Journal of Computer Assisted Learning (2003) **19**, 416-428

Reciprocal tutoring using cognitive tools

W.K. Wong, T.W. Chan*[†], *C.Y. Chou*[‡], *J.S. Heh*[‡] & *S.H. Tung*[*]

[*]National Yunlin University of Science and Technology, [†]National Central University, [‡]Chung Yuan Christian University, Taiwan, R.O.C.

Abstract Reciprocal tutoring, where peers take turns to tutor each other, is an interesting style of social learning. In the Reciprocal Tutoring System (RTS), three computational cognitive tools were designed to facilitate reciprocal tutoring of Lisp programming on the network. The first is a Petal-style of code–chunk interface, with which a tutee can enter Lisp code without making syntactic errors. The second tool is Diagnosis-Hint Tree, with which a tutor can diagnose and comment on the errors in the tutee's program. The third one is a list of dialogue templates, with which the tutee and the tutor can communicate during the tutoring process. A three-phase experiment was conducted, with each phase using different cognitive tools. In addition, with the help of the cognitive tools, RTS provides a virtual learning companion that can play tutor or tutee. Evaluation results reveal both the strengths and weaknesses of peer-based learning and intelligent tutoring, with supports of different cognitive tools. Peer-based learning supported by cognitive tools is a practical and attractive alternative to intelligent tutoring systems. Exactly which type of tutor is preferred depends on the tutee's cognitive, communication, and emotional needs in the tutorial context.

Keywords: Agents; Attitude; Collaboration; Distance; ICT-use; Summative; Undergraduate

Introduction

For about 30 years, researchers have been designing intelligent tutoring computer systems for educational purposes. A traditional intelligent tutoring system (ITS) acts like an authoritative tutor who helps students to solve problems (e.g. Carbonell, 1970; Sleeman & Brown, 1982; Wenger, 1987; Self, 1995). Such a system provides expert solutions, diagnoses student's problems, and interacts with students adaptively if it keeps tracks of the student's progress. A traditional intelligent tutoring system usually consists of four components: an expert model, a user model, a tutoring model, and a user interface (Wenger, 1987). The expert model solves problems like an expert and can diagnose the problems in a user's solution. The tutoring model uses some instruction strategy to explain the problems to the user. If

Accepted 1 November 2002

Correspondence: Dr. Wing-Kwong Wong, Department of Electronic Engineering, National Yunlin University of Science & Technology, 123 University Road Section 3, Touliu, Yunlin County, 640, Taiwan, R.O.C. Email: wongwk@yuntech.edu.tw

the tutoring model is smart enough, then the explanation should be expressed in terms of concepts that are familiar to the students (e.g. Lester & Porter, 1991). This kind of adaptive response usually uses some student model to keep track of the state of the student's knowledge level, emotional level, etc. (e.g. Lester *et al.*, 1999). The tutee's interface displays the tutor's comments, and provides friendly tools for the tutee to work out her solution and send questions to the tutor.

Although the research on ITS has produced many prototypical tutoring systems in the lab, few of them have turned into commercial software products that really help students, especially at K12 levels, learn their curricular materials. This lack of commercialisation of intelligent tutoring systems might result from the difficulties of building the three knowledge models of an ITS. First, the expert model of an ITS needs expert knowledge on problem solving but it is generally very labour-intensive, time-consuming, and expensive to extract the knowledge from experts or to design the knowledge from scratch. This is the well-known bottleneck problem of knowledge engineering in expert system construction. Second, the user model of an ITS that contains the bug patterns of users during problem solving is just as difficult to build as the expert model. It requires the collection, analysis and summarisation of a large amount of students' data on problem solving. Third, the tutoring model of an ITS that makes use of tutoring strategies presents the highest hurdle. Any teacher would agree that how to guide a student to correct her misconceptions is more an art than a science. That is one of the reasons why the teaching method of Socrates to guide his students to enlightenment through a series of provoking questions that force students to reflect deeply on the answers by themselves is highly valued (e.g. Collins & Stevens, 1982).

There are ways to address these serious questions faced by the researchers on ITS. One alternative to using a virtual tutor agent is to have students help each other solve problems through a computer network. This can be called networked peer-based cooperative learning. This mode of learning avoids the problems of knowledge engineering in ITS construction, i.e. there is no need to extract the tutoring knowledge out of the students' brains. Recall the tutoring knowledge of an ITS includes an expert model, a student model, and a tutoring model. In the learning mode of networked peer-based cooperation, these three knowledge-based components need not be built. This learning mode also eliminates the cost of getting experts to help the students. Moreover, according to educational psychologists (e.g. Slavin, 1995; Johnson & Johnson, 1999), peer-based cooperative learning offers a lot of educational benefits.

