# Automatic Modeling of Procedural Knowledge and Feedback Generation in a Computer Science Tutoring System

BY

DAVIDE FOSSATI
M.S., University of Illinois at Chicago, 2003
M.S., Politecnico di Milano, Italy, 2004

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2009

Chicago, Illinois

## ACKNOWLEDGMENTS

This work is part of a larger project, carried on by the collaborative effort of the Intelligent Tutoring Systems Group at the University of Illinois at Chicago. The ITS-UIC group is directed by Prof. Barbara Di Eugenio (Department of Computer Science) and Prof. Stellan Ohlsson (Department of Psychology). Present and past students involved in this group include Lin Chen (Computer Science), Xin Lu (Computer Science), Jack Xie (Computer Science), Bettina Chow (Psychology), David Cosejo (Psychology), Andrew Corrigan-Halpern (Psychology), Trina Kershaw (Psychology), and Justin Oesterreich (Psychology).

Additional people have been involved in the transcription and annotation of the human tutoring dataset: Gibel Buena, Aidan Feldman, and Sarah Jeziorski.

The ITS-UIC group has a fruitful collaboration with the United States Naval Academy, in particular with Prof. Christopher Brown and Prof. Don Needham.

Prof. Bhaskar DasGupta, Prof. John Lillis, and Prof. Mitchell Theys at the University of Illinois at Chicago kindly helped with the collection of student learning data and some of the iList trials in their classrooms.

Thank you to the members of my thesis committee —Barbara Di Eugenio, Martha Evens, Susan Goldman, Tom Moher, and Stellan Ohlsson— and to those fellow graduate students not already mentioned above —Anushka Anand, Joel Booth, Nitin Jindal, Cindy Kersey, Saad Sheikh, Rajen Subba, and Alberto Tretti— for all the valuable comments and suggestions on this project.

## ACKNOWLEDGMENTS (Continued)

This work is supported by the Office of Naval Research (N00014–00–1–0640), the Graduate College of the University of Illinois at Chicago (2008/2009 Dean's Scholar Award), and in part by the National Science Foundation (ALT–0536968 and IIS–0133123).

DF

# TABLE OF CONTENTS

## TABLE OF CONTENTS (Continued)

**CHAPTER** **PAGE**

## TABLE OF CONTENTS (Continued)

**CHAPTER** **PAGE**

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| ANOVA | Analysis of Variance |
| CS | Computer Science |
| ITS | Intelligent Tutoring System |
| ML | Machine Learning |
| NL | Natural Language |
| NLP | Natural Language Processing |
| UIC | University of Illinois at Chicago |

## SUMMARY

This research takes place in the larger context of the study of one-on-one tutoring, a form of instruction that has been shown to be very effective. I conducted a study of human tutoring in the domain of Computer Science data structures, to understand which features and strategies of human tutoring are important for learning. I developed an Intelligent Tutoring System, iList, that helps students learn linked lists. One of the main advances in iList is the presence of a Procedural Knowledge Model automatically extracted from student data. This model allows iList to provide effective reactive and proactive procedural feedback while a student is solving a problem. I tested five different versions of iList, differing in the level of feedback they can provide, in multiple classrooms, with a total of more than 200 students. The evaluation study showed that iList is effective in helping students learn; students liked working with the system; and the feedback generated by the most sophisticated versions of the system is helpful in keeping the students on the right path.

# CHAPTER 1

# INTRODUCTION

## 1.1 Tutoring is effective

One-on-one tutoring has been shown to be a very effective form of instruction, compared to other educational settings, like traditional classroom-based information delivery (Bloom, 1984). For more than twenty years, researchers have worked on discovering the characteristics of tutoring. One of the goals of such research is to understand the strategies tutors use, in order to design effective learning environments and tools to support learning. Among the tools, particular attention has been given to Intelligent Tutoring Systems (ITSs), which are sophisticated software systems whose goal is to provide personalized instruction to students, in some respect similar to one-on-one tutoring (Shute and Psotka, 1996; Beck et al., 1996). Many of these systems have been shown to be effective (Evens and Michael, 2006; VanLehn et al., 2005; Di Eugenio et al., 2008; Mitrović et al., 2004; Person et al., 2001). In many experiments, ITSs induced learning gains higher than those measured in a classroom environment, but lower than those obtained with one-on-one interactions with human tutors. Therefore, the belief of the research community is that knowing more about human tutoring would be beneficial to the design of better ITSs.

In particular, this dissertation is mostly concerned with the following research questions:

1. What are the important features of student-tutor interaction, such as feedback, that correlate with student learning?
2. How can we automatically build a computational model of procedural knowledge that an Intelligent Tutoring System can use to provide interactive feedback and guidance?
3. How do different forms of tutorial feedback, implemented in an Intelligent Tutoring System, impact student learning?

The rest of this chapter presents some relevant background; a summary of how this thesis addresses these questions; and an outline of the rest of this document.

## 1.2 The problem of diminishing returns

Despite the great efforts of the research community in the past decades, there is still a lot to be discovered about tutoring. One of the facts that makes this research challenging is the presence of a variety of different theories and experimental results in the literature, sometimes even contradictory. For example, (Chi et al., 2001) argued in favor of self-explanation, leading to the prediction that effective tutors should just prompt students, leaving most of the talking to them; however, (Lu, 2007) found that, in her pool of tutors, the expert tutor that engendered the greatest learning was the one that talked the most. (Aleven and Koedinger, 2000) found that students did not make effective use of the on-demand help facility of the Geometry Cognitive Tutor; on the other hand, (Renkl, 2002) showed better learning in the domain of probability calculations when students interacted with a system that allowed on-demand help.

To provide a possible explanation for this phenomenon, we can hypothesize that there might be a non-linear relationship between the sophistication of the instruction provided to students and students' learning. In particular, we can apply the model of *diminishing returns* widely known in Economics (Johns and Fair, 1999). According to that model, when students pass from having no instruction at all to receiving some limited form of instruction, we observe a substantial improvement in their learning. However, as the sophistication of instruction increases, the differential improvement in learning does not increase as rapidly. After a certain point, further increasing the complexity of instruction would not make a substantial difference in students' learning. If this model holds, we can suspect that ITS research takes place in a somewhat high part of the curve (Figure 1), since ITSs are quite sophisticated learning environments. Since there are many other variables, besides instruction, that may affect learning, the data points constituted by individual research studies inevitably are affected by a high level of "noise" with respect to the variables of interest. These two facts, i.e., noisy data and position in the curve, can explain the contradictory results and the difficulty for the research community to come up with a clear picture.

Despite the difficulties, a lot of progress has been made in the past three decades. Hopefully, as more and more studies are being conducted, the increased number of data points will help the community better understand the complexity of instructional settings like tutoring.

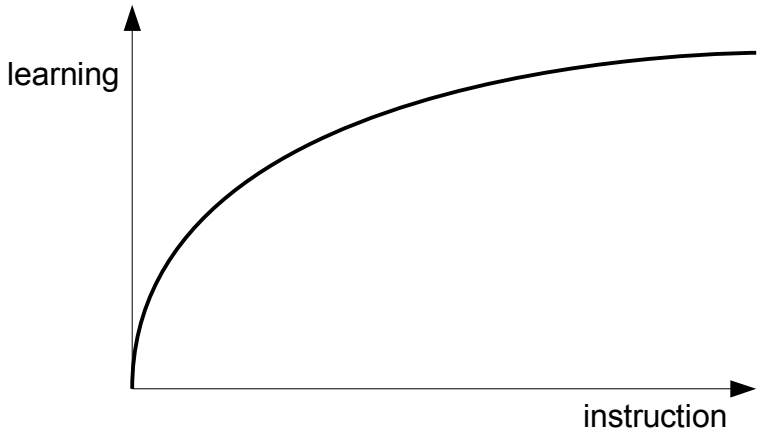



Figure 1. Hypothetical dependency between instruction and learning

### 1.3 Feedback in tutoring

One important characteristic of tutoring is the presence of *feedback*. There are many different forms of feedback that can take place in one-on-one tutorial interactions. Feedback can be provided by means of *verbal* or *non-verbal* communication. Verbal communication can be either spoken or written. Non-verbal communication includes, but is not limited to, body gestures, sounds, and pictures.

We can distinguish the *direction* of feedback, as feedback can be given to the tutor by the student or given to the student by the tutor. In the latter case we talk more specifically of *tutorial feedback*. The episodes of tutorial feedback can be of critical importance, because by means of feedback a tutor can provide targeted remediation to students' faulty knowledge

or reinforcement to students correct knowledge. We can divide tutorial feedback in two important categories: *negative feedback* and *positive feedback*.

- Negative feedback can be provided in response to students' mistakes. An effective usage of negative feedback would help the student correct that mistake and put him/her in the condition of not repeating the same (or a similar) mistake again, effectively providing a learning opportunity to the student. People can indeed learn a lot from making mistakes and correcting them (Ohlsson, 1996).
- Positive feedback is provided in response to some correct input from the student. Positive feedback can help students reinforce some correct knowledge they already have, or successfully integrate new correct knowledge, if the correct input provided by the student was originated by a random or tentative step (Ohlsson, 2008).

Notice that the distinction between positive and negative feedback depends on the correctness of the *content* that elicited the feedback, not on the surface *form* of the feedback message itself. For example, a negative feedback message can have a "positive" orientation for politeness reasons or to avoid discouraging the student.

More detailed characterizations of feedback have been reported in several studies, like the Human Tutoring Project (Fox, 1989; Fox, 1993a; Fox, 1993b) and the work of the CIRCSIM-Tutor group (Evens and Michael, 2006). Differences in tutoring behaviors with respect to feedback emerged from different studies. For example, tutors in the Human Tutoring Project tried to avoid giving direct negative feedback, opting more for questioning the students and providing hints. On the other hand, tutors in the CIRCSIM-Tutor group

tend to be more direct. Those differences might be influenced by a variety of factors, such as the subject domain, and, of course, tutors' individual characteristics.

Even though some form of positive feedback is present in many successful ITSs, the predominant type of feedback generated by those systems is negative feedback, as those systems are designed to react to student mistakes. To date, there is no systematic study of the role of positive feedback in Intelligent Tutoring Systems in the literature. However, there is an increasing amount of evidence that suggests that positive feedback may be very important in enhancing students' learning. In a detailed study in a controlled environment and domain, the letter pattern extrapolation task, (Corrigan-Halpern, 2006) found that subjects that were provided with positive feedback performed better in an assessment task than subjects that received negative feedback. In another study on the same domain, (Lu et al., 2008) found that the ratio of the positive over negative messages in her corpus of expert tutoring dialogues is about 4 to 1, and the ratio is even higher in the messages presented by her successful ITS modeled after an expert tutor, being about 9 to 1. In the human tutoring sessions that we have recorded as part of this research project, where the tutor is helping students solve problems involving Computer Science data structures, a completely different domain, we see a similar high ratio of positive to negative feedback messages, about 8 to 1. Although the overwhelming abundance of positive versus negative feedback messages per se does not imply that positive feedback is necessarily "better" than negative feedback, these findings certainly point out that positive feedback is an interesting subject, definitely worthy of further research.

In this dissertation, I will also make the distinction between *reactive feedback* and *proactive feedback*. Tutors give reactive feedback in response to statements or actions freely initiated by the student. Proactive feedback, instead, is given by the tutor in response to student statements or actions that were elicited by the tutor with a prompt or a question.

What might be the educational value of positive feedback in Intelligent Tutoring Systems? First of all, positive feedback may be seen as an effective motivational technique (Lepper et al., 1993; Lepper et al., 1997). Also, positive feedback can serve to more specific cognitive purposes. According to the ACT-R theory (Anderson, 1993), complex cognitive skills can be decomposed into smaller units, called *production rules*. Production rules are learned from examples and subsequently applied to new tasks by analogy. In order to improve learning, for example, in a problem solving setting, a tutor should help a student turn a new problem into an example from which he/she can acquire new production rules by analogy. During this process, the student can make a tentative step towards the correct solution. At this point, positive feedback from the tutor may be important in helping the student consolidate this step and learn new production rules from it (Ohlsson, 2008). Some researchers outlined the importance of self-explanation in learning (Chi, 1996; Renkl, 2002). Positive feedback has the potential to improve self-explanation, in terms of both quantity and effectiveness. The motivational boost provided by positive feedback may encourage students to engage in more self-explanation. Furthermore, positive feedback maximizes the benefits of self-explanation, according to the tentative step consolidation hypothesis. Another issue is related to how students perceive and accept feedback (Weaver, 2006), and, in

the case of automated tutoring systems, to whether students read feedback messages at all (Heift, 2001). Positive feedback might also play a role in making students more willing to accept help and advice from the tutor.

## 1.4 Research contributions

This thesis addresses the following research questions:

1. What are the important features of student-tutor interaction, such as feedback, that correlate with student learning?
2. How can we automatically build a computational model of procedural knowledge that an Intelligent Tutoring System can use to provide interactive feedback and guidance?
3. How do different forms of tutorial feedback, implemented in an Intelligent Tutoring System, impact student learning?

To address question (1), I conducted a study of one-on-one human tutoring in the domain of Computer Science data structures. I contributed to the collection, transcription, annotation, and analysis of a corpus of 54 tutoring sessions. One of the key results of this study is the importance of positive feedback resulting from a proactive interaction originating from the tutor. This result formed the evidence for the most sophisticated form of feedback implemented in the Intelligent Tutoring System, iList, presented in this dissertation.

To address question (2), I first conducted a number of classroom trials with the iList system and collected data on the interaction of students with the system. Then, I adopted

an Educational Data Mining approach to build an innovative statistical model of the solution paths taken by the students. This model allows iList to make inferences about student problem solving behavior, and generate appropriate responses. Importantly, the methodology I devised for building this model could be applied to a variety of other domains, wherever the concepts of "state space" and "actions that change the state space" can be defined.

To address question (3), I developed five versions of the iList tutoring system that differ in the level of feedback they can provide to the students. I tested the five versions of iList in multiple classrooms, with a total of more than 200 students, and compared them with a group of students working with human tutors as well. I analyzed the result of the iList trials in terms of student learning, student satisfaction, and interaction features. The main results show that students working with iList learn as well as with human tutors, with a positive trend in favor of the versions of the system with more sophisticated, human inspired feedback. Students liked working with iList and found it helpful, and judged the more sophisticated feedback more useful but also more difficult to understand and potentially misleading. The analysis of the interaction between iList and the students revealed that solving problems with iList is indeed positively correlated with learning; that solving those problems on a "better path" is also positively correlated with learning; and that the more sophisticated feedback provided by iList guides students towards "better paths."

In summary, this dissertation provides a number of theoretical and practical contributions to the community:

- I contributed to the collection, transcription, annotation, and analysis of a corpus of human tutoring in the domain of Computer Science data structures.
- I developed a tool, CSCoding, that facilitates the annotation process of the tutoring corpus with the synchronization of text, audio, and video.
- I developed a tutoring system, iList, that helps students learn linked lists.
- I developed a methodology to automatically extract procedural knowledge from a corpus of student interactions with iList.
- I developed multiple types of feedback generation for iList, the most sophisticated of those being inspired from the previous analysis of human tutoring.
- I deployed five different versions of iList in multiple classrooms, with more than 200 students, that learned on average as well as human tutors.
- I analyzed the result of the various iList trials in terms of student learning, student satisfaction, and interaction features, uncovering insights on the important features that make iList an effective system.

## 1.5 Outline

This dissertation is organized as follows:

- Chapter 2 describes the specific tutoring domain subject of this study, introductory data structures and algorithms in Computer Science.

- Chapter 3 presents a study of human tutoring. The study involved data collection, annotation, and statistical analysis based on group-wise comparisons and linear regression.
- Chapter 4 illustrates a new Intelligent Tutoring System in the domain of linked lists, a fundamental Computer Science data structure.
- Chapter 5 describes an innovative Procedural Knowledge Model automatically built from student interactions with the system.
- Chapter 6 includes a thorough evaluation of the system, comparing five different versions of the system with a control group and a group of students tutored by human tutors.
- Chapter 7 draws some conclusions and outlines the future directions of this study, as well as long-term research plans.

# CHAPTER 2

# COMPUTER SCIENCE AS A DOMAIN

## 2.1 Motivation

Like other forms of instruction, one-on-one tutoring can have characteristics that are dependent on the subject domain. Among many different disciplines, basic Computer Science has received only little attention from the educational and ITS research communities. Existing work focuses primarily on computer literacy (Graesser et al., 2005), programming languages such as Pascal (Soloway et al., 1981), Lisp (Corbett and Anderson, 1990; Chan et al., 2002), C++ (Kumar, 2002), Java (Sykes and Franek, 2003), general programming and design skills (Lane and VanLehn, 2003), databases (Mitrović et al., 2004), and special topics such as search algorithms used in Artificial Intelligence (Kalayar et al., 2001). Although the previous list might seem long, one fundamental topic has almost been neglected: basic data structures and algorithms, which are in the core of CS undergraduate curricula (AA.VV., 2008), and have been identified as difficult concepts for students to master (Katz et al., 2003). This research focuses on the tutoring of basic data structures, specifically on *linked lists*, *stacks*, and *binary search trees*.

## 2.2 Basic concepts

Almost any computer application needs to *store*, *retrieve*, and *process* information. To do so, computer scientists have devised ways to formally represent how a machine should put

the information in memory and how such information should be accessed and processed. Although these formal representations are somewhat influenced by the architecture of current computers, they provide a higher level view on the information processing task. Information storage representations are usually called *data structures*. The procedures describing how the information is processed are known as *algorithms*.

## 2.3 Linked lists

The main idea behind linked lists is that different pieces of homogeneous information can be "linked" one after each other and then accessed sequentially. The unit of information in a linked list is called a *node*. A node contains the *data* that has to be stored and a *pointer* to the following node. A *pointer* is an abstraction of the physical location of a node in a computer's memory. To retrieve the information contained in a node, it is necessary to "follow the pointer." A list starts with a pointer to the first node, called a *header*. The end of a list is indicated by the *null* pointer.

A common graphical representation of linked lists makes use of boxes and arrows. A box represents a node; it is divided to two parts, one representing the information fields, the other representing the pointer to the next node. An arrow, starting from the pointer part of a box and ending at another box, represents the link between two nodes. For example, Figure 2 shows a graphical representation of a linked list of grocery store items.

For any popular data structure there is a basic set of operations that are commonly defined to effectively use the data structure itself. More complex and application-dependent operations can be built on top of the basic ones. For linked lists, the most basic operations

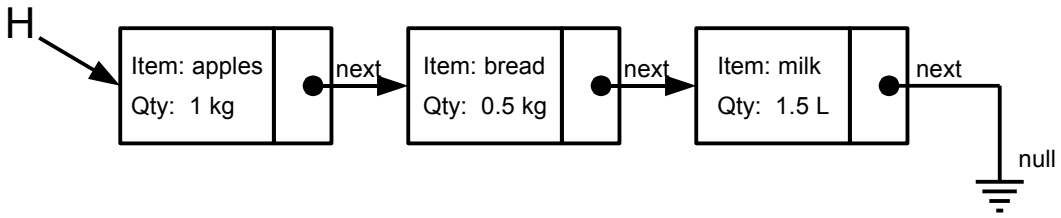



Figure 2. A linked list of grocery items

are *traversal* of the list, *insertion* of a new node in the list, and *deletion* of an existing node from the list.

## 2.4 Stacks

Like linked lists, stacks are linear structures, in which the information is stored and accessed sequentially. In fact, actual implementations of stacks are often built on top of linked lists. The main concept that characterizes a stack is the strict *policy* that has to be followed when storing and accessing the information, which is commonly referred to as the *last in, first out* (LIFO) policy. In a stack, the first element that can be retrieved, or *popped*, is always the last element that has been inserted, or *pushed*. The last element that has been pushed onto a stack is referred to as the *top* of the stack. Thus, a stack allows three fundamental operations:

- `push(element)` — Inserts an element on the top of the stack.

- `pop()` — Retrieves the element currently on the top of the stack, and removes it from the stack. It is illegal to pop an empty stack.
- `top()` or `peek()` — Retrieves the element currently on the top of the stack, without removing it from the stack. This is not really a fundamental operation, since its effect can be also obtained by popping an element and pushing it back onto the stack.

## 2.5 Binary search trees

Unlike linked lists and stacks, which are linear structures, binary search trees are two-dimensional structures. A graphical representation of a binary search tree can be seen in Figure 3. A typical node of a general binary tree is composed of three parts: the main information to be stored, and two pointers to two other nodes, called *left child* and *right child*. If a node does not have one or both children, the corresponding pointers are set to *null*.

The main property that differentiates binary search trees from general binary trees is that nodes in a binary search trees should respect an *ordering property*. First of all, there should be a total ordering relation among the content of the nodes, like the lexicographic order in the example of Figure 3. Then, for each node, the content of *all* the nodes in the *left* sub-tree must be *smaller* than the content of the node itself, and the content of *all* the nodes in the *right* sub-tree must be *greater* than the content of the reference node.

There are many operations that can be performed on a binary search tree. The fundamental ones include *searching* for a node in the tree, which can be done very efficiently thanks to the ordering property of the structure, *inserting* a new node in the tree, and

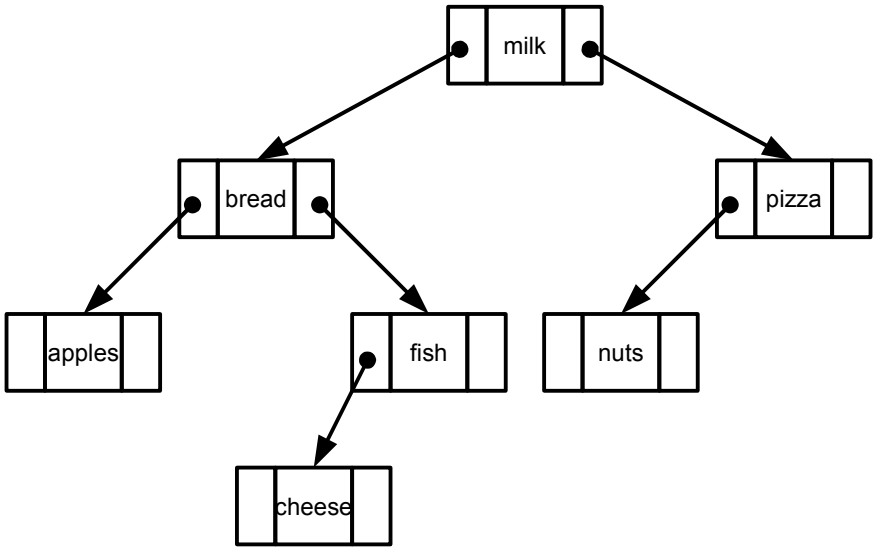



Figure 3. A binary search tree of grocery items

*removing* an existing node from the tree. Insertion and removal should be done carefully, in order to maintain the ordering property of the tree.

## 2.6 Reflections on learning and teaching

There are several issues that make learning and teaching this subject difficult.

1. What is the appropriate level of detail of an explanation? Because of the nature of the topics, high level explanations tend to be confusing. In order to understand the subject, detailed examples should be provided. However, examples expose the students to an amount of detail that can be overwhelming.

2. Top-down or bottom-up approach? A top-down approach would explain the general concepts before presenting more details and examples/exercises, whereas a bottom-up approach would start from examples and proceed with generalizations from there. A problem with the top-down approach, which is the most widely used in classrooms, is that students could understand nothing before facing the detailed examples, therefore "wasting" any previous explanation. A problem with the bottom-up approach, used mainly by self-learners, is that students can get stuck on details that do not help the abstraction/generalization process. So, what is the best mix of these two approaches, and how can we implement it?

3. Language and representation issues. When talking about data structures and algorithms, we can use at least four possible descriptions.

   (a) Natural language descriptions. Although they are easily accessible and require minimal background to be understood, natural language descriptions are way too imprecise. In order to make those descriptions more precise, often long and complicated sentences have to be used, so the advantage of simplicity is lost.

   (b) Graphical descriptions. These representations are powerful and intuitive, but have some disadvantages too. First of all, they can usually represent only static situations. If we want to use pictures to represent dynamic procedures, we have to provide a picture for each step of the procedure, ending up with very lengthy representations. Another disadvantage is that pictures represent particular in-

stances of a structure/procedure, which can make it difficult to abstract away to the most general cases.

