Supporting Computer Science curriculum: Exploring and learning linked lists with iList

Davide Fossati, Barbara Di Eugenio, Christopher Brown, Stellan Ohlsson, David Cosejo, and Lin Chen

Abstract—We developed two versions of a system, called iList, that helps students learn linked lists, an important topic in Computer Science curricula. The two versions of iList differ on the level of feedback they can provide to the students, specifically in the explanation of syntax and execution errors. The system has been fielded in multiple classrooms in two institutions. Our results indicate that iList is effective, is considered interesting and useful by the students, and its performance is getting closer to the performance of human tutors. Moreover, the system is being developed in the context of a study of human tutoring, which is guiding the evolution of iList with empirical evidence of effective tutoring.

Index Terms—K.3.1.b. Computer-assisted instruction, K.3.2.b Computer science education, I.2.1.d. Education, H.5.2.e. Evaluation/methodology, Constraint-based modeling, Intelligent tutoring systems

1 Introduction

THIS paper describes the iList project, an interdisciplinary research effort whose goal is to understand the characteristics of effective tutoring and implement them into Intelligent Tutoring Systems in the domain of Computer Science data structures and algorithms.

This project is part of a larger effort that we have undertaken at the University of Illinois at Chicago in the last eight years. We collect, analyze and mine tutorial dialogues for tutoring strategies that are cognitively plausible and correlate with learning. We computationally model those strategies in Natural Language Interfaces to Intelligent Tutoring Systems. Over the years, we have moved towards systems that generate more sophisticated feedback, in more realistic application domains. In our first project, DIAG-NLP [1], we showed that more concise and abstract feedback would lead to more learning in diagnosing simulated malfunctions of a mechanical system. In our second project, we showed that modeling various tutoring moves by an expert tutor in an abstract problem solving task again engenders more learning [2], [3]. Finally, with iList, we have moved to a real-world application that has the potential of providing substantial support in introductory Computer Science classes. Moreover, the interface of iList was an initial building block for our parallel research project on peer learning [4], which is also meant to support introductory Computer Science classes. To the previously reported results from the iList project [5], [6], [7], this paper adds important contributions:

- A more detailed description of the iList system.
- A new feedback module that delivers more sophisticated responses to students' syntax errors.
- A more comprehensive evaluation of the system, with almost four times as many students than the previous evaluation.
- New results of our study of human tutoring.

As this paper demonstrates, iList has reached a level of maturity that makes it suitable for a wider adoption in Computer Science courses. We hope that this paper will encourage researchers and educators to consider using iList with their students.

The goal of this paper is not only to present a useful system, but also to explore how the improvement of natural language feedback impacts its effectiveness. This is why in this paper we compare two versions of iList, and we describe our analysis of human tutoring dialogues from which the design of the system is guided.

2 Background and related work

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 [8]. For more than twenty years, researchers have been working 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 built to provide personalized instruction to students, in some respect similar to one-on-one tutoring [9], [10]. Many of these systems

- *Davide Fossati, Barbara Di Eugenio, and Lin Chen are with the Department of Computer Science, University of Illinois at Chicago. E-mail: davide@fossati.us, bdieugen@cs.uic.edu, linchen04@gmail.com*
- *Christopher Brown is with the Department of Computer Science, United States Naval Academy. E-mail: wcbrown@usna.edu*
- *Stellan Ohlsson and David Cosejo are with the Department of Psychology, University of Illinois at Chicago. E-mail: stellan@uic.edu, dcosej1@uic.edu*

Manuscript received December 28, 2008.

have been shown to be very effective [11], [12], [13], [14], [15]. In many experiments, ITSs induced learning gains higher than those measured in a classroom environment, but lower than those obtained with one-on-one human tutoring. Therefore, the belief of the research community is that knowing more about human tutoring would be beneficial to the design of better ITSs.

Among the many research problems in the field of ITSs, we are primarily interested in delivering effective *feedback* to students. There are many different forms of feedback 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 divide tutorial feedback in two important categories: *negative feedback* and *positive feedback* [16].

Negative feedback can be provided in response to students' mistakes. An effective usage of negative feedback would help the student correct a 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 [17].

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. Several studies, including ours, have started to provide evidence of the importance of positive feedback in tutoring [3], [6], [18], [19].

More detailed characterizations of feedback have been reported in several studies, like the Human Tutoring Project [20], [21], [22] and the work of the CIRCSIM-Tutor group [11]. Differences in tutoring behavior 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.

Like other forms of instruction, one-on-one tutoring have characteristics that are dependent on the subject domain. Among many disciplines, basic Computer Science has received only little attention from the educational and ITS research communities. Existing work focuses primarily on computer literacy [23], programming languages such as Lisp [24], [25], C++ [26], Java [27], general programming and design skills [28], databases [14], and special topics such as search algorithms used in Artificial Intelligence [29]. Although the previous list might seem long, a fundamental topic has been almost neglected: basic data structures and algorithms, which are in the core of CS undergraduate curricula [30], and have been identified as difficult concepts for students to master [31]. ADIS [32] tutors on basic data structures, but its emphasis is on visualization more than on adaptive, intelligent tutoring; also, it appears to have been more of a proof of concept than a working system, and has not been developed further.

