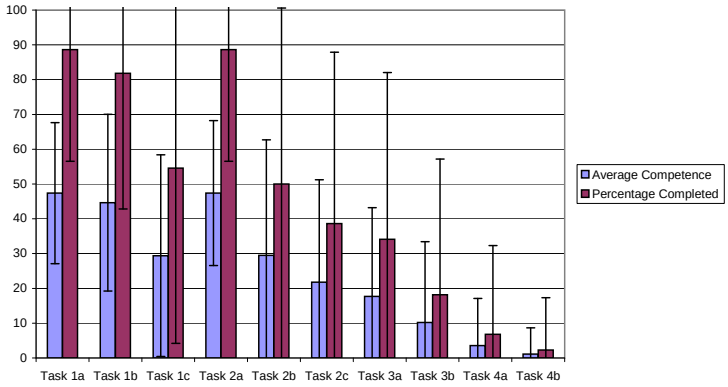


Figure 7.4: Student competency on programming tasks in dataset 2

e<Taskname>Competence. This is a composite variable intended to measure the competence displayed by the student on this task on a scale from 0 to 100. If the student failed to solve the task, an automatic 0 is given. If it is successful, the competence is calculated using the formula in equation 7.1, where *averageTime* is the average time spent on the task (i.e. the average of e<Taskname>Time for all the students) and *time* is the time this student spent on the exercise, i.e. e<Taskname>Time. This formula gives a value between 0 and 100, where 100 means the student solved the exercise in literally no time at all, 50 means the student solved it on the average time. The value approach 0 asymptotically as the time spent to successfully solve the exercise approaches infinite.

$$e < Taskname > Competence = \frac{100 * (averageTime - 1)}{time + averageTime - 1} \tag{7.1}$$

7-2.2 Overall Competence Measures

In addition to the variables representing each programming exercise, the datasets contains three variables aggregating these measures.

correctCount. The number of tasks correctly solved by the student. This is essentially an aggregation of the values of the e<Taskname>Solved variables.

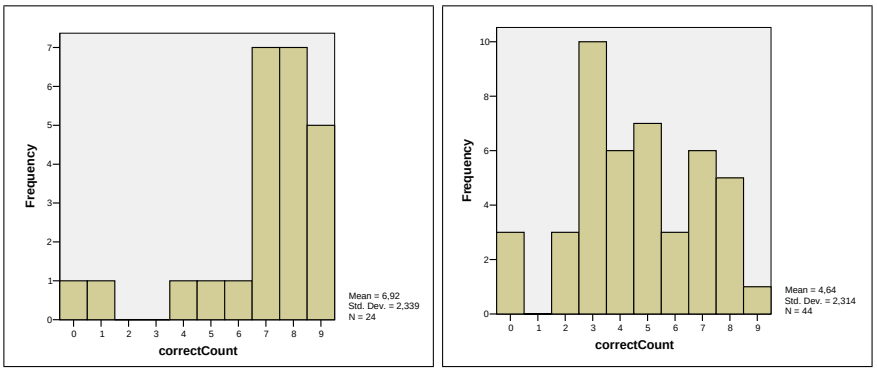


Figure 7.5: Distribution of the `correctCount` variable for dataset 1 (left) and dataset 2 (right)

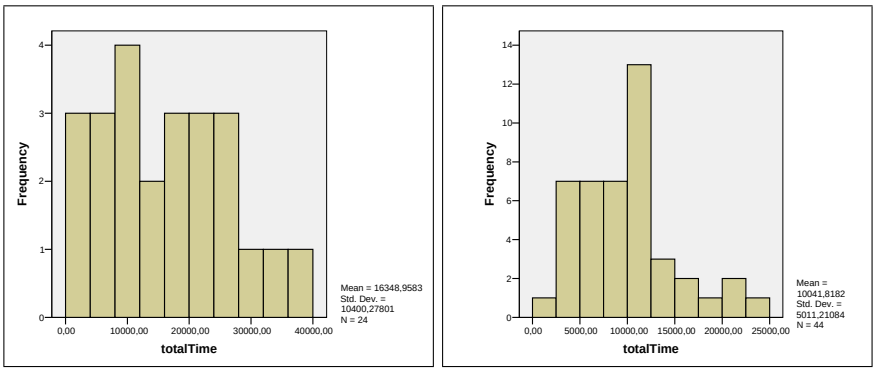


Figure 7.6: Distribution of the `totalTime` variable for dataset 1 (left) and dataset 2 (right)

Figure 7.5 contains a distribution histogram of this variable.

totalTime. The active time spent on the programming exercises, which is an aggregation of the e<Taskname>Time variables. Figure 7.6 contains a distribution histogram of this variable.

averageCompetence. The average over the e<Taskname>Competence values for the student. This is designed to be a measure of the total skill of the student taking both time and successfully solved tasks into account. The distribution of this variable is shown in Figure 7.7.

7-2.3 Concept Maps

Representing concept maps in a traditional machine-learning representation is not straightforward, but because we relate all the student concept maps to a

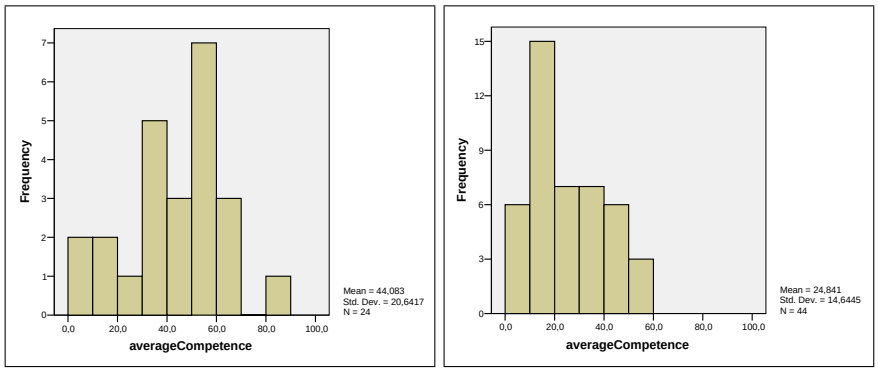


Figure 7.7: Distribution of the `averageCompetence` variable for dataset 1 (left) and dataset 2 (right)

specific teacher's map, there is a finite set of possible concept maps that can be constructed. This means that it is possible to encode any concept map, for instance by using one binary variable to represent each possible arc in the concept map. In practical terms, this is not possible because the number of possible arcs in a concept map increases exponentially with the number of labels and arcs in the teacher map (see Section 5-3.3). However, with a limited number of students as in our experiments, the total number of different arcs actually used by students are far fewer. In the first data set, only 113 unique arcs (concept-label-concept connections, such as `for kind of Loop Statement`) were formed in the first map. The second map had 89 unique arcs, and in the second group, the map had 255 unique relationships. This means that it was possible to create a binary variable for each unique arc that was present in at least one student map. This encoding allows us to use standard machine learning tools such as WEKA on the datasets off-line after the data is collected. It would not be possible to use this encoding in online use, because we rely on the ability to check which unique arcs are used by at least one student, and that is of course not known until after all concept maps have been finalized. The CREEK representation does not have this problem, however, and it is likely that specialized implementations of the machine learning methods we examine can also get around this problem.

x<Mapname>r<number>. A binary variable that represents the presence (value 1) or absence (value 0) of a specific relationship in the student's concept map. The <number> is counting number that identifies a unique relationship (concept-label-concept)

<mapname>Size. The number of arcs drawn by the student in that map. This is essentially a count of the number of "1" values on the x<Mapname>r<number> variables. Figure 7.8 displays the distribution of these variables for dataset 1, and 7.9 show it for dataset 2.

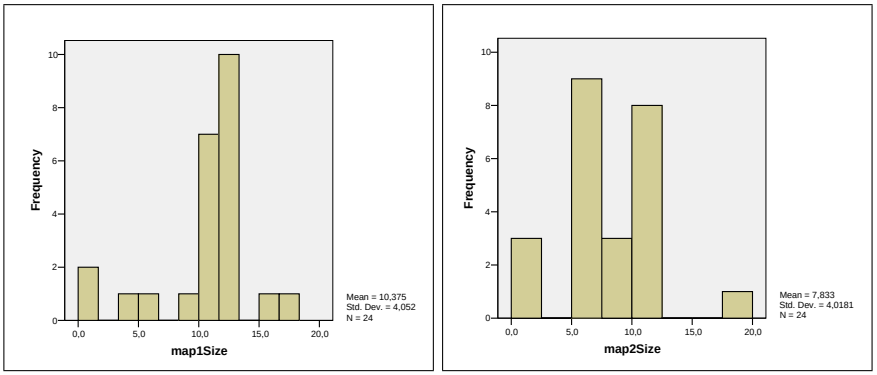


Figure 7.8: Distribution of the `map1Size` and `map2Size` variables for dataset 1

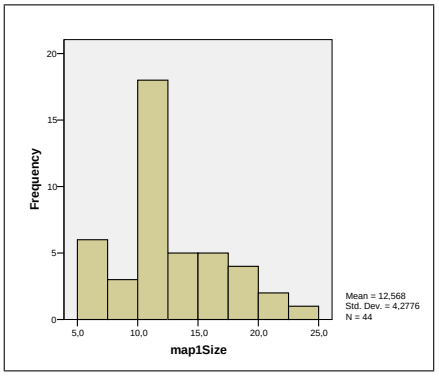


Figure 7.9: Distribution of the `map1Size` variable for dataset 2

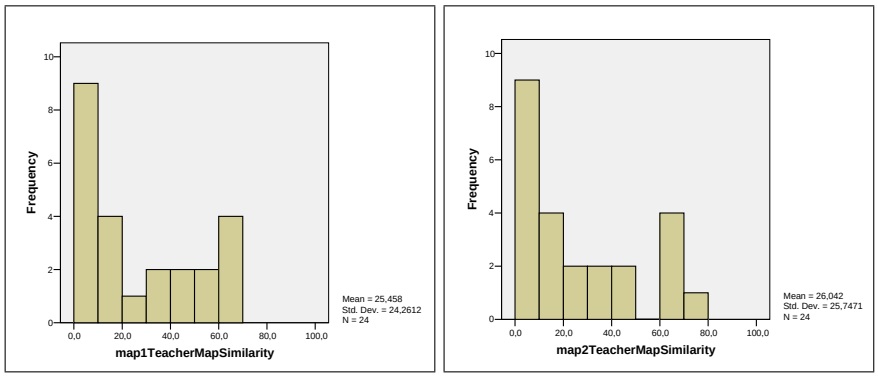


Figure 7.10: Distribution of the `map1TeacherMapSimilarity` and `map2TeacherMapSimilarity` variables for dataset 1

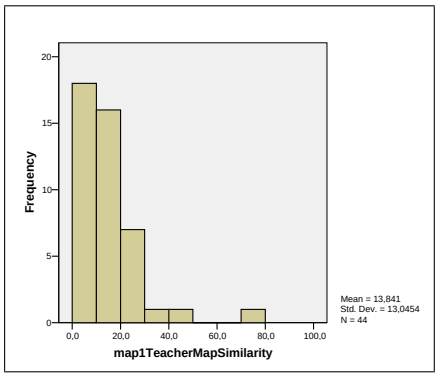


Figure 7.11: Distribution of the `map1TeacherMapSimilarity` variable for dataset 2

<mapname>TeacherMapSimilarity. The similarity between the student's map and the teacher's map, calculated by CREEK using the similarity measure described in Section 5-3.3. This is a numeric value from 0 (no similarity) to 100 (absolute similarity). Figure 7.10 displays the distribution of these variables for dataset 1, and 7.11 show it for dataset 2.

7-2.4 Background Variables

These variables represent background information about the student. This data (except the group variable) was collected in a separate survey that only went out to the second group, and as such they are only present in the second dataset.

group. The group (1 or 2) of which the student were a member. This information is only available in the second dataset.

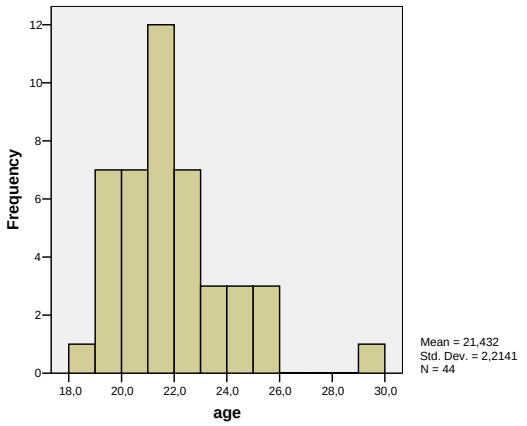


Figure 7.12: The distribution on the `age` variable

0	Never Programmed	81.8%	N = 36
1	Programmed some	13.6%	N = 6
2	Knew most of the curriculum	4.5%	N = 2

Figure 7.13: The distribution on the `exeperience` variable

age. The age of the student, in years.