An important mode of cooperative learning is learning by reciprocal teaching in a peer group setting (e.g. Palinscar & Brown, 1984). In this learning mode, each member of the peer group takes turns to act as a leader in questioning other members and summarising their answers in order to solve their problems. Students can learn from each other by considering multiple perspectives (Piaget, 1955). They might learn to detect errors in their partners or in themselves, make arguments to support themselves and criticise others, revise their solutions to accommodate others' comments and suggestions, etc. According to Vygotsky's learning theory, a child who interacts with a more knowledgeable peer can learn to become as knowledgeable as the peer. This potential learning space for the learner is called the *zone of proximity of development* (Vygotsky, 1978). In this way, each student can learn to diagnose a peer's problems, expand her own vision, get more critical of others and of herself, and thus enhance her meta-cognition and mental capacity for critical thinking. In the last decade or so, ITS researchers began to pay more attention to designing social learning environments where students interact not just with authoritative tutors but with learning companions who play the roles of competitor, collaborator, etc. (e.g. Gilmore & Self, 1988; Chan & Baskin, 1990).

In this study computer science teachers, built a reciprocal tutoring environment for students to learn programming in Lisp. This environment is called the Reciprocal Tutoring System (RTS) (Chan & Chou, 1997; Chou *et al.* 2002). In a learning session, two peer students take turns to tutor each other on solving several programming problems. In general, one problem of reciprocal tutoring is that if the problems encountered by the tutee-tutor pair get too difficult, they might feel helpless and learning can become annoying and ineffective. To address this problem, the learning environment provides several cognitive tools. The first facilitates the input of Lisp programs without making syntactic errors. The second facilitates the diagnosis of a buggy program and the formulation of hints about the bugs. The third facilitates the communication process of a tutorial dialog. The study compared the tutor's quality and the tutee's learning conditions under the guidance of peer tutors with different levels of cognitive tools. Understanding of the strengths and weaknesses of these cognitive tools can help improve the design of cognitive tools for collaborative learning environment in the future.

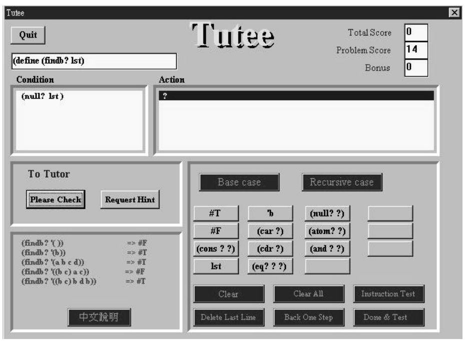
This paper is organised in the following way. The three cognitive tools used in RTS are introduced first: the code–chunk interface, the Diagnosis-Hint Tree, and the communication templates. Then the tutor agent and the tutee agent are described and the tutor agent is compared to traditional intelligent tutoring systems. An experiment on RTS is presented and the final section concludes with the findings and limitations of this study and suggests future research directions.

PETAL-like code chunk interface for tutee

When two students log in the Reciprocal Tutoring System (RTS) through the network in a learning session, the system groups them as a tutor-tutee pair. The tutee is asked to write a few LISP programs and can communicate with the tutor. After each program is finished, the tutee and tutor switch their roles and the new tutee works on the next program. This process repeats until all the programs are finished. During a tutorial session, the tutor can see what steps the tutee takes to solve a problem. When the tutee asks the tutor a question, the tutor is obliged to answer that question. The tutor cannot initiate any comments or questions to interrupt the tutee's problem solving process.

During the programming activity in RTS, the primary programming tool for the tutee is a *petal-like-system (PLS)* to simplify the task of writing a program, as shown in Fig. 1. PLS adopts the design of the Petal system, called PET1, which provides a mental-model-based programming environment using a set of predefined *code chunks* to free the programmer from dealing with the syntactic problems (Bhuiyan *et al.,* 1992). The PTS code-chunk screen for the tutee is shown in Fig. 1. At the bottom left window is the input-output behaviour of the function that the tutee is asked to write. The input-output examples provide an intuitive way of understanding the function, in addition to the verbal description of the programming problem. From the examples and the verbal description, the tutee should know that '*findb?*' takes one argument '*lst*', which is a list, and returns true if '*lst*' contains the element '*b*' at the top level, i.e. not embedded in a list within '*lst*'; otherwise, the function returns false. For example, calling (*findb? '(a b c d)*) returns true but calling (*findb? '((b c) a (c)*) returns false. A solution program for this problem is:

```
(define (findb? 1st)
    (cond
        ((null? lst) #F)            % Comment: first condition-action pair
        ((eq? (car lst) 'b) #T)     % Comment: second condition-action pair
        (#T (findb? (cdr lst)))))   % Comment: third condition-action pair
```



```
Total Score     0
Problem Score   14
Bonus           0

(define (findb? lst)

Condition: (null? lst)
Action: ?

(findb? '( ))            => #F
(findb? '(b))            => #T
(findb? '(a b c d))      => #T
(findb? '((b c) a c))    => #F
(findb? '((b c) b d b))  => #T
```

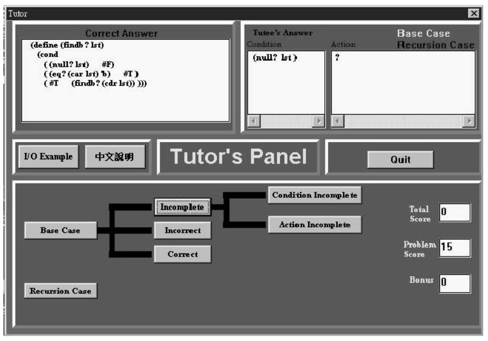
Fig. 1. Tutee's panel with code–chunk interface