(c) Programming languages can provide perfect descriptions of data structures and algorithms. The first disadvantage is that a student is supposed to have a quite extensive previous knowledge of the particular programming language chosen. Another big disadvantage is that programming languages introduce a lot of details that are essential to the correct execution of a program, but that are irrelevant to the task of teaching/learning the general subject. Students can get stuck in these details and fail to capture the underlying concepts.

(d) Pseudo-code descriptions are something in between programming languages and natural language descriptions. They use the most common constructs of programming languages (assignments, loops, conditional branches, etc.) in a more informal way, leaving out many details that are considered irrelevant to the description of the structure/algorithm. Descriptions in pseudo-code can combine the advantages of both natural and programming languages. However, they can also inherit the disadvantages of both worlds.

Since there is no "best" description, usually more than one of them is used at the same time. Of course, the multiplicity of languages and descriptions introduce another difficulty for the students, especially when the semantics of these languages is not well known or misunderstood. The issue of representation is known to the research community, especially in the field of *algorithm visualization*. Algorithm visualization is

a technology used to graphically show how algorithms, which are dynamic procedures, work. Empirical studies showed that students using it learn more only when they are actively engaged in the manipulation of the representation, not just when they passively watch it (Hundhausen et al., 2002). This finding is consistent with other research in Cognitive Science, which points to the same conclusion (Goldman, 2003).

# CHAPTER 3

# A STUDY OF HUMAN TUTORING

## 3.1 Motivation

This chapter presents a study of one-on-one tutoring in the domain of basic Computer Science data structures. This study has two primary goals. On the one hand, we want to get a better understanding of why tutoring is effective, and figure out what are the features of human tutoring that are mostly correlated with learning outcomes. On the other hand, we want to find useful guidelines and principles to direct the design and implementation of effective Intelligent Tutoring Systems.

The rest of this chapter describes the data collection and its analysis based on ANOVA and multiple regression. The main results of this study point out the importance of feedback, in particular positive feedback, and suggest the implementation of a proactive tutoring strategy in which feedback episodes are initiated by the tutor.

## 3.2 Data collection

Over the time span of three semesters, I contributed to collecting a corpus of data composed of a total of 54 one-on-one tutoring sessions in the domain of basic Computer Science data structures.

Each individual student participated in only one tutoring session, with a tutor randomly assigned from a pool of two tutors. One of the tutors is an experienced Computer Science

professor (tutor code: LOW), with more than 30 years of teaching experience. The other tutor is a senior undergraduate student in Computer Science (tutor code: JAC), with only one semester of previous tutoring experience. Overall, 30 sessions were run with tutor LOW, and 24 sessions with tutor JAC. 6 of the sessions with tutor LOW have been conducted in a slightly different setting than the remaining ones, but the resulting data has been made comparable with the rest of the corpus.

The topics covered during the sessions are linked lists, stacks, and binary search trees. Not all of the topics have been necessarily covered in every session, but the majority of the sessions include all three of them.

Each tutoring session lasts approximately 40 minutes. The time spent on each individual topic was not predefined, but was determined by the tutor case by case. Also the order in which the topics were presented was chosen by the tutor for each individual session. Most of the time, the tutors finished discussing a topic before starting the next one. In some cases, tutors got back to previously discussed topics at the end of the session, if they realized they had some additional time left. Table I shows some statistics about the length of the corpus, divided by topic.

Each tutoring session was videotaped. The camera was pointing at the sheets of paper on which tutors and students were writing during the session. The videos were all transcribed. The transcripts were produced according to the rules and conventions described in the transcription manual of the CHILDES project (MacWhinney, 2000). Additionally, they were enriched with timestamps at the beginning of each utterance, to keep track of the

TABLE I

LENGTH OF THE CS TUTORING CORPUS

| Topic | N | Session length (minutes) | | | | |
|---|---|---|---|---|---|---|
| | | Min | Max | Total | $\mu$ | $\sigma$ |
| List | 52 | 3.4 | 41.4 | 750.4 | 14.4 | 5.8 |
| Stack | 46 | 0.3 | 9.4 | 264.5 | 5.8 | 1.8 |
| Tree | 53 | 9.1 | 40.0 | 1017.6 | 19.2 | 6.6 |
| All | 54 | 12.8 | 61.1 | 2032.5 | 37.6 | 6.1 |

temporal position of the utterance in the video recording. An utterance is a natural unit of speech bounded by breaths or pauses. Figure 4 shows an excerpt from one of the transcripts.

In order to assess the learning that occurred during a session, each student was asked to take a pre-test right before the tutoring session and a post-test immediately after. The pre-test was composed of 2 problems on linked lists, 2 problems on stacks, and 4 problems on binary search trees (see Appendix A). The post-test was identical to the pre-test. The student was allowed 15 minutes to complete each test. The tutor could not see the pre-test, but he was given a short report on the student's performance on the test right before the beginning of the session. The reason for this procedure was to provide the tutor with some information about the student's prior knowledge, without biasing his teaching towards the specific test.

Pre and post tests were graded on a scale from 0 to 5 for each problem, for a total of maximum 40 points. The grading process was done blindly by two independent graders (I

```
[00:00:00.17]*JAC:  okay.
[00:00:02.21]*JAC:  alright first thing um I'm going to go over with you is uh
                    stacks.
[00:00:08.24]*JAC:  uh when we think of the stack you can think of it as um either
                    like laying bricks or like Legos # anything you can stack on
                    top of each other.
[00:00:19.23]*259:  bottom up.
[00:00:20.12]*JAC:  right, exactly.
[00:00:22.13]*JAC:  so # what we do with a stack is every time we put information
                    in it we put in on the top.
[00:00:31.22]*JAC:  I'm going to use numbers because numbers are simplest so if I
                    put the number nine it goes there.
[00:00:38.07]*JAC:  now if I want to put eight it goes immediately on top # it
                    goes on top of it.
[00:00:44.09]*JAC:  if I had seven it goes on top of that.
[00:00:47.14]*JAC:  if I want to add five it goes on top of that.
[00:00:50.12]*JAC:  if I want to add six it goes on top of that, okay?
[00:00:55.10]*JAC:  simple enough, right?
[00:00:56.20]*JAC:  always +// every time I put something new it always goes on
                    top.
[00:01:01.23]*JAC:  in order to put something on the stack we have an operation
                    which is called push.
[00:01:08.08]*JAC:  push always takes a value so in our case we would first push
                    nine followed by push eight, followed by push seven
[00:01:22.13]*JAC:  right so, okay?
[00:01:25.00]*JAC:  until we got to push six.
[00:01:26.26]*JAC:  now if I ever want to get the value of something on the stack
                    I only have one way to do that and that's called top.
[00:01:38.03]*JAC:  top doesn't take any value # it only returns a value.
[00:01:41.14]*JAC:  currently with the way we have it now top would give me the
                    value of six, okay?
[00:01:49.14]*JAC:  top only gives you the value that's on top.
[00:01:53.03]*JAC:  there's no other way to get any other value in a stack.
[00:01:55.20]*JAC:  we can only see here # this value.
[00:01:58.23]*JAC:  think of it as if you look down on a stack of bricks if we
                    wrote numbers on each one and only if we wrote them on the
                    top, right?
[00:02:07.04]*JAC:  we're looking down we'd only see the six, right?
```

Figure 4. Excerpt from a transcript

was one of the two). Appendix B shows the grading criteria adopted. For each problem, a difference in grade greater than 2 points was resolved by discussion between the graders, and the final scores were averaged. Finally, the scores were aggregated by topic and rescaled in a range from 0 to 1.

Two additional groups of students took the pre and post tests, but these students did not participate in a tutoring session. We will call these groups Control-1 (C1) and Control-2 (C2).

Group C1 is composed of 53 students, who were taking the CS201 class at UIC. Between the two tests, instead of being tutored, students in group C1 attended a lecture about a totally unrelated topic.

Group C2 is composed of 28 students, all of them taking the CS202 class at UIC. Instead of being tutored, they were allowed to read selected sections of a traditional data structures textbook (Weiss, 1999) during the time between the two tests.

The demography of group C1 is very good for the purpose of this study. At the moment in which the experiment was conducted, those students had not studied lists, stacks, and trees in class yet. However, lists and stacks are in the curriculum of CS201, and they would encounter them a few weeks later. This means that these students are supposed to be "ready to learn" these topics, and they should have all the prerequisites to understand them. Binary search trees, instead, are in the curriculum of CS202, which is a class these students are supposed to take right after CS201.

On the other hand, the composition of group C2 might be problematic. Students in CS202 had already studied lists and stacks in a previous class (CS201). This would potentially make the interpretation of the results confusing. Is the learning gain that those students exhibit really learning or is it just a "refresher" of something they had temporarily forgotten? This is a difficult question to answer. In fact, looking at the history of the project, the data with group C2 was collected first. Only after we realized that C2 would not be such a reliable control group, we decided to run the experiment involving group C1.

### 3.3 Comparison of test scores between groups

A first round of statistical analysis of the data, performed using t-tests and ANOVA, revealed interesting phenomena that helped better define the direction of further analysis.

1. *Students do learn* in most of the conditions. Paired sample t-tests revealed that post-test scores are significantly higher than pre-test scores for the two tutor conditions (inexperienced tutor, experienced tutor) and the control group C2 (the group that read a textbook), for all the topics (linked lists, stacks, binary search trees), except for linked lists with the less experienced tutor, where the difference is only marginally significant. If the two tutored groups are aggregated, there is significant difference for all the topics. In the case of group C1 (the group that attended an unrelated lecture), students showed no significant learning for linked lists; a modest but significant learning for stacks; and a marginally significant learning for binary search trees. Means and standard deviations are reported in Table II. The t-test values are reported in Table III.

TABLE II

DESCRIPTIVE STATISTICS OF TEST SCORES

| Topic | Tutor | Experienced | Pre-test | | Post-test | | Gain | |
|---|---|---|---|---|---|---|---|---|
| | | | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| List | Yes | No | .40 | .27 | .49 | .26 | .09 | .22 |
| | | Yes | .40 | .26 | .58 | .26 | .18 | .26 |
| | | All | .40 | .26 | .54 | .26 | .14 | .25 |
| | No (C1) | | .34 | .22 | .35 | .23 | .01 | .15 |
| | No (C2) | | .49 | .27 | .56 | .25 | .07 | .19 |
| | All | | .43 | .26 | .55 | .25 | .12 | .23 |
| Stack | Yes | No | .29 | .32 | .65 | .23 | .35 | .25 |
| | | Yes | .28 | .28 | .55 | .26 | .27 | .22 |
| | | All | .29 | .30 | .60 | .24 | .31 | .24 |
| | No (C1) | | .33 | .30 | .38 | .32 | .05 | .17 |
| | No (C2) | | .58 | .26 | .83 | .18 | .25 | .20 |
| | All | | .39 | .32 | .68 | .25 | .29 | .22 |
| Tree | Yes | No | .45 | .27 | .77 | .15 | .33 | .26 |
| | | Yes | .54 | .26 | .82 | .15 | .29 | .23 |
| | | All | .50 | .26 | .80 | .15 | .30 | .24 |
| | No (C1) | | .41 | .23 | .45 | .25 | .04 | .16 |
| | No (C2) | | .50 | .27 | .66 | .26 | .16 | .20 |
| | All | | .50 | .26 | .75 | .20 | .26 | .24 |
| All | Yes | No | .38 | .29 | .64 | .24 | .26 | .27 |
| | | Yes | .42 | .28 | .66 | .25 | .25 | .24 |
| | | All | .40 | .28 | .65 | .25 | .25 | .25 |
| | No (C1) | | .37 | .33 | .41 | .33 | .04 | .24 |
| | No (C2) | | .52 | .26 | .68 | .25 | .16 | .21 |
| | All | | .44 | .28 | .66 | .25 | .22 | .24 |

TABLE III

PAIRED SAMPLES T-TESTS ON PRE-SCORE, POST-SCORE

| Topic | Tutor | Experienced | $t$ | $df$ | $P$ |
|---|---|---|---|---|---|
| List | Yes | No | -2.00 | 23 | .057 |
| | | Yes | -3.85 | 29 | .001 |
| | | All | -4.24 | 53 | $< .001$ |
| | No (C1) | | -0.56 | 52 | $ns$ |
| | No (C2) | | -2.10 | 27 | .045 |
| | All | | -4.70 | 81 | $< .001$ |
| Stack | Yes | No | -6.90 | 23 | $< .001$ |
| | | Yes | -6.15 | 23 | $< .001$ |
| | | All | -9.20 | 47 | $< .001$ |
| | No (C1) | | -2.15 | 52 | .037 |
| | No (C2) | | -6.73 | 27 | $< .001$ |
| | All | | -11.3 | 75 | $< .001$ |
| Tree | Yes | No | -6.13 | 23 | $< .001$ |
| | | Yes | -6.84 | 29 | $< .001$ |
| | | All | -9.23 | 53 | $< .001$ |
| | No (C1) | | -1.78 | 52 | .08 |
| | No (C2) | | -4.24 | 27 | $< .001$ |
| | All | | -9.7 | 81 | $< .001$ |

2. There is *no significant difference* between the two tutored conditions in terms of learning gain, expressed as the difference between post-score and pre-score. This is revealed by ANOVA between the two groups of students in the tutored condition. For lists, $F(1, 53) = 1.82$, $P = ns$. For stacks, $F(1, 47) = 1.35$, $P = ns$. For trees, $F(1, 53) = 0.32$, $P = ns$. Means and standard deviations are reported in Table II.

3. The learning gain of students that received tutoring is *significantly higher* than the learning gain of the students in control group C1 (those that attended an unrelated lecture), for all the topics. This is showed by ANOVA between the group of tutored students and group C1. The tutored group is the aggregate of the inexperienced tutor's group and the experienced tutor's group. For lists, $F(1, 106) = 11.0$, $P < 0.01$. For stacks, $F(1, 100) = 41.4$, $P < 0.01$. For trees, $F(1, 106) = 43.9$, $P < 0.01$. Means and standard deviations are reported in Table II.

4. There is *no significant difference*, in terms of learning gain, between students in the tutored conditions and students in the control group C2 (those that read a textbook), for lists and stacks. For trees, the learning gains of students in the tutored condition are *significantly higher* than the learning gains of those students that read the textbook. This is showed by ANOVA between the group of tutored students and group C2. The tutored group is the aggregate of the inexperienced tutor's group and the experienced tutor's group. For lists, $F(1, 81) = 2.01$, $P = ns$. For stacks, $F(1, 75) = 1.57$, $P = ns$. For trees, $F(1, 81) = 7.0$, $P < 0.05$. Means and standard deviations are reported in Table II.

The fact that students learn from reading a textbook as much as from being tutored may seem discouraging, at a first sight pointing to the conclusion that the data collected with human tutors is not worthy of further investigation. However, as already discussed, control group C2 is composed of students enrolled in the CS202 class, whereas the tutored group is composed of a more varied —and generally less advanced— population. This fact makes it difficult to provide a reliable interpretation of results involving group C2.

Also, a closer look at the distribution of scores across the sessions shows a lot of variability (See Table II). In all the conditions, there are sessions with very high learning gains, and sessions with very low ones. Thus, the assumption that a more experienced tutor can be more effective than a less experienced one with the majority of students does not hold, at least in this context. One reason for that could be the high complexity of this domain, and the relatively shorter history of the pedagogy of these subjects compared to more "traditional" ones, like Mathematics or Physics.

This observation suggests a new direction for subsequent analysis: instead of looking at the characteristics of a particular *tutor*, it is better to look at the features that discriminate the most successful *sessions* from the least successful ones.

### 3.4 Moving away from code-and-count

A traditional approach to the analysis of tutorial dialogues involves the comparison between groups of subjects interacting with different tutors, contrasting the number of occurrences of relevant features between the groups. However, as explained in detail in

(Ohlsson et al., 2007), this traditional "code-and-count" methodology has some drawbacks that make it difficult to capture the complexity of tutoring in this domain.

The code-and-count methodology operates by first conceptualizing some features that may be related to learning, then counting the occurrence of those features in the data. The data is usually split in two or more conditions, and the features that occur significantly more frequently in one of the conditions are considered to be responsible for the learning outcomes. For example, features that have been used in the literature for analyzing tutoring dialogues are classification of *dialogue moves*, defining action categories to be associated with utterances, turns, or sequences of turns in tutorial dialogues (Chi et al., 2001; Litman et al., 2006; Lu et al., 2007). For example, tutor turns have been coded for *diagnosing*, *prompting*, *summarizing*, and *evaluating*; student turns have been classified into categories such as *questioning*, *reflecting*, and *answering*.

However, it is not necessarily true that the more frequent features are the most effective ones. The interaction between a tutor and a student is rich and complicated, and the features mostly related to learning might be relatively few and sparse within long dialogues. It may be the case that a small number of targeted actions can make the difference between a successful tutoring session and a poor one.

A possible way to address this problem is to try to discover correlations between tutoring features and learning outcomes using a multiple regression analysis. Multiple regression can tell us how much variation in learning outcomes is explained by the variation of individual

features in the data. The key here is to focus on *variation*, rather than absolute frequency of the features of interest.

## 3.5 Linear regression

The following analysis has been done adopting linear regression models. A quite delicate issue is how to aggregate the data. The CS data set is divided into "mini-sessions," each of them corresponding to a student-topic pair. One possibility is to run the regression model on all the mini-sessions together. The other option is to split the data by topic, and run the analysis independently for each topic. There are pros and cons to both types of aggregations. Analyzing all the data together has the advantage of potentially allowing us to discover generalities across topics, and has the benefit of drawing from a larger data set. On the other hand, doing a separate analysis for each topic might be more appropriate because of the large differences in the distributions of our CS data for different topics. Also, the separate analysis can reveal different effect sizes, or even different effect directions, of the various features for different topics. The results in the following sections are obtained with both partitioning methods.

### 3.5.1 Prior knowledge

First of all, we want to factor out the effect of *prior knowledge*, measured by the pre-test score. Linear regression revealed a strong effect of pre-test scores on learning gain (Table IV). However, the $R^2$ values show that there is a lot of variance left to be explained, especially for lists and stacks, although not so much for trees. Notice that the $\beta$ weights are negative. That means students with higher pre-test scores learn *less* than students with

lower pre-test scores. A possible explanation is that students with more previous knowledge have less *learning opportunity* than those with less previous knowledge.

### 3.5.2 Time on task

Another variable that is recognized as important by the educational research community is *time on task*, and we can approximate it with the length of the tutoring session. Linear regression showed a significant correlation with learning gain when all the topics are grouped together, and a significant correlation with learning gain only for linked lists when the topics are considered individually (Table IV).

### 3.5.3 Student activity

Another hypothesis is that the degree of *student activity*, in the sense of the amount of student's participation in the discussion, might relate to learning (Lepper et al., 1997; Chi et al., 2001). To test this hypothesis, the following definition of student activity was adopted:

$$\text{student activity} = \frac{\#\text{ of turns} - \#\text{ of short turns}}{\text{session length}}$$

*Turns* are the sequences of uninterrupted speech of the student. *Short turns* are the student turns shorter than three words. Subtracting the number of short turns has the effect of eliminating those turns composed exclusively by words like "okay" and "uh uh," which usually do not contribute much content to the conversation, although they are important back-channeling elements. Of course, this is just an approximation, because substantive

| | | |
|---|---|---|
| | T: | do you see a problem? |
| | T: | I have found the node a@l, see here I found the node b@l, and then I put g@l in after it. |
| *Begin +* | T: | here I have found the node a@l and now the link I have to change is +... |
| | S: | ++ you have to link e@l <over xxx.> [>] |
| *End +* | T: | [<] <yeah> I have to go back to this one. |
| | S: | *mmhm |
| | T: | so I *uh once I'm here, this key is here, I can't go backwards. |
| *Begin -* | S: | <so you> [>] <you won't get the same> [//] would you get the same point out of writing t@l close to c@l at the top? |
| | T: | oh, t@l equals c@l. |
| | T: | no because you would have a type mismatch. |
| *End -* | T: | t@l <is a pointer> [//] is an address, and this is contents. |

Figure 5. Positive and negative feedback (T = tutor, S = student)

answers that are three words or less are certainly possible. Linear regression revealed *no significant effect* of this measure of student activity on learning gain.

### 3.5.4 Feedback

The dataset was manually annotated for *episodes* where positive or negative feedback is delivered. Formal definitions of the relevant categories, along with guidelines for coders and more examples of application to the CS dataset, are reported in the Appendices. All the protocols have been annotated by one coder, and 9 of them have been double-coded by a second one (intercoder agreement: kappa = 0.67). Examples of feedback episodes are reported in Figure 5.

The counts of positive and negative feedback episodes were introduced in the regression model (Table IV). The model showed a significant correlation between feedback and learning for all the topics together, but no significant correlation for the individual topics, except a marginally significant correlation between positive feedback and learning for linked lists. Interestingly, the correlation with positive feedback is *positive*, but the correlation with negative feedback is *negative*, as can be seen from the sign of the $\beta$ values. A possible explanation for the negative correlation of negative feedback with learning could be that our tutors do not effectively remediate some of the students' misconceptions. In this case, students that exhibit more faulty knowledge end up learning less that those that do not.

I additionally annotated the episodes of positive and negative feedback for *initiative*. An episode can be initiated either by the *student* or by the *tutor*. In the first case, the student volunteers some information without being asked or prompted by the tutor, and the tutor replies with some feedback. In the second case, the tutor first asks or prompts the student (not necessarily verbally), then the student replies, and finally the tutor provides feedback on the student's answer. The distribution of initiative labels is reported in Table V. The numbers in the table are aggregated on the three topics, but splitting the three topics apart revealed similar patterns.

ANOVA revealed overall significant differences on the four groups ($F(3, 325) = 43.27$, $P < 0.01$). A Tukey post-hoc test revealed significant differences ($P < 0.01$) between positive-tutor and positive-student; positive-tutor and negative-tutor; and positive-tutor and negative-student. The difference between positive-student and negative-student is

TABLE IV

LINEAR REGRESSION – HUMAN TUTORING

| Topic | $N$ | Model | Predictor | $\beta$ | $R^2$ | $P$ |
|---|---|---|---|---|---|---|
| List | 54 | 1 | Pre-test | $-.47$ | .21 | $< .001$ |
| | | 2 | Pre-test | $-.39$ | .27 | .002 |
| | | | Session length | .28 | | .025 |
| | | 3 | Pre-test | $-.49$ | .30 | $< .001$ |
| | | | Session length | .15 | | $ns$ |
| | | | + feedback | .38 | | .051 |
| | | | - feedback | $-.20$ | | $ns$ |
| Stack | 48 | 1 | Pre-test | $-.60$ | .35 | $< .001$ |
| | | 2 | Pre-test | $-.59$ | .34 | $< .001$ |
| | | | Session length | .04 | | $ns$ |
| | | 3 | Pre-test | $-.58$ | .34 | $< .001$ |
| | | | Session length | .17 | | $ns$ |
| | | | + feedback | $-.04$ | | $ns$ |
| | | | - feedback | $-.19$ | | $ns$ |
| Tree | 54 | 1 | Pre-test | $-.83$ | .68 | $< .001$ |
| | | 2 | Pre-test | $-.81$ | .68 | $< .001$ |
| | | | Session length | .07 | | $ns$ |
| | | 3 | Pre-test | $-.81$ | .68 | $< .001$ |
| | | | Session length | .10 | | $ns$ |
| | | | + feedback | .12 | | $ns$ |
| | | | - feedback | $-.10$ | | $ns$ |
| All | 156 | 1 | Pre-test | $-.58$ | .33 | $< .001$ |
| | | 2 | Pre-test | $-.59$ | .37 | $< .001$ |
| | | | Session length | .19 | | .003 |
| | | 3 | Pre-test | $-.64$ | .40 | $< .001$ |
| | | | Session length | .13 | | .052 |
| | | | + feedback | .30 | | $< .001$ |
| | | | - feedback | $-.16$ | | .054 |

TABLE V

FEEDBACK INITIATIVE: MEAN (STD) NUMBER OF EPISODES

| | Student initiative | Tutor initiative |
|---|---|---|
| Negative feedback | 1.7 (1.2) | 2.0 (1.2) |
| Positive feedback | 3.9 (3.8) | 10.2 (9.1) |

marginally significant ($P < 0.1$) for stacks, not significant for lists and trees. We saw that positive feedback is positively correlated with learning, and that the vast majority of the positive feedback episodes are initiated by the tutor. These results suggest the importance of *proactive feedback*, which I implemented in the most recent version of the Intelligent Tutoring System described in the next chapter. As we will see in more detail later, a proactive feedback episode is composed of three steps: a question from the tutor; the answer from the student; and finally the tutor's reaction to the student's answer.