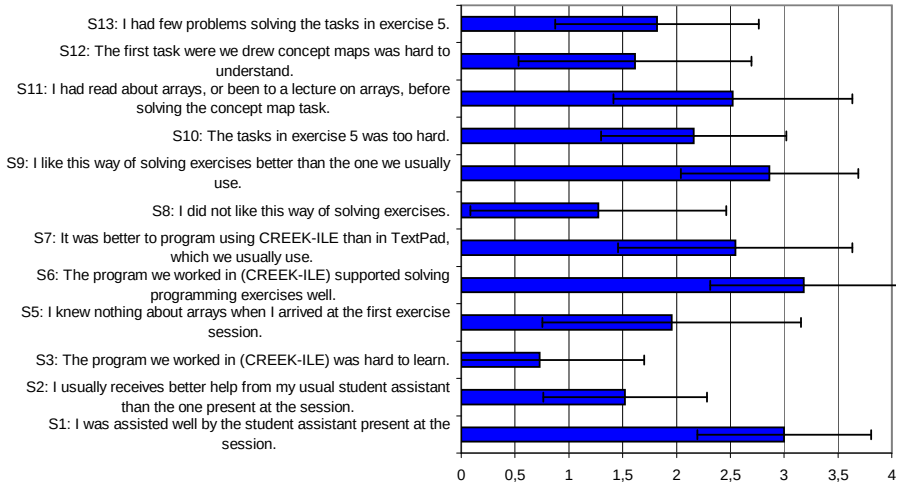
The age distribution of the selection in the second dataset ranges from 18 to 30, where the 30 is quite atypical (next highest value is 25). Figure 7.12 contains a histogram of the distribution. Although no statistics are available to us on the total student population attending the programming course, informal observations seems to suggest that this distribution is close to the population distribution.

gender. The gender of the student. A value of 0 means male, a value of 1 means female.

In the second dataset, 13.6% (6) of the students are female.

experience. An ordinal variable representing the experience level of the student. The question (translated from Norwegian) was *"How much have you programmed before starting this class (IT1103)?"*. A 0 value represents choosing the answer (again, translated) *"Never programmed"*, a 1 represents *"Programmed some, but learned a lot during the class"*, and a 2 represents *"Knew most of the curriculum of the course already"*.

Figure 7.13 contains the frequency distributions of the `experience` variable in the second dataset.



For each of the feedback questions, the average value (the blue bar) and standard deviation (the black line) is shown. A value of 0 means that the student disagrees strongly and a value of 4 means that he agrees strongly.

Figure 7.14: The feedback questions, with average response

7-2.5 Feedback Questions

These questions were also part of the survey sent only to the second group. They are asked in the form of statements the student can strongly agree, agree, be neutral, disagree or strongly disagree with. These answers are represented as a value from 0 (strongly disagree) to 4 (strongly agree). The questions were posed in Norwegian, but are translated as faithfully as possible here[2]. Figure 7.14 contains the questions and average response value for each question.

7-3 Statistical Analysis

In addition to using machine-learning methods such as case-based reasoning to predict student competency, statistical tests of correlation as well as descriptive statistics have offered some insights. In this section, we present the major conclusions we have been able to draw.

Throughout this section, one-tailed Pearson's Correlation is used to measure bivariate correlations, and only relationships are at least significant on the 5% level are reported, unless otherwise stated. When correlations are given, the coefficient is denoted r and the significance level s. If the relationship is signif-

[2]The students knew the CREEK-ILE environment as *Mind Exerciser* at the time. References to Mind Exerciser are replaced with CREEK-ILE in the translation.

icant on at least the 5% level, it is marked with a single star (*), and if it is significant on at least the 0.5% level, it is marked with a double star (**). For instance (r = 0.4, s = 0.02*) means that there is a positive correlation with the coefficient 0.4 and significance 0.02 (i.e. 2%).

7-3.1 Correlation of Competency Measures and Time

We have introduced a composite measure of competence, `averageCompetence`, combining the time spent on a task and if the student managed to solve it. It is interesting to see if this measure correlates well in the aggregate with the more direct `correctCount` competence measure, and in dataset 2 it does correlate very strongly (c = 0.967 s < 0.0005**). In dataset 1, the correlation is also strong (r = 0.775, s < 0.0005**), but not as strong as in dataset 2. This is likely because the students in dataset 1 were not time constrained in the same way as the students were in the second data collection. This means that most students would be able to solve all or most of the tasks, although they may have used longer time. Figure 7.15 shows the relationship between `totalTime` and `averageCompetence`in dataset 1, and we can see that in general, the better students use less time on the exercise. In fact, the quickest student that managed to solve all the programming tasks went through the exercise in just under an hour, while the slowest used more than 10 hours, and only solved 5 of the 9 tasks. The two students with even lower competency than this (0 and 1 tasks solved) used somewhat less time than that.

7-3.2 Student Contentment with CREEK-ILE

In general, the students seemed happy to work in the CREEK-ILE environment. A majority of the students (84.1%) disagreed with the `s3` proposition (*"The program we worked in (CREEK-ILE) was hard to learn."*), and the majority even disagreed strongly (52.3%). This is also reflected in the `s8` proposition (*"I did not like this way of solving exercises"*), where 79.5% disagreed, although fewer did so strongly (20.5%). CREEK-ILE was also viewed as a better alternative than their regular way of solving exercises (79.5% agrees with proposition `s9` – *"I like this way of solving exercises better than the one we usually use."*), and their regular programming environments (18.2% disagrees with `s7` – *"It was better to program using CREEK-ILE than in TextPad, which we usually use."*).

Interestingly, the more experienced students seemed to find the CREEK-ILE environment harder to learn than the less experienced students – the `experience` variable correlates with `s3` (r = 0.445, s = 0.001**). This suggests that the CREEK-ILE environment is particularly friendly to beginners. In the free-form comments some users requested more advanced editor features such as automated indentation. The lack of such features as compared to their usual editors may explain why the more experienced users found the learning curve steeper.

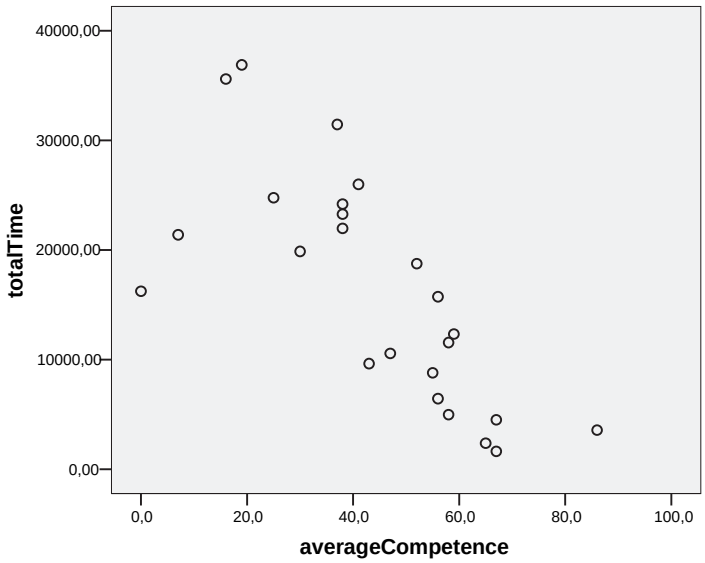


Figure 7.15: Scatterplot on the `totalTime` and `averageCompetence` variables in dataset 2

.

7-3.3 Young and Lazy?

The young seems to spend less active time solving the exercises – there is a positive correlation between `age` and `totalTime` (r = 0.373, s = 0.006*). Older students were also better prepared (negative correlation with `s5` *"I knew nothing about arrays when I arrived at the first exercise session"* (r = -0.351, s = 0.010*)). This did not mean that they were necessarily more experienced programmers before starting the class as there is no correlation between `age` and `experience`.

The question, then – does the extra effort pay off for the older students? To some degree. Age has a weak positive correlation with the `correctCount` competency measure, but this is not quite significant (r = 0.245, s = 0.055*). This seems to be an effect of the extra time and preparations, and not some other effect of ages, as this effect disappears completely if we control for the `totalTime` and `s5` in a linear multivariate regression.

7-3.4 Gender Differences

The great disparity between the genders (only 6 females are present in dataset 2) suggests that we should be careful in drawing any conclusions on gender differences from this dataset. That said, there is a fairly strong correlation suggesting that the female students perceived the exercise as harder than the male students (`gender` correlates with `s13` with r = -0.418, s = 0.002**). However,

there is not a significant correlation between gender and the actual competence measures (`correctCount` and `averageCompetency`), and there are differences between the male and female population, such as lower levels of previous experience and slightly lower average age in the female selection that may explain these findings. Still, it might be interesting in a larger experiment to test the hypothesis that female students perceive the exercises as being harder.

Another interesting finding is that the women have concept maps significantly more similar to the teacher than the men (r = 0.432, c = 0.002**). This finding remains if we control for age, experience and the feedback questions – gender remains the strongest indicator of teacher map similarity of all of these. We do not have a good explanation for this, but it might be an interesting topic for further research.

7-3.5 Does Experience Matter?

The students with higher `experience` has scored slightly higher on the competency measures than less experiences students, but these differences are not significant. However, the experienced students did tend to agree more strongly with proposition S13 (*"I had few problems solving the tasks in exercise 5."*), resulting in a significant correlation between `experience` and `s13` (r = 0.489, s < 0.0005**). This seems to suggest that the experience level reported by the student affects perceived difficulty more than actual competence. However, the number of students in the sample with any experience (value 1 or 2 on the `experience`) variable is only 8, so like the gender differences, this conclusion is tentative.

7-3.6 Similarity to Teacher Map

One of the major hypothesis we would like to test in this experiment is if the degree of similarity between the student's and the teacher's concept map has any predictive power of procedural tasks – in this case programming tasks. This correspond to the *Method 1* of using concept maps in exercise selection, described in Section 5-4.1.

Throughout this subsection, we use two-tailed Pearson's Correlation.

In our datasets, the student-teacher concept map similarity are almost completely independent of the major programming competence measures (in dataset 2, `map1TeacherMapSimilarity` and `correctCount` has r = 0.059, s = 0.704). This is also illustrated in a scatter diagram in Figure 7.16.

However, there is some correlation on a few individual task measures. In dataset 2, `map1TeacherMapSimilarity` correlates with `eOppgave1cCompetence` (r = 0.348, s = 0.021*), and `eOppgave2bTime` (r = 0.323, s = 0.032*).

Similar results were found for both concept maps in dataset 1. Neither correlate at all with any of the overall competence measures, but on a few variables for specific tasks, there were correlations. Specifically on `map2TeacherMapSimilarity`, which correlates with `eOppgave2bCompetence` (r = 0.464, s = 0.022*) and `eOppgave2bSolved` (r = 0.526, s = 0.008**).

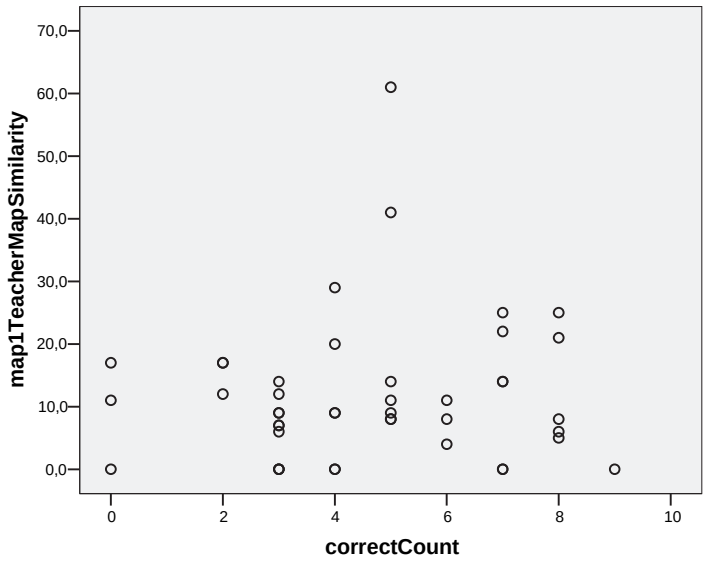


Figure 7.16: Scatterplot on the `correctCount` and `map1TeacherMapSimilarity` variables in dataset 2

These relationships may seem somewhat suspicious. In light of the number of siginficance tests done, a couple could very well show up by chance. The positive correlation between the teacher map similarity and the time spent on task 2b in dataset 2 is particularly strange. This means that the students with a concept map more similar to the teacher were likely to spend longer time on task 2b! On the other hand, this may just mean that fewer of these students gave up. If we look to Section 7-4, the variables with correlations here matches those found to have predictive effect there, so the consistency of results implies that this is not mere chance.

A stronger relationship is found between the `map1TeacherMapSimilarity` and `map2TeacherMapSimilarity` measures in dataset 1 ($r = 0.737$, $s < 0.0005$**). This means that if a student forms a map that is similar to the teacher's on the first task, he is more likely to form a concept map similar to the teacher's in the second. This, at least, suggests that the concept maps contains some measurable information about the student, presumably the conceptualization given in lectures and the text book. In dataset 1, we also see a correlation between the map size and the similarity to the teacher map (`map1Size` and `map1TeacherMapSimilarity` at $r = 0.404$, $s = 0.050$*, `map2Size` and `map2Teacher-MapSimilarity` at $r = 0.538$, $s = 0.007$*). This relationship is also present in dataset 2 ($r = 0.308$, $s = 0.042$*).