How does the tutee use the *code chunk* buttons to write Lisp programs? Suppose the tutee works on the condition part of the first base case of the '*findb'* function. She need not type in the code *'(null? lst)'* as one would in a traditional Lisp interpreter. Instead, all she needs to do is to push the button *'(null? ?)'* followed by the button *'lst'.* Both buttons are in the panel of code chunk buttons below the two 'Case' buttons, as shown in Fig. 1. When the tutee clicks the button labelled *'(null? ?)',* the code *'(null? ?)'* appears on the Condition column on the left part of the window just below the code *'(define (find? lst)'* near the top left corner. Then when she clicks the button labelled *'lst',* the code *'(null? ?)'* will be converted to *'(null lst)',* since the standalone *'?'* symbol is replaced by *'lst'.* The code is not 'complete' if it contains any standalone *'?'* symbol. If the code contains one or more *'?'* symbols, then the leftmost such symbol would be replaced by the code chunk whose button is clicked. The tutee repeats the process until the code is complete. For another example, the Lisp code for the condition part of the second condition-action pair is *'(eq? (car lst) 'b)'.* This code is constructed with the button sequence *'(eq? ? ?)', '(car ?)', 'lst', "'b".* The corresponding resulting code sequence is *'(eq? ? ?)', '(eq? (car ?) ?)', '(eq? (car lst) ?)', '(eq? (car lst) 'b)'.*

Diagnosis-Hint tree for the tutor

The tutor's major responsibility is to guide the tutee to write a correct Lisp program for computing the function given in each exercise. Since each student needs to play tutor during a learning session, she is obliged to help the tutee. But there is no guarantee that the student tutor is knowledgeable enough to give the tutee a fruitful learning experience. Therefore, RTS provides a 'super-tutor' to help a student play tutor. The super-tutor will give sufficient guidance to the tutor to diagnose the bugs in the tutee's program. In this way, the tutor is actually learning the concepts of recursive programming, though in a different way from that of the tutee. This super-tutor is called the 'Diagnosis-Hint Tree' and will be described in detail in this section. The working interface window for the tutor is shown in Fig. 2.

When the tutor receives a message from the tutee asking for help, she would use the Diagnosis-Hint Tree to diagnose the tutee's program code. Such a tree provides a search space for possible Lisp codes that the tutee might produce for one specific exercise. A typical tree is shown in the bottom half of Fig. 2. The default *'Root'* node, which is invisible in the figure, represents a state when the tutee has not written any code yet. The children of the root are *'Base Case'* node and *'Recursive Case'* node. Each of these two nodes has three children: *'Incomplete', 'Incorrect', 'Complete'*. *'Incomplete'* means that the tutee's Lisp code lacks some part, i.e. the code contains a standalone*'?'* symbol. *'Incorrect'* means that the tutee's code uses an incorrect function or argument. *'Correct'* means that the code is correct and complete. Each of the *'Incomplete'* and *'Incorrect'* nodes has two children: *'Condition'* and *'Action'*. The *'Condition'* child node of the *'Incorrect'* node means that the condition part of the recursive condition-action pair is incorrect while the *'Action'* child node of the *'Incorrect'* node means that the action part is incorrect. If the tutee's current code is correct and complete, then the tutor's task is to go down the tree along the path leading to the node that represent the correctness of the last condition-action pair in the tutee's code. If the tutee's code has a bug, i.e. incomplete or incorrect, the tutor's task is to identify the first bug that appears in the tutee's program code by exploring a branch of the tree starting from the root and moving to the right along a path whose destination node represents the bug. Thus any possible Lisp code from the tutee, whether buggy or not, corresponds to a unique path in the diagnosis tree and the tutor is responsible for finding such a path, which is called the *diagnosis path.*



```
(define (findb ? lst)
  (cond
    ((null? lst)    #F)
    ((eq? (car lst) 'b)    #T)
    (#T    (findb ? (cdr lst)) )))
```

```
(null? lst)
```

```
?
```

Total Score: 0
Problem Score: 15
Bonus: 0

Fig. 2. Tutor's panel in Phase A

Suppose the tutee has produced the following Lisp code in writing the *'findb?'* function:

```
(define (findb? 1st)
  (cond
    ((null? lst) #F)
    ((eq? (car lst) 'b) #T)
    (#T (findb? (car ?)))))
```

Now the tutee requests hints from the tutor, who is responsible for diagnosing the tutee's code and responding accordingly. This code is both incorrect and incomplete. It is incorrect because in the third condition-action pair '*(#T (findb? (car ?)))', * the first argument *'(car ?)'* of the action code should be *'(cdr ?)'*. It is incomplete because there exists the *'?'* symbol in *'(car ?)'*. Since Lisp code is read from left to right, the incorrectness occurs before the incompleteness so the proper diagnosis should be 'incorrect code'. In this example, the diagnosis path the tutor needs to identify is *['Root'-'Recursive case'-'Incorrect'-'Action'],* which is already found by the system. The tutor must click the nodes one after another along the path. After the tutor identifies a proper diagnosis for the tutee's program, she is then in a position to give hints to the tutee on how to proceed. The moment when the tutor finds the diagnosis path, i.e. when she clicks the last (destination) node of the path, a list of prescribed hints about the bug is displayed for the tutor to pick. This is why the tree is called Diagnosis-Hint Tree, not just diagnosis tree. These hints range from general and abstract remarks to specific and correct answer. To continue with the above example, some hints corresponding to the bug *'(car ?)'* are:

- The action of the third condition-action pair is incorrect.
- If '*b* exists in the tail of *lst*, this function should return #T.
- We have already checked (*car lst*), there is no need to check it again.
- We have to check whether '*b* is in the tail of *lst*.
- The correct action should be (*findb?* (*cdr lst*)).