### 3.5.5 Direct procedural instruction

The dataset was annotated for *direct procedural instruction* (DPI). In the context of problem solving, DPI occurs when the tutor directly tells the student what to do. This includes correct steps that lead to the solution of a problem (e.g., "and there is nothing there, so we put six right there"); high-level steps or subgoals (e.g., "it wants us to put the new node that contains G in it, after the node that contains B"); and tactics and strategies (e.g., "so with these kind of problems, the first thing I have to say is always draw pictures").

Linear regression showed a significant positive correlation between DPI and learning gain for lists ($\beta = 0.0038$, $t(49) = 2.69$, $P < 0.01$, $R^2 = 0.11$) and trees ($\beta = 0.0024$, $t(50) = 3.07$, $P < 0.01$, $R^2 = 0.14$). However, the significance is lost when including DPI as additional variable in the multiple regression models showed in the previous sections. The reason is that DPI has a very high degree of collinearity with session length, making those two variables almost interchangeable.

## 3.6 The CSCoding tool

Some of the measures used in the analysis presented in this chapter were extracted automatically from the transcribed corpus. However, to capture more complex phenomena, it was also necessary to proceed with some manual annotation of the CS tutoring dataset. To facilitate the annotation process, I developed a specialized tool, called *CSCoding*. The tool allows an efficient annotation of the transcripts through an intuitive graphical user interface, and enables the coders to browse the video-recording of the session, automatically linking the video to the transcript. The tool is written in Java, and interfaces to the popular software *MPlayer* for playing video files. A screenshot of CSCoding can be seen in Figure 6.

## 3.7 Important results

The data analysis reported in this chapter revealed some important results that directly guided the design and implementation of the most successful versions of the Intelligent Tutoring System described in the rest of this thesis. First of all, feedback is important in tutoring, and it appears that positive feedback is even more important than negative feedback in helping students learn. This is an important lesson, as the majority of tutoring

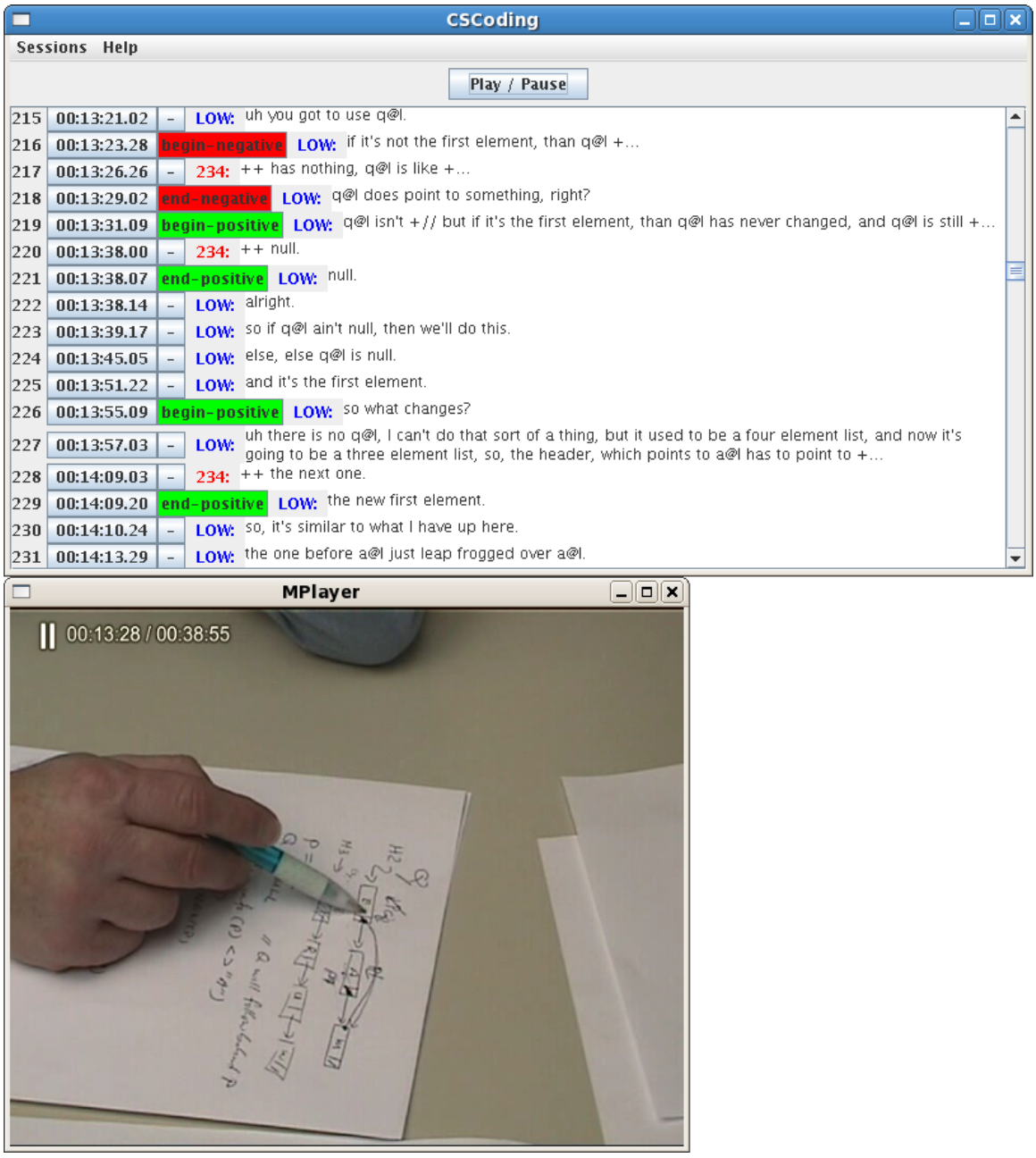



Figure 6. The CSCoding annotation tool

systems, including the first versions of ours, deliver mostly negative feedback in order to help students correct their mistakes. Another important discovery from this human tutoring study is the "proactive interaction" mechanism with which our tutors elicit mostly correct responses from students, that are finally acknowledged with positive feedback. It appears that our tutors often guide their students by asking questions that are easy enough so that students may answer correctly most of the times. This kind of proactive interaction is in fact the one mimicked by the latest version of our Intelligent Tutoring System. Finally, a comparison between this study of human tutoring and the behavior of many Intelligent Tutoring Systems, including ours, points out that there is a delicate balance between being directive —for example, telling the student what to do— and letting students explore the subject domain by themselves. Our tutors are very directive and leave little space to student exploration. On the opposite end of the spectrum, the first versions of our system lets students do whatever they want, mildly constrained by the user interface of the system itself. Probably, the maximal effectiveness lies somewhere between these two extremes. In fact, the latest versions of our system tries to balance out the extreme exploratory behavior of students with some guidance provided by procedural feedback, which will be described later in this thesis.

# CHAPTER 4

# A TUTORING SYSTEM FOR LINKED LISTS

## 4.1 Motivation

This chapter describes iList, an Intelligent Tutoring System in the domain of linked lists. As illustrated earlier, linked lists are one of the fundamental data structures taught in Computer Science curricula. In the development of the system, I chose to focus on linked lists for several important reasons, conceptualized by Prof. Christopher Brown at the United States Naval Academy.

- Linked lists are usually presented early in Computer Science curricula. Thus, more students see this topic.
- According to the experience of the Computer Science instructors collaborating with our research group, students struggle with linked lists more than with many other data structures.
- The fundamental concepts of linked structures, pointer manipulations, object allocation, and traversals, which students learn in the context of linked lists, are all necessary for more complicated data structures, such as trees. Linked lists are important because students can learn these concepts in a relatively simple context and will be already familiar with them when trying to understand more complicated structures.

- Part of what students learn while they struggle with linked lists is to think about an abstract visual model of their data, and to think of steps in a program/algorithm as making changes to that model. Mastering that way of thinking (Wing, 2008) is a huge step for students, and one that they need to make to continue successfully in Computer Science.

## 4.2 Architecture

The iList system provides a student with a simulated environment where linked lists can be seen and manipulated. The student is supposed to already know at least a basic definition of linked lists. Lists are represented graphically, and can be manipulated with programming language commands. Students are asked by the system to solve problems in this environment, such as to insert new nodes in a given linked list, remove nodes, or perform other more complicated operations. As a student is working towards a solution, the system provides feedback to help the student make progress.

The architecture of iList is depicted in Figure 7. It is composed of a graphical user interface that manages the interaction with the student; a problem model that represents the problems to be solved; a constraint evaluator that determines the correctness of student solutions; a feedback manager that generates appropriate messages for the student; a student model that includes the history of student actions and various measures of student performance; and a procedural knowledge model that represents information about the goodness of solution paths within individual problems. These modules implement the main

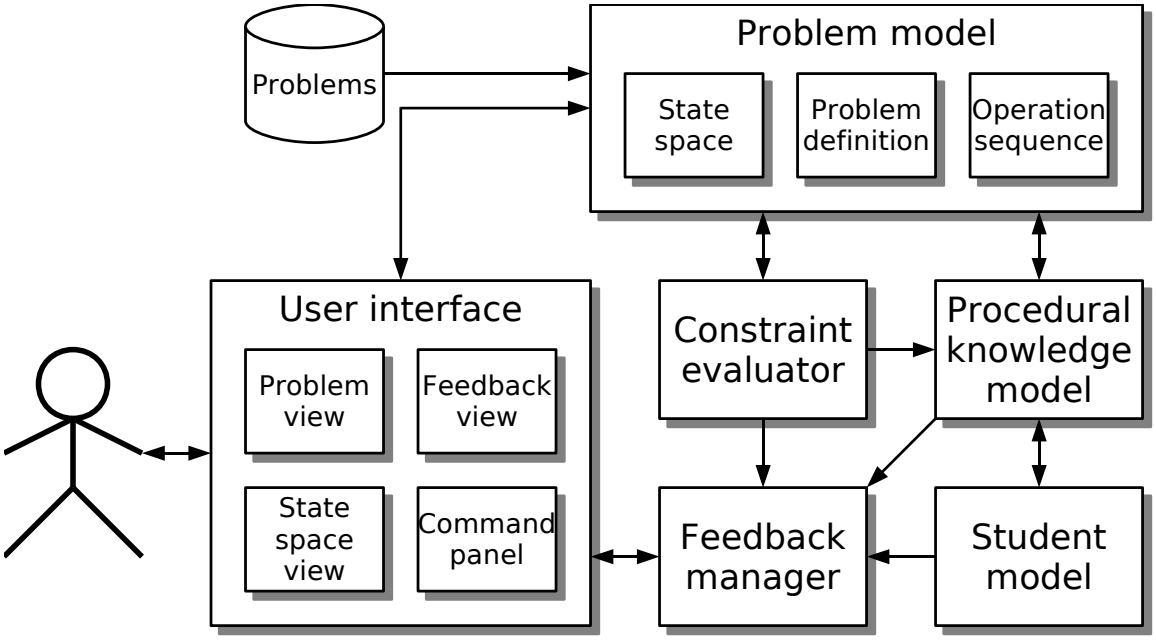



Figure 7. Architecture of iList

functionalities present in canonical architectures of Intelligent Tutoring Systems (Beck et al., 1996).

### 4.2.1 Graphical user interface

The graphical user interface is responsible for the main interaction with the student. Snapshots of the interface can be seen in Figure 8 and Figure 9. The interface is divided in four main parts: an area containing the description of the problem to be solved; an area reporting the history of feedback messages given to the student; an area representing the current state of the linked list virtual machine; and an area where students can enter

commands and see a history of the previously executed operations. Using this interface, students can interactively manipulate the data structures using C++ or Java commands. Depending on the problem type, either the effects of individual commands are reflected immediately on the graphical representation, or a block of commands is executed at once in the simulated environment. The command interpreter is quite resilient and tries to understand the user input even if it is slightly inaccurate. This allows the student to focus more on the semantics of statements rather than on language dependent details.

### 4.2.2 Problem model

A problem is given to the student in the form of a textual description and one or more initial scenarios. The initial scenario is integrated into the working *state space*, which includes relevant domain elements like variables and nodes. The student is asked to progressively modify the state space by interactively providing a *sequence of operations*, until the desired configuration of the data structure has been reached.

The iList system supports two types of problems. The first kind of problems can be solved interactively, step-by-step (Figure 8). Students can enter a command into the system, and the system simulates the effect of that command, showing the effect of the action immediately on the simulated scenario. Problems of the second type require the student to write a complete snippet of code, typically involving iterative constructs like loops (Figure 9). Problems of this type usually introduce more than one initial scenario and ask the student to write code that should work correctly in all of them. This encourages the student

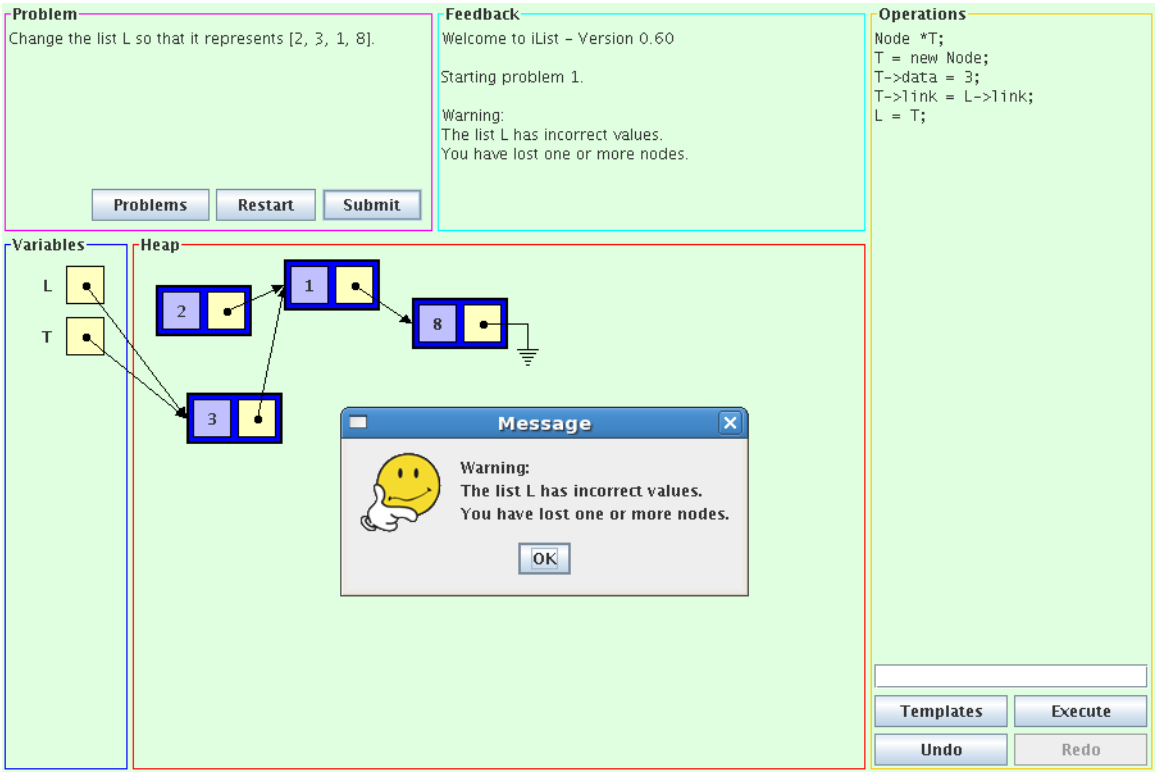



Figure 8. Screenshot of iList – Step-by-step problem

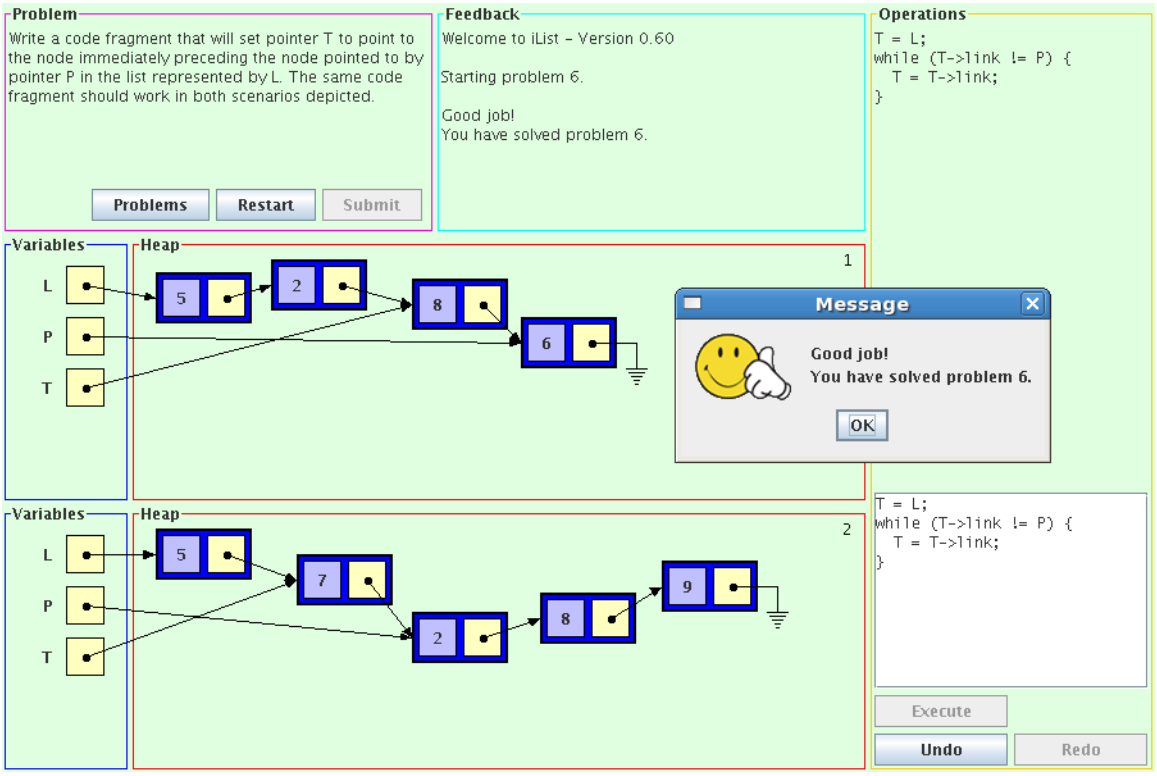



Figure 9. Screenshot of iList – Block of code problem

to abstract away the specific details of a scenario and think about more general algorithms for solving problems involving a wider range of situations.

The curriculum included in iList is currently composed of seven problems, five of the first type and two of the second type. These problems have been carefully crafted based on our experience as computer science educators and on published CS curricula, such as that by the ACM (AA.VV., 2008). The goal is to challenge the students with common difficulties in manipulating linked lists. The problems are defined in iList using a human-readable XML format, which makes it easy to add new problems as needed.

### 4.2.3 Constraint evaluator

When the student believes he/she is done with the current problem, the current state space is submitted to a constraint evaluator that checks the correctness of the solution. The usage of constraints in iList is motivated by a methodology called *constraint-based modeling*. We now briefly describe how constraint modeling works and then explain its application in the linked list domain.

Originally developed from a cognitive theory of how people might learn from performance errors (Ohlsson, 1992; Ohlsson, 1996), constraint-based modeling has grown into a methodology used to build full-fledged ITSs (Mitrović et al., 2004), and an alternative to the model tracing approach adopted by other ITSs, such as (VanLehn et al., 2005). In a constraint-based system, domain knowledge is modeled with a set of *constraints*. A constraint is a unit composed of a *relevance condition* and a *satisfaction condition*. A constraint is irrelevant when the relevance condition is not satisfied; it is satisfied when both

relevance and satisfaction conditions are satisfied; it is violated when the relevance condition is satisfied but the satisfaction condition is not.

In the context of tutoring, constraints are matched against student solutions. Constraints that are satisfied correspond to knowledge that students have acquired, whereas violated constraints correspond to gaps or incorrect knowledge. An important feature is that there is no need for an explicit model of students' mistakes, as opposed to buggy rules in model tracing. The possible errors are implicitly specified as the possible ways in which constraints can be violated. This property greatly simplifies the difficult and time consuming task of knowledge modeling in an ITS.

Computationally, the evaluation of constraints is fairly simple and efficient. Each constraint is implemented as a computational unit with three fundamental functions: a Boolean function checking the *relevance* of the constraint with respect to the solution, a Boolean function checking the *satisfaction* of the constraint, and a *feedback* function responsible to return relevant information used to generate feedback for the student. A constraint is violated if the logic implication $isRelevant \Rightarrow isSatisfied$ is false for that particular state space.

In the linked list domain, there are several properties that a solution should have in order to be correct. For example, a list should contain the correct values, as specified in the description of each problem; lists should be free of cycles; lists should not terminate with undefined or incorrect pointers; no nodes should be made unreachable from any of the variables, i.e., lost in the heap space; nodes should be correctly deleted when necessary (this

applies specifically to non-garbage collected languages, like C++). With these properties in form of constraints, iList can catch many common mistakes students make.

The constraint evaluator has access to two sources of information: the current student solution, and an exemplary correct solution provided with the definition of the problem, which is not necessarily the only possible correct solution of the problem. Having a correct reference solution allows iList to evaluate the problem-dependent properties of a student's solution, like the expected values of the final lists.

Overall, the adoption of a constraint-based paradigm to evaluate student solutions provides us with the main advantage that many different correct student solutions are recognized and accepted by the system. This is important in a domain like data structures, where alternative procedures can be used to achieve the same results. On the downside, this type of constraint evaluation cannot tell if the student is following a path that will never lead to a correct solution before it is too late for the student to recover from that path. In the current work section, we will briefly touch on an additional model that we are implementing to overcome this difficulty.

### 4.2.4 Feedback manager

The feedback manager is responsible for generating feedback messages for the students. Currently, iList can give five main types of feedback.

1. The student enters a command that iList cannot understand. We will call the feedback corresponding to this situation *syntax feedback*.

2. The student enters a command and iList understands it, but the command cannot be executed because of the contingent state of the virtual machine. For example, the student might try to access a variable that has never been declared, or reference a node that does not exist. We will refer to this type of feedback as *execution feedback*.

3. The student explicitly asks for his/her solution to be evaluated by pressing the "submit" button on the user interface. The system in this case will deliver what we will call *final feedback*.

4. The student enters a command that gets correctly executed, and the student has not completed the problem yet. In some circumstances, iList can deliver feedback about the move that the student just made. This is *reactive procedural feedback*.

5. During the problem, iList can look forward to the next moves students are likely to make. In some cases, iList can intervene with *proactive procedural feedback*.

The amount and sophistication of feedback differentiate the multiple versions of iList that have been developed and evaluated. More detailed descriptions of the different feedback strategies are illustrated in a separate section.

### 4.2.5 Student model

The student model includes information about individual students' problem-solving behavior. In particular, it stores the history of student actions, as well as measures of a student's speed and "undo/redo" behavior. This information is used by the Feedback Manager to decide when and what type of feedback to provide. For example, timing information

and "undo/redo" behavior are used to estimate student uncertainty, which is used as a condition to provide reactive procedural feedback, as described later on.

The history of student actions is tightly coupled with the Procedural Knowledge Model. In fact, each record in the history is linked to a node of the Procedural Knowledge Model, and the model itself is updated as new actions are performed.

### 4.2.6 Procedural knowledge model

The Procedural Knowledge Model is a probabilistic model that includes information about global students' problem-solving behavior. This model is used by the Feedback Manager to keep track of a student's progress towards a solution, and intervene with appropriate feedback when necessary. As opposed to the traditional models used by Model Tracing ITSs, iList's Procedural Knowledge Model is automatically computed from a corpus of student interactions with the system. A detailed description of the Procedural Knowledge Model is reported in a following section.

## 4.3 Feedback in iList

This section describes the five types of feedback provided by iList. So far, five versions of the system have been deployed in classroom and evaluated. The main differences between these five systems is the type and level of feedback. These differences are summarized in Table VI.