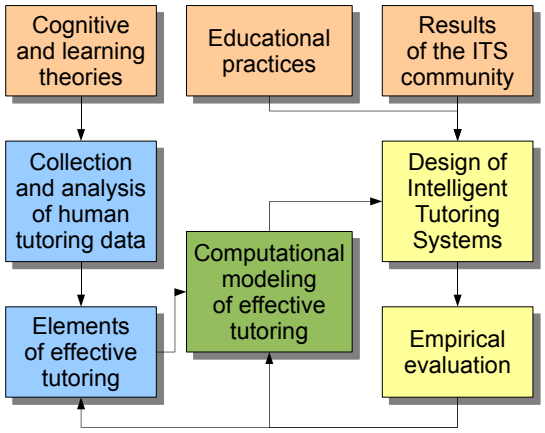


Fig. 1. Research methodology

This research focuses on the tutoring of basic data structures, specifically on *linked lists*.

3 Methodology

This research has two main goals. On the one hand, we want to build an effective system that can be successfully used in introductory computer science classes around the world. On the other hand, we want to discover the pedagogical elements that make human tutoring effective, in order to build computational models that can improve the performance of our system, and get a better understanding of complex dynamics of teaching and learning that can be transfered to other contexts and domains. To accomplish these goals, we are following an iterative methodology, represented in Figure 1. Guided and motivated by cognitive and learning theories, we are collecting naturalistic tutoring sessions, transcribing, annotating, and analyzing them using statistical approaches. Cognitive theories allow us to formulate hypotheses about features that could correlate with learning, and we are testing those hypotheses by regressing those features against the learning outcomes of the students interacting with our human tutors. In this way, we are able to understand which features are worthwhile modeling and implementing in our system. At the same time, motivated by the success and advancements of the Intelligent Tutoring Systems community, we are developing a robust system, deploying it in classrooms, and analyzing the recorded interaction between students and system. When new results are obtained we feed them back into the process so they can guide the subsequent analysis, design, and implementation steps.

4 The Linked List Domain

Many of the readers are certainly familiar with *linked lists*, as they are one of the fundamental data structures

at the core of Computer Science curricula. The main idea behind linked lists is that different units of information can be "linked" one after each other and then accessed sequentially. The unit of information in a linked list is called *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 uses boxes and arrows. A box represents a node; it is divided in 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 to another box, represents a link between two nodes. For example, Figures 3 and 4 show representations of linked lists within the interface of iList.

To effectively master linked lists, students need to understand the static properties of the structure and the dynamic operations necessary to store, organize, and retrieve information from it. More complex operations are built on top of basic ones such as *traversal* of a list, *insertion* of a new node, and *deletion* of an existing node.

We chose to focus on linked lists for several reasons.

- Linked lists are usually presented early in Computer Science curricula. Thus, more students see this topic.
- According to our experience as Computer Science instructors, 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 [33] is a huge step for students, and one that they need to make to continue successfully in Computer Science.

There are several issues that make learning and teaching this subject difficult. For example, what is the appropriate level of detail of an explanation? Because of the abstract nature of this topic, high level explanations of linked lists tend to be confusing. In order to understand them, detailed examples should be provided. However, these examples can expose the students to an overwhelming amount of detail.

Another teaching issue is the choice of a top-down or bottom-up strategy. In a top-down approach, the general concepts would be explained before presenting more details and examples/exercises, whereas a bottom-up explanation would start from examples and proceed with generalizations from there. A problem with top-down, which is the most widely used approach in classrooms, is that students could understand nothing before facing the 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.

There are also language and representations issues. When talking about data structures and algorithms, we can use natural language descriptions, graphics and diagrams, programming languages, and pseudo-code descriptions. All these representations have their advantages and disadvantages, and 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 engender 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. A study 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 [34]. This finding is consistent with other research in Cognitive Science, which points to the same conclusion [35].

5 Description of iList

We are developing multiple versions of iList, capable of delivering feedback of increasing complexity. We will now describe the first two versions of the system and their evaluation with students in classrooms. Our methodology crucially relies on controlled comparisons between versions of iList that differ in few important features. Only in this way we can assess whether a new feature that appears to be educationally or empirically motivated is really conducive to learning and should be retained in further enhancements to iList.

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 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 reflects the typical scheme of an Intelligent Tutoring System. It is depicted in Figure 2.