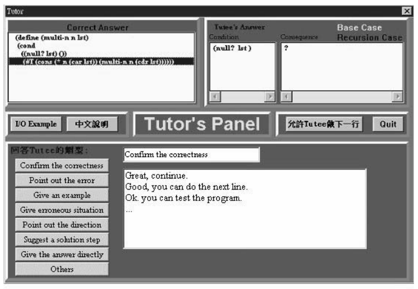
There is a scoring mechanism to encourage the tutor to use more general comments rather than giving away the correct answer directly. A typical tutoring session is summarised in Table 1. In this session, the tutor provided five hints for the tutee, clicked nine correct nodes and two incorrect nodes. In general, the tutor would not make too many mistakes before finding the diagnosis path since the DHT is a rather small tree for simple programming problems.

Table 1. Summary of a tutoring session using DHT

Tutee's code	Diagnosis path in DHT	No. of hints requested by tutee before changing the code	Tutor clicks nodes of DHT before sending hint	No. of correct nodes clicked by tutor	No. of incorrect nodes clicked by tutor
Code A	Root- Base case- Incomplete- Action	2	(Root-) Base case- Incomplete- Action	3	0
Code B	Root- Base case- **Incorrect-** Condition	2	(Root-) Base case- **Correct**	1	1
			(Root- Base case-) Incorrect- Condition	2	0
Code C	Root- **Recursive case-** Incorrect- Action	1	(Root-) **Base case**	0	1
		1	(Root-) Recursive case- Incorrect- Action	3	0
Code D	**No bug**				
		Total = 5		**Total = 9**	**Total = 2**

Dialogue templates for tutee and tutor

The Diagnosis-Hint Tree is a powerful tool in providing expert knowledge for the tutor to diagnose the tutee's LISP programs and pick hints for the tutee. Since the study was interested in investigating the effect of this tool on the reciprocal tutoring process, an alternative tool for the tutor, which uses no diagnostic know-ledge was designed. The tool does not provide any diagnostic capability but a set of generic dialogue templates for the tutee and the tutor to discuss the tutee's problems (Fig. 3). Therefore, the tutor's diagnosis is based purely on her own knowledge.



```
(define (multi-n n lst)
 (cond
  ((null? lst) ())
   (#T (cons (* n (car lst)) (multi-n n (cdr lst))))))
```

(null? lst)

?

Confirm the correctness

Great, continue.
Good, you can do the next line.
Ok. you can test the program.
...

Fig. 3. Tutor dialogue templates in phase B

The templates are modelled after the theory of *Inquiry teaching strategies* (Collins & Stevens, 1982). Many works use dialogue templates like these (e.g. Pilkington, 2001).

For example, a list of *initial questions* is provided for the tutee to choose after she initiates a request for help. The five initial questions are:

- I0: Is that right?
- I1: What should I do?
- I2: What to do next?
- I3: What should I consider?
- I4: Can you tell me the answer?

Upon receiving a request from the tutee, the tutor can choose a sample hint from a given list, modify it, and send to the tutee. After getting the tutor's hint, the tutee can pick up some sample responses from a given list to respond to the tutor. These responses include both follow-up questions to continue the dialogue and final acknowledgements that report the tutee's understanding about the tutor's hint and end the conversation. *'Could you provide more hints?'* is a typical follow-up question and *'Great! I think I understand. Let me try again.'* is a typical acknowledgement message. All the dialogue templates are shown in Table 2.

Table 2. Dialogue templates between tutee and tutor

Tutee's initial questions:	
I0	Is that right?
I1	What should I do?
I2	What to do next?
I3	What should I consider?
I4	Can you tell me the answer?
Tutee's follow-up questions:	
F0	Can you give me more hints?
F1	I cannot understand. Can you explain clearly?
F2	What should I consider?
F3	Why?
F4	What is the point?
Tutee's final acknowledgement:	
A0	Great! I understand. Let me try.
A1	Perhaps I understand. Let me try.
A2	Although I understand little, let me try anyway.
A3	I do not understand completely, but let me try.
Tutor's responses:	
R0	Confirm the correctness of the tutee's program.
R1	Point out the error.
R2	Give an example.
R3	Give a possible erroneous situation.
R4	Point out the direction to pursue.
R5	Suggest a solution step.
R6	Give the answer directly.
R7	Others