TABLE VI

FEEDBACK TYPES IN ILIST

| Feedback type | iList-1 | iList-2 | iList-3 | iList-4 | iList-5 |
|---|---|---|---|---|---|
| Syntax | Minimal | Yes | Yes | Yes | Yes |
| Execution | Minimal | Yes | Yes | Yes | Yes |
| Final | Yes | Yes | Yes | Yes | Yes |
| Reactive procedural | No | No | Yes | Yes | Yes |
| Proactive procedural | No | No | No | Yes but infrequent | Yes |

### 4.3.1 Syntax feedback

Most of the ideas presented in this section and the actual implementation of the second version of the syntax feedback module are from Prof. Christopher Brown at the United States Naval Academy.

In the first version of iList, iList-1, syntax feedback is very simple. Syntax error messages are of the form "I'm sorry, I can't understand XXX," where XXX is the command entered by the student.

The domain of tutoring interest for iList is really the visual model of linked lists, problem-solving in the visual model, and the correspondence between code actions and actions in the visual model. Thus, the original iList system did not tutor on syntax in any way. Syntax errors were flagged as errors, but students received no more information than the message "I didn't understand that." Students were expected to be advanced enough to correct their syntax errors on their own.

The system responded with "I didn't understand that" in another circumstance as well, namely when it received syntactically correct C++/Java code using constructs outside of the language subset understood by iList. iList restricts the C++/Java constructs it allows not only to simplify the system, but also to force students towards "the right solution," which means a solution that generalizes, or which does not contain unneeded complexity. For example, early problems have the students entering statements one-at-a-time that operate on a concrete set of variables and nodes. If the problem is to change the node in list L with data value 6 to have data value 42, a valid C++ solution might be `L->link->link->link->data = 42`. However, this solution does not generalize to the case in which L is an *arbitrary* list containing a node with value 6. Thus, iList does not allow this kind of chaining of `->`'s. The student is forced into a solution like

```
Node *T = L;
T = T->link;
T = T->link;
T = T->link;
T->data = 42;
```

which generalizes to

```
Node *T = L;
while(T->data != 6) {
  T = T->link;
}
T->data = 42;
```

The "I didn't understand that" messages generated quite a bit of frustration in students, which was voiced in survey responses (see the evaluation section). The expectation that students knew enough to understand and correct their own syntax errors was ill-founded, as was the expectation that they would remember/realize that if's, while's and for's were not allowed in single-statement input. It became clear that the system would have to respond to syntax issues with something more.

A more advanced feedback module for syntax errors was introduced in the second version of iList, iList-2. The syntax error response module generates feedback for input that iList is unable to understand. Recognizing the "for," "while," and "if" keywords is trivial, and the module responds to the presence of these keywords by explaining that, although they are valid C++/Java constructs, iList doesn't allow them because it wants students to solve the problem a different way. Responding to genuine syntax errors is trickier. Generating good syntax error messages, whether in iList or in actual interpreters/compilers, requires good guesses as to what the programmer actually *intends*, and the remainder of this section is a brief description of how the module makes such guesses, and how it generates feedback.

Compilers and interpreters generally build a tree representation of a program from input text by (1) tokenization (grouping input text into chunks called tokens) and (2) parsing (hierarchically organizing tokens based on the rules of some grammar) (Aho et al., 2006). For valid input in a well-defined programming language the process is unambiguous in the sense that the grouping of characters into tokens is unique, and the organization of tokens based on grammar rules is unique. The theory of tokenization and parsing is very well developed,

so that huge programs in complex programming languages can be tokenized and parsed quickly. For invalid inputs, theory has much less to say. Detecting that input is invalid is no problem, but generating good error messages is hard. Most compilers/interpreters do not deal with errors in the tokenization phase unless they are actually faced with a sequence of characters that cannot be tokenized. Thus errors are dealt with primarily through the parser alone, and they are dealt with by throwing away tokens until what is left fits the grammar rules. This approach is efficient and produces error messages quickly even for large programs in complex languages. However, the approach is limited by not considering alternate tokenizations; it only subtracts from the input, never adding or reinterpreting. Moreover, the approach is to parse according to the actual grammar, instead of allowing "error" grammar rules embodying common misconceptions. In our case, the input is a single statement, and the language is a small subset of C++/Java. Thus efficiency is not much of an issue, and we can pursue a more wide-ranging approach to understanding incorrect input.

The module tokenizes text and parses token streams with respect to a grammar, just as standard parsers do. However, it produces many tokenizations and many parses, each weighted by some measure of likelihood. Valid input gets tokenized and parsed with weight zero. For invalid input, higher weighted tokenizations and parses are deemed to be less likely. The module returns the lowest weighted tokenization and parse, provided one exists below a prescribed threshold, along with an error message if that weight is non-zero.

Tokenizations are generated by adding error-keywords to the set of actual keywords, and by using the standard edit distance (Damerau-Levenshtein distance) metric to find plausible interpretations accounting for typos and misspellings (see e.g. (Cormen et al., 2000), p.364). Though there are many potential tokenizations, the system only generates them one-by-one in order of increasing weight, until a solution is found or a threshold is reached. Tokenization steps of positive weight have error messages associated with them.

Tokenized input is parsed according to a grammar, but the grammar includes "error rules", e.g. $pexp \rightarrow num$, which allows a number to be interpreted as a pointer expression. Each error rule has a positive weight associated with it and, just as with tokenizations, parses are generated one-at-a-time in order of increasing weight. Each error rule also has a message associated with it.

The module described is limited in many ways. Most notably, it models errors as independent — for example, the weight of the parse for `2->link = 5->link` is twice the weight of the rule $plval \rightarrow num$. This isn't really appropriate, since making the error the first time makes it much more likely it will happen again. In fact, in some sense, there is only one error here; or perhaps more accurately, only one misconception. Conversely, while a single typo is not uncommon and should get small weight, there are not likely to be many typos in a single statement. So the costs of typos should increase with their frequency in a given input. In short, the system would be improved by making error weights context-dependent. Another important limitation of the current implementation is that the module

TABLE VII

EXAMPLES OF SYNTAX ERRORS AND MESSAGES

| | |
|---|---|
| input | `p>link = NULL;` |
| g++ | `error: comparison between distinct`<br>`pointer types Node* and int`<br>`(*)(const char*, const char*)`<br>`throw () lacks a cast`<br>`error: lvalue required as left`<br>`operand of assignment` |
| iList | `You're trying to write a pointer`<br>`assignment statement, right?`<br>`* Did you mean "->" instead of ">"?` |
| note | Here an alternate tokenization actually interprets `">"` as `"->"`, with small penalty since the edit distance is small. |
| input | `8 = p->link;` |
| g++ | `error: lvalue required as left`<br>`operand of assignment` |
| iList | `You're trying to write a pointer`<br>`assignment statement, right?`<br>`* You're using a number like`<br>`it's a Node pointer object.` |
| note | Here the grammar rule $plval \rightarrow num$, which has positive error weight, allows the parse succeed by interpreting a number as a pointer "l-value". |
| input | `delete *p;` |
| g++ | `error: type class Node argument`<br>`given to delete,`<br>`expected pointer` |
| iList | `You're trying to write a delete`<br>`statement, right?`<br>`* You should give delete a pointer`<br>`to a Node, not the Node itself,`<br>`so there's no need to dereference`<br>`with *.` |
| note | Here the system can parse by ignoring the `*`, e.g. tokenizing `*p` as the *name* `p`, or by applying the error grammar rule $dltstmt \rightarrow dlt\ star\ plval$. The weight of the later is less, therefore that is the parse the system generates. |

works only for the step-by-step problems, and not for those requiring and entire block of code as input. This limitation will be removed in a future version of iList.

Despite its limitations, the syntax module can generate more explanatory messages than those usually generated by standard compilers, such as g++. Examples of messages can be seen in Table VII.

### 4.3.2 Execution feedback

In iList-1, also execution feedback is very simple. Execution error messages are of the form "You tried to execute XXX. I'm sorry, I can't do that." In the following versions, execution error message are more detailed and they point out the reason why the command can't be executed, such as "Variable T does not exist." The generation of these messages is straightforward. Individual messages are hard-coded into the different error conditions in the iList virtual machine and are assembled into the output as needed.

### 4.3.3 Final feedback

Final feedback is the feedback that is given to the student when he/she presses the "submit" button in the iList interface. In all versions of iList, final feedback comes from a collection of feedback units associated with the individual constraints that have been violated. The feedback manager collects these units and assembles them into a message directed to the student. An example of such feedback can be seen in Figure 8.

The generation of final feedback in iList is done with a very small number of constraints. On the one hand, this is good because it limits the complexity of the system. On the other hand, these messages are not very informative. The main problem is that final feedback

messages can't say anything about the procedure that have been followed to reach the solution. A solution to this problem is provided by the reactive and proactive feedback messages based on the Procedural Knowledge Model.

### 4.3.4 Reactive procedural feedback

In a tutoring context, *reactive feedback* is given in response to student actions that were not explicitly prompted by the tutor. In an exploratory environment like iList, this is the dominant case, as students are working on solving problems on their own. To decide whether to give reactive procedural feedback, iList evaluates a student's move on two main factors: the *goodness* of a student move and the level of *uncertainty* of the student in making that move. Both factors can be quantitatively estimated from the Procedural Knowledge Model, as explained later. The goodness of a move is directly related to the probability that a student will eventually reach a correct solution, starting from the current state of his/her solution. Student's uncertainty is estimated by monitoring the time taken by the student to make the move, and the student's "undo behavior." If the time taken by the student is more than a standard deviation greater than the average time spent by past students at that same point, plus a correction factor based on a student's personal history, then the student is considered uncertain. Also, if the student performed an "undo," "redo," or "restart" operation at that point in the past, he/she is considered uncertain there.

More specifically, the feedback generation algorithm works as follows. If the student just got into a "hopeless" state, i.e., a state from which the estimated probability of success is zero, then a *negative feedback* message is generated, to help the student get unstuck. If the

student has made a good move, i.e., has improved his/her probability of reaching a correct solution, *and* the student showed uncertainty, then a *positive feedback* message is provided. The rationale is that a student could have performed a correct but tentative move. In this situation, positive feedback can help consolidate correct knowledge that the student has not fully acquired yet. Moreover, it has been recently shown that human tutors may regulate their feedback according to student uncertainty (Forbes-Riley and Litman, 2008).

The procedural reactive feedback messages have two main components. The first part is a content-free sentence expressing the goodness of the student's move, such as "Mmmhh... Probably you can't go very far from here" (negative feedback) and "Good move!" (positive feedback). This is followed by a summarization of the effects of that move on the problem state space, for example "Node 2 was pointing to node 1, now it points to node 3. Node 1 was being targeted by node 2 but now it is abandoned." This explanation is dynamically generated comparing the previous state with the current state, then reporting the differences between those states. The facts to be communicated are chosen using a set of rules. Finally, the surface realization is performed using the SimpleNLG library (Reiter, 2007).

### 4.3.5 Proactive procedural feedback

Sometimes, waiting for the student to make a move and then providing feedback is too late. As emerged from our study of human tutoring (see the results on tutor initiative within feedback episodes), tutors often proactively anticipate the next student move and intervene with appropriate instruction to guide them in the right direction. Sometimes, tutors explicitly tell students what to do (direct procedural instruction), other times they

try to elicit the right move from the student using more subtle strategies, such as a hint "hidden" within a question or prompt. In the strategy implemented in iList, we decided to combine elements of direct procedural instruction hidden into a tutor-student interaction which includes the following elements:

1. A question from the tutor. It is composed of three parts:

   (a) a statement of the goal to be achieved by the following move;

   (b) the explicit question about how to accomplish that goal;

   (c) a set of up to four choices, which include the correct answer and some of the most frequent incorrect answers given by students.

   Example: "Let's see what we can do now... Pointer T is pointing to node 5, we want it to point to null. How would you do that? (1) T = NULL; (2) delete T;"

2. An answer from the student. The answer is simply given by clicking on one of the given choices.

3. Feedback from the tutor. If the answer was right, the message is a positive reinforcement statement such as "That sounds right! I suggest you to try it now." If the answer was incorrect, the message points out the mistake and illustrates the consequences of that hypothetical action. Example: "Uhmm... This is probably not a good idea. Here is what will happen if you do what you suggested. You will delete the node that is pointed by pointer T and that contains 2. Variable T is now pointing to node 2, then it will point to garbage."

Once the interaction has started, the student must complete the three steps described above before he/she can continue working on the problem. To decide when to start this type of interaction, iList monitors the student's activity. The current state of the problem is matched against the Procedural Knowledge Model. If the state is "critical" and enough time has elapsed since the last move, iList initiates the proactive interaction. The criticality $c$ of a state is defined as the probability that a student will go to a hopeless state (a state with goodness = 0) on the next step. The exact time to pop up with a proactive interaction is also determined by the context. The amount of time is determined by four variables: mean and standard deviation of the time that previous students spent into the current state, as it is stored in the Procedural Knowledge Model; the criticality of the current state, as defined above; and a measure of the correctness of the current student's general behavior $B$, defined as a smoothed function of how often the student visits "good" or "bad" states ($n$ is the current step, $n-1$ is the previous step, $g$ is the goodness of the current state):

$$B(n) = \alpha x + (1-\alpha)B(n-1)$$

$$\alpha = 0.5$$

$$x = \begin{cases} 0 & \text{if } g = 0 \\ 1 & \text{if } g > 0 \end{cases}$$

The delay $T$ after which the proactive interaction should start is determined by the following formula ($\mu_T$, $\sigma_T$ are the mean and standard deviation of think time of past students in the current state; $B$ is the student's behavior; $c$ is the current state's criticality):

$$T_{min} = \max(5, \mu_T - \sigma_T)$$

$$T_{max} = \max(5, \mu_T + \sigma_T)$$

$$T = T_{min} + (T_{max} - T_{min})B(1 - c);$$

The only difference between iList-4 and iList-5 is the time window in which the delay $T$ is chosen. In the formula above, used in iList-5, $T$ lies within plus and minus a standard deviation from the mean time to exit from the current state. In iList-4, the delay is much longer, and it is chosen between zero and two standard deviations from the mean. In practice, the choice of a longer delay in iList-4 resulted in too few proactive interactions. This lead to the choice of an earlier delay window in iList-5.

# CHAPTER 5

# THE PROCEDURAL KNOWLEDGE MODEL

## 5.1 Motivation

In order to generate the feedback explained in the previous chapter, we need a model that is able to assess the goodness of a state and that can support the determination of student uncertainty. Traditional model tracing techniques would allow us to do that (Anderson et al., 1995). A model tracing system explicitly incorporates "expert rules," that encode the possible correct steps to solve a problem; "buggy rules," that model the most common student errors; and a mechanism to trace student actions during the solution of a problem according to the mentioned sets of rules. However, to build a comprehensive knowledge model for model tracing, a significant amount of manual knowledge engineering work is required. Even though helpful authoring tools are available, such as (Blessing and Gilbert, 2008), these tools do not yet fully automate the knowledge acquisition process.

For iList, I wanted to avoid the expensive, time consuming, and rigid process of manually encoding procedural models for each problem in the system. A more fundamental reason is that problems in the linked list domain allow a great degree of flexibility, and many different paths can lead to a successful solution. Anticipating all the possible correct and incorrect paths and manually encoding them into the system would be almost impossible.

Thus, I decided to explore a machine learning approach to automatically generate a useful model from the past interactions of students with iList. Researchers in the new field of Educational Data Mining are working on extracting meaningful information from past student data, such as (Merceron and Yacef, 2005; Perera et al., 2007; Shih et al., 2008). Particularly relevant to this project is the work of Barnes and Stamper, who automatically extracted Markov Decision Processes from past student interactions with their logic proof tutor (Barnes and Stamper, 2007; Barnes and Stamper, 2008). Among the differences between the model I developed for iList and theirs, I mention the absence of an explicit reward function in iList's model; the representation of states, which are "virtual machine snapshots" in iList and sequences of steps in Barnes and Stamper's; and the computation of the "goodness" and "criticality" measures that iList uses in order to decide when to generate reactive and proactive procedural feedback.

## 5.2 Description of the model

The core of our model is a probabilistic graph equivalent to a Markov Chain. Its main components are *states* and *actions*. A state is a snapshot of iList's virtual machine, which includes the simulated linked lists. Linked lists in iList are internally represented with graphs. This representation allows the flexibility of modeling unusual or inconsistent linked list configurations, which can happen as a result of student exploratory actions. Actions in iList are first-class objects, which are created by the students from C++ or Java-like commands. Actions can modify a state into a different one. The model is represented with a *simple directed graph* with two types of vertices, *state vertices* and *action vertices*, and the

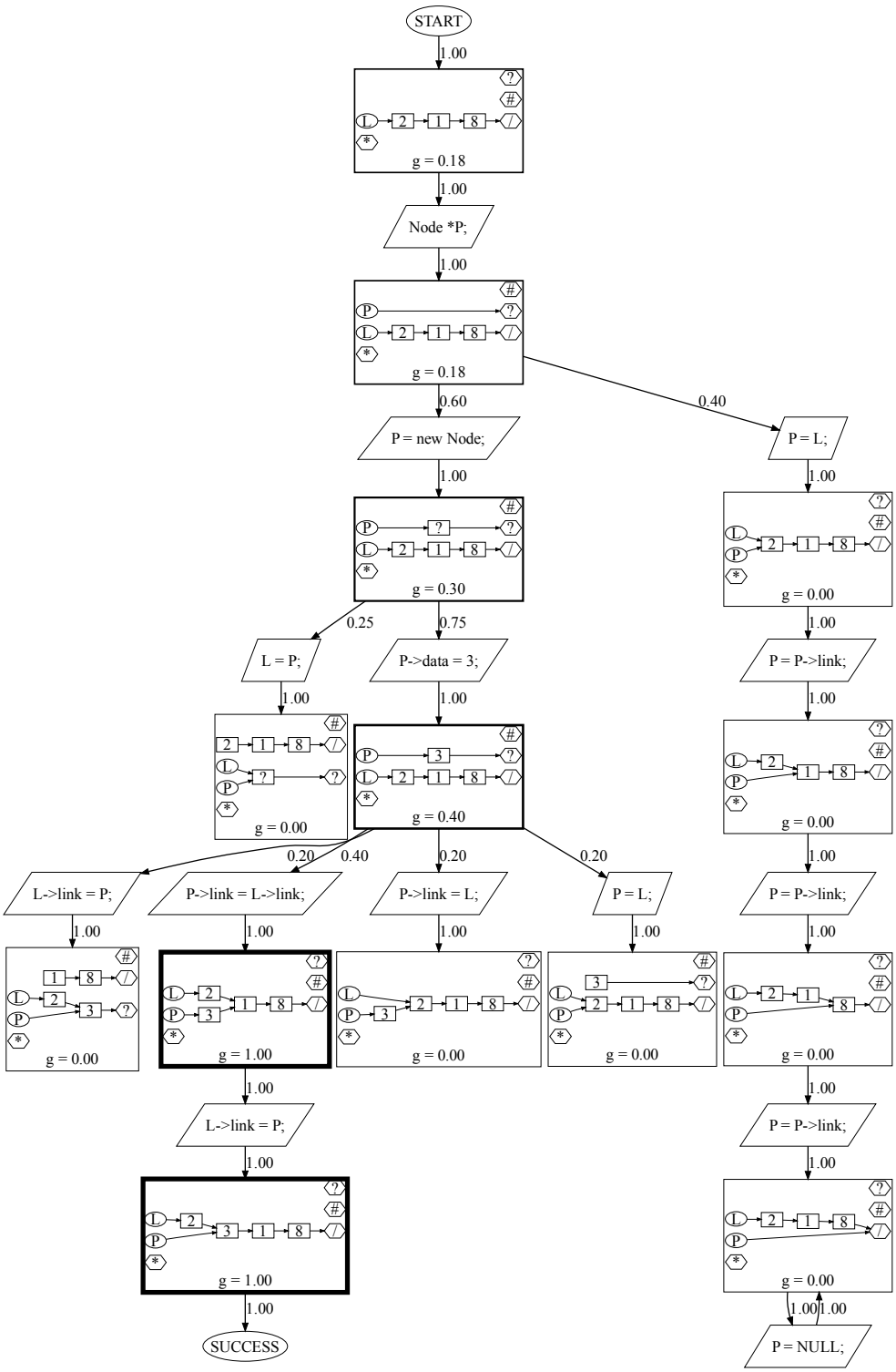



Figure 10. Example of generated graph. The thickness of the border is proportional to the $g$ value

constraint that a state vertex can only point to action vertices, and that an action vertex must point to exactly one state vertex. The set of actions in the graph is associated with a probability mass function. So, each action is associated with the probability that a student will take that action. Figure 10 shows an example of a graph for problem 1, which requires the student to insert a new node that contains "3" in the presented linked list. This graph is generated from only one student session for reading clarity, as the graphs generated with the entire dataset are much larger (Table VIII). "Undo," "redo," and "restart" operations are not represented in this graph. In the figure, large boxes represent states, and parallelograms represent actions. Each state contains a smaller graph representing linked lists and the variables to which some of the nodes of the list are linked. The figure also reports the "goodness" value ($g$) of each state, which will be explained in detail later. The edges in the figure are annotated with an estimate of the probability that a student would traverse them while solving a problem. Other important measures, such as node criticality and students' timing information, are omitted from the figure. They will be explained in more detail in the next section.

## 5.3 Training of the model

The algorithm to build the graph works as follows. First, iList scans and executes the student actions recorded in past log files. For each action, a new state is generated. If the new state can be matched to a state already present in the graph, the frequency of the pre-existing state is updated; otherwise, the new state is added to the graph. A

TABLE VIII

SIZE OF THE PROCEDURAL KNOWLEDGE MODELS FOR DIFFERENT PROBLEMS

| Training dataset | Problem | Dataset size | | Model size | |
|---|---|---|---|---|---|
| | | Sessions | Actions | State nodes | Action nodes |
| iList-1 + iList-2 | Problem 1 | 149 | 1360 | 272 | 353 |
| | Problem 2 | 155 | 2688 | 742 | 960 |
| | Problem 3 | 136 | 1165 | 269 | 356 |
| | Problem 4 | 89 | 1078 | 356 | 467 |
| | Problem 5 | 65 | 758 | 240 | 290 |
| All | Problem 1 | 259 | 2160 | 295 | 399 |
| | Problem 2 | 267 | 4130 | 1034 | 1363 |
| | Problem 3 | 236 | 2033 | 314 | 460 |
| | Problem 4 | 184 | 2037 | 549 | 721 |
| | Problem 5 | 150 | 1724 | 323 | 424 |

similar procedure is performed for actions. Matching states and actions in a sensible way is important in order to reduce the size of the graph and to avoid overfitting the model.

The matching process for states and actions is not trivial, as iList needs to be able to match semantically equivalent state spaces. A state space in iList is also represented with a properly annotated graph. Matching is performed by looking for isomorphism relations between two states. In the case of iList's state spaces, the search for isomorphism relations is highly constrained by their specific structures. In particular, the set of nodes that are considered "interchangeable" when checking for isomorphism are temporary variables and data nodes with identical values. If more than one isomorphism relation is found, iList

looks back at the matching history to disambiguate and choose the appropriate one, which is used as mapping function.

After a state is added to the model, iList checks if that state is a correct solution for the current problem. In that case, the state is connected to a special "success" node.

Each state gets also annotated with statistics about the time students took to exit from that state. This information is used by iList to assess student uncertainty.

Individual students' histories are recorded into separate structures, together with sets of mappings that establish a correspondence between the "real" states in iList and the matched states in the graph, as well as "real" actions and matched actions. This means that iList keeps a history of all the original student information together with mappings to the global graph.

After the construction of the graph is completed, the entire graph is traversed and two important quantities are computed. The frequencies associated with states and actions are converted into probabilities using maximum likelihood estimation. These probabilities are stored in the edges of the graph. Then, iList computes a "goodness" value ($g$) for each state of the graph. The $g$ value represents a lower bound on the probability that a student traversing that state will eventually reach a correct solution. It is calculated by summing the probabilities of the $k$ most likely paths (with $k$ empirically set to 10) from the current state to the special "success" node to which all the correct states are linked. This computation can be done efficiently with a variation of the traditional Bellman-Ford algorithm, a label correcting algorithm that computes shortest paths in weighted directed

graphs. The algorithm is implemented in the JGraphT library (Naveh and Sichi, 2003). Additionally, iList computes and stores the "criticality" ($c$) of each node. The criticality of a node is the probability that a student will get into a "hopeless" state (a state with $g = 0$) at the next step from the current node.