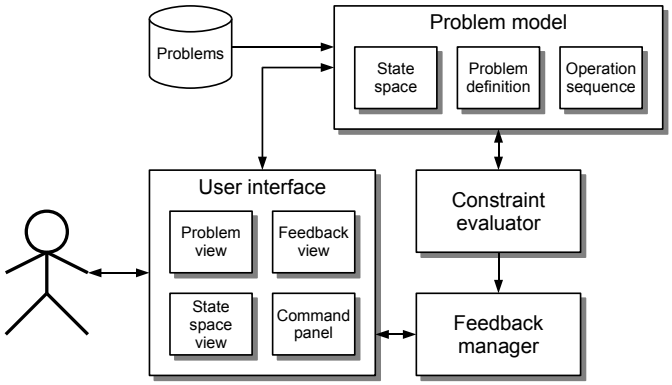


Fig. 2. Architecture of iList

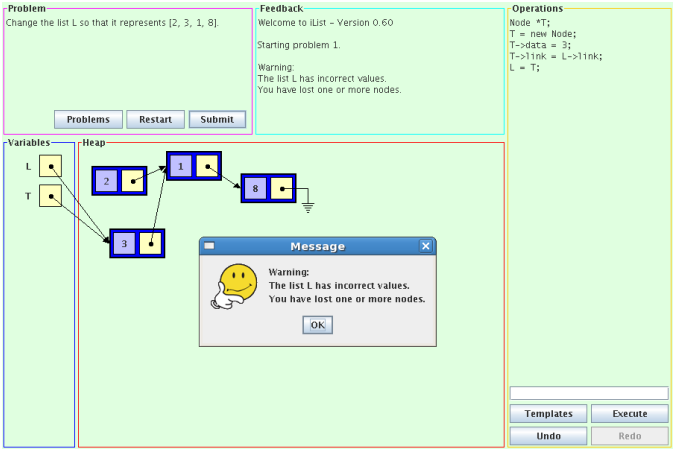


Fig. 3. Screenshot of iList – Step-by-step problem

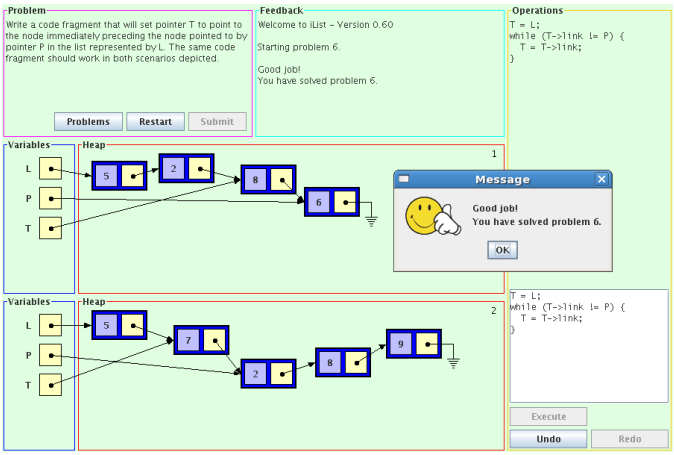


Fig. 4. Screenshot of iList – Block of code problem

5.1 Graphical user interface

The graphical user interface is responsible for the main interaction with the student. Snapshots of the interface can be seen in Figures 3 and 4. 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 effect 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.

5.2 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 3). 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. The second type of problems requires writing a complete snippet of code, typically involving structured conditional constructs like loops (Figure 4). 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 to abstract away the specific details of a scenario and think about more general algorithms for solving problems on 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 ACM [30]. 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.

5.3 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 [17], [36],

constraint-based modeling has grown into a methodology used to build full-fledged ITSs [14], and an alternative to the model tracing approach adopted by other ITSs, such as [12]. 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.

5.4 Feedback manager

The feedback manager is responsible for generating feedback messages for the students. Currently, feedback is given by iList in three main circumstances.

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*.

The amount and sophistication of feedback differentiates the versions of the system developed and tested so far. In particular, the main difference between the two versions reported in this paper is the quality of syntax feedback and execution feedback.

In both versions of iList, final feedback comes from a collection of feedback units associated to 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 3.

In the first version of iList, *iList-1*, both syntax and execution feedback are 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. Similarly, execution error messages are of the form "You tried to execute XXX. I'm sorry, I can't do that."

The second version of iList, *iList-2*, provides substantial improvements to both syntax and execution feedback. Execution feedback messages in iList-2 are of the same form of those delivered by iList-1, plus a brief explanation of the reason why the command cannot be executed. For example, if a student tried to reference a variable T that was never declared, the system would report that "variable T does not exist." This information is available from the execution engine of the virtual machine, and it was not difficult to implement this additional feedback. The most challenging improvement is on syntax feedback, explained in the following section.

5.5 Improved syntax feedback in iList-2

This section describes a new module in iList-2 that seeks to provide useful feedback in the presence of incorrect

C++/Java syntax, or syntax outside the language subset understood by iList. The lack of meaningful feedback for syntax-related problems caused a great deal of frustration in our first iList trials.

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 — something that at least explained what was wrong with the input, if not actually initiating new tutoring actions.

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) [37]. 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. [38], p.364). Though there are many potential tokenizations, the system only generates them one-by-one in order

TABLE 1
Examples of syntax errors and messages

input	`p>link = NULL;`
g++	`error: comparison between distinct pointer types Node* and int (*)(const char*, const char*) throw () lacks a cast` `error: lvalue required as left operand of assignment`
iList	`You're trying to write a pointer assignment statement, right?` `* 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 operand of assignment`
iList	`You're trying to write a pointer assignment statement, right?` `* You're using a number like 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 given to delete, expected pointer`
iList	`You're trying to write a delete statement, right?` `* You should give delete a pointer to a Node, not the Node itself, so there's no need to dereference 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.

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 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 1.

6 Evaluation of iList

Over three school semesters, we deployed the two versions of iList in three classrooms at the University of Illinois at Chicago and the United States Naval Academy. More than 120 students worked with the system as part of their coursework. From those that consented to participate in our evaluation study, we collected data on learning outcomes, user satisfaction, and logs of the interaction of those students with iList. In this section we describe our experimental procedure and results on learning outcomes, user satisfaction, and log analysis. We will show a positive learning trend among our four experimental conditions; an increase in user satisfaction with iList-2, the system with more sophisticated feedback; and an indication of higher efficiency for those students that interacted with iList-2.

6.1 Experimental procedure

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. More recently, starting with those students working with iList-2, we also asked them to complete a working memory capacity test.

All the students were taking an introductory data structures class, and the lab session with iList was scheduled right at the time when the topic of linked lists was just being introduced. The students had no previous experience with iList. The system was presented to them on the day of the experiment. Originally, 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; later, we replaced the instructor intervention with a brief written tutorial providing some minimal information on the system and demonstrating how to solve the same first problem. Also, in the original experiments, pre-test, post-test, and survey were hand written; later, we converted them into electronic format, without altering their content. We are aware that those

changes on the experimental procedures might have had an effect on the results, but we believe the impact is negligible compared to the many other sources of noise that affect real-world experimental settings like ours.