The only other variables that correlates with the `map1TeacherMapSimilarity` variable is, `age` ($r = 0.291$, $s = 0.028$*) and, as already mentioned, `gender` (r

= 0.431, s = 0.002**). We currently have no explanation for this relationship. One theory is that a student's conceptualization will grow closer to the teacher's as he approaches the teacher's age, but the result on the gender variable would seem to contradict this, as it was the men's maps who were less similar to the teacher's map, which was also created by a man.

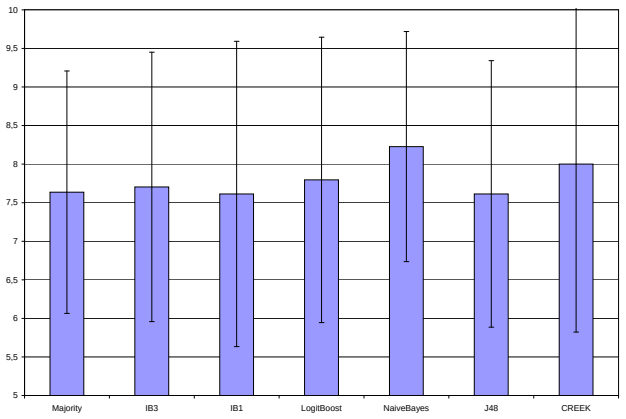
The lack of any significant relationship to teacher map similarity also includes the propositions S11 (*"I had read about arrays, or been to a lecture on arrays, before solving the concept map task."*) and S5 (*"I knew nothing about arrays when I arrived at the first exercise session."*), which one might expect to correlate with a measure of conceptual knowledge. This could again reflect the fact that the particular concept map in dataset 2 focused more on actual code examples and less on organizing the terms, but we also observed that some students spent time with their text books, looking up how different terms related while forming their concept maps. This suggests that the concept map tasks formed the motivation to seek the conceptual knowledge they did not have in advance.

7-4 Predicting Student Exercise Competence

In the previous subsection, we have shown that similarity to teacher maps did not correlate with any of our measures of student exercise competence. This is a fairly strong indication that Method 1 ("Model first, Compare to Teacher") from Section 5-4.1 will not work. However, this does not necessarily mean that there is no information content in the student concept maps that will help predict student exercise competence. The teacher map, although presumably a competent conceptualization, represents only one individual's conceptualization. A comparison of the teacher's map with students' maps rests heavily on the quality of the teacher's map. If her conceptualization is in some way unnatural or too complex for the average student, similarity will be low and a possible correlation between concept map and exercise competence hidden.

The alternative approach suggested in Method 2 ("Model first, Compare to Students") is an alternative that relies less on the quality of the single conceptualization represented in the teacher map. The concepts and relation names are still constrained by the teacher, but instead of comparing a student's map to the teacher's, it is compared to the other student maps. Using machine learning techniques, classifiers and regressors may be trained on a subset of the student maps to predict the competence of the students. The ability of the machine learning methods can then be tested on the remaining student instances. In this section, we will test the hypothesis that machine learning on concept maps can predict student competence by applying standard and specialized machine learning techniques to the datasets produced by the data collection efforts described in Section 7-1.

Our first attempt at using machine learning methods to predict student exercise competence used the CREEK case-based reasoning mechanism with the similarity measure from Section 5-3.3, and was reported in [77]. The CREEK



This figure shows how many of the ten tasks in this exercise each classifier predicted the correct result for. The baseline method predicts by always picking the majority class for each task. The thin error lines represent a standard deviation.

Figure 7.17: Evaluation of CREEK's predictions on individual exercise tasks

case-based reasoner used the concept map of a new student and compared his map to those of previous students. The previous student with the most similar map was then used as a basis for predicting how the new student would fare on the programming exercises. This experiment was performed offline on the data collected from the second round of data collection (see Section 7-1)[3]

Throughout this section, the one-tailed Student's T-Test will be used for testing significance as the distribution of values that are approximates normal. When using other machine-learning methods than CREEK, the machine-learning library Weka (version 3.4.8) [92] is used. Unless otherwise stated, the default parameters for these methods are the defaults supplied by Weka.

7-4.1 Classification Test

In the datasets we have collected, the e<task>Solved variables are binary variables (solved or not solved), which may be predicted using standard machine-learning classifiers. In addition to CREEK, we have used several methods from the Weka machine-learning library on the two datasets. We have used Weka's

[3] There are some small differences in the numbers reported in [77] and this thesis. The reason for this is that the experiment in the [77] paper included the four instances that was excluded from the second dataset here. The differences are minor and only in one situation does it impact a significance test, but it does not impact the overalls results.

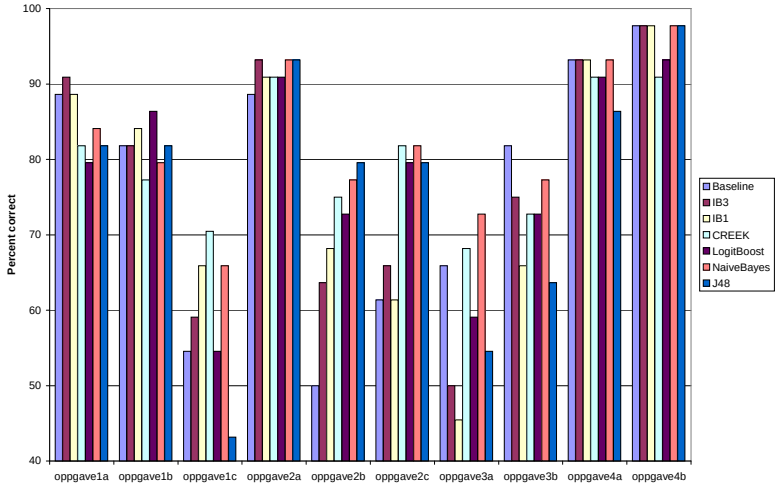
instance-based nearest neighbor IBk classifier (with k set to 1 and 3, called IB1 and IB3 respectively), as well as the LogitBoost classifier as an example of a boosting method, NaiveBayes as a simple statistical method, and J48[4] as a information entropy tree-based classifier.

Using these classifiers, we performed a leave-one-out cross validation test, where all classifiers were trained on the all but one of the instances in the dataset. The classifiers were then tested on the excluded instance by predicting which tasks the student represented by that instance managed to solve. This was repeated so that in turn, all instances were excluded from the training data and used as a test. Figure 7.18 shows the correct prediction percentage for each classifier on every task in the second dataset. We compared this with a baseline where for each task the majority class was chosen as the prediction. Although almost all students were able to solve the easy tasks, very few solved the most difficult tasks. This caused the frequency of the majority class to vary widely from one task to the next. In particular the very easy and very hard tasks had a very high frequency for the majority class, but the middle difficulty tasks discriminated better between the students, giving a baseline close to 50%. On some of these, we see significant improvement over the baseline on several of the classifiers, including CREEK. An aggregation of these results found in Figure 7.17, shows how many correct classifications each classifier got, on average, for each student. A perfect classifier would get all ten predictions for each student correct every time, giving an average of ten, while a random guess would give an average of five. This shows that CREEK and NaiveBayes have higher rates of prediction than the other methods, but only NaiveBayes is significantly better ($t < 0.05$).

The way the first dataset was collected means that it is harder to test using the classifiers. Here, the students were given unlimited time and no option to be given credit for the exercise was given unless the student solved, or tried hard to solve, all tasks. This means that we would expect that fewer tasks would go unsolved, and a higher baseline on most tasks, which makes it harder to distinguish the classifiers from the baseline. In Figure 7.19, we see that although most tasks have a high baseline, some tasks, like 2b, 3a, and 3b has a low enough baseline that several classifiers improve in the baseline (including CREEK on tasks 2b and 3b, although it is lower than the baseline on 3a). However, none of these differences are significant for this dataset.

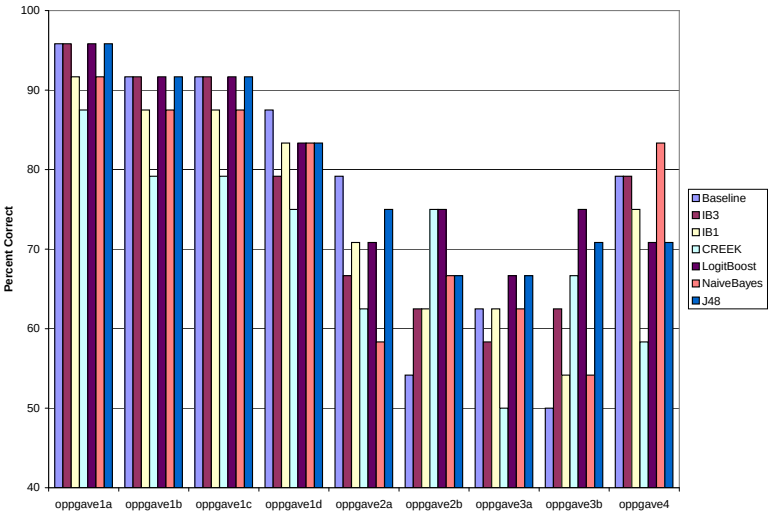
A curious result in these graphs is that CREEK seems to be doing better than the instance-based methods. This may seem strange because CREEK is not using any of its knowledge-intensive methods here – the cases are matched based only on the concept maps using the simple similarity measure from Section 5-3.3. This means that one might think that CREEK should perform similarly to IB1. This is not the situation, because of a subtle difference in the similarity measures involved. IBk counts the number of matches in a pre-defined feature vector, which means that if two maps have 12 relationships in common and 7 that are not in common, IBk will find these as similar as two maps that have 3

[4] J48 is Weka's implementation of the C4.5 algorithm



This figure shows how the percentage of correct classifications found for each classifier on each task for dataset 2. This is compared to the baseline for each programming task given to the student. While the baseline for the very easy tasks (1a, 1b and 2a) and very hard tasks (3b, 4a and 4b) are high and hard to beat, several machine learning methods (including CREEK) show an improvement over the baseline in the more discriminate tasks (2b and 2c). On task 2b, IB1 ($t = 0.042$), CREEK ($t = 0.007$), LogitBoost ($t = 0.014$), NaiveBayes ($t = 0.003$), and J48 ($t = 0.002$) was significantly better than the baseline. On task 2c, CREEK ($t = 0.016$), LogitBoost ($t = 0.031$), NaiveBayes ($t = 0.034$) and J48 ($t = 0.031$) was significantly better than the baseline.

Figure 7.18: Classification on Tasks from Dataset 2



This figure shows how the percentage of correct classifications found for each classifier on each task for dataset 1. This is compared to the baseline for each programming task given to the student. Only the LogitBoost classifier on task 3b is showing a significant increase ($t = 0.038$) over the baseline here.

Figure 7.19: Classification on Tasks from Dataset 1

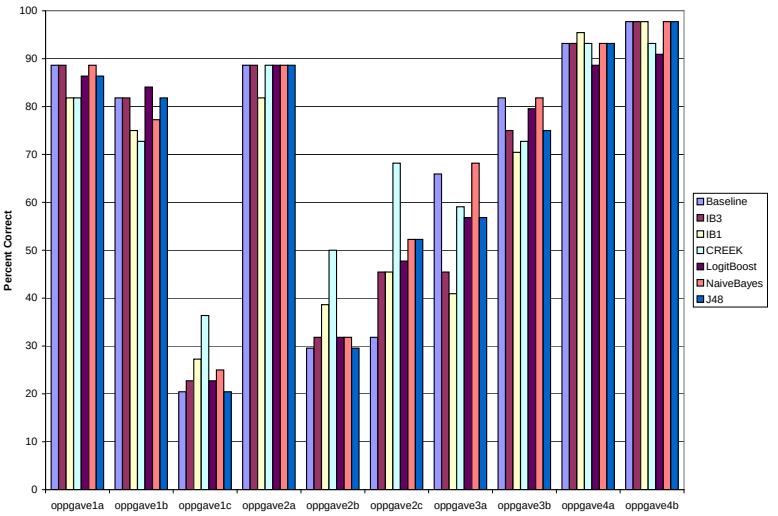
relationships in common and 7 that are not. This because IBk will include the number of relationships that does not exists in both maps as matching features. On the other hand, CREEK excludes the relationships that does not exists in any of the maps and only uses the ratio of the number of relationships existing in both maps and the number of relatioships that exist only in one. CREEK would thus find the first two maps above much more similar than the second set of maps.

In Section 7-1, we identify cooperation among students as a possible error source when using machine-learning classifiers to predict which tasks a student would be able to solve. The problem is that if two students cooperate to the degree that they effectively produce two instances that are the same, a case-based reasoning in a leave-one-out cross validation methodology would always identify the cooperating student as the most similar case. This would of course give a case-based reasoner a perfect record on these cooperating students. Other machine-learning methods might not be as vulnerable to this as instance-based methods, but because our dataset has so many attributes, it is likely that generalizing methods would be influenced by cooperation. For instance, there may be attribute values unique to pairs of cooperating instances.

Unfortunately, we do not know which students cooperated, but we do know that the data collection for dataset 2 was done in two separate groups, and we know which students belong to what group. Because the students were limited to working on the exercises during the group sessions, we know that no one from group A cooperated with anyone from group B. This allows us to control for cooperation by dividing the second dataset in two along the group lines. By using the first the set representing group A as training data and then testing on group B, and then repeating the same process in reverse, we know that any effects here are not due to cooperation. Figure 7.20 shows the result of this experiment. As in Figure 7.18, the greatest differences can be seen on the tasks with the low baselines. CREEK is signficantly better than the baseline on task 1c ($t = 0.050$), 2b ($t = 0.025$) and 2c ($t < 0.0003$). Of the other classifiers, on task 2c, J48 ($t = 0.026$) and Naïve Bayes ($t = 0.026$) is significantly better than the baseline.