Comparing the tutor agent of RTS to intelligent tutoring systems

The tutor agent of RTS is similar to the Lisp Tutor, which is a well-known intelligent tutoring system (Anderson *et al.*, 1985). Both tutor agents guide the tutee to finish the program code for specific, prescribed problems. This is the general approach of learning by doing. The task of programming provides a tutorial context in which the tutee acquires skills of coding simple recursive programs. Both systems use a model-directed tutoring strategy called *model-tracing* by the Lisp Tutor. The student model is assumed to be an overlay on the merging of an ideal expert model and a bug model. The expert solutions to a few specific problems are enumerated and prescribed as knowledge structures. Such a compiled expert model is different from an articulated model that can generally compute a solution dynamically by searching through many evolving hypotheses. Examples of articulated expert model include that of the medical diagnosis in MYCIN (Clancey *et al.*, 1984) and that of planning program writing in SPADE (Miller, 1979). In RTS and the Lisp Tutor, the tutee is kept on track with the tutor agent's hints, as the bugs in the tutee's program are detected and reported by the tutor agent. For the student model, all possible bugs are enumerated. Diagnosis is mostly done with a one-step search, which is much more efficient than general planning that might spend a lot of time searching for a solution path. This enumerative theory of bugs assumes that bugs originate in the tutee's mislearning and forgetting, that is, the tutee fails to follow, organise, or internalise an instructional sequence.

An enumerative theory differs from a reconstructive theory and a generative theory of bugs (Wenger, 1987; pp. 348–350). In a reconstructive theory, bugs are reconstructed from observed errors, generally in a bottom-up, data-driven manner. Such theory usually uses a performance model with articulated knowledge for reconstructing observed bugs. Examples include PROUST (Johnson & Soloway, 1985), ACM (Langley & Ohlsson, 1984), MACSYMA ADVISOR (Genesereth, 1982), and PIXIE (Sleeman, 1985). In a generative theory, psychologically plausible mechanisms are proposed on top of a performance model. In this way, surface errors can be explained and even predicted with misconceptions at deeper levels of the performance model, independently of specific contexts or diagnosis method. Examples include REPAIR/STEP (Brown & VanLehn, 1980; VanLehn, 1983), WHY (Stevens *et al.*, 1979), ENVISION (de Kleer & Brown, 1984), MENO-II (Soloway *et al.*, 1983), and Matz (1982). Learners are regarded as agents actively working on theory formation and progressing with an evolving model of the world, rather than simply following instructions. However, how to come up with such an ideal learning model for the task of programming or other nontrivial problem solving domains is still an open research question. Although the enumerative theory of bugs is considered to provide the least explanatory value among the three theories, it is practical and efficient in well-structured, restricted domains for inexperienced learners.

Both RTS and the Lisp Tutor used the pedagogical principle of 'immediate feedback' during tutorial intervention (Wenger, 1987; p. 296). In RTS, whenever the tutee asks for help, the tutor agent diagnoses the first bug in the tutee's program. Being more active than the tutor agent in RTS, the Lisp Tutor monitors every piece of code the tutee enters and, if the code is incorrect, actively reports the bug. This immediate feedback prevents the tutee from travelling far down along an erroneous path. If feedback is delayed and the erroneous path gets too long, the problem of assigning blame to a node on the path will generally get more difficult. This results in local corrections for the coding problem in an incremental overlay on the ideal and buggy models of the tutee's knowledge.

Despite the many similarities between RTS' tutor agent and the Lisp Tutor, they differ in their representation of expert skill and bug library. In RTS, expert skill and bug library are represented as the Diagnosis-Hint Tree. The tutee is expected to differentiate between the base cases and the recursive cases, which are represented as different branches in DHT, between the condition and the action codes, which again form sub-branches in DHT, etc. In the Lisp Tutor, these conceptual distinctions are represented as production rules. The following is an example of a production rule (Anderson *et al.*, 1985):

```
IF      The goal is to code a recursive function, FUNCTION, with an integer
        argument, INTEGER,
THEN    Use a conditional structure and set as subgoals
        (1) to code the terminating condition when INTEGER is 0, and
        (2) to code the recursive case of FUNCTION(INTEGER – 1)
```

Besides differentiating the concepts of base case and recursive case, production rules have subgoal structures. The task of programming is thus done through the achievement of a series of subgoals and embedded subgoals. Moreover, the rule specifies constraints on the parameter INTEGER. Such constraints are not explicitly represented in the current implementation of DHT but can be added as extended branches of DHT.

Experiment

Students in two introductory programming courses took part in an experiment with RTS. One of the courses was taught at the Department of Information Engineering, National Central University, the other at the Department of Electronic Engineering, National Yunlin University of Science and Technology. These two universities are about 150 km apart. The total number of students in these two courses was 53. These students used the Reciprocal Tutoring System with computers at their own schools. When a student in RTS chose to team up with a student who logged onto RTS at the same time, they then communicated with each other through the Internet. In other words, students from different schools used RTS to do distance learning. Two grading policies were adopted to motivate students to use RTS. First, each student was required to use RTS at least once to avoid a grade penalty. In addition, some bonus points would be given to the students who used RTS more than once.

