A Review of AI-Supported Tutoring Approaches for Learning Programming

Nguyen-Thinh Le, Sven Strickroth, Sebastian Gross, and Niels Pinkwart

Clausthal University of Technology,
Department of Informatics,
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
{nguyen-thinh.le,sven.strickroth,sebastian.gross,
niels.pinkwart}@tu-clausthal.de

Abstract. In this paper, we review tutoring approaches of computer-supported systems for learning programming. From the survey we have learned three lessons. First, various AI-supported tutoring approaches have been developed and most existing systems use a feedback-based tutoring approach for supporting students. Second, the AI techniques deployed to support feedback-based tutoring approaches are able to identify the student's intention, i.e. the solution strategy implemented in the student solution. Third, most reviewed tutoring approaches only support individual learning. In order to fill this research gap, we propose an approach to pair learning which supports two students who solve a programming problem face-to-face.

Keywords: Computer-supported collaborative learning, Intelligent Tutoring Systems, programming, pair learning.

1 Introduction

For programming as a Computer Science subject, researchers have found that learning cannot only be achieved through memorizing facts or programming concepts, rather students need to apply the learned concepts to solve programming problems [1]. That is, researchers suggest focusing on constructivistic learning/teaching approaches for Computer Science courses, because the goal is to develop and to master programming skills. For this purpose, a variety of AI-supported tutoring approaches have been developed in order to help beginner students learn programming.

How can AI techniques be deployed to engage students in the process of learning, and can students play an active role in these settings? That is the motivation of this paper. For this purpose, we have reviewed several AI-supported tutoring approaches for the domain of programming: example-based, simulation-based, collaboration-based, dialogue-based, program analysis-based, and feedback-based approaches. All the tutoring approaches under review in this paper have been evaluated in empirical studies with students.

N.T. Nguyen, T. van Do, and H.A. Thi (Eds.): *ICCSAMA 2013*, SCI 479, pp. 267–279.
DOI: 10.1007/978-3-319-00293-4_20

2 AI-Supported Tutoring Approaches

2.1 Example-Based Approaches

One popular approach to teaching programming is explaining example problems and solutions and then asking students to solve similar new problems. This way, students transfer learned ways of solving problems to new problems of the same type. Yudelson and Brusilovsky [2] applied this tutoring approach for building learning systems for programming. NavEx [2] is a web-based tool for exploring programming examples. Each example is annotated with textual explanations for each important line. Thus, students can open the explanation line-by-line, or go straight to the lines which are difficult to understand, or display explanations of several lines at the same time. This system has several benefits compared to normal textbooks: 1) the code of each example can be shown as a block, instead of being dissected by comments as usually found in textbooks; 2) explanations can be accessed in different ways in accordance with the student's need; and 3) students can learn with examples in an exploratory manner, instead of reading examples passively. NavEx provides adaptive navigation support to access a large set of programming examples, that is, for each individual student only examples which correspond to the knowledge level of that student are recommended. This is the feature which requires the contribution of AI techniques. The adaptive navigation support is based on a *concept-based mechanism*. Applying this technique, first, a list of programming concepts is identified in each example and is separated into *pre-requisite concepts* and *outcome concepts*. The process of separation is supported by the structure of a specific course, which is defined by having the teacher assign examples to the ordered sequence of lectures. This process assumes that for the first lecture, all associated examples have only outcome concepts. The algorithm advances through lectures sequentially until all example concepts are effectively covered. The pre-requisites and the outcome concepts of each example and the current state of the individual user model determine which example should be recommended next to the student. The student's degree of engagement within an example is measured by counting the number of clicks on annotated lines of code.