It should be noted, however, that the the baseline for the 1c, 2b and 2c tasks is very low. For instance, on task 1c, using the majority class from one group to predict how the other group will do, gets a correct answer only 20% of the time! This suggests that the populations of these groups are quite different in composition.

This result may give some indication that the effect observed in the leave-one-out cross validation experiment in Figure 7.18 is not solely the effect of cooperation, as for CREEK, the results in Figure 7.20 seem to mirror the earlier results, although with lower significance values. However, this is only an indication, and it is even less robust for the other machine learning methods we have tested. In particular, Naive Bayes, which did well on the leave-one-out cross validation but did not on this test.



This figure shows how the percentage of correct classifications found for each classifier on each task for dataset 2. Unlike the result presented in Figure 7.18, this test does not use a leave-one-out cross validation method, but divides the dataset in two so that different groups are used to train and test the classifier.

Figure 7.20: Classification on Tasks from Dataset 2, Divided by Group

7-4.2 Regression Tests

While the discrete e<Task>Solved variables were possible to predict using classifiers, the variables measuring time spent on a task and the composite competence measure requires regression methods. The competence measure was introduced in part as a more fine-grained measurement for the programming skill of a student on a complete exercise, and in part to allow us to use machine learning methods on tasks where a large majority of student succeeded in solving the task. In the classification tests, these are the tasks with very high baselines. In particular, we assumed this would be relevant for dataset 1, as when collecting this dataset, students were operating under the normal assumption that all tasks should be solved. Presumably this would lead to many tasks with very high baselines. The competence measure addresses this by using the active time spent by the student on the task, with the assumption that lower time spent before succeeding indicates a more skilled student. It might also be possible to predict the time spent on a task or the complete exercise, however, as we have shown, the time spent on the exercise is lowest at the extremes – i.e. the best and the worst students spends the least amount of time on an exercise, with the students in the middle spending more time. In order to form a more linear variable, we created the competence measure. Because this measure assigns a score of 0 on tasks that was not solved, the worst performing students would still get a lower score than average students that spent more time but managed to solve more tasks. This creates a more linear relationship between skill and the competence measure, which should be easier to model with standard statistical techniques.

Case-based reasoning and instance-based methods such as CREEK and IBk are able to do both classification and regression, but the other machine-learning methods we used in the classification tests were not able to handle regression tasks. Instead, we used MP5 (the Weka implementation of the M5 algorithm [61]), the REPTree fast decision tree learner, RegressionByDiscretization, AdditiveRegression and LinearRegression. All algorithms used the Weka defaults. In addition, we used IB1, IB3 and CREEK.

In Figure 7.21, the set of regression methods were used to predict the total time spent on the whole exercise in a leave-one-out cross validation method similar to that used in the classification test, while in Figure 7.22, the average competence on the whole exercise is predicted. Both of these figures are for dataset 2. These figures, as well as the other figures in this section, show the average error, so unlike the classification tests, lower values are better. As a baseline, the mean value for the training set was used as the prediction. In Figure 7.21, the M5P method is significantly better than the baseline ($t = 0.027$), with CREEK ($t = 0.057$) and RegressionByDiscretization ($t = 0.076$) approaching significantly better. However, no methods performs significantly different than the baseline on the average competence prediction shown in Figure 7.22. However, when we break this down to the individual tasks (Figure 7.23), we see significantly lower errors on task 2b for CREEK ($t = 0.028$), REPTree ($t = 0.002$), RegressionByDiscrtization ($t = 0.003$) and LinearRegression ($t =$

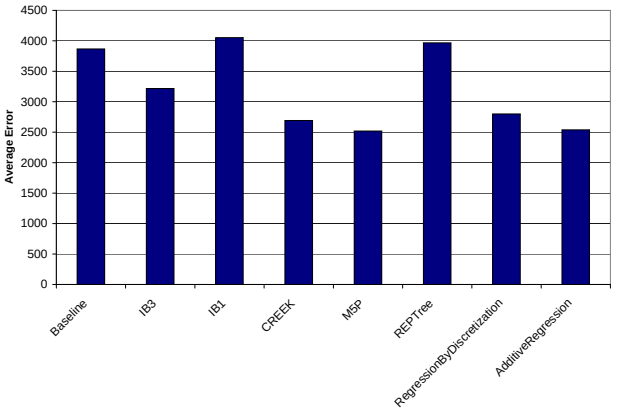


Figure 7.21: Regression on total time spent on exercise (totalTime) in dataset 2

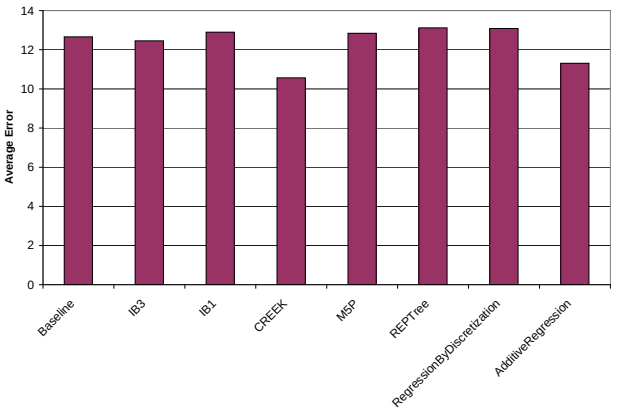


Figure 7.22: Regression on average competence (averageCompetence) on dataset 2

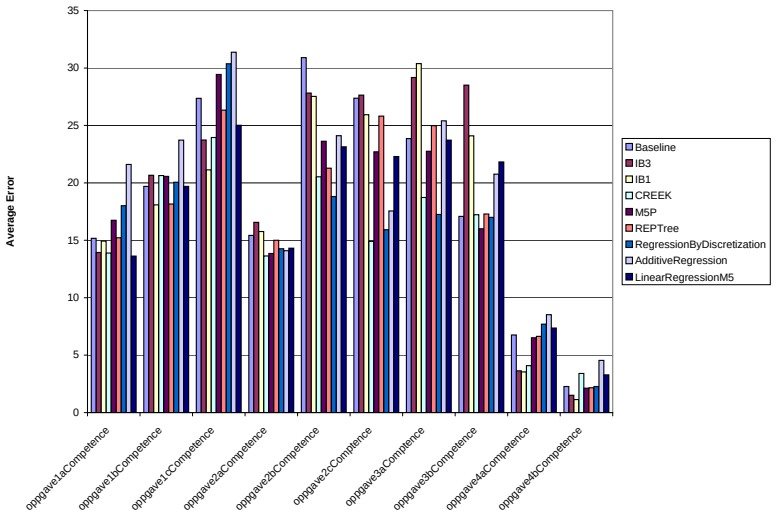


Figure 7.23: Regression on the competence variable on each task in dataset 2

0.013). On task 2c CREEK (t = 0.003), RegressionByDiscretization (t = 0.003), and AdditiveRegression (t = 0.009) are significantly better than the baseline, and on task 3a, RegressionByDiscretization is significantly better (t = 0.024). On several tasks, the IB1 and IB3 methods are significantly worse than the baseline.

However, all significant effects disappear when we instead of a leave-one-out cross validation method, divides the dataset along the group lines so as to control for cooperation. Figure 7.24 shows the regression on the total time spent on the exercise using this method, and Figure 7.25 shows the same for the competence measure on each task. As is apparent in these figures, the predictive effect observed in the earlier tests disappears completely. Where some of the corresponding classification tests showed effects that approached significance on the 5%-level in line with the effects we observed in the leave-one-out tests, these regression tests show no similar trend. Neither are there any significant effects for total time, average competence or task-specific competence on dataset 1.

The results of these tests suggest that we are unable to predict the finer-grained measure of competence that contains time information. The only effects observed are for those tasks with a low baseline where we also observed effects in classification, but the inclusion of the time component did not make it possible

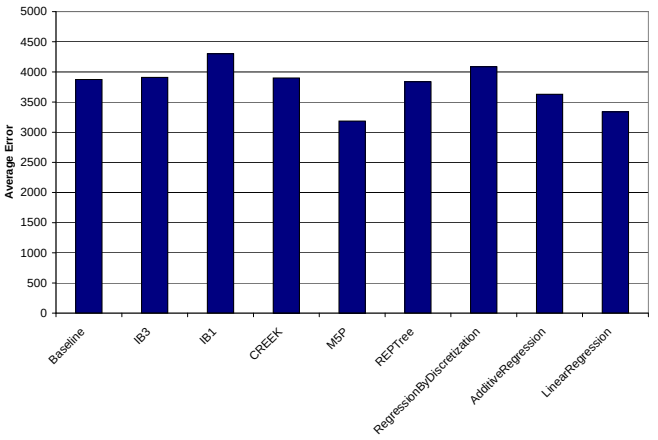


Figure 7.24: Regression on total time spent on exercise (totalTime) in dataset 2 divided by groups

to predict competence well on other tasks. To the contrary, inclusion of time data may have added a source of noise so that even the effect on the low-baseline tasks disappeared in the tests where the dataset were divided by group. While this does not mean that the competence measure or time information cannot be predicted on the basis of other student model components, it seems unlikely that it can be predicted using concept map information.

7-5 The Effect of Inference on Concept Maps

Of the three maps we have used in this experiment, the "Control Structure" map is most relevant for examining the effect of inference. This map uses the *type of* relation to structure concepts hierarchically and intuitively, this relation should be interpreted transitively. This means that if a concept map contains "A type of B", and "B type of C", it makes sense to interpret that "A type of C" as well. As we examined in Section 5-3.3, inferred relationships allows us to match the teacher map to students that for instance skipped the intermediate level by modeling "A type of C" directly. Without inference, this would be entirely dissimilar from the "A type of B type of C" model above. However, if inference is applied, the students "A type of C" relationship would match the inferred teacher relationship "A type of C", and the maps would match partially.

We would have liked to test this mechanism by repeating the experiments

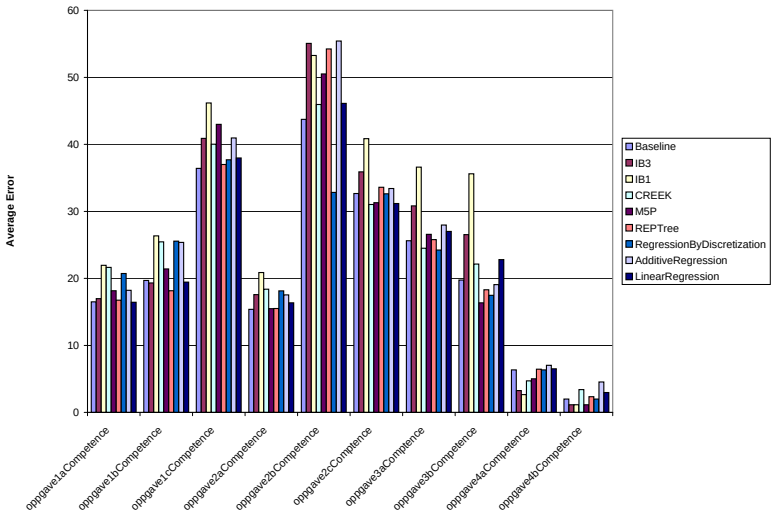


Figure 7.25: Regression on the competence variable on each task in dataset 2 divided by groups

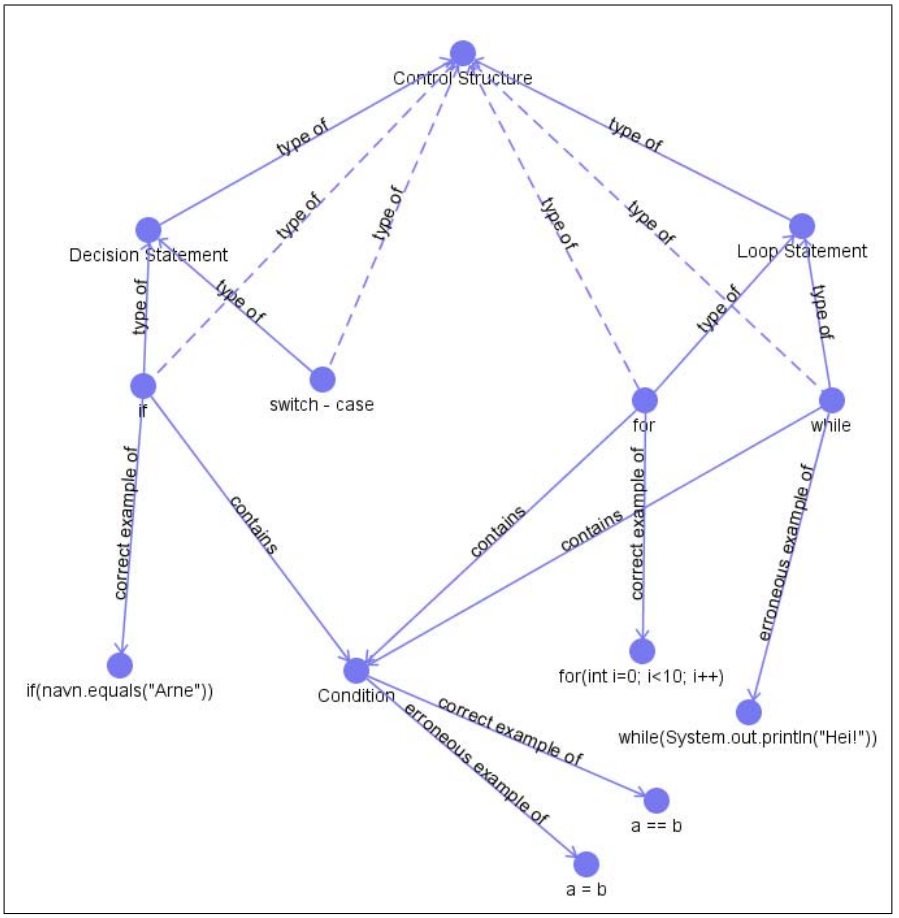


Figure 7.26: The teacher map for "Control Structures" from exercise 3, with inference.