The pre-test and post-test are identical, and they derive directly from those we developed for our study of human tutoring [6], [7]. In particular, our pre/post test includes the first two questions from that study (the only two problems on linked lists), plus a third one developed specifically for the experiments with iList. We decided to add a question because we believed that two problems would not be enough to accurately assess the knowledge of our students. Thus, although scaled to make them comparable, the scores of students working with human tutors are based on two questions, whereas the scores of those working with iList are based on three questions.

The questions in the test have been 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.

6.2 Learning outcomes

Our primary measure of learning outcome is *learning gain*, defined as the difference between post-test score and pre-test score. We ran a four-way comparison of learning outcomes on the following groups:

1) Students doing an irrelevant activity between pre and post test (control group).
2) Students working with iList-1, the original version with simple feedback.
3) Students working with iList-2, the version with more sophisticated syntax and execution feedback.
4) Students working with human tutors.

Number of students, pre-test score, post-test score, and learning gain for each group are reported in Table 2.

ANOVA revealed an overall significant difference across the four groups ($F(3, 220) = 4.93$, $P < 0.01$). Post-hoc Tukey test showed only a significant difference between the control group and the human group ($P < 0.01$), and a marginally significant difference between the control group and iList-2 ($P < 0.1$). The difference between iList-1 and iList-2 is not significant, but the progression of effect sizes indicates that the performance of iList-2 is even less distinguishable from human tutors than that of iList-1. Although this is not a strong evidence that iList-2 is better than iList-1, this result is encouraging and, overall, the performance of iList is very respectable compared to human tutors.

TABLE 2
Test scores (range: 0 to 1)

Tutor	N	Pre-test		Post-test		Gain		
		μ	σ	μ	σ	μ	σ	Eff. Size
None	53	.34	.22	.35	.23	.01	.15	-
iList-1	61	.41	.23	.49	.27	.08	.14	.49
iList-2	56	.31	.17	.41	.23	.10	.17	.59
Human	54	.40	.26	.54	.26	.14	.25	.88

TABLE 3
Survey: scaled questions (1=No to 5=Yes)

Question	iList-1		iList-2	
	μ	σ	μ	σ
1. Do you feel that iList helped you learn about linked lists?	2.9	1.1	2.9	1.1
2. Do you feel that working with iList was interesting?	4.0	1.1	3.8	1.1
3. Did you read the verbal feedback the system provided?	4.1	1.1	4.1	1.0
4. Did you have any difficulty understanding the feedback?	2.8	1.5	2.9	1.2
5. Did you find the feedback useful?	**2.6**	1.2	**3.0**	1.0
6. Did you ever find the feedback misleading?	2.3	1.3	2.4	1.1
7. Did you find the feedback repetitive?	**3.8**	1.2	**3.1**	1.1

6.3 User satisfaction

In addition to learning outcomes, we conducted a survey to assess the satisfaction of the students using iList. Our survey includes eight questions. The first seven are scaled questions, to which students replied with a number between 1 (meaning "no") and 5 (meaning "yes"). 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 for each version of the system are reported in Table 3. Notice that each individual student has seen only one version of the system.

ANOVA revealed no significant differences in scores between the two groups in questions 1, 2, 3, 4, and 6; a marginally significant difference on question 5 ($F(1, 113) = 3.64$, $P < 0.1$); and a significant difference on question 7 ($F(1, 113) = 11.8$, $P < 0.01$). These differences indicate that students working with iList-2 found the feedback more useful and less repetitive than those working with iList-1.

Linear regression of survey answers on learning gain revealed some correlations between students' feelings about the system and learning. The students who felt

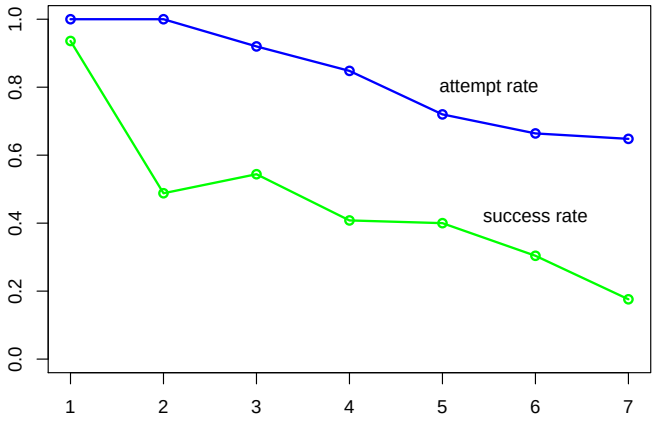


Fig. 5. Attempt and success rates per problem

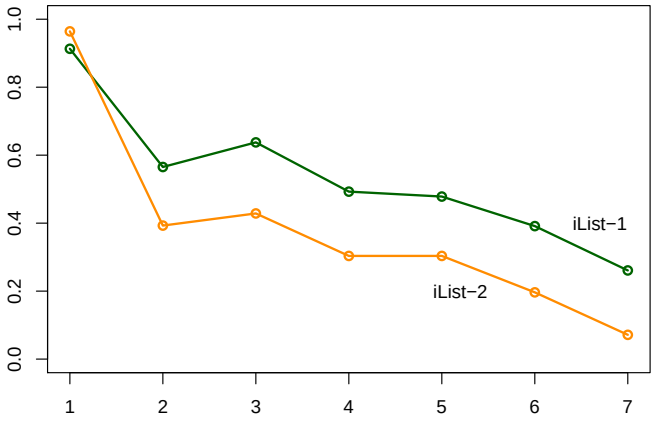


Fig. 6. Success rates per problem, per system

that iList helped them the most or found the feedback useful did indeed learn the most (question 1: $\beta = 0.04$, $t(113) = 3.67$, $P < 0.01$; question 5: $\beta = 0.03$, $t(113) = 2.84$, $P < 0.01$). Those who had trouble understanding the feedback or found the feedback repetitive learned less (question 4: $\beta = -0.02$, $t(113) = -1.88$, $P < 0.1$ – marginally significant; question 7: $\beta = -0.04$, $t(113) = -3.32$, $P < 0.01$). The answers to questions 2, 3, 5, and 6 were not significantly correlated with learning gain.