At run-time, when a new student comes in, his/her actions are matched against the graph. The comparison between the student's behavior and the model allows iList to make inferences and generate feedback using the strategies explained in the previous chapter.

## 5.4 Computational complexity

The computational complexity of the construction of the graph can be broken down in several components. The matching operation that compares two state spaces is potentially exponential, as it involves checking for isomorphism relations between subgraphs. However, in practice, the size of the state spaces does not grow arbitrarily, as students work on problems involving only small lists, which lead to small and structurally constrained graph representations that can be compared quickly. For the sake of the computational complexity analysis of the entire graph construction, the matching operation is considered performed in constant time. Let $N$ be the size of the dataset, i.e., the number of student actions recorded in the log files. The worst case time complexity of the first phase of the graph construction is $O(N^2)$, as each new node could be potentially be checked for equivalence with every other node in the graph. In practice, the number of comparisons is heuristically reduced by starting the matching process from those nodes close to the most recent action of the student. The Bellman-Ford algorithm used to compute the shortest paths in the

final graph is $O(mn)$, where $m$ is the number of edges and $n$ is the number of nodes of the final graph. Given the structure of the model, $m = n$. In the worst case, $n = 2N$, as each student action could be mapped to a new action node and a new state node. However, in practice $n$ is much smaller than $2N$, as a great fraction of the students' actions and the resulting state spaces could be matched in the first stage of the construction of the model.

The practical construction of the Procedural Knowledge Model is indeed very fast. With the data set currently available, the entire model can be built in a few minutes even on a modest personal computer.

## 5.5 Learning curve of the model

For the model to be useful in practice, it is important that a high percentage of the actions of new students can be matched with the model. The main hypothesis behind the model is that even though the state space of the graph is potentially infinite, only a relatively small number of states can represent most student actions. I calculated the learning curve of our models by training them incrementally and counting the percentage of matched states as each session is added to the model. I repeated the procedure 10000 times, randomly shuffling the dataset each time, and averaged the resulting curves.

Figure 11 shows the learning curves for all the problems. In these experiments, the maximum standard error of the mean is 0.003, which is too small to be plotted on the figure. Notice that the models can be learned quickly. For problem 1, the 80% match level is reached after just 3 training sessions; the 85% match level after 9 session; and more than 90% of the states can be matched after 30 sessions. For the other problems the learning rate

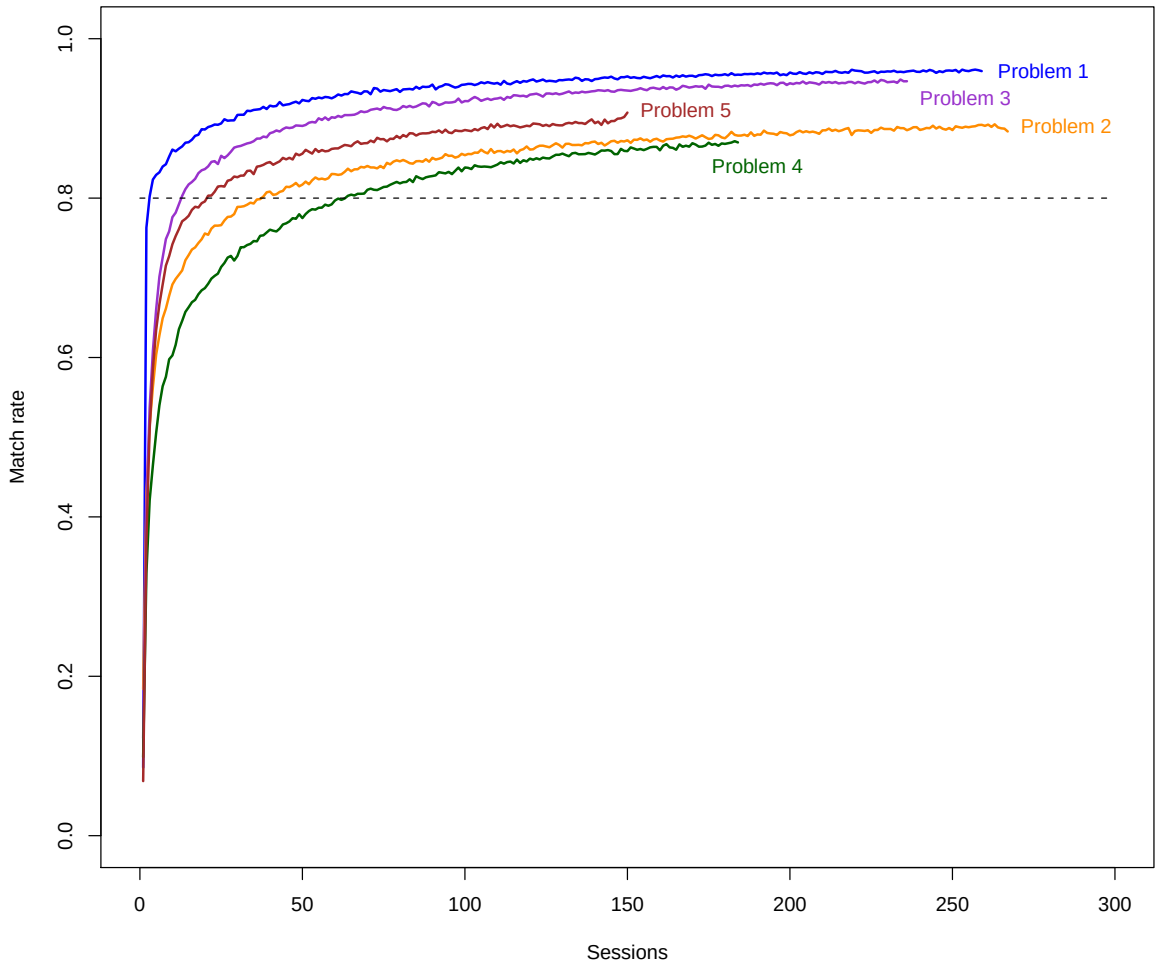



Figure 11. Learning curve of the procedural knowledge model

is slower, with problem 4 being the slowest. For problem 4, the 80% match level reached after 64 training sessions; the 85% match level after 127 session; and the 90% level is never reached with the available data. Notice that the lines of problem 4 and 5 are shorter, because fewer students worked on these problems in our past iList trials (see Table VIII). I believe that the steepness of the learning curve depends on the complexity of a problem. On easier problems, students make fewer "creative" moves than they do on more difficult ones. This leads to more predictable actions and consequently a faster learning of the model.

# CHAPTER 6

# EVALUATION

## 6.1 Summary of findings

One of the main goals of this project is to assess the impact of different types and levels of feedback on the effectiveness of Intelligent Tutoring Systems. To do so, I developed five versions of iList that differ in the feedback they provide (see Table VI). The five versions have been deployed in multiple classrooms at the University of Illinois at Chicago and at the United States Naval Academy over the span of several semesters. An important characteristic of this study is that the versions of iList have been developed and tested incrementally, based on the results and insights provided by the previous trials. The differences in student populations between the different groups introduced a high level of experimental uncertainty. This was inevitable and reflects the "real world" approach adopted in this project, similar in many aspects to the paradigm of Design Based Research (Brown, 1992). Examples of potentially confounding variables are reported in the next section.

The main findings of this study are the following:

- There is a positive trend in terms of student learning gain through the different versions of iList, which are statistically indistinguishable from human tutors.
- Students liked working with iList and found it useful.

- Students considered the most sophisticated feedback more helpful, but also more difficult to understand and potentially more misleading.
- Solving more problems with iList generally correlates with better learning.
- The versions of iList with more sophisticated feedback help students solve more problems.
- The more sophisticated procedural feedback (reactive and proactive) provided by the newest versions of iList effectively guides students towards "the right path" while solving the problems.
- The procedural feedback manager generated more positive feedback messages than negative feedback messages.

In summary, iList proved to be an effective system, well accepted by the students; and the more sophisticated feedback introduced in the most advanced versions of iList produced a measurable positive change in students' problem solving behavior.

## 6.2 Experimental design

During their regular class time, students participated in a single lab session 1 hour 15 minutes to 1 hour 30 minutes long. We asked them to complete a pre-test, then work with iList, complete an identical post-test, and finally fill in a survey. Starting with iList-2, students also completed a working memory capacity test. In the first version of iList, pre-test, post-test, and final survey were in paper-and-pencil format; starting from iList-2, tests and survey have been administered in electronic format. Also, in the first version of iList, the

instructor was in charge of giving the student a short tutorial on the system by solving the first problem of iList's curriculum in front of them; from iList-2, the instructor intervention has been replaced with a brief written tutorial providing some minimal information on the system and demonstrating how to solve the same first problem. These changes might have had an effect on the results, but the assumption is that the impact was negligible compared to the many other sources of noise that affect this kind of real-world experimental setting.

All the students were taking an introductory data structures class, and the lab sessions with iList were scheduled right at the time when the topic of linked lists was just being introduced. An exception is the group that worked with iList-4, that used the system approximately three weeks after being taught linked lists in classroom. This fact might have seriously affected their interaction with iList.

Students had no previous experience with iList. The system was presented to them on the day of the experiment. An exception is the group that worked with iList-3, that was forced to repeat part of the experiment one week later because of a network crash in their classroom on the scheduled day of the experiment with iList. In practice, most students took the pre-test and some of them started working with iList for a few minutes before being interrupted. In the make-up session, students repeated the pre-test. The pre-test scores were slightly higher the second time, although not in a statistically significant amount. This incident generated quite a bit of frustration in students, which was reflected in their survey responses.

The pre-test and post-test are identical and are based on the test developed for the study of human tutoring described in Chapter 3. In particular, the iList pre/post test includes the first two problems on linked lists of the human tutoring test, plus an additional third problem on linked lists that was deemed appropriate for testing skills on operating with linked lists. Notice that the addition of a third problem makes the comparison of test scores between iList and human tutors not ideal; however, this problem was partly alleviated by proper re-scaling of the scores, that made them at least numerically comparable. The questions in the test were carefully crafted to assess a deep level of knowledge of the topic, and they are somewhat difficult for students at that level. The first problem presents a fragment of code and an initial scenario. Students have to draw the final state of the given linked lists after the execution of the code. The second question presents the same code, with a syntactically correct but semantically wrong variation which would cause the code to malfunction. The students are asked to explain why the modified code does not work as expected. The third question requests the students to write a sequence of operations that moves the first node of a given list to the end of that list. These three questions assess a mix of important analytical, diagnostic, and operational skills in the linked list domain. All the questions have been graded by the researchers on a scale from 0 to 5, following written guidelines. For the reader's convenience, all the scores have been rescaled to a 0 to 1 decimal scale. The text and grading scheme of the pre/post test are reported in the appendices.

## 6.3 Learning outcomes

The first and most important question to be answered is whether students learn when using iList, and to what degree. Indeed, students do learn when working with iList. A measure of learning gain is the difference between post-test score and pre-test score. Table IX reports pre-test, post-test, and learning gain scores in the five iList conditions, plus the human tutoring condition and the control group of students that did not receive any relevant instruction (see Chapter 3).

ANOVA revealed an overall significant difference among the seven groups ($F(6, 319) = 3.04$, $P = .007$). Tukey post-hoc tests revealed significant differences only between the control group and the human tutored group ($P = .004$), and between the control group and iList-5 ($P = 0.021$). The point-to-point differences between the multiple versions of iList is not significant, but the progression of effect sizes indicates an overall positive performance trend. Although this is not a strong evidence that the latter versions of iList are better than the earlier ones, this result is encouraging and, overall, the performance of iList is very respectable compared to human tutors.

## 6.4 User satisfaction

After working with iList, students filled in a survey to assess their satisfaction with the system. The survey includes eight questions. The first seven are 5-point Likert scaled questions. The eighth question is an open ended question, asking the students for general comments on the system. Mean and standard deviation of students' scores on each question

TABLE IX

LEARNING GAIN OF STUDENTS IN SEVEN CONDITIONS

| Tutor | $N$ | Pre-test | | Post-test | | Gain | |
|---|---|---|---|---|---|---|---|
| | | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| None | 53 | .34 | .22 | .35 | .23 | .01 | .15 |
| iList-1 | 61 | .41 | .23 | .49 | .27 | .08 | .14 |
| iList-2 | 56 | .31 | .17 | .41 | .23 | .10 | .17 |
| iList-3 | 19 | .53 | .29 | .65 | .26 | .12 | .24 |
| iList-4 | 53 | .53 | .24 | .63 | .22 | .10 | .16 |
| iList-5 | 30 | .37 | .24 | .51 | .26 | .14 | .17 |
| Human | 54 | .40 | .26 | .54 | .26 | .14 | .25 |

for each version of the system are reported in Table X. Notice that each individual student has seen only one version of the system.

ANOVA found overall significant differences among the groups for question 1 ($F(4, 209) = 4.44$, $P = .002$), question 3 ($F(4, 209) = 3.56$, $P = .008$), question 5 ($F(4, 209) = 4.46$, $P = .002$), question 6 ($F(4, 209) = 4.27$, $P = .002$), and question 7 ($F(4, 209) = 5.33$, $P = .0004$). Tukey post-hoc tests pointed out the following significant differences:

- Question 1: iList-5 is more helpful than iList-1 ($P = .003$) and iList-2 ($P = .001$).
- Question 3: students read the feedback of iList-3 less than that of iList-4 ($P = .04$) and iList-5 ($P = .006$).
- Question 5: the feedback of iList-1 is less useful than that of iList-4 ($P = .004$) and iList-5 ($P = .01$).

TABLE X

SURVEY: SCALED QUESTIONS (1=NO TO 5=YES)

| Question | iList-1 | | iList-2 | | iList-3 | | iList-4 | | iList-5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| 1. Do you feel that iList helped you learn about linked lists? | 2.9 | 1.1 | 2.9 | 1.1 | 3.2 | 1.4 | 3.3 | 1.4 | 3.9 | 1.2 |
| 2. Do you feel that working with iList was interesting? | 4.0 | 1.1 | 3.8 | 1.1 | 3.3 | 1.3 | 3.8 | 1.2 | 4.0 | 1.2 |
| 3. Did you read the verbal feedback the system provided? | 4.1 | 1.1 | 4.1 | 1.0 | 3.5 | 1.5 | 4.3 | 1.1 | 4.6 | 0.8 |
| 4. Did you have any difficulty understanding the feedback? | 2.8 | 1.5 | 2.9 | 1.2 | 3.5 | 1.2 | 2.9 | 1.4 | 3.4 | 1.3 |
| 5. Did you find the feedback useful? | 2.6 | 1.2 | 3.0 | 1.0 | 2.8 | 1.4 | 3.4 | 1.1 | 3.4 | 1.2 |
| 6. Did you ever find the feedback misleading? | 2.3 | 1.3 | 2.4 | 1.1 | 3.2 | 1.4 | 2.8 | 1.4 | 3.2 | 1.3 |
| 7. Did you find the feedback repetitive? | 3.8 | 1.2 | 3.1 | 1.1 | 4.2 | 1.0 | 3.2 | 1.4 | 3.2 | 1.3 |

- Question 6: the feedback of iList-3 is more misleading than that of iList-1 ($P = .097$, marginally significant); the feedback of iList-5 is more misleading than that of iList-1 ($P = .01$) and iList-2 ($P = .02$).
- Question 7: the feedback of iList-1 is more repetitive that that of iList-2 ($P = .01$) and iList-4 ($P = .04$); the feedback of iList-3 is more repetitive than that of iList-2 ($P = .008$), iList-4 ($P = .02$), and iList-5 ($P = .06$, marginally significant).

Notice how the user satisfaction scores dropped in iList-3, most likely due to the bad experience students had because of the network incident that forced them to repeat the experiment. The acceptance of a tutoring system can be highly affected by external factors. This is an important concern when migrating from a controlled, experimental setting to the real world.

Overall, students liked the system, especially the latest versions. Interestingly, they found the feedback of the newer versions of iList more useful, but more difficult and sometimes misleading. It is not clear why the feedback may be considered misleading, as iList does not communicate incorrect information.

Linear regression showed significant correlations between some of the questions and learning gain. In particular:

- There is a positive correlation between question 1 (system helpfulness) and learning gain ($R^2 = .08$, $\beta = .29$, $F(1, 210) = 18.7$, $P < .001$).
- There is a positive correlation between question 5 (feedback usefulness) and learning gain ($R^2 = .02$, $\beta = .15$, $F(1, 210) = 4.94$, $P = .027$).

- There is a negative correlation between question 7 (feedback repetitiveness) and learning gain ($R^2 = .05$, $\beta = -.23$, $F(1, 210) = 11.3$, $P < .001$).

## 6.5 Log analysis

Now that we know that iList is effective in helping students learn and is well perceived by the students, it is important to understand in more detail the features of the interaction between the students and the systems that are mostly correlated with learning. The interaction of iList and the students has been comprehensively logged. From the log files, several features have been extracted and compared using ANOVA and linear regression (Table XI and Table XII). In the regression study, each variable has been used as a predictor of learning gain in multiple independent models.

### 6.5.1 Pre-test scores

Linear regression found a significant negative correlation between pre-test score and learning gain in iList-3, iList-4, and in the combined dataset (Table XI). It is not surprising that this correlation is stronger in the groups of students that worked with iList-3 and iList-4, as they had the highest pre-test scores (Table IX), suggesting a stronger ceiling effect that does not show up in the other three groups instead.

### 6.5.2 Working memory capacity

Although the collection of pre-test scores aims to take into account students' previous knowledge, there is much more to students' individual characteristics than what can be captured with the pre-test, and these "hidden" student features might have a profound impact on their learning. With the introduction of iList-2, I started to collect a measure

TABLE XI

LINEAR REGRESSION: FEATURES OF ILIST ON LEARNING GAIN

| Feature | System | Descriptive | | | Regression on learning gain | | | |
|---|---|---|---|---|---|---|---|---|
| | | $N$ | $\mu$ | $\sigma$ | $\beta$ | $t$ | $P$ | $R^2$ |
| Pre-test score | iList-1 | 61 | .41 | .23 | *ns* | | | |
| | iList-2 | 56 | .31 | .17 | *ns* | | | |
| | iList-3 | 19 | .53 | .29 | −.53 | −2.54 | .02 | .23 |
| | iList-4 | 53 | .53 | .24 | −.48 | −3.90 | < .001 | .21 |
| | iList-5 | 30 | .37 | .24 | *ns* | | | |
| | All | 219 | .42 | .24 | −0.23 | −3.40 | < .001 | .05 |
| Operation span (words) | iList-1 | 0 | N/A | | N/A | | | |
| | iList-2 | 56 | 29.8 | 8.5 | −.24 | −1.78 | .08 | .04 |
| | iList-3 | 18 | 30.1 | 8.4 | *ns* | | | |
| | iList-4 | 53 | 29.4 | 10.0 | *ns* | | | |
| | iList-5 | 30 | 29.4 | 9.4 | *ns* | | | |
| | All | 157 | 29.6 | 9.1 | *ns* | | | |
| Operation span (math) | iList-1 | 0 | N/A | | N/A | | | |
| | iList-2 | 56 | 41.2 | 1.3 | −.28 | −2.12 | .04 | .06 |
| | iList-3 | 18 | 41.4 | .9 | *ns* | | | |
| | iList-4 | 53 | 40.9 | .9 | *ns* | | | |
| | iList-5 | 30 | 41.0 | 1.3 | *ns* | | | |
| | All | 157 | 41.1 | 1.2 | *ns* | | | |
| Time (minutes) | iList-1 | 61 | 41.9 | 17.6 | *ns* | | | |
| | iList-2 | 56 | 32.7 | 8.6 | *ns* | | | |
| | iList-3 | 19 | 30.0 | 14.8 | *ns* | | | |
| | iList-4 | 53 | 24.9 | 10.9 | *ns* | | | |
| | iList-5 | 30 | 54.7 | 16.3 | *ns* | | | |
| | All | 219 | 36.2 | 16.7 | *ns* | | | |

TABLE XII

LINEAR REGRESSION: MORE FEATURES OF ILIST ON LEARNING GAIN

| Feature | System | Descriptive | | | Regression on learning gain | | | |
|---|---|---|---|---|---|---|---|---|
| | | $N$ | $\mu$ | $\sigma$ | $\beta$ | $t$ | $P$ | $R^2$ |
| Problems attempted | iList-1 | 61 | 6.2 | 1.3 | .24 | 1.90 | .06 | .04 |
| | iList-2 | 56 | 5.6 | 1.6 | *ns* | | | |
| | iList-3 | 19 | 7.0 | 0 | *ns* | | | |
| | iList-4 | 53 | 5.7 | 1.9 | .35 | 2.66 | .01 | .10 |
| | iList-5 | 30 | 6.8 | .5 | *ns* | | | |
| | All | 219 | 6.1 | 1.5 | .21 | 3.15 | .002 | .04 |
| Problems solved | iList-1 | 61 | 4.0 | 2.5 | .35 | 2.84 | .006 | .11 |
| | iList-2 | 56 | 2.7 | 2.2 | .42 | 3.37 | .001 | .16 |
| | iList-3 | 19 | 6.7 | 1.1 | *ns* | | | |
| | iList-4 | 53 | 4.3 | 2.3 | .29 | 2.16 | .04 | .07 |
| | iList-5 | 30 | 6.0 | 1.1 | *ns* | | | |
| | All | 219 | 4.2 | 2.5 | .31 | 4.73 | $< .001$ | .09 |
| Student actions | iList-1 | 61 | 159 | 67 | *ns* | | | |
| | iList-2 | 56 | 110 | 51 | *ns* | | | |
| | iList-3 | 19 | 96 | 38 | *ns* | | | |
| | iList-4 | 53 | 80 | 40 | *ns* | | | |
| | iList-5 | 30 | 132 | 56 | *ns* | | | |
| | All | 219 | 118 | 61 | *ns* | | | |
| Actions density (act/min) | iList-1 | 61 | 4.2 | 1.8 | *ns* | | | |
| | iList-2 | 56 | 3.4 | 1.2 | *ns* | | | |
| | iList-3 | 19 | 3.5 | 1.2 | *ns* | | | |
| | iList-4 | 53 | 3.3 | 1.0 | *ns* | | | |
| | iList-5 | 30 | 2.4 | .7 | *ns* | | | |
| | All | 219 | 3.5 | 1.4 | *ns* | | | |
| Path goodness | iList-1 | 61 | .22 | .14 | .33 | 2.72 | .008 | .10 |
| | iList-2 | 56 | .15 | .11 | .43 | 3.53 | $< .001$ | .17 |
| | iList-3 | 19 | .38 | .09 | *ns* | | | |
| | iList-4 | 53 | .28 | .11 | .32 | 2.40 | .02 | .08 |
| | iList-5 | 30 | .30 | .10 | *ns* | | | |
| | iList-1+2 | 117 | .19 | .13 | .35 | 3.97 | $< .001$ | .11 |
| | iList-3+4+5 | 102 | .31 | .11 | .20 | 2.00 | .049 | .03 |
| | All | 219 | .24 | .13 | .28 | 4.30 | $< .001$ | .08 |

of *working memory capacity* (Kyllonen and Christal, 1990; Conway et al., 2003), assessed with an operation span test (Conway et al., 2005) integrated in the interface of iList. The operation span test requests the student to remember a sequence of words that briefly appear on the screen, interleaved by simple but distracting arithmetic operations that the student has to solve. Previous research showed that working memory capacity correlates very well with other measures of general cognitive abilities, and the operation span test can be taken quickly and easily by the students.

Linear regression revealed significant correlations only for iList-2, but no correlation overall. Specifically, a marginally significant correlation was found between the word score (which is the main score) of the operation span test and learning gain, and a significant correlation was found between the math score of operation span and learning gain (Table XI). Notice that, in both cases, the correlation is negative, suggesting that students with higher working memory capacity learned less than those with lower working memory capacity.

A note of caution should be made on this result. During the experiments, some students were caught cheating while taking the operation span test, i.e., they wrote down the words to be remembered instead of memorizing them. If this behavior was common, which is not known, then the working memory capacity measures can not be considered reliable.