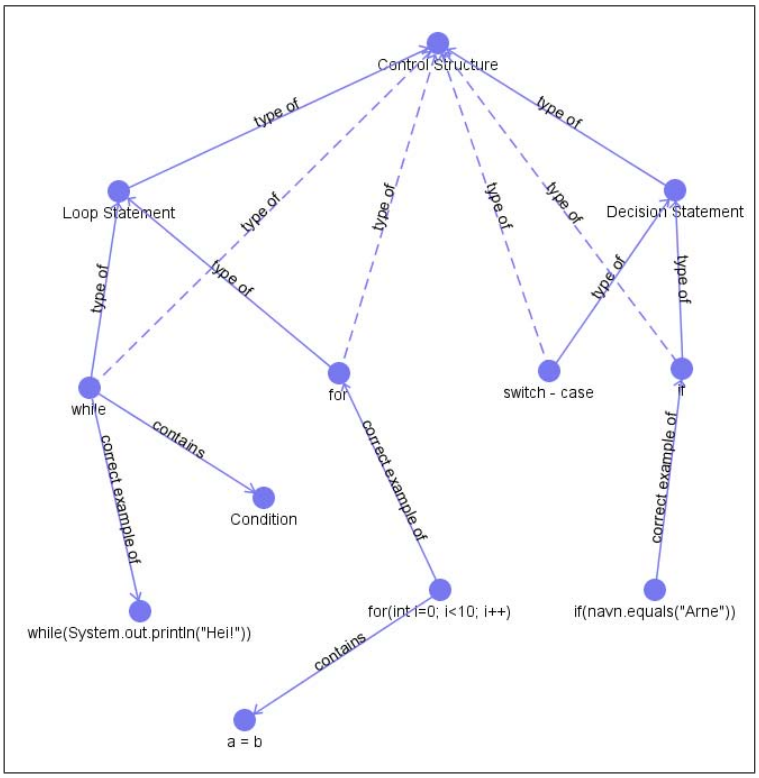


Figure 7.27: *Anne*'s student map.

above using inference and comparing them to the results where inference was not used. However, these maps were not originally designed with this experiment in mind, and as such the amount of inference that can be usefully applied is quite limited. This means that the effect of the inference should be low as well, and combined with the quite tenuous effects we have observed, we have concluded that we do not have the data to do quantitative analysis on this.

However, we can see some indications through descriptive statistics and by examining individual student maps. In order to do this, we applied the single inheritance rule that *type of* is transitive (i.e. *type of* inherits over *type of*). Figure 7.26 shows the teacher map for "Control Structures" with the four inferred relationships found by the inference mechanism (inferred relationships are marked as dotted lines in the figure).

The average similarity between student and teacher maps increased from 25,5% where no inference was used, to 31,0% after inference was applied. However, simply choosing a relationship at random and adding that relationship to all student maps would also increase the average similarity, so this on its own is not necessarily an indication of an interesting effect.

When we examine individual student maps we can see some nice examples

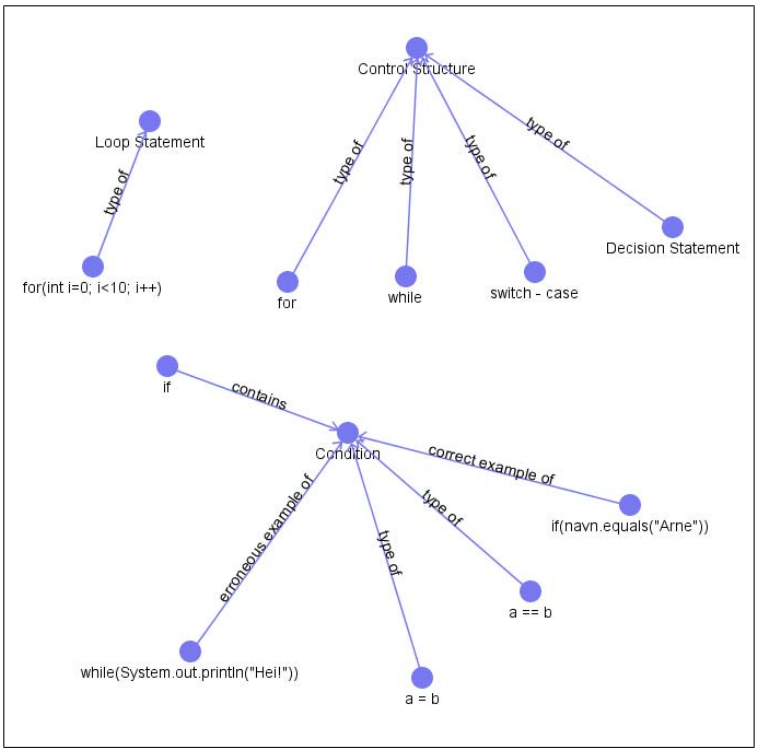


Figure 7.28: *Bill*'s student map.

of how this works out in practice. In Figure 7.27 we can see how the inference affects the map of "Anne"[5]. Her map is quite similar to the teacher map, and the inference mechanism adds the same four relationships as for the teacher. This does increase the similarity of the maps slightly, from 56% to 65%. This is the most common type of inference seen in the student maps – inferring some or all of the same relationships inferred in the teacher map.

In Bill's student map (7.28), the situation is different. The similarity to the teacher map is very low – only 4%. One of the things that lowers Bill's similarity score is that he has drawn direct relationships from the FOR, WHILE, and SWITCH-CASE concepts to the CONTROL STRUCTURE concept, while the teacher has an intermediate layer. Although the inference mechanism finds no additional relationships in Bill's map, the implicit *type of* relationships found in the teacher map matches Bill's relationships. This increases the similarity of the maps from 4% to 21%. Intuitively, this seems like a more accurate representation of Bill's conceptual knowledge.

A common mistake in the student's map is that they reverse the direction of the relationships. In Claire's student map (Figure 7.29), she has drawn a

[5] Although all student maps are real maps drawn by real students during the experiments, the student names are fictional and only used for ease of reference.

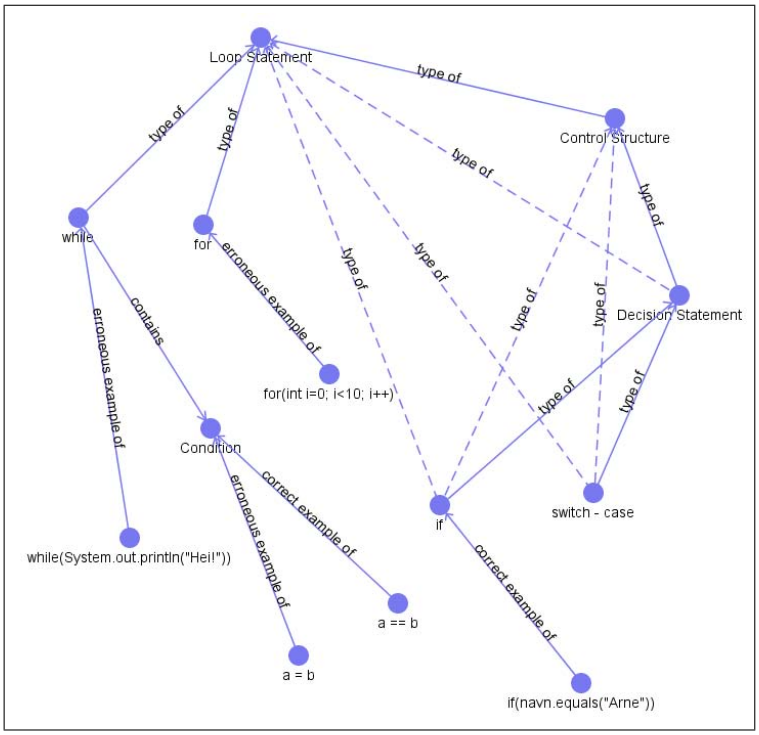


Figure 7.29: *Claire*'s student map.

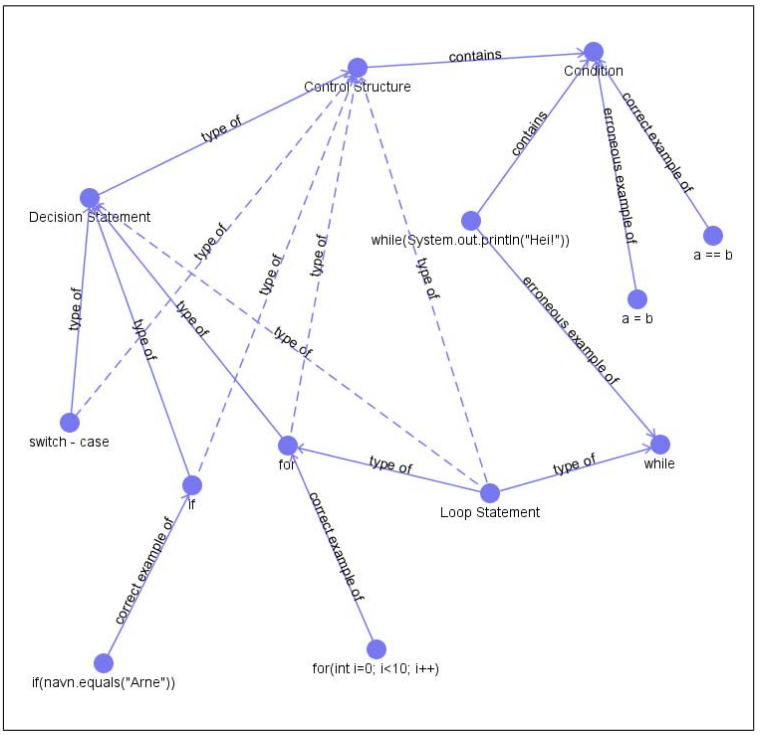


Figure 7.30: *Daniel*'s student map.

type of relationship from CONTROL STRUCTURE to LOOP STATEMENT. Claire seems to have been careful about the direction in which relationships are drawn elsewhere in the map, so this is likely a mistake. A part from this mistake, the map is quite similar to the teacher's – only a few *contains* relationships to the CONDITION concept is missing. Without inference, Claire's similarity is in fact 63%, which is among the highest in the dataset. However, when inference is applied, new relationships are inferred using the incorrect *type of* relationship, and with inference the similarity to the teacher map drops to 52%. In other words, the inference mechanism may magnify the effect of small mistakes. This is not the only example of this, but probably the most extreme. If this single relationship was inversed, Clarie would have seen an increase in similarity to the teacher if inference was applied. Instead, the similarity is decreased by 11%-points.

Daniel's student map (Figure 7.30) also have a few examples of relationships drawn in the wrong direction (the *type of* relationships from the LOOP concept), but in this map it does not affect the similarity as much. Curiously, it even allows the inference mechanism to conclude that LOOP is a *type of* CONTROL STRUCTURE – via FOR and DECISION STATEMENT! Another interesting feature of Daniel's map is that while the teacher map has the *contains* CONDITION

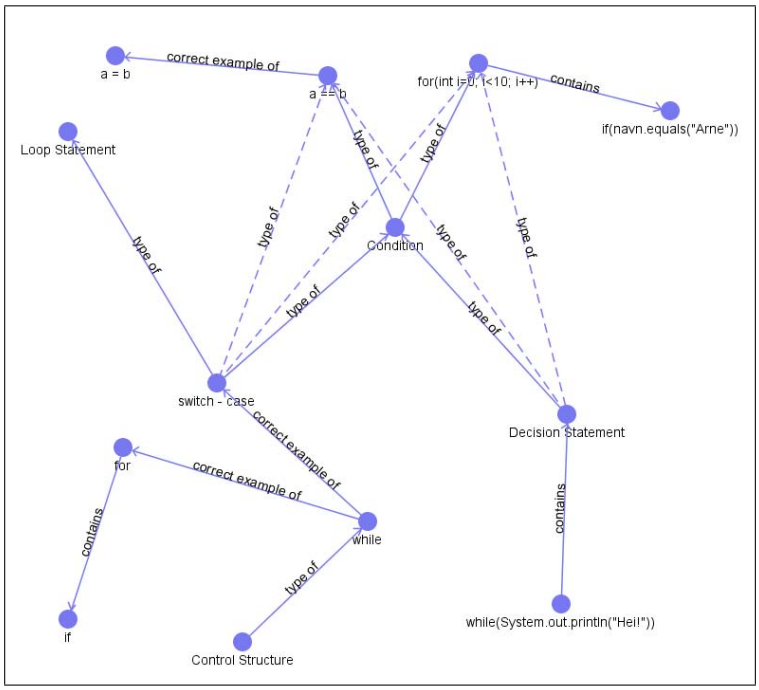


Figure 7.31: *Ellen*'s student map.

relationship from concepts such as FOR, IF and WHILE, Daniel has instead chosen to attach this to the superclass of these concepts – the CONTROL STRUCTURE. The reason this is not done in the teacher map is that it is not universially true – the SWITCH – CASE statement is an exception to this, but it is generally true. However, it might be argued that this representation by Daniel is generally correct, and that we should be able to infer that he meant this to be inherited down to e.g. FOR. Using a plausible rule saying that *contains* is inherited over *type of*, this would indeed have been possible, although we have not tried to extend the set of plausible inheritance rules used in this analysis.