Finally, the answers to the last open question provided us with useful insights on several practical issues that students had with the system, which will be taken into account as new versions are developed.

6.4 Log analysis

The interaction of the students with iList has been comprehensively logged. From the logs, we extracted several features. We compared these features across the two systems using ANOVA, and we tested their impact on learning using linear regression (Table 4). In this regression study, each variable has been used as a predictor of learning gain in multiple independent models.

6.4.1 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 5). We found a strong positive correlation between success rate and learning gain: the more problems the students solved, the more they learned. The correlation between attempt rate and learning gain is also positive but only marginally significant (Table 4). Interestingly, if we look at the differences in problem solving performance of the students working with the two versions of the system, we can see that the students working with iList-2 solved *fewer* problems than those working with iList-1 (Figure 6 and Table 4).

The question now is, if solving more problems leads to more learning, and students working with iList-2 solved fewer problems, why these students did *not* learn less that their peers working with iList-1. There should be some other reason for which problem solving in iList-2 turned out to be more "efficient" than in iList-1. This difference in efficiency is even more evident looking at the β coefficients of the linear regression of learning gain on problem solving for the two iList groups separately. With separate groups, the correlation is still strongly significant ($P < 0.01$ in both cases), but we have $\beta = 0.13$ for iList-1 and $\beta = 0.22$ for iList-2. Thus, we can see that iList-2 is more "problem efficient" than iList-1, in the sense that students needed to solve fewer problems to learn the same amount. Given that the main structural difference between iList-1 and iList-2 is the sophistication of feedback, it looks plausible that the difference mentioned before can be at least in part justified by the feedback itself.

6.4.2 Pre-test scores

Linear regression revealed no significant correlation between pre-test score and learning gain. This is a notable difference with respect to our study of human tutoring, where there was a significant negative correlation between pre-test score and learning gain.

6.4.3 Working memory capacity

Although we are collecting pre-test scores to take into account students' previous knowledge, we feel that there is much more to students' individual characteristics than what we can capture with our pre-test, and we believe that many of these "hidden" student features might have a profound impact on their learning. With the introduction of iList-2, we started to collect a measure of *working memory capacity* [39], [40], assessed with an operation span test [41], which we implemented in iList. We chose to record working memory capacity because previous research showed that it correlates very well with other measures of general cognitive abilities, and the test can be taken quickly and easily by the students.

We found a marginally significant correlation between the word score (which is the main score) of the operation span test and learning gain, and a significant correlation

TABLE 4
Comparison of the two systems and correlation with learning

Feature	iList-1		iList-2		Difference of means				Regression on learning				
	μ	σ	μ	σ	Δ_{21}	df	F	P	β	df	t	P	R^2
Problem attempt rate	88%	19%	80%	23%	−8%	1, 115	4.91	$< .05$	.12	115	1.73	$< .1$	.02
Problem success rate	56%	36%	38%	31%	−18%	1, 115	8.53	$< .01$	.15	115	4.00	$< .01$	.11
Operation span (words)	N/A		29.8	8.55	N/A				−12.1	54	−1.78	$< .1$	.08
Operation span (math)	N/A		41.2	1.35	N/A				−2.24	54	−2.12	$< .05$	.06
Time (minutes)	42.5	17.2	33.0	8.6	−9.5	1, 115	13, 78	$< .01$	ns				
Student actions	159	67	110	51	−49	1, 115	19.2	< 0.01	ns				
Action density (act/min)	4.0	1.6	3.3	1.2	−0.7	1, 115	6.48	< 0.05	ns				
Syntax errors	20.4	14.1	16.7	12.2	ns				ns				
Syntax error ratio	0.21	0.13	0.22	0.13	ns				ns				
Execution errors	12.9	10.5	7.9	6.0	−5.0	1, 115	9.54	< 0.01	ns				
Execution error ratio	0.12	0.07	0.11	0.06	ns				ns				

between the math score of operation span and learning gain (Table 4). 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.

6.4.4 Time on task

Students working with iList-2 spent significantly less time with the system than those working with iList-1 (Table 4). Linear regression 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.

6.4.5 Student activity

We counted 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. As Table 4 shows, students that worked with iList-2 took significantly fewer actions than those working with iList-1. Also, there is a significant difference in action density (number of actions over time), which might indicate that students with iList-2 spent more time thinking before taking an action. We found no significant correlation between the number of student actions and learning gain, nor between action density and learning gain, for any category of actions.

6.4.6 Syntax and execution errors

We wanted to test whether the better syntax end execution feedback in iList-2 had a direct effect on the number of syntax errors and execution errors that students make when they solve problems. As reported in Table 4, we found no significant difference between the number of syntax errors that students make with the two versions of iList. We also found that students interacting with iList-2 make significantly less execution errors than those working with iList-2. However, both syntax and execution error ratio, defined as the ratio of the number of errors over the number of programming commands given by the student, are statistically indistinguishable. Finally, we found no significant correlation between the number of errors and learning gain.

7 A STUDY OF HUMAN TUTORING

As we mentioned in the methodology section, we are also conducting a study of human tutoring, in order to uncover empirical evidence for effective tutoring strategies, which we will incorporate in future versions of iList [6], [7]. This section briefly describes the study and reports some recent findings. In particular, our findings about the importance of positive feedback and feedback initiative are providing direct guidance for the further development of iList (see the current work section).