### 6.5.3 Time on task

ANOVA revealed an overall significant difference in the time the different groups spent working with iList ($F(4, 214) = 27.0$, $P < .001$). Tukey post-hoc confirmed that all the pairs are indeed different, except the pairs iList-2/iList-3 and iList-3/iList-4. Linear regression

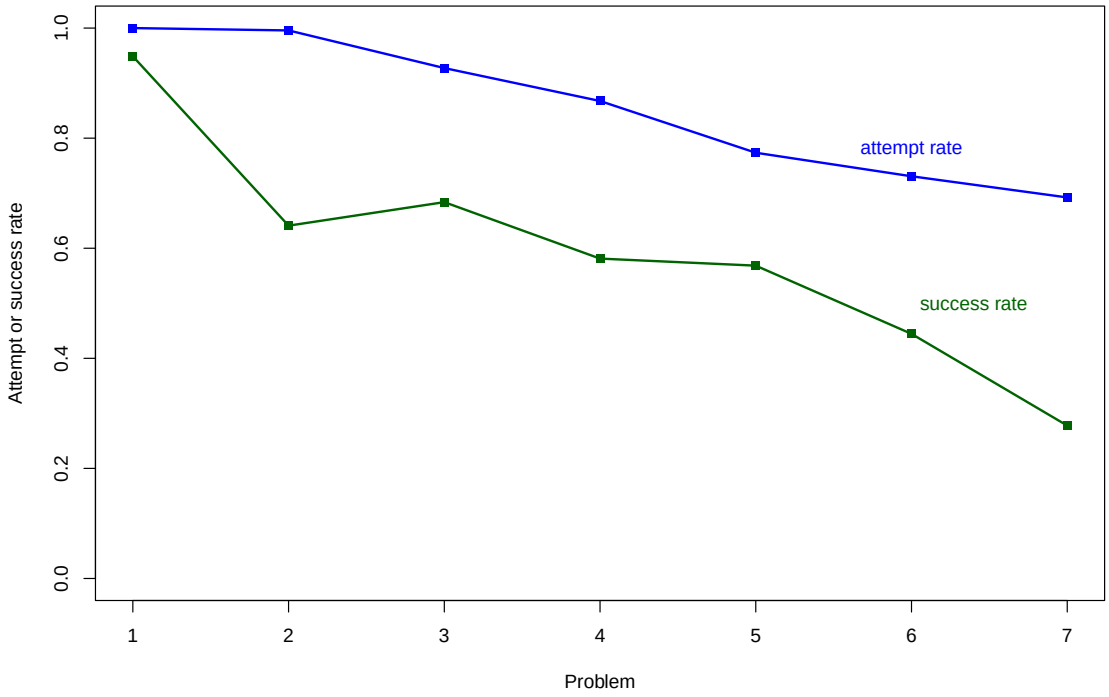



Figure 12. Attempt and success rates per problem

revealed no significant correlation between the time spent by the students with the system and learning gain. This is surprising, because it contradicts our result with human tutors, where we found a significant positive correlation between the time a student interacted with the tutor and learning gain.

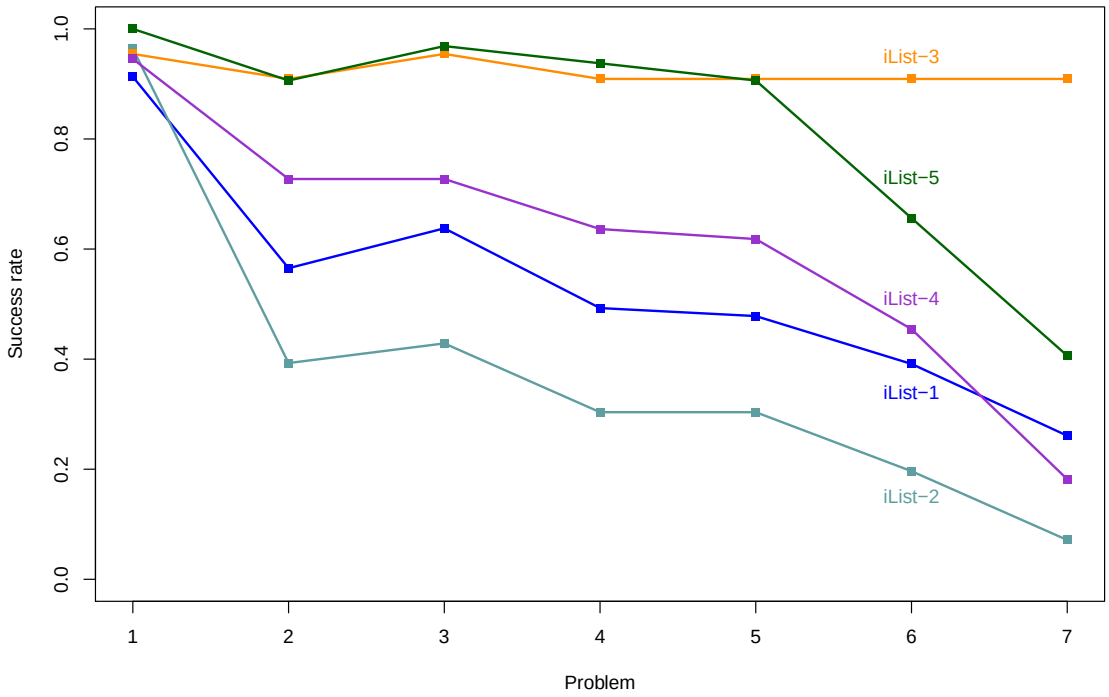



Figure 13. Success rates per problem, per system

### 6.5.4 Problem solving

The problems included in iList's curriculum are of increasing difficulty, as can be seen from the success rate for each problem (Figure 12 and Figure 13). ANOVA revealed overall significant differences among the five groups on the number of problems attempted ($F(4, 214) = 6.68$, $P < .001$) and the number of problems successfully solved by the students ($F(4, 214) = 19.5$, $P < .001$). In particular, Tukey post-hoc test pointed out significant differences in the following pairs:

- On the number of problems attempted, iList-2 < iList-3 ($P = .002$); iList-2 < iList-5 ($P = .002$); iList-4 < iList-3 ($P = .007$); iList-4 < iList-5 ($P = .007$).
- On the number of problem solved, all the pairs are significantly different ($P < .05$), except iList-1/iList-4 and iList-3/iList-5.

Linear regression showed an overall positive correlation between problem solving and learning (Table XII). In particular, stronger correlations can be seen in the number of problems solved and learning: generally, students that solved more problems also learned more. Notice that the three groups that worked with a version of iList enhanced with procedural feedback (iList-3, iList-4, and iList-5) generally solved more problems, possibly because of the help provided by the feedback.

### 6.5.5 Student activity

A measure of student activity is the number of actions students took while solving problems. An action is either a programming command (correct or incorrect), or an

undo/redo/restart meta-command. Action density is defined as number of actions per minute (Table XII). A larger action density suggests a more pronounced "point and click" student behavior; a smaller action density suggests that students spent more time thinking or reading feedback.

ANOVA showed overall significant differences in the number of student actions ($F(4, 214) = 17.3$, $P < .001$) and in action density ($F(4, 214) = 9.1$, $P < .001$). In particular, Tukey post-hoc tests indicated the following:

- Students working with iList-1 took more actions than students working with iList-2 ($P < .001$), iList-3 ($P < .001$), and iList-4 ($P < .001$); students working with iList-4 took fewer actions than those working with iList-2 ($P = .03$) and iList-5 ($P < .001$).
- Students working with iList-1 had a larger action density than those working with iList-2 ($P = .01$), iList-4 ($P = .009$), and iList-5 ($P < .001$); students working with iList-5 had a smaller action density than those working with all the other versions of iList ($P$ values ranging from $P = .055$ to $P < .001$).

Linear regression revealed no significant correlation between the number of student actions and learning gain, nor between action density and learning gain.

### 6.5.6 Feedback messages

Since the main design difference between the various versions of iList is the type and level of feedback they provide, it is interesting to take a look at the number of feedback messages that the system actually generated during the classroom experiments (Table XIII,

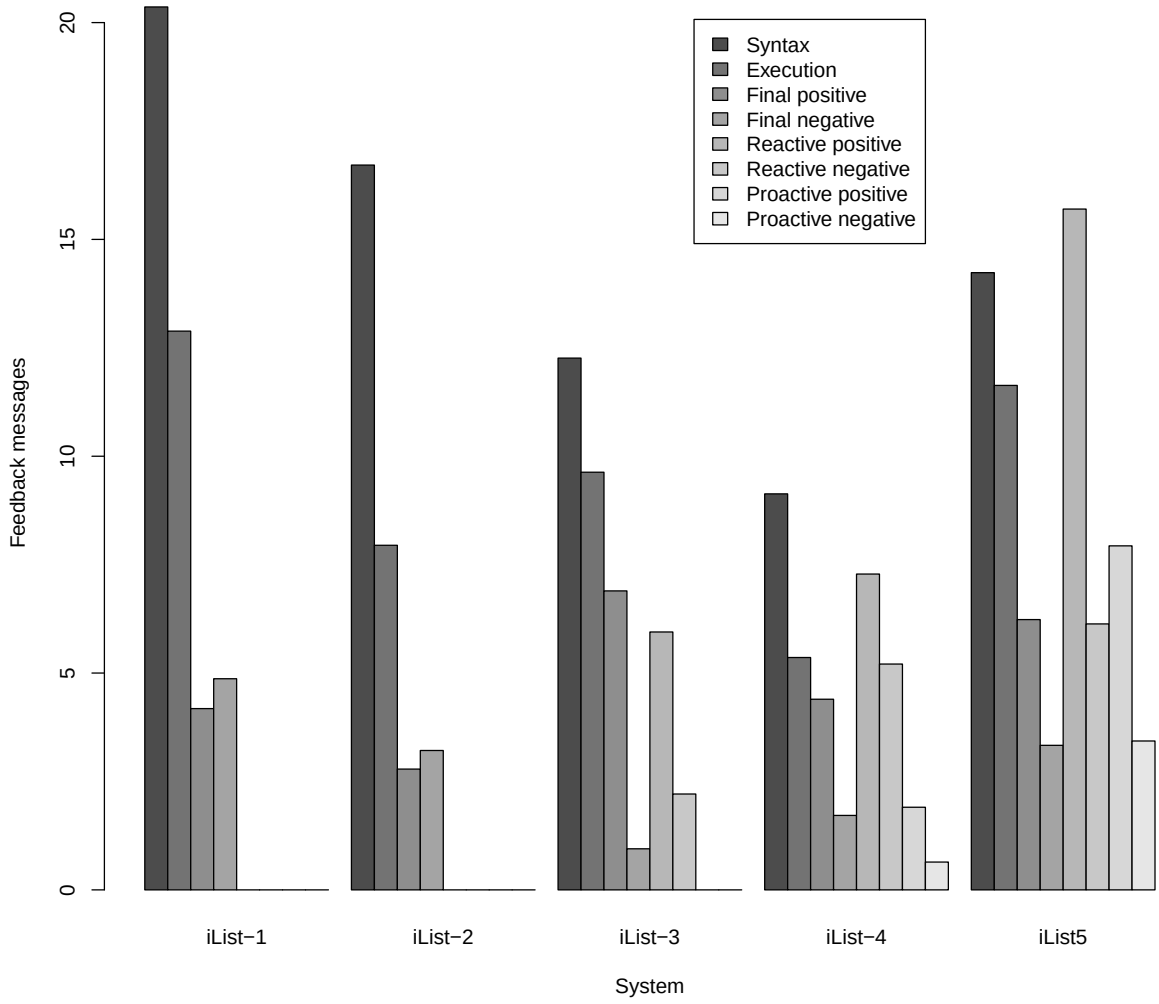



Figure 14. Bar chart: number of feedback messages of all types

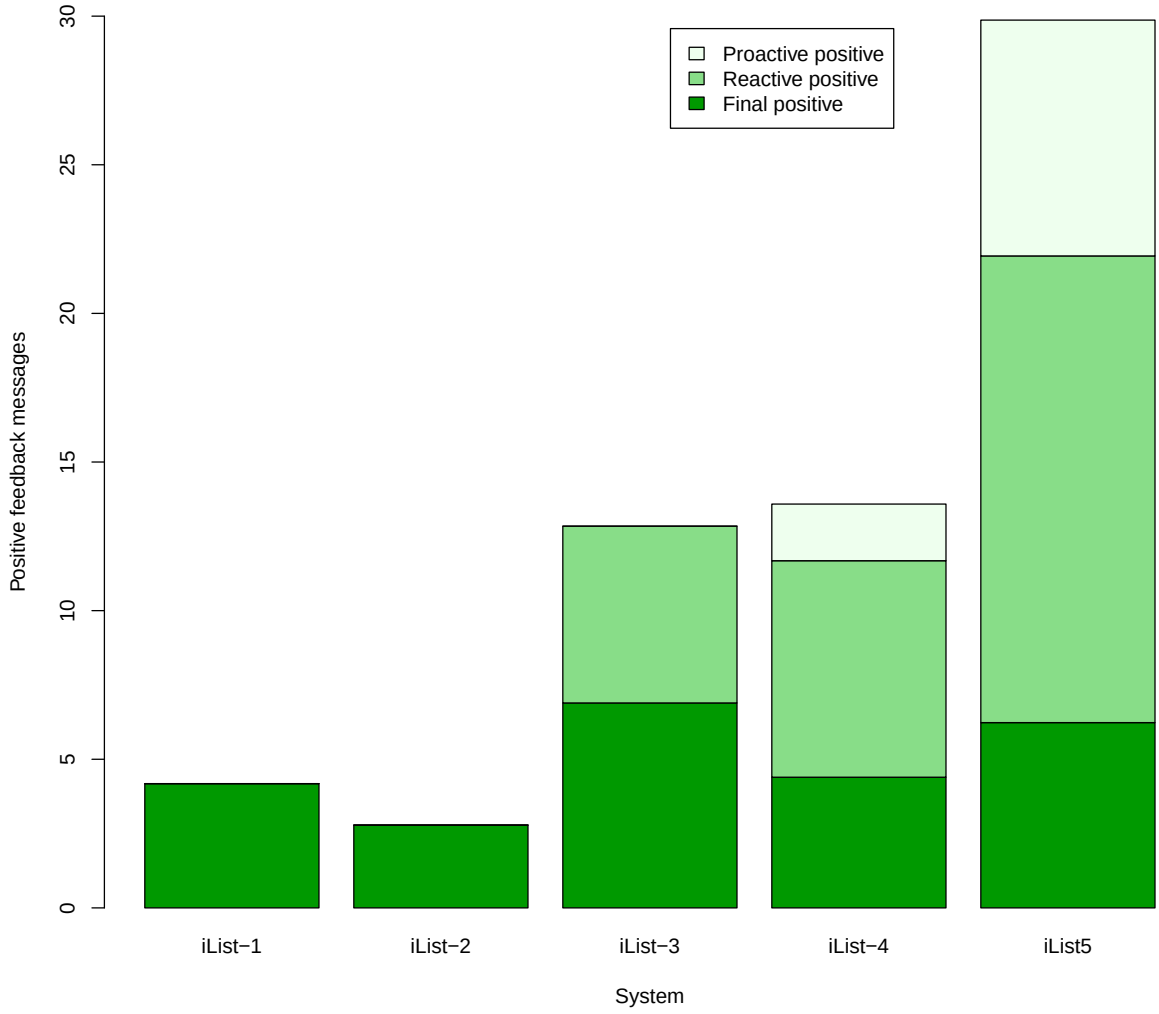



Figure 15. Number of positive feedback messages

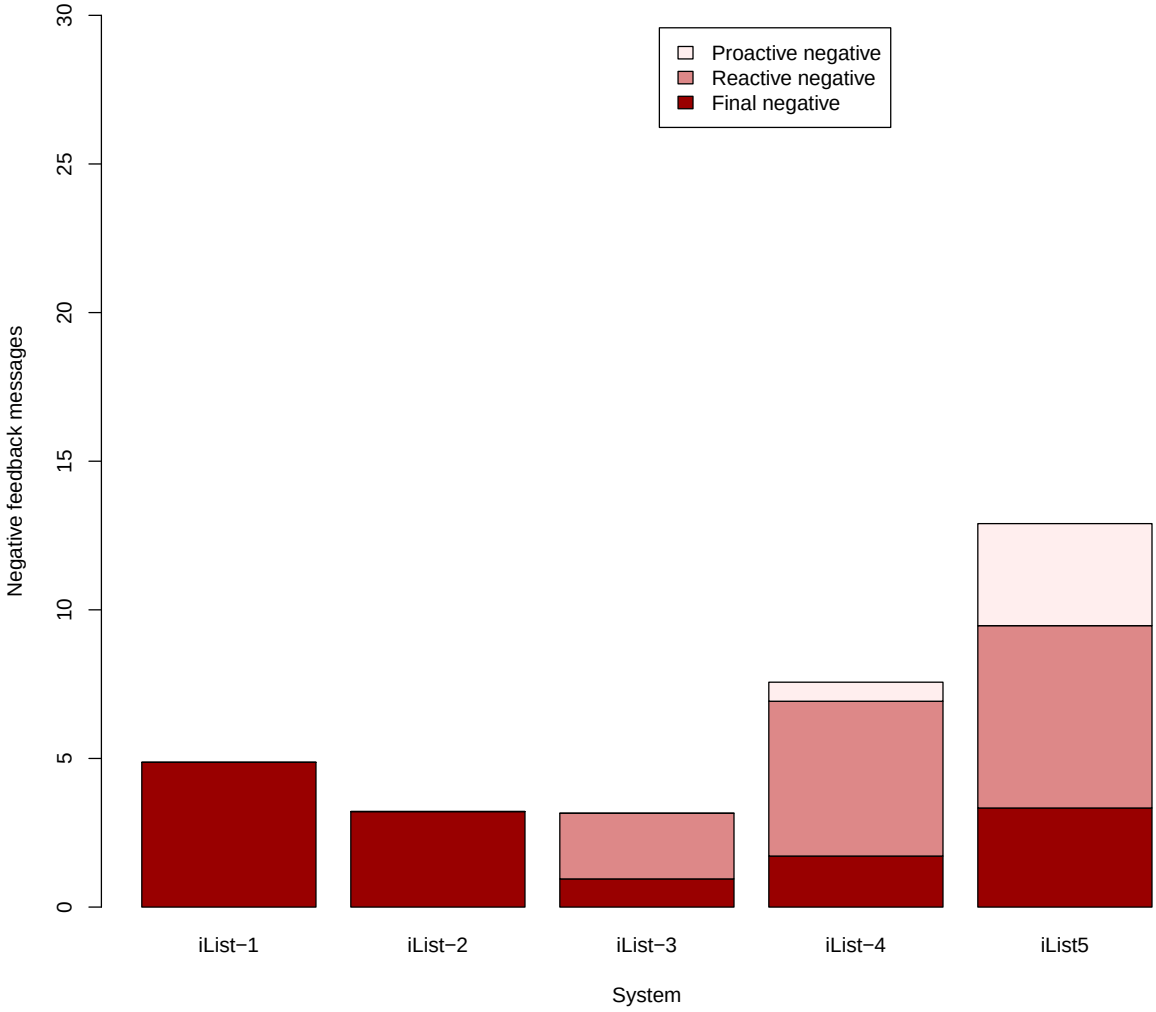



Figure 16. Number of negative feedback messages (except syntax and execution)

TABLE XIII

NUMBER OF FEEDBACK MESSAGES OF ALL TYPES

| Feedback type | iList-1 | | iList-2 | | iList-3 | | iList-4 | | iList-5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ |
| Syntax | 20.4 | 14.1 | 16.7 | 12.2 | 12.3 | 7.1 | 9.1 | 6.1 | 14.2 | 12.7 |
| Execution | 12.9 | 10.5 | 7.9 | 6.0 | 9.6 | 5.7 | 5.4 | 4.6 | 11.6 | 8.8 |
| Final, positive | 4.2 | 2.7 | 2.8 | 2.3 | 6.9 | 1.3 | 4.4 | 2.4 | 6.2 | 1.1 |
| Final, negative | 4.9 | 5.3 | 3.2 | 2.9 | .9 | 1.3 | 1.7 | 1.9 | 3.3 | 4.3 |
| Reactive, positive | 0 | 0 | 0 | 0 | 5.9 | 5.1 | 7.3 | 5.4 | 15.7 | 9.2 |
| Reactive, negative | 0 | 0 | 0 | 0 | 2.2 | 2.5 | 5.2 | 4.5 | 6.1 | 5.3 |
| Proactive, positive | 0 | 0 | 0 | 0 | 0 | 0 | 1.9 | 1.7 | 7.9 | 4.3 |
| Proactive, negative | 0 | 0 | 0 | 0 | 0 | 0 | .64 | .86 | 3.4 | 2.0 |

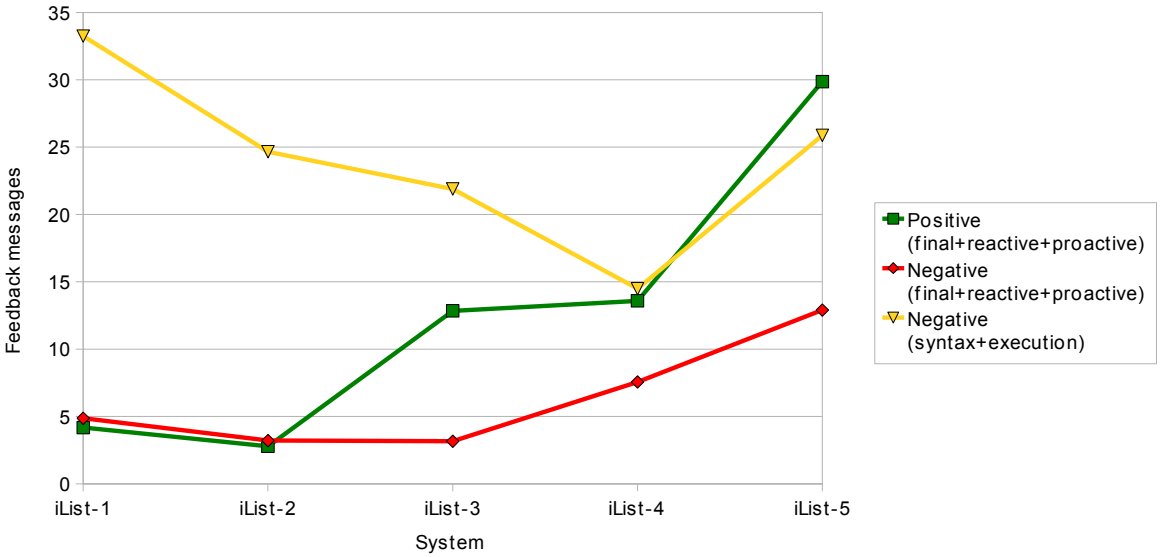



Figure 17. Number of feedback messages grouped in three categories

Figure 14, Figure 15, Figure 16, Figure 17). The Procedural Knowledge Model allows iList to provide a greater amount of feedback, in particular positive feedback, in addition to the more primitive syntax, execution, and final feedback. This makes the latest three versions of iList able to cover more learning opportunities than the first two versions of the system. The different groups of students received significantly different amounts of feedback of every type (all the ANOVAs have overall $P < .001$, pairwise differences vary). Notice that the number of positive final feedback messages is almost equal to the number of problems solved by the students, as usually a student receives only a single "good job, you have solved the problem" message before moving to the next problem. Also, notice that the amount of positive procedural (reactive + proactive) feedback increased remarkably in the most recent versions of the system. In iList-5, the ratio of positive procedural feedback over negative procedural feedback is approximately 2.5 to 1.

Interestingly, linear regression couldn't find any significant correlation between the number of feedback messages of various types and learning gain (excluding positive final messages, which we already know are equivalent to number of problems solved).

### 6.5.7 Student path goodness

A way to measure the behavioral change that procedural feedback could have induced on students is to use the Procedural Knowledge Model to assess the "goodness" of student paths, defined as the average goodness of the states visited by a student while solving a problem. If the procedural feedback had an effect, we should see an increase in the overall path goodness from the first two versions of iList, which did not provide procedural feedback,

and the latest three, which provided procedural feedback of varying degree. Indeed, we can see such an increase (Table XII). ANOVA found overall significant differences ($F(4, 214) = 19.1$, $P < .001$). Tukey confirmed all the pairwise differences ($P < .05$), except iList-3/iList-5 and iList-4/iList-5 which are not significantly different, and iList-1/iList-4 which are marginally different ($P = .089$). The difference between iList-1 and iList-2 is unexpected. Linear regression found significant positive correlations between path goodness and learning gain. This finding supports the validity of the Procedural Knowledge model and the strategy of iList, which guides the students towards the "right path."