In addition to reversing relationships, a fairly common mistake in this concept map task is that the students are not sure which relation types to use. In particular, some are confused by the separation of *type of* and *example of* relation types. One student created a nice map using only the *type of* relation type, and no others. Others say that FOR is a *correct example of* LOOP, or that IF(NAVN.EQUALS("ARNE")) is a *type of* IF. This is understandable in that while the *type of* relation type is a subclass relation, the *example of* relation types can be said to be kinds of specialized instancing relations. What is a subclass and what is an instance can often be hard to identify. One lesson from this seems to be that the relation types offered to the student should be as distinct from each other as possible.

The student maps examined this far has ranged from decent to quite good,

but we also see inference in maps that seem far more random. Ellen's map (Figure 7.31) is an example of this. It contains all the concepts offered and connects them all to an interconnected map, but seemingly in a random fashion. This also introduces some inferred relationships (the dotted lines), but this does not change that this map has a 0% similarity to the teacher map.

The use of inference seems to have both good and bad consequences, although in this case we believe the net effect is positive. In many situations, the same relationships were inferred in both student and teacher. However, in some situations, relationships inferred either in the student or teacher map reveals implicit relationships, which allows the inference mechanism to see similarity beyond the purely syntactic.

Chapter 8

Conclusion

We have presented a framework for analyzing and comparing exercise-oriented intelligent tutoring systems, which we have used to analyze six intelligent tutoring systems. These systems can be divided into two categories. In the first category, the ELM and PACT Cognitive Tutors attempt to model the student's procedural problem solving ability in order to identify and rectify "bugs". In the second category are systems that we call intelligent learning environments (ILEs); CATO, Ambre-AWP, and BLITS. These systems do not have a student model in the same sense as ELM and the PACT Cognitive Tutors. Instead, they seek to provide a problem-solving environment that is conductive to learning. To do this, they use general AI methods such as case-based reasoning to assist the student, as well as domain-specific measures.

The CREEK-ILE approach is closer to the second of these categories. It does not seek to build a complete procedural cognitive model of problem solving, but to create an intelligent learning environment. We suggest that some of the benefits of student modeling can be achieved without modeling at the procedural level. The CREEK-ILE system has knowledge of the student on the episodic level, through storing which exercise tasks a student has attempted to solve, and knowledge on the conceptual level, through student concept maps.

In the next sections, we will review the research goals presented in Section 1-2, and discuss how this work has contributed to these, and look at some future avenues of research.

8-1 Contributions

8-1.1 Student Modeling in Weak Theory Domains

Strong student modeling, in the sense that it is a procedural, cognitive model of the student's thought processes, is intractable for many, if not most domains. The domains chosen for the ELM and PACT cognitive tutors are domains where this seems to be possible, although with great effort. The idea that incomplete student models may still provide useful information is not new – a particularly

compelling technique is to use episodic knowledge in the form of earlier exercises to get some notion of what the student knows.

We have presented a method that combines concept maps and reasoning traces from earlier students to give partial support in in exercise selection, in-exercise support, conceptualization and explanation. This method, while it seeks to address weak theory domains similar to those addressed by CATO, Ambre-AWP and BLITS, goes beyond providing a learning environment for the student and include capabilities to assist the student actively and tailor the experience to him. We call our system CREEK-ILE, where the three last letters indicate that we view it as an intelligent learning environment, although our interpretation of this is that it also has tutoring capabilities, and requires a student model to achieve this.

8-1.2 Concept Maps as Student Models

Although concept maps and similar techniques have been used before in tutoring systems, we have not found any work that uses it for student modeling purposes.

Computational tractability

We have demonstrated that although concept maps may seem like complex structures, they need not be computationally expensive to compare, for instance in a case-based reasoning process, as long as they relate to a common teacher map that defines the concepts and relation labels. We claim that this limitation is both pedagogically and computationally useful. Pedagogically, it allows the student a large room of possibilities to form conceptualizations, but it constraints them to a particular topic decided by the teacher. The example student maps in Section 7-5 illustrate that this freedom does result in quite a wide variation in conceptualizations – including some that differ greatly from the teacher's model but still seem to make sense. Computationally, the limitation to concepts and relation types defined by the teacher allows us to avoid the intractable task of deciding whether two graphs are isomorph. We avoid this problem because the common list of concepts and relation labels create a direct mapping between the nodes of the graph.

Concept Maps as Procedural Skill Indicators

In Section 5-4.1 we postulate four methods for how concept maps can be used to predict competence on procedural skills. Of these, we have tested basic versions of the first two methods in Chapter 7. Here we found that there is only very weak correlation between a student map's similarity to the teacher map, and how well that student performs on procedural skill. Such a correlation is only found for some maps, and only on one or two measures of skill on specific tasks. The conclusion of this is that we cannot support the hypothesis that is the basis for Method 1 ("Model First, Compare to Teacher."). There may be some correlation there, but certainly not anything strong enough for practical use.

On method 2, however, the results ("Model First, Compare to Students.") are more positive. There are significant results that suggest that by comparing a new student's concept map to previous students' maps, some information can be gained about his procedural skill. These indications are not strong, in that they only appear for one of two datasets, and only on those tasks that best divide the student groups and are thus easier to improve over the baseline. However, this effect is robust when we control for spurious factors that would not be present in an online environment, such as cooperation. In the end, we cannot rely entirely on the statistical significance tests here. On the one hand, we have performed hundreds of significance tests on the datasets, and with a 5% significance threshold, which should yield some "significant" results by pure chance. While there are statistical techniques to compensate for this, the consistency of where the results show up should not be ignored either. For instance, both the classification and regression tests show quite good results for predicting competence on task 2b in the second dataset. This holds for many of the classifiers, and some level of effect remains even when we divide the dataset by group to control for cooperation. This consistency of results, more than any single significance test, leads us to conclude that there is an effect.

An important question is if the effect we have observed is strong enough for student modeling purposes. In fact the age of the students had almost as large a predictive effect on the procedural skill as the concept map. It is, after all, easier to ask for the student's age than to ask him to spend half an hour to create a concept map, if the only goal is to have some measure of the procedural skill. From this perspective, although we can find support for the hypothesis of method 2 ("Model First, Compare to Students."), it is not a practical approach to student modeling as it stands.

We have observed during these experiments that very few students seemed to have good conceptual models when they arrived at the exercise session. This is also reflected in the students' answers on the feedback questions. Although many had attended lectures, the students with good concept maps were generally those that took the time to study how the concepts were related by searching their textbooks. In other words, our experience fits well with how Schank describes the learning process. A result of this is that the two last methods ("Method 3: Model Repeatedly" and "Method 4: Exercise-Specific Models, Explicit Generalization") more tightly couple the concept mapping tasks with the procedural tasks. This also blurs the lines between the exercise selection, in-exercise, conceptualizations and explanation support tasks. For instance, we have earlier pointed out the practical problem in method 3 of asking the student to refine the same concept map between each task. To the student, this looks like he is asked to solve the same task again and again. He might be better equipped to solve it again after each task, but the boredom of revisiting a previously solved task will likely work against this approach. However, this might be avoided if the system can provide hints or ask learning questions related to the task just solved. A simple form of this, suggested in method 3, is that a part of the teacher map related to that task can be revealed, and the student asked to compare it to his own. A more complex variant of this is suggested in method

4, where the student is asked to form a concept map for each individual task, and is then given assistance to generalize the task-specific maps to a common concept map. Both of these methods may perhaps give a closer coupling to the procedural skill of the student, but they also assist in conceptualization, explanation and in the case of method 4, in-exercise support.

The Practical Value of Exercise Selection

The practical value of exercise selection may be questionable. In our experiments using programming exercises, we saw how the constructivist critique of the reductionist approach applies in practice. In our exercises, we used the same exercises as were given in the normal exercise groups. These exercises were constructed so that the student would go through a series of tasks that built on each other. Often, the result of the exercise would be one or two quite large programs, where the tasks simply defined sub goals and served to guide the student to the final destination. This is only possible if the path through the tasks is effectively pre-defined. If the tutoring system has the option of skipping tasks, or reordering them based on individual student needs, the tasks cannot be dependant on each other. Further, with many students using several hours to solve each task, tasks would likely have to be smaller in order to ensure that the student went through a meaningful number of iterations of exercise selection. It also seems that this is not the area where the student needs help the most. The motivation of the student is to solve the gatekeeper problems, and finish the exercise. This makes it hard to motivate the student to solve extra tasks, although the insight gained from solving this task may help in solving the gatekeeper problem. Such assistance is likely more efficiently done by offering examples relevant to the task at hand, as is by ELM.

Similar conclusions to the question of exercise selection have been reached by many of the other tutoring systems we have studied. Of the systems we have examined, only the PACT Cognitive Tutors does some exercise selection, but this is effectively limited to creating sets of exercises with increasing difficulty and only allowing the student to progress to a harder set once the system is satisfied the student is sufficiently skilled at the level of the current set.

In conclusion, methods 3 and 4 represent our current views for how concept maps should be used in exercise-oriented tutoring systems. We believe that the goal of exercise selection should be de-emphasized in favor of providing conceptualization, explanation and in-exercise support. In fact, we think that methods 3 and 4, although listed under exercise selection, are at least as useful for conceptualization, explanation and possibly in-exercise support, and can be used even if the task order is fixed in advance.

8-1.3 Concept Maps, Knowledge Representation and Inference

Typically, knowledge representations have sacrificed some ease of use in order to get a higher degree of semantic agreement between human and computer in

how the contents of the representation is interpreted. However, we suggest that semantic-net based representations like our CREEK knowledge representation may be used to store concept maps without compromising the ease of use. This allows us to attach some semantics to concept maps terms, which allows the use of inference to find implied knowledge. The inference mechanisms presented in this work will never understand the contents of a map on the same level as a human, but may be able to draw some of the same conclusions about implied knowledge. We suggest that this is useful for instance in comparing concept maps, where knowledge that is explicit in one map may be implicit in another.

In this work, we have provided a proof-of-concept that a knowledge representation language with inference can be used to store concept maps created by students with no prior training. In fact, the CREEK knowledge representation did not impose any further constraints on the students' ability to form concept maps. Because the CREEK plausible inheritance inference does not require global consistency, can handle loops in the graphs and operates in polynomial time, it may be used without placing strong requirements on student or teacher.

In Section 7-5 we illustrate how inference affects computational operation on the concept maps, in particular measuring the similarity between them. Although we were unable to do quantitative tests, qualitative analysis revealed that:

Inference may reveal useful, implicit knowledge. This is illustrated clearest in Bill's student map (Figure 7.28), where several relationships he models explicitly are found to match implicit relationships in the teacher's map, found through inference.

Inference may magnify mistakes. Because a single relationship in the concept map may allow multiple inferred relationships to be found, a single mistake may have much larger consequences than if no inference is used. In Claire's student map (Figure 7.29) a single reversed relationship causes the similarity of her map to drop 11%-points.

Computational analysis of concept maps may place too great a weight on minor mistakes. A generalization of the above finding is that any computational method that analyzes or uses concept map tends to be stricter in interpretation than a human. This causes minor mistakes or differences of interpretation to seem more important than a human normally would rate them. In addition to the example from Claire's map above, we saw that a confusion on when to use the *type of* and when to use the *... example of* relation types caused some students to be rated with lower similarity to the teacher. Here, some additional inference may actually help, for instance by recognizing that the *type of* and *... example of* relation types can be used to mean the same thing, or to recognize that sometimes a student may unintentionally draw a relationship in the wrong direction. Although the inference methods described

in this work do not support these particular kinds of conclusions, such methods should not be hard to create.

Earlier, we claimed that our knowledge representation and inference methods do not place additional constraints on the student and teacher. While this is true on a purely syntactical level, allowing the computer to find implicit relationships does have consequences, as we have seen above. This introduces issues on the semantic level. In particular, we found that it is useful to avoid having several relation types with overlapping semantics, in the sense that there may be situations where it is not clear which one to use. For instance, the semantic difference between *contains* and *type of* seemed clear to our students – we never saw anyone use *contains* where the teacher used *type of*. However, the confusion we saw between the *type of* and *... example of* relations seems to suggest that there is some semantic overlap between these relation types. The "reverse relationship" issue is also a result of the additional semantic importance given to the relationships, including their direction. This suggests that if inference is used on concept maps, greater care should be given to semantic issues when the concept mapping task is constructed. This can for instance mean that greater care is taken to make sure that the set of relation types used in the teacher map is semantically distinct.