7.1 Description of the study

We collected a corpus of 54 one-on-one tutoring sessions on data structures, specifically on *linked lists*, *stacks*, and *binary search trees*. 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, with more than 30 years of teaching experience. The other tutor is a senior undergraduate student in Computer Science, with only one semester of previous tutoring experience. Each tutoring session lasted approximately 40 minutes. The tutoring sessions were videotaped and transcribed. The transcripts were produced according to the rules and conventions described in the transcription manual of the CHILDES project [42]. Additionally, they were enriched with timestamps at the beginning of each utterance, to keep track of the temporal position of the utterance in the video recording. An utterance is a natural unit of speech bounded by breaths or pauses, manually identified by the transcribers. Students took a pre-test right before the tutoring session, and an identical post-test immediately after. The test had 2 problems on linked lists, 2 problems

TABLE 5
Learning gains and t-test statistics

Topic	Tutor	μ	σ	t	df	P
List	Novice	.09	.22	-2.00	23	.057
	Expert	.18	.26	-3.85	29	$< .01$
	Combined	.14	.25	-4.24	53	$< .01$
	None	.01	.15	-0.56	52	ns
Stack	Novice	.35	.25	-6.90	23	$< .01$
	Expert	.27	.22	-6.15	23	$< .01$
	Combined	.31	.24	-9.20	47	$< .01$
	None	.05	.17	-2.15	52	$< .05$
Tree	Novice	.33	.26	-6.13	23	$< .01$
	Expert	.29	.23	-6.84	29	$< .01$
	Combined	.30	.24	-9.23	53	$< .01$
	None	.04	.16	-1.78	52	ns

on stacks, and 4 problems on binary search trees. An additional control group of 53 students took the pre and post tests, but instead of participating in a tutoring session they attended a lecture about an unrelated topic.

7.2 Learning outcomes

Paired samples t-tests revealed that post-test scores are *significantly higher* than pre-test scores in the two tutored conditions for all the topics, 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. Students in the control group did *not* show significant learning for linked lists and binary search trees, and only marginally significant learning for stacks. Means, standard deviations, and t-test statistic values are reported in Table 5.

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, $F(1, 53) = 0.32$, $P = ns$.

The learning gain of students that received tutoring is *significantly higher* than the learning gain of the students in the control group, for all the topics. This is showed by ANOVA between the group of tutored students (with both tutors) and the control 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 5.

7.3 Regression analysis

The distribution of scores across sessions shows a lot of variability (Table 5). In all the conditions, there are sessions with very high learning gains, and sessions with very low ones. This observation and the previous results suggest 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. As advocated in [7], a sensible way to do that is to adopt an approach based on multiple regression of learning outcomes per tutoring session onto the frequencies of the different features. The following analysis has been done with linear regression models.

7.3.1 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 6). 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 β weights are negative. That means students with higher pre-test scores learn *less* then 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.

7.3.2 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. Surprisingly, session length has a significant effect only on linked lists (Table 6).

7.3.3 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 [43], [44]. To test this hypothesis, the following definition of student activity has been 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-channelling elements. Of course, this is just an approximation, because substantive 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.

7.3.4 Feedback

The dataset has been manually annotated for *episodes* where positive or negative feedback is delivered. All the protocols have been annotated by one coder, and some of them have been double-coded by a second one (intercoder agreement: kappa = 0.67). Examples of feedback episodes are reported in Figure 7.

	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.

Fig. 7. Positive and negative feedback (T = tutor, S = student)

TABLE 6
Linear regression – human tutoring

Topic	Model	Predictor	β	R^2	P
List	1	Pre-test	-.45	.18	$< .05$
	2	Pre-test	-.40	.28	$< .05$
		Session length	.35		$< .05$
	3	Pre-test	-.35	.36	$< .05$
		Session length	.33		.05
		+ feedback	.46		.05
		- feedback	-.53		$< .05$
Stack	1	Pre-test	-.53	.26	$< .01$
	2	Pre-test	-.52	.24	$< .01$
		Session length	.05		ns
	3	Pre-test	-.58	.33	$< .01$
		Session length	.01		ns
		+ feedback	.61		$< .05$
		- feedback	-.55		$< .05$
Tree	1	Pre-test	-.79	.61	$< .01$
	2	Pre-test	-.78	.60	$< .01$
		Session length	.03		ns
	3	Pre-test	-.77	.59	$< .01$
		Session length	.04		ns
		+ feedback	.06		ns
		- feedback	-.12		ns
All	1	Pre-test	-.52	.26	$< .01$
	2	Pre-test	-.54	.29	$< .01$
		Session length	.20		$< .05$
	3	Pre-test	-.57	.32	$< .01$
		Session length	.16		.06
		+ feedback	.30		$< .05$
		- feedback	-.23		.05

The counts of positive and negative feedback episodes have been introduced in the regression model (Table 6). The model showed a significant correlation between feedback and learning for linked lists and stacks, but no significant correlation for trees. 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 β values.

We 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 7. The numbers in the table are aggregated on the three topics, but splitting the three topics apart revealed similar patterns.

TABLE 7
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)

ANOVA revealed overall significant differences on the four groups ($F(3, 325) = 43.27$, $P < 0.01$). 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 marginally significant ($P < 0.1$) for stacks, not significant for lists and trees. This result suggests the importance of *proactive feedback*, which we will briefly introduce in the current work section.

7.3.5 Direct procedural instruction

We annotated the dataset 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.