## 6.6 Summary

Despite the difficulties that arose from differences in student population between the groups and other unanticipated variables, the evaluation study showed that iList, in all its versions but particularly in the most advanced ones, is a useful system that helps students learn. The learning gain obtained in these experiments is statistically indistinguishable from the human tutors used in this study. Students liked working with iList and found it useful. The different forms of feedback are correlated with differences in student behavior which are in turn correlated with learning gain. In particular, the procedural feedback generated by the most sophisticated versions of the system is helpful in keeping the students on the right path.

# CHAPTER 7

# CONCLUSIONS AND FUTURE WORK

## 7.1 Contributions

This dissertation presents a number of significant research contributions. This work further substantiates the methodological validity of the development of Intelligent Tutoring Systems grounded on empirical evidence of effective tutorial strategies extracted from a careful analysis of human tutoring data.

The human tutoring study presented in this dissertation contributes a new corpus of human tutoring dialogues in the domain of Computer Science data structures; a new tool for the annotation of text, audio, and video corpora; and compelling results on the importance of positive feedback elicited from proactive interactional episodes.

The tutoring system developed in this project, iList, is innovative in many ways. First of all, iList is among the first systems that tutor Computer Science data structures. The graphical interface of iList is designed to help bring an abstract and difficult concept like linked lists to a more "tangible" level. Moreover, iList brings together the static representations of lists to the procedures used to manipulate them. I believe the most innovative feature of iList is its Procedural Knowledge Model automatically extracted from previous interactions of students with the system. To date, iList is among the first systems that can

build such a statistical model automatically and is the first one that uses it to provide the kind of reactive and proactive procedural feedback described in this dissertation.

The evaluation of iList was conducted in a "real world" setting, where more than 200 students worked with the system in multiple classrooms. The results showed that iList is as effective as human tutors in helping students learn; students liked working with the system and found it useful; and the procedural feedback automatically generated by the most advanced versions of iList effectively guides students towards the right solution paths.

The research community already appreciated this work in a number of occasions, as many of the results discussed in this dissertation have already been published in prestigious conference and journal articles (Ohlsson et al., 2007; Fossati, 2008; Fossati et al., 2008; Fossati et al., 2009a; Fossati et al., 2009b; Di Eugenio et al., 2009).

## 7.2 Future directions

This research offers many possibilities for future work. The following list includes some interesting directions.

- The study of human tutoring can be enriched with the annotation and analysis of additional tutorial features that are potentially correlated with learning. Our research group is currently working on the annotation of features such as *prompts* and *direct declarative instruction*.
- The iList system is now mature for a public release, so the community at large could benefit from it. I am planning to release iList under an Open Source licensing. In this way, the research and educational communities could also contribute to its fur-

ther development. For example, Computer Science educators could easily write new problems that can be integrated in the curriculum of iList.

- The effectiveness and the interactional features of iList could be evaluated in a more controlled setting, without necessarily losing the "real world" character of the current evaluation approach. In particular, a rigorous randomized study involving groups extracted from the same student population (e.g., different sections of the same course, in the same semester, with the same instructor) would reduce the statistical uncertainty and improve the clarity of the results.
- Currently, the feedback messages generated by iList are "self-contained" and they are not organized in a dialogue history where messages can refer to or build on previous ones. Adding this capability would be an interesting improvement to the feedback generator.
- It would be interesting to directly compare iList with KSC-PaL, an educational system about linked lists that adopts an innovative "peer behavior" paradigm (Kersey et al., 2008; Kersey et al., 2009).

## 7.3 Long term research directions

Working on Intelligent Tutoring Systems made me realize the importance of understanding the behavior and knowledge of students to provide better education. In the future, I would like to explore how computational models can be applied to enhance the practice of *assessment* in educational contexts.

Impressive advances in the fields of Data Mining and Artificial Intelligence, combined with the insights provided by Cognitive and Educational Psychology, make it possible to envision the creation of Intelligent Assessment Systems (IASs), tools that can help teachers collect and analyze data to facilitate *formative assessment*. The concept of formative assessment, originally expressed by educational psychologists in the late sixties, formalizes the idea of testing the knowledge of students in order to personalize instructional intervention and enhance their learning opportunities (Bransford et al., 2002; Pellegrino et al., 2001). This is in contrast to the traditional practice of summative assessment, where the main purpose of testing is to assign grades and rank student performance. Although summative assessment is certainly important for practical reasons, the inclusion of more formative assessment into instructional practice could significantly improve the effectiveness of teaching.

Formative assessment, although very elegant in theory, can be difficult to apply in practice. One of the reasons is that a teacher may need to collect large quantities of data, thoroughly analyze it to discover important trends and patterns, and use the findings to design appropriate teaching responses at the level of individual students and at the scale of the entire classroom. This process may be overwhelming and unrealistically expensive in terms of teacher time and class time. This is why Intelligent Assessment Systems can be very helpful. Such tools would noninvasively monitor students' learning processes, by creating appropriate tests and automatically evaluating them; store, filter, and classify relevant data; and automatically or semi-automatically discover important patterns that are promptly reported to teachers so they can make informed decisions about what to do

next. This could be seen as an educational equivalent of the Management Information Systems that are successfully used in business.

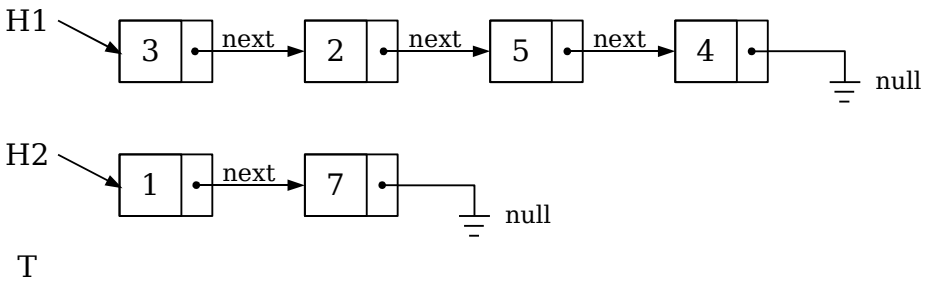
Research on Intelligent Assessment Systems can rely on the results of the Intelligent Tutoring System (ITS) community, but it has peculiarities that require different approaches and methods. For example, an IAS would need to collect and interpret data across a whole classroom, whereas an ITS usually focuses on individual students. Moreover, an IAS would need to work on a larger scope in terms of time and topics, for example to cover an entire course, whereas ITSs usually focus on specific topics for shorter periods of time.

# APPENDICES

# Appendix A

## PRE/POST TEST FOR HUMAN TUTORING

You have the following two *linked lists*, starting from the head pointers H1 and H2. You also have a temporary pointer T.




1. Look at the following procedure. The procedure is written in pseudo C/C++/Java, but don't worry about programming details such as declarations, etc. What is the status of the data structures after its execution? Draw a picture representing them.

```
T = H2;
while (T.next ≠ null) {
    T = T.next;
}
T.next = H1;
```

2. Consider the following "variation" of the same procedure. Why doesn't it work?

```
T = H2;
while (T ≠ null) {
    T = T.next;
}
T.next = H1;
```

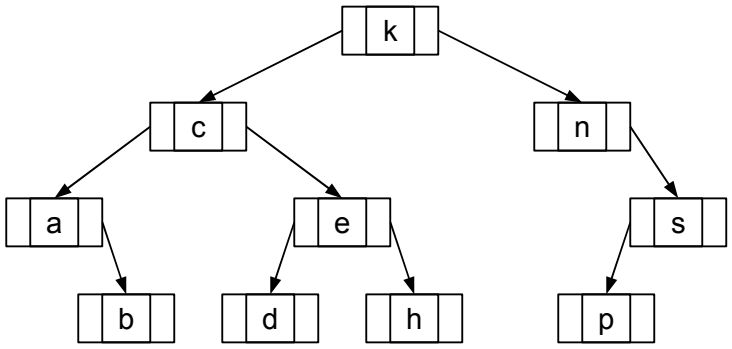
You have a *stack* data structure. The operations defined on it are *push*, *pop*, and *top*, with the usual semantics for stacks.

3. Your stack is initially empty. Write down the state of the stack and the content of the variable x after executing each of the following operations.

| *Operation* | *Stack* | *x* |
|---|---|---|
| `x = "A";` | | |
| `push("B");` | | |
| `push("C");` | | |
| `push(x);` | | |
| `pop();` | | |
| `x = top();` | | |
| `push("A");` | | |
| `push(top());` | | |

4. Mention an application (either a computer application or a real world situation) in which you think a stack structure may be appropriate.

The following picture represents a *binary search tree* (BST).




5. Show the keys with which "d" will be compared if you search for "d" in the given tree.

6. Insert a node containing "m" into the binary search tree. Draw the resulting tree.

7. Delete the node that contains "c" from the original binary search tree. Draw two possible final trees.

8. In general, is it more difficult to insert a new node in a BST or to delete an existing node from a BST? Explain why.

# Appendix B

## PRE/POST TEST GRADING SCHEME (HUMAN TUTORING)

### General guidelines

Cover the student information and test type (pre/post) on the top of each test with a removable label. Shuffle all the tests, and grade them in random order. Each problem shall be graded with a number from 0 to 5. Do not write on the original test.

| **Problem** | **Score** | **Criteria** |
|---|---|---|
| 1 | 5 | The solution is correct and complete. |
| | 4 | The solution is correct, but something minor is missing (e.g., the state of temporary pointers, or the null pointer at the end of the lists) |
| | 3 | There are some minor mistakes. The overall solution reveals some understanding of the topic. |
| | 2 | There are some serious mistakes, revealing flaws in the understanding of the topic. |
| | 1 | The solution is totally wrong or garbled. |
| | 0 | The answer is blank. |
| 2 | 5 | The explanation is correct, complete, and well formulated. |
| | 4 | The explanation is correct but not so well formulated. |
| | 3 | The explanation makes some sense, but it is somewhat fuzzy. |
| | 2 | The explanation does not make much sense. |
| | 1 | The explanation is wrong or garbled. |
| | 0 | The answer is blank. |
| 3 | 5 | The solution is correct and complete. |

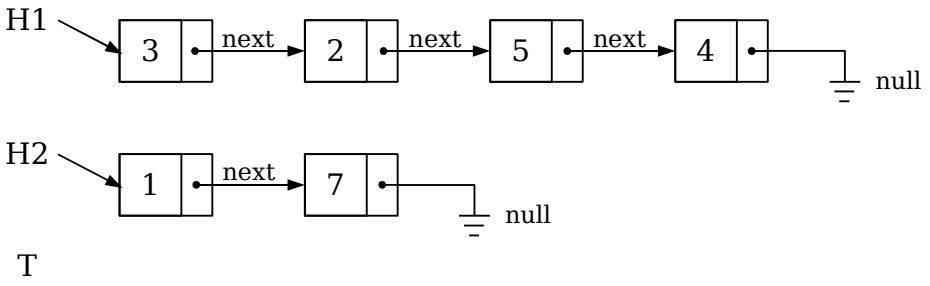
| **Problem** | **Score** | **Criteria** |
|---|---|---|
| | 4 | The solution is essentially correct, there may be at most one non-repeating mistake due to distraction. |
| | 3 | There are some mistakes or the solution is incomplete. |
| | 2 | There are repeating mistakes, revealing flaws in the understanding of the topic. |
| | 1 | The solution is totally wrong or garbled. |
| | 0 | The answer is blank. |
| 4 | 5 | The example provided is correct and well formulated. The example is also well centered: a stack is the only or the best model fitting the example. |
| | 4 | The example provided is correct, but a stack structure does not necessarily best fit the example, or the example is not well formulated. |
| | 3 | The example makes some sense, but it is somewhat confused. |
| | 2 | The example does not make much sense or is very confused. |
| | 1 | The answer is totally wrong or garbled. |
| | 0 | The answer is blank. |
| 5 | 5 | The solution is correct. |
| | 4 | The solution misses the last comparison. |
| | 3 | There are some mistakes, but the answer reveals some understanding. |
| | 2 | There are serious mistakes. |
| | 1 | The solution is totally wrong or garbled. |
| | 0 | The answer is blank. |
| 6 | 5 | The solution is correct. |

| **Problem** | **Score** | **Criteria** |
|---|---|---|
| | 4 | The solution is correct but somewhat "non standard." |
| | 3 | The solution is incorrect but reveals some understanding. |
| | 2 | The solution is incorrect and reveals serious flaws in understanding. |
| | 1 | The solution is totally wrong or garbled. |
| | 0 | The answer is blank. |
| 7 | 5 | The solution is correct and complete. |
| | 4 | The solution is correct but incomplete (only 1 solution) or somewhat "non standard." |
| | 3 | The solution is correct but incomplete and "non standard." |
| | 2 | The solution is somewhat wrong. |
| | 1 | The solution is seriously wrong or garbled. |
| | 0 | The answer is blank. |
| 8 | 5 | The answer is correct, and the explanation is sound and well formulated. |
| | 4 | The answer is correct, the explanation is essentially correct but not so well formulated. |
| | 3 | The answer is correct, the explanation makes some sense, but it is somewhat fuzzy and does not capture the main points. |
| | 2 | The explanation doesnt make much sense or is contradictory. |
| | 1 | The answer is totally wrong or garbled. |
| | 0 | The answer is blank. |

# Appendix C

## PRE/POST TEST FOR ILIST

You have the following two *linked lists*, starting from the head pointers H1 and H2. You also have a temporary pointer T.




1. Look at the following procedure. The procedure is written in pseudo C/C++/Java, but don't worry about programming details such as declarations etc. What is the status of the data structures after its execution? Draw a picture representing them.

```
T = H2;
while (T.next != null) {
    T = T.next;
}
T.next = H1;
```

2. Consider the following "variation" of the same procedure. Why doesnt it work?

```
T = H2;
while (T != null) {
    T = T.next;
}
T.next = H1;
```

3. Write a sequence of operations, in pseudo-code or in a programming language of your choice, that moves the first node of the original list H1 to the end of the list.

# Appendix D

## PRE/POST TEST GRADING SCHEME (ILIST)

### General guidelines

Each problem shall be graded with a number from 0 to 5. Do not write on the original test.

| **Problem** | **Score** | **Criteria** |
|---|---|---|
| 1 | 5 | The solution is correct and complete. |
| | 4 | The solution is correct, but something minor is missing (e.g., the state of temporary pointers, or the null pointer at the end of the lists) |
| | 3 | There are some minor mistakes. The overall solution reveals some understanding of the topic. |
| | 2 | There are some serious mistakes, revealing flaws in the understanding of the topic. |
| | 1 | The solution is totally wrong or garbled. |
| | 0 | The answer is blank. |
| 2 | 5 | The explanation is correct, complete, and well formulated. |
| | 4 | The explanation is correct but not so well formulated. |
| | 3 | The explanation makes some sense, but it is somewhat fuzzy. |
| | 2 | The explanation does not make much sense. |
| | 1 | The explanation is wrong or garbled. |
| | 0 | The answer is blank. |
| 3 | 5 | The solution is completely correct. |

| **Problem** | **Score** | **Criteria** |
|---|---|---|
| | 4 | There is one mistake. |
| | 3 | There are two mistakes. |
| | 2 | There are some mistakes, but the solution makes some sense. |
| | 1 | The solution is totally wrong or garbled. |
| | 0 | The answer is blank. |

# Appendix E

## CODING SCHEME FOR CS TUTORING DATA: FEEDBACK

We will code the CS tutoring protocols for *episodes*. An episode is a sequence of consecutive utterances that can be grouped together into one of the following categories.

### Positive feedback

The student says or does something correct, either spontaneously or after being prompted by the tutor. The tutor acknowledges the correctness of the student's claim, and possibly elaborates on it with further explanation. Example:

| | | |
|---|---|---|
| | [00:13:15.03]*LOW: | do you see a problem? |
| | [00:13:17.09]*LOW: | I have found the node a@l, see here I found the node b@l, and then I put g@l in after it. |
| *Begin positive* | [00:13:27.08]*LOW: | here I have found the node a@l and now the link I have to change is +... |
| | [00:13:36.21]*112: | ++ you have to link e@l <over xxx.> [>] |
| *End positive* | [00:13:38.08]*LOW: | [<] <yeah> I have to go back to this one. |
| | [00:13:39.29]*112: | *mmhm |
| | [00:13:41.02]*LOW: | so I *uh once I'm here, this key is here, I can't go backwards. |

### Negative feedback

The student says or does something wrong, either spontaneously or as an answer to a tutor's prompt. The tutor reacts to the mistake and possibly provides some form of explanation or remediation. Example:

| | | |
|---|---|---|
| *Begin negative* | [00:06:23.08]*112: | <so you> [>] <you won't get the same> [//] would you get the same point out of writing t@l close to c@l at the top? |
| | [00:06:29.02]*LOW: | oh, t@l equals c@l. |
| | [00:06:30.02]*LOW: | no because you would have a type mismatch. |
| *End negative* | [00:06:32.07]*LOW: | t@l <is a pointer> [//] is an address, and this is contents. |

**General guidelines and special cases**

1. When consecutive positive or negative feedback episodes are present in the data, it may happen that the end of an episode overlaps with the beginning of the following one. In this case, start the beginning of the new episode one utterance later. Example:

| | | |
|---|---|---|
| *Begin positive* | [00:25:32.06]*112: | +< six is xxx xx so you would have to move to the right. |
| *End positive* | [00:25:34.24]*LOW: | it would have to be over there. |
| *Begin positive* | [00:25:36.00]*LOW: | if the 6 was in this tree it would have to be +... |
| | [00:25:39.22]*112: | ++ to the left of nine. |
| *End positive* | [00:25:40.22]*LOW: | and then it would have to be +... |
| *Begin positive* | [00:25:41.26]*112: | ++ left of eight. |
| *End positive* | [00:25:42.28]*LOW: | and then it would have to be +... |
| *Begin positive* | [00:25:43.21]*112: | ++ left of seven |
| *End positive* | [00:25:44.28]*LOW: | and that is nothing there, so we put six right there. |

2. Sometimes the tutor prompts the student, expecting an answer from him/her. In this case, include the tutor's prompt in the episode. In other cases, the tutor is just interrupted by the student, and the tutor provides feedback on what the student says. In this case, do *not* include the utterance of the tutor right before the interruption. Examples:

| | | |
|---|---|---|
| *Begin positive* | [00:30:25.19]*LOW: | or if you pick the smallest one over here +... |
| | [00:30:28.08]*112: | it'll be greater than. |
| *End positive* | [00:30:29.11]*LOW: | it'll be greater than five, since we entered all these guys automatically, and it's the smallest of these so it's less than these guys. |

| | | |
|---|---|---|
| | [00:29:51.01]*LOW: | <six would be ok because lets see> [//] ok six would be ok on that side +... |
| *Begin positive* | [00:29:55.13]*112: | and to move this to this I would move four. |
| *End positive* | [00:29:57.13]*LOW: | yeah exactly. |

3. Sometimes the participants of the conversation repeat things not to provide feedback, but just because they do not hear each other. If this happens at the beginning or at the end of a feedback episode, do not include the repetition in the episode. Example:

| | | |
|---|---|---|
| | [00:33:14.22]*112: | I also had a [?] top. |
| | [00:33:17.11]*LOW: | I'm sorry? |
| *Begin positive* | [00:33:18.03]*112: | you also had an a [?] top. |
| | [00:33:19.28]*LOW: | yeah, okay. |
| *End positive* | [00:33:21:01]*LOW: | right. |

4. Sometimes there is no real feedback. The tutor might say something that appears as feedback, but in fact he is ignoring what the student just said. Make sure *not* to mark such an episode as feedback. These cases might be difficult to detect. Watching the video helps in these situations. Example:

| | |
|---|---|
| [00:16:28.16]*LOW: | it has to be +... |
| [00:16:30.04]*112: | it has to be a step <before q@l.> [>] |
| [00:16:31.21]*LOW: | [<] <it has to be before> we update it. |
| [00:16:33.04]*112: | yeah |
| [00:16:33.15]*LOW: | otherwise, t@l would be s@l's new value. |
| [00:16:35.15]*LOW: | so this is s@l equals t@l, and s@l is t@l's old value. |

5. Mark only those episodes where the topic is relevant to the subject domain (in our case, linked lists, stacks, and binary search trees). The following example can be considered feedback about social behavior, and should *not* be included in your annotation:

```
[00:28:46.03]JAC:  so now George is here in the line.
[00:28:48.08]JAC:  right?
[00:28:49.21]JAC:  okay.
[00:28:50.14]248:  that's very rude.
[00:28:51.19]JAC:  yes he's rude and *uh # Bob's no
                   better for letting him behind him.
```

6. Pay particular attention to those words like "alright" and "okay." Such words might be part of the feedback, but sometimes they are just used in the sense of "let's move on." In these cases, if they occur at the end of an episode, do not include them in the episode. Example:

```
Begin positive  [00:24:25.15]LOW:  and d@l?
                [00:24:27.12]257:  these two should be reversed so +...
                [00:24:28.25]LOW:  yeah, in fact, the whole thing is
                                   totally reversed.
                [00:24:31.19]257:  yeah.
                [00:24:33.04]LOW:  yeah, five and eleven are reversed,
                                   and then ten is on the right, and
                                   twelve is on the left, and four is on
                                   the right, everything is flipped.
End positive    [00:24:39.03]LOW:  total flip.
                [00:24:40.03]LOW:  alright.
```

# Appendix F

## CODING SCHEME FOR CS TUTORING DATA: DIRECT PROCEDURAL INSTRUCTION

In the context of problem solving, the category direct procedural instruction (DPI) should be applied when the tutor directly tells the student what to do. This label can be applied only to tutor's utterances. Do not apply it to student utterances. Always refer to the coding scheme when you encounter ambiguous utterances.

1. Correct steps that lead to the solution of a problem should be marked as DPI.

```
F05-12 :  468-469
and that is nothing there, so we put six right there.
and obviously as you said a minute ago, you put four right here.

F05-12:  77
It doesn't matter where you part point f@l xxx just point at that node.
```

2. High-level steps or subgoals should be marked as DPI. Subgoals also include top-level goals. For example, the tutor may say, "here, they want us to delete the parent." This is a DPI.

```
F05-12 :  40
it wants us to put the new node that contains g@l in it, after the node that
contains b@l.

F05-12:  87
So let me show you how to find a node that has b@l in it.
```

However, subgoals posed in the form of a question are NOT to be marked as DPI. For example, the tutor may say, "how do we do action X?"

Finally, some utterances may seem like subgoals but are indeed not relevant to the task. For example, the tutor may say, "lets look at this problem now." These instances are NOT DPI.

3. Tactics and strategies should be marked as DPI.

```
F05-12 :  41
so with these kind of problems, the first thing I have to say is always draw
pictures.

F05-12 :  474
Well alright so that's why I want to do them in this order, search, and then
if it fails, put it in there.
```

4. General domain knowledge that the tutor is telling the student (declarative instruction) should NOT be marked as DPI. Sometimes instances of declarative instruction can be mistaken as procedural instruction. The main difference is that elements of declarative instruction are usually not bound to a specific problem. In other words, declarative instruction tends to be more context-general and not context-specific. Furthermore, DPI is when the tutor tells the student what task to perform whereas declarative instruction is when the tutor tells the student about context-general computer science knowledge.

An ambiguous utterance may be encountered when the tutor describes the problem or gives a description of the problem to the student. For example, the tutor may say, "now node m@l points to node k@l." This utterance is NOT a DPI because the tutor is not directly instructing the student what task to perform. In other words, it is NOT a DPI when the tutor merely provides a description of the problem. However, there could be exceptions. For example, "the only thing you can do in this situation is X," is ostensibly a descriptive statement, but it is also clearly a DPI, that is, to do X next.

Similarly, the tutor may describe the end result of an action. This is NOT to be marked as DPI. For example, the tutor may say, "Because we did X, node Y points to node Z."