The benefit of using inference on concept maps extends well beyond similarity comparisons, however. For instance, when the plausible inheritance mechanism is able to recognize that the direct FOR *type of* CONTROL STRUCTURE relationship in Bill's model matches the inferred relationship from the teacher's map, it allows the system to suggest concrete improvements to Bill's map. In this case, Bill's relationship represents a simplification of the more complex chain in the teacher's map (FOR *type of* LOOP STATEMENT and LOOP STATEMENT *type of* CONTROL STRUCTURE). Since Bill's relationship may be inferred from the teacher's relationship, the system can for instance suggest that Bill adds the LOOP STATEMENT concept to his concept map as an intermediate level. Alternatively, it can use the same knowledge to form a *learning question* tailored to his map, such as *"How would you say* LOOP STATEMENT *relates to* FOR *and* CONTROL STRUCTURE*?"*. Such concrete advice and learning questions are exactly what is required for the "Model Repeatedly" and "Exercise-Specific Models, Explicit Generalization" methods.

8-2 Future Directions

There are many questions that remain unanswered with regards to how and why concept maps may be useful in case-based tutoring systems, and our research has also identified some additional venues.

First, it is necessary to extend the experiments already conducted to control for such noise factors such as cooperation and cheating, and to develop our experiment design to do so. The current datasets are also limited in that they are based on only two exercises with only one or two teacher concept map each. Doing more experiments with more maps is necessary before any strong

conclusion can be drawn.

However, our experiments have also caused us to question, on a more fundamental level, how concept maps should be used in case-based tutoring. We have used a single concept map at the beginning of the exercise, but as we have seen, many students report that they do have little knowledge of the subject matter of the exercise before attempting to solve it. This is consistent with the theory that motivation for conceptual knowledge is found through solving exercises and that many students approach learning in this way. In Section 5-4, we suggest two refined methods, where concept maps are attached to each task in the exercise, and that the system provides support for generalizing these task-oriented maps into more generalized, conceptual models. These methods have not been tested yet, and represent a possible direction for future work.

Through our reported research, we have come to believe that the use of inference on concept maps is a promising venue of research. While concept maps should maintain their ease of use and focus on human expression, it is possible and useful to use limited inference to interpret implicit knowledge in concept maps. We have identified some potential uses, such as improved similarity measurement, and assistance in improving student concept maps gradually by forming concrete hints or learning questions. This work contains a proof-of-concept of inference on concept maps, and identifies some issues related to using inference on concept maps, but only dips its toes in the water when it comes to how concept map inference can be of use to computer systems.

Bibliography

[1] G. Aakvik, A. Aamodt, and I. Nordbø. A Knowledge Representation Framework Supporting Knowledge Modelling. In *Proceedings of the Fifth European Knowledge Acquisition for Knowledge-based Systems Workshop (EKAW-91)*. 1991.

[2] Agnar Aamodt. *A Knowledge-Intensive, Integrated Approach to Problem Solving and Sustained Learning*. PhD thesis, Norwegian Institute of Technology, Department of Computer Science, Trondheim, May 1991.

[3] Agnar Aamodt. Knowledge-Intensive Case-Based Reasoning in CREEK. In Peter Funk and Pedro A. González Calero, editors, *Advances in Case-Based Reasoning: Proceedings ECCBR 2004*, number 3155 in LNAI, pages 1–15, Berlin Heidelberg, 2004. Springer.

[4] Agnar Aamodt and Enric Plaza. Case-based reasoning; Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[5] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Mach. Learn.*, 6(1):37–66, 1991.

[6] F. N. Akhras and J. A. Self. Beyond intelligent tutoring systems: Situations, interactions, processes and affordances. *Instructional Science*, 30:1–30, 2002.

[7] V. Aleven. Using background knowledge in case-based legal reasoning: A computational model and an intelligent learning environment. *Artificial Intelligence*, 150:183–237, 2003.

[8] V. Aleven and K. D. Ashley. Teaching Case-Based Argumentation through a Model and Examples: Empirical Evaluation of an Intelligent Learning Environment. In B. du Boulay and R. Mizoguchi, editors, *Artificial Intelligence in Education, Proceedings of AI-ED 97 World Conference*, pages 87–94, Amsterdam, 1997. IOS Press.

[9] A. T. Corbett amd H. J. Trask, K. C. Scarpinatto, and W. S. Hadley. A formative evaluation of the PACT Algebra II Tutor: Support for simple hierarchical reasoning. In B. Goettl, H. Halff, C. Refield, and V. Shute,

editors, *Intelligent Tutoring Systems: Fourth International Conference, ITS'98*, New York, 1998. Springer.

[10] J. R. Anderson. *The Architecture of Cognition*. Harvard University Press, Cambridge, MA, 1983.

[11] J. R. Anderson, C. F. Boyle, A. T. Corbett, and M. W. Lewis. Cognitive Modeling and Intelligent Tutoring. *Artificial Intelligence*, 42:7–49, 1990.

[12] J. R. Anderson, F. G. Conrad, and A. T. Corbett. Skill Acquisition and the LISP Tutor. *Cognitive Science*, 13:467–505, 1989.

[13] J. R. Anderson, A. T. Corbett, K.R. Koedinger, and R. Pelletier. Cognitive tutors: Lessons learned. *Journal of the Learning Sciences*, 4:167–207, 1995.

[14] J. R. Anderson, R. Farrell, and R. Sauers. Learning to Program in LISP. *Cognitive Science*, 8:87–129, 1984.

[15] J. R. Anderson, J. G. Greeno, P. J. Kline, and D. M. Neves. Acquisition of problem-solving skill. In J. R. Anderson, editor, *Cognitive Skills and their acquisition*, pages 191–230. Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.

[16] J. R. Anderson and C. Lebiere. *The Atomic Components of Thought*. Lawrence Erlbbaum Associates, Mahwah, NJ, 1998.

[17] J. L. Arcos and R. López De Mántaras. An interactive case-based reasoning approach for generating expressive music. *Applied Intelligence*, 14(1):115–129, 2001.

[18] K. D. Ashley. *Modeling Legal Argument: Reasoning with cases and hypotheticals in HYPO*. MIT/Bradford Books, Cambridge, MA, 1990.

[19] K. D. Ashley. Reasoning with cases and hypotheticals in HYPO. *International Journal of Man-Machine Studies*, 34:753–796, 1991.

[20] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

[21] R. S. Baker, A. T. Corbett, and K. R. Koedinger. Detectinc Student Misuse of Intelligent Tutoring Systems. In J. C. Lester, R. M. Vicari, and F. Paraguaçu, editors, *Intelligent Tutoring Systems: Seventh International Conference, ITS'04*, pages 531–540, 2004.

[22] W. H. Bares, L. S. Zettlemoyer, and J. C. Lester. Habitable 3D Learning Environments for Situated Learning. In *ITS '98: Proceedings of the 4th International Conference on Intelligent Tutoring Systems*, pages 76–85, London, UK, 1998. Springer-Verlag.

[23] P. Boylan, A. Micarelli, V. Pirrottina, and F. Sciarrone. Constructivism, self-directed learning and case-based reasoners: a winning combination. *Learning How to Do Things/ Papers from the 2000 AAAI Fall Symposium. (Technical Report FS-00-02)*, pages 18–23, 2000.

[24] R.J. Brachman and J.G. Schmolze. An overview of of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9(2):171–216, 1985.

[25] J. D. Brown and R.R. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitve Science*, 2:155–191, 1978.

[26] J. D. Brown and R.R. Burton. Reactive learning environments for teaching electronic troubleshooting. In D. H. Sleeman and J. S. Brown, editors, *Advances in Man-Machine Systems Research*. JAI Press, Greenwich, Connecticut, 1986.

[27] J. D. Brown, R.R. Burton, and A. G. Bell. SOPHIE: a sophisticated instructional environment for teaching electronic troubleshooting. *BBN Report*, 2790, 1974.

[28] J. D. Brown, R.R. Burton, and A. G. Bell. SOPHIE: a step towards a reactive learning environment. *International Journal of Man-Machine Studies*, 7:675–696, 1975.

[29] J. D. Brown, R.R. Burton, and J. de Kleer. Pedagogical, natural language, and knowledge engineering techniques in SOPHIE I, II and III. In D. H. Sleeman and J. S. Brown, editors, *Intelligent Tutoring Systems*. Academic Press, London, 1982.

[30] J. R. Carbonell. AI in CAI: an arificial intelligence approach to computer-assistend instruction. *IEEE Transactions on Man-Machine Systems*, 11:190–202, 1970.

[31] B. Carr and I. P. Goldstein. Overlays. a theory of modeling for computer-aided instruction. *AI Lab Memo 406 (Logo Memo 40)*, 1977.

[32] P.A. Champin and C. Solnon. Measuring the Similarity of Labeled Graphs. In *Case-Based Reasoning Research and Development: Proceedings of ICCBR 2003*, pages 80–95, Trondheim, Norway, 2003. Springer.

[33] W. J. Clancey. The epistemology of a rule-based expert system: A framework for explanation. *Artificial Intelligence*, 20(3):215–251, 1983.

[34] W. J. Clancey. *Knowledge-based Tutoring: The GUIDEON Program*. MIT Press, Cambridge, MA, 1987.

[35] A. Corbett, M. McLaughlin, and K. C. Scarpinatto. Modeling Student Knowledge: Cognitive Tutors in High School and College. *User Modeling and User-Adapted Interaction*, 10:81–108, 2000.

[36] J. de Kleer and J. S. Brown. A physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[37] K. Forbus. Qualitative procecss theory. *Artificial Intelligence*, 24:85–168, 1984.

[38] P.P Gómez-Martín, M.A. Gómez-Martín, B. Díaz-Agudo, and P.A. González-Calero. Opportunities for CBR in Learning by Doing. In *Case-Based Reasoning Research and Development: Proceedings of ICCBR 2005*, pages 267–281, Berlin, 2005. Springer Verlag.

[39] Shirley Gregor and Izak Benbasat. Explanations From Intelligent Systems: Theoretical Foundations and Implications for Practice. *MIS Quarterly*, 23(4):497–530, 1999.

[40] N. Guin-Duclosson, S. Hean-Daubias, and S. Nogry. The Ambre ILE: How to Use Case-Based Reasoning to Teach Methods. In S.A. Cerri, G. Gouardères, and F. Paraguaçu, editors, *Intelligent Tutoring Systems: Sixth International Conference, ITS'02*, pages 789–791, Berlin Heidelberg, 2002. Springer-Verlag.

[41] L. W. Hawkes, S. J. Derry, and E. A. Rundensteiner. Individualized tutoring using an intelligent fuzzy temporal database. *International Journal of Man-Machine Studies*, 33:409–429, 1990.

[42] G.G. Hendrix. Expanding the utility of semantic networks through partitioning. In *Proceedings of the 4th International Conference on Artifical Intelligence*, pages 115–121, Cambridge, Mass., 1975. IJCAI.

[43] J. D. Hollan, E. L. Hutchins, and L. Weizman. STEAMER: an interactive inspectable simulation-based training system. *AI Magazine*, 5:15–27, 1984.

[44] K. R. Koedinger and J. R. Anderson. Effective use of intelligent software in high school math classrooms. In *Artificial Intelligence in Education, Proceedings of AI-ED 93 World Conference*, Charlottsville, VA, 1993. AACE.

[45] K. R. Koedinger and J. R. Anderson. Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education*, 8:30–43, 1997.

[46] K. R. Koedinger and K. Cross. Tutoring answer-explanation fosters learning with understanding. In S.P. Lajoie and M. Vivet, editors, *Artificial Intelligence in Education, Open Learning Enviornments: New Computational Technologies to Support Learning, Exploration, and Collaboration, Proceedings of AIED-99*, Amsterdam, 1999. IOS Press.

[47] D. B Leake, A. Maguitman, and A. Canas. Assessing Conceptual Similarity to Support Concept Mapping. In *Proceedings of the Fifteenth International Florida Artifical Intelligence Research Socity Conference (FLAIRS'02)*, pages 186–172, Menlo Park, 2002. AAAI Press.

[48] Ji-Ye Mao and Izak Benbasat. The Use of Explanations in Knowledge-Based System: Cognitive Perspectives and a Process-Tracing Analysis. *Journal of Managment Information Systems*, 17(2):153–179, 2000.

[49] J. E. McKendree. Effective feedback content for tutoring complex skills. *Human-Computer Interaction*, 5:281–413, 1990.

[50] T. Murray. Authoring Intelligent Tutoring Systems: An Analysis of the State of the Art. *International Journal of Artificial Intelligence in Education*, 10:98–129, 1999.

[51] Mark A. Musen. An overview of knowledge acquisition. In *Second generation expert systems*, pages 405–427, Secaucus, NJ, USA, 1993. Springer-Verlag New York, Inc.

[52] A. Ca nas, D. B Leake, and A. Maguitman. Combining Concept Mapping with CBR: Towards Experienced-Based Support for Knowledge Modeling. In *Proceedings of the Fortheenth International Florida Artifical Intelligence Research Socity Conference (FLAIRS'01)*, pages 286–290, Menlo Park, 2001. AAAI Press.