8 Current directions and conclusions

The results in this paper suggest that iList is a useful and effective system, and that improving the sophistication of feedback can be beneficial to its performance. Another important point comes from our study of human tutoring, that is giving us clear directions to guide the evolution of iList. The importance of positive feedback in human tutoring calls for an implementation of such behavior in iList, as iList is currently delivering mostly negative feedback in response to students' mistakes. Also, the predominance of tutor-initiated feedback and the importance of direct procedural instruction indicate that iList should not just wait for student actions before delivering feedback, but should create opportunities such that feedback can be provided earlier.

In order for iList to deliver more feedback, the system should be able to monitor more closely the solution paths of the students, and intervene with appropriate responses after or even before specific student actions. To do so, we are currently building an innovative model that is automatically generated from the logs of previous students that worked with iList in our classroom trials. This model is able to estimate the goodness of student actions and solution paths. We are going to use this model to implement two new behaviors in iList: *reactive feedback* and *proactive feedback*. Reactive feedback will be delivered in response to student actions that are syntactically correct and have been successfully executed in iList's virtual machine. These actions can trigger negative or positive feedback, depending on their correctness and pedagogical importance in the context of the current solution path. Proactive feedback will be delivered by iList after a student response has been elicited by a prompt from iList, which will also be decided according to the solution context. This behavior will involve a shift of initiative from the student to the tutor, in the same way human tutors frequently behave.

Now that we have showed that iList is effective in helping students learn linked list, we will be glad to broaden the diffusion of iList, allowing access to our system to all the instructors who wish to use it in their classroom, free of charge. Instructors are welcome to contact us for further details.

Acknowledgments

This work is supported by Award N00014–07–1–0040 from the Office of Naval Research, the 2008/2009 Dean's Scholar Award from the Graduate College of the University of Illinois at Chicago, and additionally by Awards ALT–0536968 and IIS–0133123 from the National Science Foundation.

References

[1] B. Di Eugenio, D. Fossati, S. Haller, D. Yu, and M. Glass, "Be brief, and they shall learn: Generating concise language feedback for a computer tutor," *International Journal of AI in Education*, vol. 18, no. 4, pp. 317–345, 2008.

[2] X. Lu, B. Di Eugenio, T. C. Kershaw, S. Ohlsson, and A. Corrigan-Halpern, "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*, Mexico City, 2007, pp. 456–467, best Student Paper Award.

[3] X. Lu, "Expert tutoring and natural language feedback in intelligent tutoring systems," Ph.D. dissertation, University of Illinois at Chicago, 2007.

[4] C. Kersey, B. D. Eugenio, P. Jordan, and S. Katz, "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.

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

[6] D. Fossati, "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.

[7] S. Ohlsson, B. Di Eugenio, B. Chow, D. Fossati, X. Lu, and T. C. Kershaw, "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.

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

[9] V. J. Shute and J. Psotka, "Intelligent tutoring systems: Past, present and future," *Handbook of Research for Educational Communications and Technology*, pp. 570–600, 1996, macmillan Library Reference USA.

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

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

[12] K. Van Lehn, C. Lynch, K. Schulze, J. A. Shapiro, R. H. Shelby, L. Taylor, D. J. Treacy, A. Weinstein, and M. C. Wintersgill, "The Andes physics tutoring system: Five years of evaluations," in *Artificial Intelligence in Education Conference*, G. I. McCalla and C. K. Looi, Eds. Amsterdam: IOS Press, 2005.

[13] B. Di Eugenio, D. Fossati, D. Yu, S. Haller, and M. Glass, "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.

[14] A. Mitrović, P. Suraweera, B. Martin, and A. Weerasinghe, "DB-suite: Experiences with three intelligent, web-based database tutors," *Journal of Interactive Learning Research*, vol. 15(4), pp. 409–432, 2004.

[15] N. K. Person, A. C. Graesser, L. Bautista, E. C. Mathews, 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*, J. D. Moore, C. L. Redfield, and W. L. Johnson, Eds. Amsterdam: IOS Press, 2001, pp. 286–293.

[16] S. Ohlsson, "Computational models of skill acquisition," in *The Cambridge handbook of computational psychology*, R. Sun, Ed. Cambridge, UK: Cambridge University Press, 2008, pp. 359–395.

[17] ——, "Learning from performance errors," *Psychological Review*, vol. 103, pp. 241–262, 1996.

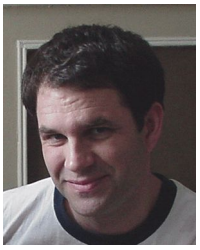
[18] D. Barrow, A. Mitrović, S. Ohlsson, and M. Grimley, "Assessing the impact of positive feedback in constraint-based tutors," in *ITS 2008, The 9th International Conference on Intelligent Tutoring Systems*, Montreal, Canada, June 2008.

[19] A. Corrigan-Halpern, "Feedback in complex learning: Considering the relationship between utility and processing demands," Ph.D. dissertation, University of Illinois at Chicago, 2006.