Another type of example-based learning is the completion learning strategy. Learning tools present a programming problem and a solution template to be filled in. Gegg-Harrison [3] developed the tutoring system ADAPT for Prolog applying this tutoring approach. ADAPT exploits the existence of Prolog schemata which represent solutions for a class of problems. The tutor first introduces students with several solution examples and shows significant components of the schema underlying these solutions. After that, the tutor asks students to solve a similar problem given a solution template with slots to be filled. If the student is stuck and needs help, then the tutor breaks the template down into its recursive functional components and explains the role of each component. After explaining each of these components, the student has a chance to complete them or she can ask for further help. If the student still needs help, the tutor provides her with a more specific template, i.e., a template which is already filled with

a partial solution. If the student is still unable to solve the problem, then the complete solution is given and explained. The AI technique which supports this tutoring approach is using a library of Prolog schemata. The purpose is giving feedback to student solutions. A Prolog schema represents the common structure for a class of programs. Prolog schemata are used in ADAPT in order to diagnose errors in students' solutions. The process of diagnosis consists of two phases. First, ADAPT tries to recognize the algorithm underlying the student solution. It uses the normal form program for each schema in the schema library to generate a class of representative implementations and then transforms the most appropriate implementation into a structure that best matches the student solution. In the second phase, ADAPT tries to explain mismatches between the transformed normal form program produced in the first phase and the student solution. For this purpose, ADAPT uses a hierarchical bug library to classify bugs found in incorrect programs. If there are no mismatches, then the student solution is correct.

Chang and colleagues [4] followed a similar completion strategy. In addition to the fill-in-the-blank approach proposed in ADAPT, they also proposed modification and extension tasks. Modification tasks are to rewrite a program to improve it, to simplify it, or to use another program to perform the same function. Extension tasks include adding new criteria or new commands to original questions (for each question, several exercises are proposed), and students are asked to rewrite a program by adding or removing some aspects. In this system, the AI technique has been deployed to give feedback to a student solution. Here, program templates are used. For each problem (from which several completion, extension, or modification tasks are generated), a model program is defined by several templates which represent required semantic components of that model program. Each template is defined as a representation of a basic programming concept. The basic programming concepts are those which the system wants the students to learn working with the problem. In order to understand the student solution, pre-specified templates for each problem are expanded into source code and thus, the model program is generated. The generated model program serves to create exercises (completion, modification, and extension tasks) and to evaluate the student solution. Mismatches between the model program and the student solution are considered as errors and feedback is returned to students.

In this subsection we have introduced three AI techniques. While the concept-based mechanism has been used for providing adaptive navigation support, schemata-based and template-based techniques have been developed to provide feedback to student's solutions. The schemata-based technique is really useful if a schema can represent a broad class of programs which can be considered solutions for a class of programming problems. This is fulfilled for Prolog, a declarative programming language. However, for imperative programming languages, schemata can hardly be determined. Instead, templates which represent a basic programming concept can be used as proposed by Chang and colleagues [4].

2.2 Simulation-Based Approaches

The simulation-based teaching approach deals with the problem that a program has a dynamic nature (i.e., the state of variables is changing at run time) and students can not realize how a piece of program code works. Researchers have suggested visualizing the process underlying an algorithm. One way to lower the abstraction of programs is using micro worlds which are represented by concrete entities, e.g., turtles using NetLogo[1]. The student can program the movements of entities and control their behavior. Several environments have been developed following this approach, e.g. Alice[2] and Scratch[3]. The goal is to make program flows more visible. Through such simulation-based environments, students can develop their basic programming skills in order to later use them within more complex programming environments.