[53] A. J. Ca nas, J. W. Coffey, M. J. Carnot, P. F. Feltovich, R. R. Hoffman, J. Feltovich, and J. D. Novak. A Summary of Literature Pertaining to the Use of Concept Mapping Techniques and Technologies for Education and Performance Support. 2003.

[54] A. Newell. The Knowledge Level. *Artificial Intelligence*, 8:87–127, 1982.

[55] S. Nogry, S. Jean-Daubias, and N. Duclosson. ITS Evaluation in Classroom: The Case of Ambre-AWP. In J. C. Lester, R. M. Vicari, and F. Paraguaçu, editors, *Intelligent Tutoring Systems: Seventh International Conference, ITS'04*, pages 511–520, 2004.

[56] J. Novak and D. Gowin. *Learning how to Learn.* Cambridge University Press, New York, 1984.

[57] S. Ohlsson. Constraint-based student modeling. In J. E. Greer and G. I. McCalla, editors, *Student Modelling: The Key to Individualized Knowledge-Based Instruction*, NATO ASI Series, pages 127–146. Springer-Verlag, Berlin, 1990.

[58] H. F. O'Neil and D. C. D. Klein. Feasibility of Machine Scoring of Concept Maps. *CSE Technical Report*, 460, 1997.

[59] M. Papagni, V. Cirillo, A. Micarelli, and P. Boylan. Teaching through Case-Based Reasoning: An ITS Engin Applied to Business Communication. In *Proceedings of the 8th World Conference on Artificial Intellience in Education*, pages 111–118. IOS Press, 1997.

[60] S. J. Payne and H. R. Squibb. Understanding algebra errors: the psychological status of mal-rules. *CeRCLe Technical Report*, 43, 1987.

[61] J.R. Quinlan. Learning with Continous Classes. In *Proceedings of AI'92*, pages 343–348, Singapore, 1992. World Scientic.

[62] R.Davis, H. Shrobe, and P. Szolovits. What is a Knowledge Representation. *AI Magazine*, 14(1):17–33, 1993.

[63] M. Redmond. Educational implications of CELIA: Learning by observering and explaining. In *Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society*, Atlanta, GA, 1994. Lawrence Erbaum Associates.

[64] M. Redmond and S. Phillips. Encouraging self-explanation through case-based tutoring: a case study. In *Case-Based Reasoning Research and Development. Second International Conference on Case-Based Reasoning, ICCBR-97 Proceedings*, pages 132–43, Providence, RI, USA, 1997.

[65] M. A. Ruiz-Primo. Examining Concept Maps as an Assessment Tool. In *Proceedings of the First International Conference on Concept Mapping (CMC'04)*, 2004.

[66] R. C. Schank. CBR Meets Learning by Doing. In D. B. Leake, editor, *Case-Based Reasoning: Experiences Lessons & Future Directions*, pages 295–347. AAAI Press, Meno Park, CA, 1996.

[67] R. C. Schank and R. P. Abelson. *Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures.* Erlbaum, Hillsdale, NJ, 1977.

[68] R. C. Schank and C. Clearly. *Engines for Education.* Laerence Erlbaum Associates, 1995.

[69] Roger C. Schank. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge University Press, Cambridge, 1982.

[70] A. Seitz. A case-based methodology for planning individualized case oriented tutoring. In *Case-Based Reasoning Research and Development. Third International Conference on Case-Based Reasoning, ICCBR-99. Proceedings (LNAI Vol.1650)*, pages 318–328, Berlin, 1999. Springer.

[71] J. Self. Bypassing the Intractable Problem of Student Modeling. In C. Frasson and G. Gauthier, editors, *Intelligent Tutoring Systems: At the Corssroads of Artificial Intelligence and Education*, pages 107–123. Greenwood Publishing Group, Norwood, NJ, 1990.

[72] J. Self. The defining characteristics of intelligent tutoring systems research: ITSs care, precisely. *CeRCLe Technical Report*, 10:350–364, 1999.

[73] B. Selman. *Tractable Default Reasoning.* PhD thesis, University of Toronto, Toronto, 1991.

[74] L. Shastri. Semantic networks: a formalization. *Artificial Intelligence*, 39:283–355, 1980.

[75] M. E. A. Shiri, E. Aïmeur, and C. Frasson. SARA: A Case-Based Student Modelling System. In B. Smyth and P. Cunningham, editors, *Proceedings of the European Workshop on Case-Based Reasoning (EWCBR)*, pages 425–436, Berlin Heiderlberg, 1998. Springer-Verlag.

[76] F. Sørmo. Plausible Inheritance: Semantic Network Inference for Case-Based Reasoning, 1999.

[77] Frode Sørmo. Case-Based Student Modeling using Concept Maps. In *Case-Based Reasoning Research and Development: Proceedings of ICCBR 2005*, pages 492–506, Berlin, 2005. Springer Verlag.

[78] Frode Sørmo, Jörg Cassens, and Agnar Aamodt. Explanation in case-based reasoning—perspectives and goals. *Artificial Intelligence Review*, 24(2):109–143, 2005.

[79] R. J. Spiro, P. L. Feltovich, M. J. Jacobson, and R. L. Coulson. Cognitive flexibility, constructivism and hypertext: Random access instruction for advanced knowledge acquisition in ill-structured domains. *Educational Technology*, 35:24–33, 1991.

[80] R. J. Spiro and J. C. Jehng. Cognitive flexibility and hypertext: Theory and technology for the multidimensional traversal of complex subject matter. In D. Nic and R. J. Spiro, editors, *Cognition, education and multimedia: Exploring ideas in high technology.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1990.

[81] A. L. Stevens and A. Collins. The goal structure of a Socratic tutor. In *Proceedings of the National ACM Conference*, pages 256–263, New York, 1977. Association for Computing Machinery.

[82] W. R. Swartout. What Kind of Expert Should a System be? XPLAIN: A System for Creating and Explaining Expert Consulting Programs. *Artificial Intelligence*, 21:285–325, 1983.

[83] W. R. Swartout and S. W. Smoliar. On Making Expert Systems More Like Experts. *Expert Systems*, 4(3):196–207, 1987.

[84] D. S. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann, Los Altos, CA, 1986.

[85] M. Villano. Probabilistic student models: Bayesian belief networks and knowledge space theory. In *Intelligent Tutoring Systems: Second International Conference, ITS'92*, pages 491–498, Montreal, Canada, 1992.

[86] G. Weber. Episodic Learner Modeling. *Cognitive Science*, 20:195–236, 1996.

[87] G. Weber and P. Brusilovsky. ELM-ART: And Adaptive Versatile System for Web-based Instruction. *International Journal of Artificial Intelligence in Education*, 12:351–384, 2001.

[88] G. Weber, H.-C. Kuhl, and S. Weibelzahl. Developing adaptive internet based courses with authoring system NetCoach. In *Proceedings of Third workshop on Adaptive Hypertext and Hypermedia*, pages 35–48, Sonthofen, Germany, 2001. Technical University Endhoven.

[89] E. Wenger. *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge.* Morgan Kaufmann, Los Altos, CA, 1987.

[90] Barbara Y. White and John R. Frederiksen. Causal model progressions as a foundation for intelligent learning environments. *Artifical Intelligence*, 42(1):99–157, 1990.

[91] W. Winn. A Constructivist Critique of the Assumptions of Instructional Design. In T. M. Duffy, J. Lowyck, and D. H. Jonassen, editors, *Designing Environments for Constructive Learning*, pages 189–212. Springer, Berlin Heidelberg, 1990.

[92] I.H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques, 2nd Edition.* Morgan Kaufman, San Fransisco, USA, 2005.

[93] D. Zapata-Rivera and J. E. Greer. Inspecting and Visualizing Distributed Bayesian Student Models. In Gi. Gauthier, C. Frasson, and K. VanLehn, editors, *Intelligent Tutoring Systems: Fifth International Conference, ITS'00*, pages 545–553, Berlin Heidelberg, 2000. Springer-Verlag.

[94] D. Zapata-Rivera and J. E. Greer. Exploring Various Guidance Mechanisms to Support Interactions with Inspectable Learner. In S.A. Cerri, G. Gouardères, and F. Paraguaçu, editors, *Intelligent Tutoring Systems: Sixth International Conference, ITS'02*, pages 442–452, Berlin Heidelberg, 2002. Springer-Verlag.

Appendix A

Java Program Creating a Simple CREEK model.

This appendix contains a sample Java programs using CREEK to create a mode using the CREEK knowledge representation.

```
package jcreek.examples;

/**
 * A simple CREEK knowldege model.
 *
 * @author Frode Srmo
 */

import jcreek.util.CreekException;
import jcreek.representation.*;
import jcreek.representation.cbr.*;

public class SimpleModel
{
  public SimpleModel()
  {
  }

  public static void main(String argv[])
  {
    try
    {
      // Create a new CREEK KnowledgeModel. By default, this
      // is based on thebasic model created by the CBRModel class.
      LocalKnowledgeModel model = new LocalKnowledgeModel();
```

```
// Add concepts to represent types of findings. The age
// finding shouldbe a number.
Entity age = new Entity(model, "Age", "The age of the student");
age.addRelation("subclass of", "Number");

// Gender should be a symbol type, with male and
// female as possiblevalues.
Entity gender = new Entity(model, "Gender",
       "The gender (male/female) of the student");
gender.addRelation("subclass of", "Symbol");
Entity male = new Entity(model, "Male", "The male gender");
male.addRelation("instance of",gender);
Entity female = new Entity(model, "Female",
 "The female gender");
female.addRelation("instance of",gender);

// Create a few cases. These are also concepts in the
// model, and we coulduse the Entity class to make them,
// but it is more convinient to use the Case class. New
// cases made in this way are automaticallymade instances
// of the "Case" concept.
Case firstStudent = new Case(model, "Student Case #1",
                             "A male 19-year old");

firstStudent.addRelation("has finding", male);
NumberEntity firstStudentAge =
  new NumberEntity(model, new Integer(19), age);
firstStudent.addRelation("has finding", firstStudentAge);
firstStudent.setStatus(Case.UNSOLVEDCASE);

Case secondStudent = new Case(model, "Student Case #3",
                             "A female 23-year old");

secondStudent.addRelation("has finding", female);
NumberEntity secondStudentAge =
  new NumberEntity(model, new Integer(23), age);
secondStudent.addRelation("has finding", secondStudentAge);
secondStudent.setStatus(Case.SOLVEDCASE);

// Now, we print information about the firstStudent case
// to screen.
System.out.println("Concept name: "+firstStudent.getName());

// The basic getRelations() method uses the
// PlausibleInheritanceMethodto also find inherited
// relationships.
```

```
      Relation[] relationships = firstStudent.getRelations();
      for(int i=0;i<relationships.length;i++)
      {
        if (relationships[i].isInherited())
          System.out.println("  " + relationships[i].toString() +
                             " (inherited)");
        else
          System.out.println("  " + relationships[i].toString());
      }


      // The is saved as a binary .km file so it can be used later.
      model.saveAs("simpleModel.km");

    }
    catch(CreekException e)
    {
      e.printStackTrace();
    }
    catch(java.io.IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

Running this program, results in the output:

```
Concept name: Student Case #1
  Student Case #1 has finding Male (strength 0.5)
  Student Case #1 has finding NumberEntity#82 (strength 0.5)
  Student Case #1 has case status Unsolved Case (strength 1.0)
  Student Case #1 instance of Case (strength 0.9)
  Student Case #1 has comparator CaseComparison (strength 0.9)
(inherited)
```

Appendix B

Java Progam for Simple Case-Based Reasoning

This appendix contains a sample Java programs using the CREEK model created by the program in Appendix A to retrieve the best matching case for a new problem.

```
package jcreek.examples;

/**
 * A simple Case-Based Reasoning example, based on the SimpleModel.
 *
 * @author Frode Srmo
 */

import jcreek.util.CreekException;
import jcreek.representation.*;
import jcreek.representation.cbr.*;
import jcreek.reasoning.*;

public class SimpleCBR
{
  public SimpleCBR()
  {
  }

  public static void main(String argv[])
  {
    try
    {
      // Load the CREEK model created in the SimpleModel program.
      LocalKnowledgeModel model =
```

```
       new LocalKnowledgeModel("simpleModel.km");

      // Find the first student case in the model by name.
      Entity inputEntity = model.getEntity("Student Case #1");
      // We need it as a case instance, though. Below, we create
      // a new case instance that refer to the same concept - it
      // does not create a new concept.
      Case inputCase = new Case(inputEntity);

      // The RetrieveResult class runs retrieve by creating one
      // CaseComparisonbetween the input case and any solved cases
      // in the case base.
      RetrieveResult retrieve = new RetrieveResult(inputCase);

      // Print the number of comparisons that were made. In this
      // model, itshould only be two, as there is only two solved
      // cases - the "Student Case #2" and the "Student Case #3".
      System.out.println("We compared against "+
                  retrieve.getAllComparisons().length+" cases.\n");

      // Retrieve the CaseComparison for this match.
      CaseComparison bestMatch = retrieve.getBestComparison();

      // Printing information about this match to output.
      System.out.println("Best match: "+bestMatch.getTarget());
      System.out.println("Similarity: "+bestMatch.getStrength());
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Running this program, results in the output:

```
We compared against 2 cases.

Best match: Student Case #2
Similarity: 0.45
```