A three-phase experiment was conducted (Phases A, B and C), with each phase using different cognitive tools. The goal was to test how these tools affect the learning behaviour of the students. In Phase A, the tutee used the code–chunk interface and the tutor used the Diagnosis-Hint Tree to select hints for the tutee. In Phase B, the Diagnosis-Hint Tree was replaced with dialogue templates. Without any help from the system, the tutor needed to diagnose the tutee's program on her own. Moreover, Phase B used harder problems so that the tutoring dialogue would be more demanding. The '*sum-all*' procedure, which adds all the numbers in a nested list, was an example of the problems the students solved in Phase B:

```
(sum-all '(1 2 3)) = > 6
(sum-all '(1 (2 3) ((4)))) = > 10
```

In Phase C, the dialogue templates were removed so that the tutor must key in hints by hand. In addition, each student was arranged to work with a fellow student and a virtual companion at different times so that she could compare the performance of the virtual companion to that of a fellow student.

Conclusion and discussion

Intelligent tutoring systems attempt to provide smart tutorial guidance to improve the effectiveness and efficiency of learning. However, serious knowledge engineering problems are encountered when builders of ITS try to formalise and organise the knowledge of the expert model of problem solving, student model of bugs, and model of tutoring strategies. One of the goals of this study is to check the feasibility of using peer-based reciprocal tutoring to bypass the knowledge engineering problems of intelligent tutoring systems. Another goal is to design a learning companion agent that can play tutor or tutee in reciprocal tutoring.

To get an idea of the similarities and differences between learning with a peer tutor and learning with a virtual tutor, an experiment was run in three phases during which students used the Reciprocal Tutoring System with different cognitive tools. During Phase A, tutee programmed with the code chunk tool while tutor used the DHT for diagnosis and hinting. During Phase B, tutee used the same tool but tutor communicated with dialogue templates. During Phase C, tutee typed programs manually and tutor made diagnosis with no DHT and typed hints manually. Using the experimental data and students' comments collected after the experiment, it is possible to compare the tutor's quality (seen from the tutee's perspective) for different tutor types and do the same for the tutee's learning conditions when guided by different types of tutors. These tutor types are differentiated according to the cognitive tools that they use (or not use). The first cognitive tool is the list of dialogue templates, which has some common communication knowledge but little knowledge of Lisp programming compared with the second knowledge-rich tool of the Diagnosis-Hint Tree. In short, the scenarios of four types of tutor, in order of increasing cognitive assistance, are compared. The first is a peer tutor diagnosing the tutee's code and typing tutorial comments by herself. The second is a peer tutor making diagnosis by herself and using dialogue templates. The third is a peer tutor using DHT for diagnosis and hints. The last is a virtual tutor, again based on DHT, which is similar to the tutor agent of an intelligent tutoring system. Thus these tutor types lie on a spectrum that extends, with increasing support of cognitive tools, between two extremes of a peer tutor with no cognitive tools and a virtual tutor.

Table 3. Tutor's quality for different tutor types from tutee's perspective

Tutor Types: **Tutor's quality:**		**Peer tutor (no tools)** **vs.** **Human tutee + tutee (no tools)**	**Peer tutor + dialogue templates** **vs.** **Human tutee + dialogue templates**	**Peer tutor + DHT** **vs.** **Human tutee + code chunks**	**Virtual tutor (with DHT)** **vs.** **Human tutee + code chunks**
Diagnosis	Knowledge	Medium	Medium	High	High
	Responsiveness	Low	Low	Medium	High
	Variation	High	High	Low	Low
Hint	Authority	Low	Medium	High	High
	Responsiveness	Medium	High	Medium	High
	Flexibility	High	Medium	Low	Low

Tutor's quality, from the tutee's perspective, is rated in two aspects: diagnosis and hinting (Table 3). Diagnostic skills include the knowledge level, responsiveness, and variation. Hinting styles include authority, responsiveness, and flexibility. Without the support of Diagnosis-Hint Tree, a peer tutor has similar diagnosis knowledge as the tutee on the average, is less responsive in diagnosis, and has greater variation due to individual difference. In contrast, a peer tutor using DHT is similar to a virtual tutor, whose design is based mainly on DHT, in that both tutors are knowledgeable in diagnosis and there is little variation in diagnostic performance despite individual difference among peer tutors. Nevertheless, the peer tutor using DHT is less responsive than the virtual tutor during the diagnosis process because the peer tutor needs to spend time figuring out the diagnosis path and might make mistakes from time to time. With respect to the tutor's authority in the hinting process, the peer tutors not using DHT have lower authority than the other two tutor types. Also, the tutor using no tools is less authoritative than the tutor using dialogue templates. For hint responsiveness, both the peer tutor using dialogue templates and the virtual tutor are more responsive than the peer tutor without tools and the peer tutor with DHT. Finally, the flexibility of hints is greatest for the peer tutor with no tools, less so for the peer tutor with dialogue templates, and least for the other two tutor types.