[20] B. A. Fox, "Cognitive and interactional aspects of correction in tutoring," in *Teaching knowledge and intelligent tutoring*, P. Goodyear, Ed. Norwood, NJ: Ablex, 1989, pp. 149–172.
[21] ——, "Correction in tutoring," in *Proceedings of Fifteenth Annual Meeting of the Cognitive Science Society*, M. Polson, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, 1993, pp. 121–126.
[22] ——, *The Human Tutorial Dialogue Project: Issues in the design of instructional systems*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1993.
[23] A. C. Graesser, N. Person, Z. Lu, M. Jeon, and B. McDaniel, "Learning while holding a conversation with a computer," in *Technology-based education: Bringing researchers and practitioners together*, L. PytlikZillig, M. Bodvarsson, and R. Brunin, Eds. Information Age Publishing, 2005.
[24] A. T. Corbett and J. R. Anderson, "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*, Cambridge, MA, 1990, pp. 796–803.
[25] T. W. Chan, C. J. Lin, and C. Y. Chou, "An approach to developing computational supports for reciprocal tutoring," *Knowledge-Based Systems*, vol. 15, pp. 407–412, 2002.
[26] A. N. Kumar, "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.
[27] E. Sykes and F. Franek, "An intelligent tutoring system for learning to program in Java," in *IEEE International Conference on Advanced Learning Technologies*, 2003.
[28] H. C. Lane and K. VanLehn, "Coached program planning: Dialogue-based support for novice program design," in *Proceedings of the Thirty-Fourth Technical Symposium on Computer Science Education (SIGCSE 03)*. ACM Press, 2003, pp. 148–152.
[29] M. Kalayar, H. Imekatsu, T. Hirashima, and A. Takeuchi, "An intelligent tutoring system for search algorithms," in *Proceedings of ICCE01*, 2001, pp. 1369–1376.
[30] AA.VV., "Computing Curricula 2001 – Computer Science," 2001, report of the Joint Task Force (IEEE Computer Society, Association for Computing Machinery). http://www.sigcse.org/cc2001/.
[31] S. Katz, D. Allbritton, J. M. Aronis, C. Wilson, and M. L. Soffa, "Gender and race in predicting achievement in computer science," *IEEE Technology and Society, Special Issue on Women and Minorities in Information Technology*, vol. 22, no. 3, pp. 20–27, 2003.
[32] K. Warendorf and C. Tan, "ADIS - an animated data structure intelligent tutoring system or putting an interactive tutor on the WWW," in *Intelligent Educational Systems on the World Wide Web Workshop. Eighth World Conference of the AIED Society*, 1997.
[33] J. M. Wing, "Computational thinking and thinking about computing," *Philosophical Transactions of the Royal Society*, vol. 366, pp. 3717–3725, July 2008.
[34] C. D. Hundhausen, S. A. Douglas, and J. T. Starko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages and Computing*, vol. 13(3), pp. 259–290, 2002.
[35] S. R. Goldman, "Learning in complex domains: When and why do multiple representations help (commentary)," *Learning and Instruction*, vol. 13, pp. 239–244, 2003.
[36] S. Ohlsson, "Constraint-based student modelling," *Journal of Artificial Intelligence in Education*, vol. 3(4), pp. 429–447, 1992.
[37] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.
[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 2nd edition*. MIT Press, McGraw-Hill Book Company, 2000.
[39] P. C. Kyllonen and R. E. Christal, "Reasoning ability is (little more than) working-memory capacity?!" *Intelligence*, vol. 14, pp. 389–433, 1990.
[40] A. R. Conway, M. J. Kane, and R. W. Engle, "Working memory capacity and its relation to general intelligence," *TRENDS in Cognitive Sciences*, vol. 7, no. 12, pp. 547–552, 2003.
[41] A. R. Conway, M. J. Kane, M. F. Bunting, D. Z. Hambrick, O. Wilhelm, and R. W. Engle, "Working memory span tasks: A methodological review and user's guide," *Psychonomic Bulletin & Review*, vol. 12, no. 5, pp. 769–786, 2005.
[42] B. MacWhinney, *The CHILDES project: Tools for analyzing talk*, 3rd ed. Mahwah, NJ: Lawrence Erlbaum Associates, 2000.
[43] M. R. Lepper, M. Drake, and T. M. O'Donnell-Johnson, "Scaffolding techniques of expert human tutors," in *Scaffolding student learning: Instructional approaches and issues*, K. Hogan and M. Pressley, Eds. New York: Brookline Books, 1997, pp. 108–144.
[44] M. T. Chi, S. A. Siler, H. Jeong, T. Yamauchi, and R. G. Hausmann, "Learning from human tutoring," *Cognitive Science*, vol. 25, pp. 471–533, 2001.

Davide Fossati received a MSc in Computer Science at the University of Illinois at Chicago in 2003 and a second MSc in Computer Engineering at Politecnico di Milano, Italy in 2004. He is currently a PhD candidate in Computer Science at the University of Illinois at Chicago. His research interests include Intelligent Tutoring Systems, Computer Science Education, and Natural Language Processing. *http:// www.fossati.us/*

Barbara Di Eugenio , PhD, is an Associate Professor at the University of Illinois at Chicago. She has published extensively in Natural Language Processing, and she is a past treasurer of the North American Chapter of the Association for Computational Linguistics. She is a recipient of the NSF CAREER award.



Christopher W. Brown has a PhD in Computer Science from the University of Delaware in 1999. He is an Associate Professor in the Department of Computer Science at the United States Naval Academy. His research interests lie in symbolic mathematical computing, and computer science education.

Stellan Ohlsson has a PhD in Psychology from the University of Stockholm, Sweden in 1980. He is a Professor of Psychology at the University of Illinois at Chicago. His research interests are in Cognitive Psychology, with special focus on the acquisition of complex knowledge, computer simulation, protocol analysis, and educational and technological applications.

David G. Cosejo is a graduate student of cognitive psychology at the University of Illinois at Chicago. His research interests include science education, the inhibitory memory processes involved in conceptual change, and the cognitive mechanisms that underlie the restructuring of knowledge structures.

Lin Chen received a BA and MA in Linguistics from Tsinghua University, China in 2000 and 2004. He is currently pursuing a PhD degree in Computer Science at the University of Illinois at Chicago. He is a member of the Natural Language Processing Lab at UIC, and his research interests include natural language processing, information extraction, and information retrieval.