Furthermore, the tutor may be writing something relevant to the discussion. Here, there are two cases: the tutor may be writing out what actions the student wants to perform. This is NOT a DPI. The second case may involve the tutor writing out an instruction. This is DPI. For example, the tutor says, "do action X" and concurrently writes out the matching pseudo-code. Be careful to notice the context of the utterance in order to disambiguate mere description or declarative instruction versus DPI.

```
(NO!) F05-12:  101
Right now t@l is pointing at this node.

(NO!) F05-12 :  38
yeah, well, so in link list you could put something in the middle real easy.
```

```
(NO!) F05-12 :  78-79
and then uh +// let me just write that down +// each node has two parts.
you call that the info, you call this the next field.

(NO!) F05-12 :  337-341
so a binary tree, first of all okay link list is s@l one dimensional kind of
data structure
mmhm.
and a tree is two dimensional data structure.
okay.
And the way you keep track of the tree is through a pointer to the first node
which is called the root.
```

5. ALWAYS watch the video. Some cases can only be disambiguated by listening to the prosody of the speakers, or by seeing what they are writing on their scratch paper.

6. When the tutor is repeating the same or a similar concept in different utterances, and both utterances are instances of DPI, then mark both of them.

```
F05-12:  41 and 47, respectively.
So with these kind of problems, the first thing I have to say is always draw
pictures.
So always draw pictures, particularly the after picture, which you want.
```

In the case of adjacent repeated utterances, that is, the tutor repeating the same or a similar concept one after the other, then the adjacent utterances are to be marked as DPI.

Sometimes, the tutor may include an explanation that follows the instruction. If the instruction and the subsequent explanation are in the same utterance then it is marked as DPI. In contrast, if the instruction and the subsequent explanation are in separate yet adjacent utterances, then only the instruction and NOT the explanation are to be marked as DPI.

```
(YES!) F05-12:  248
So, the initialization is s@l is null (be)cause theres nothing behind the e@l.
```

7. In many occasions the tutor talks in first person ("I do this... I have done that...") or in third person ("This is what people do..."), but what the tutor really means is to provide direct instruction to the student.

```
F05-12 :  611-612
um, so I'm pushing this value onto a stack.
so I'm pushing g@l back on.

F05-12 :  525
so what people try to do is move two up, (be)cause two has two kids, and by the
way two can't come up because <three and four can't be on the left side> [>]
```

Sometimes, the tutor may provide "suggestive" DPI. For example, the tutor may say, "maybe you could do action X?" In such cases, the utterance is to be marked as DPI provided that the instruction is a correct procedure.

```
F05-12:  577
Or, we could have brought the four up, and once again the four wouldn't necessarily
be a leaf if I put the three and a half right here xxx right here.
```

8. Sometimes the tutor is creating an incorrect example or following an incorrect procedure on purpose, to make the student reason about possible incorrect scenarios. In these cases, the "wrong" steps should NOT be marked as instances of DPI.

```
(NO!) F05-12 :  216-219
so this isn't a@l so we moved t@l over here we found a@l.
do you see a problem?
I have found the node a@l, see here I found the node b@l, and then I put g@l
in after it.
here I have found the node a@l and now the link I have to change is +...
```

Specifically, during discussion of stacks, there are cases that are difficult to code. The tutor LOW tends to discuss how to count a stack and while doing so he discusses some wrong examples. The descriptions are below.

```
(NO!) Pop the original stack.

(NO!) Pop the original stack and push to a new stack.

(YES!) Push elements back from the new stack.
```

The tutor may utter a DPI that is actually wrong but the tutor believes to be correct. In these instances, the utterance is to be marked as DPI. To err is human.

Sometimes, an incorrect procedure is difficult to determine especially for coders who have little or no background in computer science. In such cases, one should discuss the utterance in question with a coder who has substantial computer science experience.

9. Sometimes the tutor is using incomplete sentences. Mark them as DPI only if those incomplete sentences are instructing the student about what task to perform.

```
(NO!) F05-12 :  265
now look, s@l's new value is +...

(NO!) F05-12 :  267
But there's a simple thing you can say, s@l's new value is +...
```

Sometimes, the tutor completes his or her own incomplete utterance. For example, tutor may say, "so if you want to delete node X then you should ... [short pause] ... do action Y." In this example, the tutor leaves his or her utterance incomplete so the student can answer but the student does not answer thus, the tutor complete the utterance. This is DPI.

10. Sometimes the tutor discusses hypothetical or "suppose you..." utterances. For example, if the tutor says, "suppose that so and so is the case, then we need to do X and Y." This is DPI. However, hypothetical or "suppose you..." scenarios that are invented for the purpose of showing the consequences of errors are NOT a DPI (refer to number 8 above).

11. Negative "don't do X" utterances either are or are not DPI, depending on how tightly they constraint the learner's actions. "Don't do X," in a situation or context in which that statement leaves completely open what to do instead, are NOT DPI; but, "don't do X," in scenarios where NOT doing X clearly implies doing Y instead, are DPI. For example, "don't leave the door open," is DPI because the only way to follow the instruction is to close the door, so it tightly constrains the students possible actions. "Don't paint the house red," is NOT a DPI, because it is very ambiguous, given that there are many other colors to paint with.

## CITED LITERATURE


AA.VV.: Computer science curriculum 2008: An interim revision of CS 2001, December 2008. Report from the Interim Review Task Force, Association for Computing Machinery and IEEE Computer Society. http://www.acm.org/education/curricula-recommendations.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley, 2006.

Aleven, V. and Koedinger, K. R.: Limitations of student control: Do students know when they need help? In Proceedings of the 5th International Conference on Intelligent Tutoring Systems, ITS 2000, eds. C. F. G. Gauthier, C. Frasson, and K. VanLehn, pages 292–303, Berlin, 2000. Springer-Verlag.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R.: Cognitive tutors: Lessons learned. The Journal of the Learning Sciences, 4(2):167–207, 1995.

Anderson, J. R.: Rules of the Mind. Hillsdale, NJ, Erlbaum, 1993.

Barnes, T. and Stamper, J. C.: Toward the extraction of production rules for solving logic proofs. In AIED07, 13th International Conference on Artificial Intelligence in Education, Educational Data Mining Workshop, pages 11–20, Marina Del Rey, CA, July 2007.

Barnes, T. and Stamper, J. C.: Toward automatic hint generation for logic proof tutoring using historical student data. In ITS 2008, The 9th International Conference on Intelligent Tutoring Systems, pages 373–382, Montreal, Canada, June 2008.

Beck, J., Stern, M., and Haugsjaa, E.: Applications of AI in education. ACM crossroads, 1996. http://www.acm.org/crossroads/xrds3-1/aied.html.

Blessing, S. B. and Gilbert, S.: Evaluating an authoring tool for model-tracing intelligent tutoring systems. In ITS 2008, The 9th International Conference on Intelligent Tutoring Systems, pages 204–215, Montreal, Canada, June 2008.

Bloom, B. S.: The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. Educational Researcher, 13:4–16, 1984.

Bransford, J. D., Brown, A. L., Cocking, R. R., Donovan, M. S., and Pellegrino, J. W.: How People Learn: Brain, Mind, Experience, and School - Expanded edition. Washington, D.C., National Academies Press, 2002.

Brown, A. L.: Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. The Journal of the Learning Sciences, 2(2):141–178, 1992.

Chan, T. W., Lin, C. J., and Chou, C. Y.: An approach to developing computational supports for reciprocal tutoring. Knowledge-Based Systems, 15:407–412, 2002.

Chi, M. T.: Constructing self-explanations and scaffolded explanations in tutoring. Applied Cognitive Psychology, 10:33–49, 1996.

Chi, M. T., Siler, S. A., Jeong, H., Yamauchi, T., and Hausmann, R. G.: Learning from human tutoring. Cognitive Science, 25:471–533, 2001.

Conway, A. R., Kane, M. J., Bunting, M. F., Hambrick, D. Z., Wilhelm, O., and Engle, R. W.: Working memory span tasks: A methodological review and user's guide. Psychonomic Bulletin & Review, 12(5):769–786, 2005.

Conway, A. R., Kane, M. J., and Engle, R. W.: Working memory capacity and its relation to general intelligence. TRENDS in Cognitive Sciences, 7(12):547–552, 2003.

Corbett, A. T. and Anderson, J. R.: The effect of feedback control on learning to program with the Lisp tutor. In Proceedings of the Twelfth Annual Conference of the Cognitive Science Society, pages 796–803, Cambridge, MA, 1990.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C.: Introduction to Algorithms, 2nd edition. New York, NY, MIT Press, McGraw-Hill Book Company, 2000.

Corrigan-Halpern, A.: Feedback in Complex Learning: Considering the Relationship Between Utility and Processing Demands. Doctoral dissertation, University of Illinois at Chicago, 2006.

Di Eugenio, B., Fossati, D., Haller, S., Yu, D., and Glass, M.: Be brief, and they shall learn: Generating concise language feedback for a computer tutor. International Journal of AI in Education, 18(4):317–345, 2008.

Di Eugenio, B., Fossati, D., Ohlsson, S., and Cosejo, D.: Towards explaining effective tutorial dialogues. In CogSci 2009, The Annual Meeting of the Cognitive Science Society, Amsterdam, The Netherlands, 2009.

Evens, M. and Michael, J.: One-on-one Tutoring by Humans and Machines. Mahwah, NJ: Lawrence Erlbaum Associates, 2006.

Forbes-Riley, K. and Litman, D. J.: Analyzing dependencies between student certainness states and tutor responses in a spoken dialogue corpus. In Recent Trends in Discourse and Dialogue, eds. L. Dybkjaer and W. Minker, chapter 11, pages 275–304. Springer, 2008.

Fossati, D.: The role of positive feedback in intelligent tutoring systems. In ACL 2008, The 46th Annual Meeting of the Association for Computational Linguistics, Student Research Workshop, Columbus, OH, June 2008.

Fossati, D., Di Eugenio, B., Brown, C., and Ohlsson, S.: Learning linked lists: Experiments with the iList system. In ITS 2008, The 9th International Conference on Intelligent Tutoring Systems, pages 80–89, Montreal, Canada, June 2008.

Fossati, D., Di Eugenio, B., Brown, C., Ohlsson, S., Cosejo, D., and Chen, L.: Supporting computer science curriculum: Exploring and learning linked lists with iList. IEEE Transactions on Learning Technologies, Special Issue on Real-World Applications of Intelligent Tutoring Systems, 2009. In press.

Fossati, D., Di Eugenio, B., Ohlsson, S., Brown, C., Chen, L., and Cosejo, D.: I learn from you, you learn from me: How to make ilist learn from students. In AIED 2009, The 14th International Conference on Artificial Intelligence in Education, Brighton, UK, July 2009.

Fox, B. A.: Cognitive and interactional aspects of correction in tutoring. In Teaching knowledge and intelligent tutoring, ed. P. Goodyear, pages 149–172. Norwood, NJ, Ablex, 1989.

Fox, B. A.: Correction in tutoring. In Proceedings of Fifteenth Annual Meeting of the Cognitive Science Society, ed. M. Polson, pages 121–126, Hillsdale, NJ, 1993. Lawrence Erlbaum Associates.

Fox, B. A.: The Human Tutorial Dialogue Project: Issues in the design of instructional systems. Hillsdale, NJ, Lawrence Erlbaum Associates, 1993.

Goldman, S. R.: Learning in complex domains: When and why do multiple representations help (commentary). Learning and Instruction, 13:239–244, 2003.

Graesser, A. C., Person, N., Lu, Z., Jeon, M., and McDaniel, B.: Learning while holding a conversation with a computer. In Technology-based education: Bringing researchers and practitioners together, eds. L. PytlikZillig, M. Bodvarsson, and R. Brunin. Charlotte, NC, Information Age Publishing, 2005.

Heift, T.: Error-specific and individualized feedback in a web-based language tutoring system: Do they read it? ReCALL Journal, 13(2):129–142, 2001.

Hundhausen, C. D., Douglas, S. A., and Starko, J. T.: A meta-study of algorithm visualization effectiveness. Journal of Visual Languages and Computing, 13(3):259–290, 2002.

Johns, K. E. and Fair, R. C.: Principles of Economics. Prentice-Hall, 5th edition, 1999.

Kalayar, M., Imekatsu, H., Hirashima, T., and Takeuchi, A.: An intelligent tutoring system for search algorithms. In Proceedings of ICCE01, pages 1369–1376, 2001.

Katz, S., Allbritton, D., Aronis, J. M., Wilson, C., and Soffa, M. L.: Gender and race in predicting achievement in computer science. IEEE Technology and Society, Special Issue on Women and Minorities in Information Technology, 22(3):20–27, 2003.

Kersey, C., Di Eugenio, B., Jordan, P., and Katz, S.: Modeling knowledge co-construction for peer learning interactions. In ITS 2008, The 9th International Conference on Intelligent Tutoring Systems, Student Research Workshop, Montreal, Canada, June 2008.

Kersey, C., Di Eugenio, B., Jordan, P., and Katz, S.: Knowledge co-construction and initiative in peer learning interactions. In AIED 2009, The 14th International Conference on Artificial Intelligence in Education, Brighton, UK, July 2009.

Kumar, A. N.: Model-based reasoning for domain modeling, explanation generation and animation in an ITS to help students learn C++. In ITS-02 Workshop on model-based systems and qualitative reasoning for Intelligent Tutoring Systems, 2002.

Kyllonen, P. C. and Christal, R. E.: Reasoning ability is (little more than) working-memory capacity?! Intelligence, 14:389–433, 1990.

Lane, H. C. and VanLehn, K.: Coached program planning: Dialogue-based support for novice program design. In Proceedings of the Thirty-Fourth Technical Symposium on Computer Science Education (SIGCSE 03), pages 148—-152. ACM Press, 2003.

Lepper, M. R., Drake, M., and O'Donnell-Johnson, T. M.: Scaffolding techniques of expert human tutors. In Scaffolding student learning: Instructional approaches and issues, eds. K. Hogan and M. Pressley, pages 108–144. New York, NY, Brookline Books, 1997.

Lepper, M. R., Woolverton, M., Mumme, D. L., and Gurtner, J.: Motivational techniques of expert human tutors: Lessons for the design of computer-based tutors. In Computers as cognitive tools, eds. S. P. Lajoie and S. J. Derry, pages 75–105. Hillsdale, NJ, Erlbaum, 1993.

Litman, D. J., Rosé, C. P., Forbes-Riley, K., VanLehn, K., Bhembe, D., and Silliman, S.: Spoken versus typed human and computer dialogue tutoring. International Journal of Artificial Intelligence in Education, 16:145–170, 2006.

Lu, X.: Expert Tutoring and Natural Language Feedback in Intelligent Tutoring Systems. Doctoral dissertation, University of Illinois at Chicago, 2007.

Lu, X., Di Eugenio, B., Kershaw, T. C., Ohlsson, S., and Corrigan-Halpern, A.: Expert vs. non-expert tutoring: Dialogue moves, interaction patterns and multi-utterance turns. In CICLing-2007, Eight International Conference on Computational Linguistics and Intelligent Text Processing, pages 456–467, Mexico City, 2007. Best Student Paper Award.

Lu, X., Di Eugenio, B., Ohlsson, S., and Fossati, D.: Simple but effective feedback generation to tutor abstract problem solving. In INLG 2008, 5th International Natural Language Generation Conference, Salt Fork, OH, June 2008.

MacWhinney, B.: The CHILDES project: Tools for analyzing talk. Mahwah, NJ, Lawrence Erlbaum Associates, third edition, 2000.

Merceron, A. and Yacef, K.: Educational data mining: A case study. In AIED05, 12th International Conference of Artificial Intelligence in Education, Amsterdam, The Netherlands, 2005. IOS Press.

Mitrović, A., Suraweera, P., Martin, B., and Weerasinghe, A.: DB-suite: Experiences with three intelligent, web-based database tutors. Journal of Interactive Learning Research, 15(4):409—-432, 2004.

Naveh, B. and Sichi, J. V.: JGraphT: A free Java graph library, 2003. http://jgrapht.sourceforge.net.

Ohlsson, S.: Constraint-based student modelling. Journal of Artificial Intelligence in Education, 3(4):429–447, 1992.

Ohlsson, S.: Learning from performance errors. Psychological Review, 103:241–262, 1996.

Ohlsson, S.: Computational models of skill acquisition. In The Cambridge handbook of computational psychology, ed. R. Sun, pages 359–395. Cambridge, UK, Cambridge University Press, 2008.

Ohlsson, S., Di Eugenio, B., Chow, B., Fossati, D., Lu, X., and Kershaw, T. C.: Beyond the code-and-count analysis of tutoring dialogues. In AIED07, 13th International Conference on Artificial Intelligence in Education, Marina Del Rey, CA, July 2007.

Pellegrino, J. W., Chudowsky, N., and Glaser, R.: Knowing What Students Know: The Science and Design of Educational Assessment. Washington, D.C., National Academies Press, 2001.

Perera, D., Kay, J., Yacef, K., and Koprinska, I.: Mining learners' traces from an online collaboration tool. In AIED07, 13th International Conference on Artificial Intelligence in Education, Educational Data Mining Workshop, pages 60–69, Marina Del Rey, CA, July 2007.

Person, N. K., Graesser, A. C., Bautista, L., Mathews, E. C., and the Tutoring Research Group: Evaluating student learning gains in two versions of AutoTutor. In Artificial intelligence in education: AI-ED in the wired and wireless future, eds. J. D. Moore, C. L. Redfield, and W. L. Johnson, pages 286–293. Amsterdam: IOS Press, 2001.

Reiter, E.: An architecture for data-to-text systems. In ENLG07, 11th European Workshop on Natural Language Generation, Saarbruecken, Germany, June 2007.

Renkl, A.: Learning from worked-out examples: Instructional explanations supplement self-explanations. Learning and Instruction, 12:529–556, 2002.

Shih, B., Koedinger, K. R., and Scheines, R.: A response time model for bottom-out hints as worked examples. In EDM08, 1st International Conference on Educational Data Mining, pages 117–126, Montreal, Canada, 2008.

Shute, V. J. and Psotka, J.: Intelligent tutoring systems: Past, present and future. Handbook of Research for Educational Communications and Technology, pages 570–600, 1996. Macmillan Library Reference USA.

Soloway, E. M., Woolf, B., Rubin, E., and Barth, P.: Meno-II: An intelligent tutoring system for novice programmers. In IJCAI-81, pages 975–977, Vancouver, B.C., Canada, August 1981.

Sykes, E. and Franek, F.: An intelligent tutoring system for learning to program in Java. In IEEE International Conference on Advanced Learning Technologies, 2003.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J. A., Shelby, R. H., Taylor, L., Treacy, D. J., Weinstein, A., and Wintersgill, M. C.: The Andes physics tutoring system: Five years of evaluations. In Artificial Intelligence in Education Conference, eds. G. I. McCalla and C. K. Looi. Amsterdam: IOS Press, 2005.

Weaver, M. R.: Do students value feedback? Student perceptions of tutors' written responses. Assessment and Evaluation in Higher Education, 31(3):379–394, 2006.

Weiss, M. A.: Data Structures and Agorithm Analysis in C++. Addison–Wesley, second edition, 1999.

Wing, J. M.: Computational thinking and thinking about computing. Philosophical Transactions of the Royal Society, 366:3717–3725, July 2008.

# VITA

| | |
|---|---|
| NAME: | Davide Fossati |
| EDUCATION: | Ph.D., Computer Science<br>University of Illinois at Chicago, 2009 |
| | M.S., Computer Engineering<br>Politecnico di Milano, Italy, 2004 |
| | M.S., Computer Science<br>University of Illinois at Chicago, 2003 |
| WORK EXPERIENCE: | Research Assistant and Teaching Assistant<br>Computer Science Department<br>University of Illinois at Chicago, 1/2005–Present |
| | Rock Climbing Instructor<br>Campus Recreation<br>University of Illinois at Chicago, 5/2008–08/2008 |
| | Consultant<br>Reply @logistics, Milano, Italy, 4/2004–12/2004 |
| | Intern<br>Accenture, Milano, Italy, 1/2004–3/2004 |
| | Teacher<br>Elementary School of Gaggiano, Milano, Italy, 1/2003–6/2003 |
| | Research Assistant and Teaching Assistant<br>Computer Science Department<br>University of Illinois at Chicago, 8/2001–7/2002 |
| AWARDS: | 50 for the Future Award, Illinois Technology Foundation. Honoring Illinois' most promising technology students.<br>http://www.50forthefuture.org. 2009. |

Chancellor's Student Service and Leadership Award, University of Illinois at Chicago. 2009.

Dean's Scholar Award and Fellowship, Graduate College, University of Illinois at Chicago. 2008–2009.

Leadership and Service Award, Department of Computer Science, University of Illinois at Chicago. 2008.

As an officer of Engineers Without Borders, I contributed to the achievement of the Student Organization of the Year Award (2007), and a formal recognition from the University of Illinois Board of Trustees (2008).

Chancellor's Student Service Award, University of Illinois at Chicago. 2007.

Outstanding Teaching Assistant Award, Department of Computer Science, University of Illinois at Chicago. 2006.

PUBLICATIONS:

Fossati, D., Di Eugenio, B., Ohlsson, S., Brown, C., Chen, L., and Cosejo, D.: I learn from you, you learn from me: How to make ilist learn from students. In AIED 2009, The 14th International Conference on Artificial Intelligence in Education, Brighton, UK, July 2009.

Fossati, D., Di Eugenio, B., Brown, C., Ohlsson, S., Cosejo, D., and Chen, L.: Supporting computer science curriculum: Exploring and learning linked lists with iList.
IEEE Transactions on Learning Technologies, Special Issue on Real-World Applications of Intelligent Tutoring Systems, 2009. In press.

Di Eugenio, B., Fossati, D., Ohlsson, S., and Cosejo, D.: Towards explaining effective tutorial dialogues. In
CogSci 2009, The Annual Meeting of the Cognitive Science Society, Amsterdam, The Netherlands, 2009.

Fossati, D., Di Eugenio, B., Brown, C., and Ohlsson, S.: Learning linked lists: Experiments with the iList system. In ITS 2008, The 9th International Conference on Intelligent Tutoring Systems, pages 80–89, Montreal, Canada, June 2008.

Fossati, D.: The role of positive feedback in intelligent tutoring systems. In ACL 2008, The 46th Annual Meeting of the Association for Computational Linguistics, Student Research Workshop, Columbus, OH, June 2008.

Lu, X., Di Eugenio, B., Ohlsson, S., and Fossati, D.: Simple but effective feedback generation to tutor abstract problem solving. In INLG 2008, 5th International Natural Language Generation Conference, Salt Fork, OH, June 2008.

Fossati, D., and Di Eugenio, B.: I saw TREE trees in the park: How to correct real-word spelling mistakes. In LREC 2008, Sixth International Conference on Language Resources and Evaluation, Marrakech, Morocco, May 2008.

Di Eugenio, B., Fossati, D., Haller, S., Yu, D., and Glass, M.: Be brief, and they shall learn: Generating concise language feedback for a computer tutor. International Journal of AI in Education, 18(4):317–345, 2008.

Ohlsson, S., Di Eugenio, B., Chow, B., Fossati, D., Lu, X., and Kershaw, T. C.: Beyond the code-and-count analysis of tutoring dialogues. In AIED 2007, 13th International Conference on Artificial Intelligence in Education, 2007.

Fossati, D., and Di Eugenio, B.: A mixed trigrams approach for context sensitive spell checking. In CICLing-2007, Eighth International Conference on Intelligent Text Processing and Computational Linguistics, 2007.

Fossati, D., Ghidoni, G., Di Eugenio, B., Cruz, I., Xiao, H., and Subba, R.: The problem of ontology alignment on the web: a first report. In EACL 2006, Workshop on Web as Corpus, Proceedings of the 11th conference of the European Association of Computational Linguistics, 2006.

Di Eugenio, B., Fossati, D., Yu, D., Haller, S., and Glass, M.: Aggregation improves learning: experiments in natural language generation for intelligent tutoring systems. In ACL05, Proceedings of the 42nd Meeting of the Association for Computational Linguistics, Ann Arbor, MI, 2005.

Di Eugenio, B., Fossati, D., Yu, D., Haller, S., and Glass, M.: Natural language generation for intelligent tutoring systems: a case study. In AIED 2005, 12th International Conference on Artificial Intelligence in Education, Amsterdam, The Netherlands, 2005.