Table 4. Tutee's conditions when guided by different tutor types

Tutor Types: / **Tutee's condition:**		**Peer tutor (no tools)** vs **Human tutee + tutee (no tools)**	**Peer tutor + dialogue templates** vs **Human tutee + dialogue templates**	**Peer tutor + DHT** vs **Human tutee + code chunks**	**Virtual tutor (with DHT)** vs **Human tutee + code chunks**
Problem solving	Programming knowledge	Medium	Medium	Low	Low
	Reliance on guidance	Low	Medium	High	High
	Self reflection	High	Medium	Low	Low
Communi-cation	Flexibility	High	Medium	Low	Low
	Effort	High	Medium	Low	Low
	Willingness to challenge	High	Medium	Low	Low
Emotion	Devotion	High	High	High	Medium
	Pressure	Medium	Medium	Medium	Low

Tutee's learning conditions include three main aspects: problem solving, communication, and emotion (Table 4). Problem solving conditions include the knowledge level, reliance on tutorial guidance, and self-reflection. Communication conditions include flexibility, effort, and amount of challenge posed to the tutor. Emotional aspects include devotion and pressure. For the aspect of problem solving, the tutee has similar programming knowledge as the peer tutor not using DHT and is less knowledgeable than the other two tutor types. The tutee relies the least on the peer tutor with no tools since she needs to take the greatest effort in requesting hints that are less helpful compared to the hints from the other more 'knowledgeable' tutor types. The tutee relies more on the peer tutor with dialogue templates and most on the other two tutor types. With respect to communication, flexibility is greatest for tutee when tutor and tutee exchange comments by typing manually, less so when tutor and tutee both use dialogue templates, and the least for the other two tutor types. The tutee spends the most effort in communicating with the tutor with no tools, less so with the tutor using dialogue templates, and the least when communicating with the other two tutor types. Similarly, the tutee challenges the peer tutor most when no tools are used, less so for the peer tutor using dialogue templates, and the least for the other two tutor types. With respect to emotion, the tutee is more devoted but might feel more pressure when guided by the three types of peer tutors compared to the virtual tutor.

Furthermore, it is possible to compare how the three types of human tutor learn from tutoring. The basic patterns are very similar to the learning conditions of the tutee. When a cognitive tool is used, the tutor relies more on the tool, spends less effort and reflects less on her own but the tutoring process proceeds more smoothly. When the tutee is a human peer, the tutor is more devoted and feels more pressure. When the tutee is a virtual agent, the tutor is less devoted and feels less pressure.

The above comparison shows the strengths and weaknesses of the four tutor types. While the virtual tutor is knowledgeable in diagnosing bugs in the tutee's code and leads the tutee to produce correct code with strong guidance, the tutee might seldom challenge the tutor and become less active in reflective thinking.

While a peer tutor with no cognitive tools might be less knowledgeable than a virtual tutor, the tutee might challenge the tutor more often and try to solve problem by herself. If a peer tutor is needed to get the tutee more devoted to the learning activity and it is necessary to know the that tutor is competent, it may still be best to use a peer tutor and provide cognitive tools to increase her tutoring skills of problem solving or communication. Exactly what tools to provide depends on the purpose and the specific needs of the tutee in the tutorial context. Therefore, for a particular task, if the engineering knowledge problems of an intelligent tutoring system cannot be solved, peer-based tutoring with cognitive tools can be a very practical and attractive alternative. Furthermore, if the needs of the tutee can be detected dynamically in the tutorial process, then a virtual tutor can adapt to the tutee's needs by adjusting the level of cognitive support. This idea points to the important research problems concerning how to detect the changing needs of the tutee dynamically as tutoring proceeds. Some of the problems are addressed by various knowledge negotiation methods (for example, Moyse & Elsom-Cook, 1992).

Acknowledgements

We acknowledge the support of National Science Council (NSC 91–2520-S-224–001) and the Ministry of Education (89-H-FA07-1-4), Taiwan, R.O.C. Shi-Sheng Yang and Ruei-Liou Huang are also appreciated for their assistance in implementing the system. Two anonymous reviewers' critical comments are much appreciated for driving us to rethink and rephrase some of the issues of an early draft of the paper. We also thank Meihua Liao for editing and commenting on the paper.

References

Anderson, J.R., Boyle, C.F. & Reiser, B.J. (1985) Intelligent tutoring systems. *Science,* **228,** 4698, 456-462.

Bhuiyan, S., Greer, J.E. & McCalla, G.I. (1992) Learning recursion through the use of a mental model-based programming environment, In *Lecture Notes in Computer Science, 608.* (eds. C. Frasson, G. Gauthier & G.I. McCalla) pp. 50-57. Springer-Verlag, Berlin.

Brown, J.S. & VanLehn, K. (1980) Repair theory: a generative theory of bug in procedural skills. *Cognitive Science*, **4**, 379–426.

Carbonell, J.R. (1970) AI in CAI: an artificial-intelligence approach to computer-assisted instruction. *IEEE Transactions on Man-Machine Systems*, **11**, 4, 190–202.

Chan, T.W. & Baskin, A.B. (1990) Studying with the prince: The computer as a learning companion. In *Intelligent Tutoring Systems: at the Crossroad of Artificial Intelligence and Education* (eds. C. Frasson & G. Gauthier) pp. 6-33. Ablex, Norwood, NJ.

Chan, T.W. & Chou, C.Y. (1997) Exploring the Design of Computer Supports for Reciprocal Tutoring. *Intl. Journal of Artificial Intelligence in Education*, **8**, 1-29.

Chou, C.Y., Lin, C.J. & Chan, T.W. (2002) An Approach to Developing Computational Supports for Reciprocal Tutoring. *Knowledge-Based Systems*, **15**, 7, 407–412.

Clancey, W.J., Shortliffe, E.H. & Buchanan, B.G. (1984) Intelligent computer-aided instruction for medical diagnosis. In *Medical Artificial Intelligence: the First Decade*. (eds. W.J. Clancey, & E.H. Shortliffe). pp. 256-274. Addison-Wesley, Reading, MA.

Collins, A. & Stevens, A. (1982) Goals and strategies of inquiry teachers. In *Advances in Instructional Psychology Vol 2.* (ed. R. Glaser) pp. 65–119. Erlbaum, Hillsdale, NJ.

Genesereth, M.R. (1982) The role of plans in intelligent teaching systems. In *Intelligent Tutoring Systems*. (eds. D.H. Sleeman & J.S. Brown) pp. 137-156. Academic Press, London.

Gilmore, D. & Self, J. (1988) The application of machine learning to intelligent tutoring systems. In *Artificial Intelligence and Human Learning, Intelligent Computer-Aided Instruction* (ed. J. Self) pp. 179–196. Chapman & Hall, New York.

Johnson, W.L. & Soloway, E. (1985) PROUST: An automatic debugger for Pascal programs. *Byte,* **10**, 4, 170-190.

Johnson, D.W. & Johnson, R.T. (1999) *Learning Together and Alone*. Allyn and Bacon, Boston, MA.

de Kleer, J. & Brown, J.S. (1984) A physics based on confluences. *Artificial Intelligence*, **24**, 7–83.

Langley, P. & Ohlsson, S. (1984) Automated cognitive modeling. In *Proceedings of the National Conference on Artificial Intelligence*. pp. 193-197. AAAI Press, Menlo Park.

Lester, J.C. & Porter, B.W. (1991) Generating context-sensitive explanations in interactive knowledge-based systems. *AI Technical Report 91–160.* The University of Texas at Austin, Department of Computer Sciences, Austin, Texas.

Lester, J.C., Towns, S.G. & Fitzgerald, P.J. (1999) Achieving affective impact: visual emotive communication in lifelike pedagogical agents. *International Journal of Artificial Intelligence in Education*, **10**, 278–291.

Matz, M. (1982) Towards a process model for high school algebra. In *Intelligent Tutoring Systems*. (eds. D.H. Sleeman & J.S. Brown) pp. 25-50. Academic Press, London.

Miller, M.L. (1979) A structured planning and debugging environment for elementary programming. *International Journal of Man-Machine Studies*, **11,** 79–95. Reprinted in *Intelligent Tutoring Systems*. (eds. D.H. Sleeman & J.S. Brown). pp. 119-135. Academic Press, London.

Moyse, R. & Elsom-Cook, M.T. (eds.) (1992) *Knowledge Negotiation*. Academic Press, London.

Pilkington, R. (ed.) (2001) Special issue on analysing educational dialogue interaction -Part II. *International Journal of Artificial Intelligence in Education,* **12**, 1.

Palinscar, A.S. & Brown, A.L. (1984) Reciprocal teaching of comprehension monitoring activities. *Cognition and Instruction*, **2**, 117–175.

Piaget, J. (1955) *The Language and Thought of the Child.* New American Library, NY.

Self, J. (1995) An Introduction to Artificial Intelligence in Education. In *Emerging Technologies in Education* (eds. T.W. Chan & J. Self) pp. 3-20. AACE, Charlottesville.

Slavin, R.E. (1995) *Cooperative Learning: Theory, Research, and Practice*. Allyn and Bacon, Boston, MA.

Sleeman, D.H. (1985) Inferring (mal) rules from pupils' protocols. In *Progress in Artificial Intelligence* (eds. L. Steels & J.A. Campell) Ellis Horwood, Chichester.

Sleeman, D. & Brown, J. (1982) *Intelligent Tutoring Systems*. Academic Press, London.

Soloway, E.M., Rubin, E., Woolf, B.P., Bonar, J. & Johnson, W.L. (1983) MENO-II: an AI-based programming tutor. *Journal of Computer-Based Instruction*, **10**, 1, :20–34.

Stevens, A.L., Collins, A. & Goldin, S. (1979) Misconceptions in students' understanding. *Proceedings of the 5th European Conference on Artificial Intelligence,* pp. 160-164. Reprinted in *Intelligent Tutoring Systems* (1985) (eds. D.H. Sleeman & J.S. Brown). pp. 13-24. Academic Press, London.

VanLehn, K. (1983) Human procedural skill acquisition: theory, model, and psychological validation. In *Proceedings of the Third National Conference on Artificial Intelligence,* pp. 420–423. AAAI Press, Menlo Park, CA.

Vygotsky, L. (1978) *Mind in Society*. (Translation: M. Cole, V. John-Steiner, S. Scribner & E. Souberman). Harvard University Press, Cambridge, MA.

Wenger, E. (1987) *Artificial Intelligence and Tutoring Systems.* Morgan Kaufmann, Los Altos, CA.