Recently, other simulation-based approaches have been developed to make visualizations more helpful: engaging visualization, explanatory visualization and adaptive visualization. Engaging visualization emphasizes the active role of student in learning, instead of just observing a teacher prepared animations. Explanatory visualization augments visual representations with natural language explanations that can help students understand simulation. Adaptive visualization adapts the details of visual representations to the difficulties of underlying concepts and to the knowledge level of each student. Several simulation tools have been developed applying the first two approaches in order to engage students and provide them with explanatory feedback: SICAS [5], OOP-AMIN [6], and PESEN [7]. SICAS and OOP-AMIN were developed to support learning of basic procedural programming concepts (e.g., selection and repetition) and object-oriented concepts (class, object or inheritance), respectively. PESEN is another simulation system which is intended to support weak students. Here, the goal is learning basic concepts of sequential, conditional and repetition execution which are frequently used in computer programs. These systems focus on the design and implementation of algorithms. Algorithms are designed by using an iconic environment where the student builds a flowchart that represents a solution. After a solution to a problem has been created, the algorithm is simulated and the student observes if the algorithm works as expected. The simulator can be used to detect and correct errors. While the tasks for students using SICAS and OOP-AMIN include designing and implementing an algorithm, PESEN just asks the student to program a very simple algorithm which consists of movements of elementary geometrical shapes using simple commands. The AI technique supporting these systems for detecting and correcting errors is relatively simple. Each programming problem is composed of a statement that describes it, one or more algorithms that solve it, and a solution with some test cases. The existence of several algorithms for the same problem serves to provide different ways of reasoning about the solution strategy chosen by the student. The test cases allow a student to verify if the designed algorithm really solves

[1] `http://ccl.northwestern.edu/netlogo`
[2] `http://www.alice.org`
[3] `http://scratch.mit.edu`

the problem. Both the existence of alternative algorithms and test cases allow the simulation system to diagnose errors in the student solution.

Brusilovsky and Spring [8] developed WADEin to allow students to explore the process of expression evaluation. The goal of WADEin is to provide adaptive and engaging visualization. The student can work with the system in two different modes. In the exploration mode, the user can type in expressions or select them from a menu of expressions. Then, the system starts visualizing the execution of each operation. The student can observe the process of evaluating a C expression step-by-step. The evaluation mode provides another way to engage students and to evaluate their knowledge. The student works with the new expression by indicating the order of execution of the operators in the expression. After that, the system corrects errors, shows the correct order of execution, and starts evaluating the expression. Here, the contribution of AI is very little. What the system has to do is storing the operators of an expression and possible reformulations a priori. Using this information, the order of operators in an expression submitted by the student is evaluated. In addition, the value of expressions can be evaluated "on-the-fly".

The AI techniques which have have been deployed to support the simulation-based tutoring approaches introduced in this subsection have been used for detecting errors in student solutions in order to provide feedback.

2.3 Dialogue-Based Approaches

One way of coaching students is to communicate with them in form of mixed-initiative dialogues. That is, both the tutor and the student are able to initiate a question. Lane [9] developed PROPL, a tutor which helps students build a natural-language style pseudocode solution to a given problem. The system initiates four types of questions: 1) identifying a programming goal, 2) describing a schema for attaining this goal, 3) suggesting pseudocode steps that achieve the goal, and 4) placing the steps within the pseudocode. Through conversations, the system tries to remediate student's errors and misconceptions. If the student's answer is not ideal (i.e., it can not be understood or interpreted as correct by the system), sub-dialogues are initiated with the goal of soliciting a better answer. The sub-dialogues will, for example, try to refine vague answers, ask students to complete incomplete answers, or to redirect to concepts of greater relevance. The AI contribution of PROPL is the ability of understanding student's answers and communicating with students using natural language. For this purpose, PROPL has a knowledge source which is a library of Knowledge Construction Dialogues (KDCs) representing directed lines of tutorial reasoning. They consist of a sequence of tutorial goals, each realized as a question, and sets of expected answers to those questions. The KCD author is responsible for creating both the content of questions and the forms of utterances in the expected answer lists. Each answer is either associated with another KCD that performs remediation or is classified as a correct response. KCDs therefore have a hierarchical structure and follow a recursive, finite-state based approach to dialogue management.

2.4 Program Analysis-Based Approaches

In contrast to other tutoring approaches which request students to synthesize a program or an algorithm, the program analysis-based approach intends to support students learn by analyzing a program.

Kumar [10] developed intelligent tutoring systems to help students learn the C++ programming language by analyzing and debugging C++ code segments. These systems focus on tutoring programming constructs rather than on the entire programming. Semantic and run-time errors in C++ programs are the objectives of tutoring. The problem-solving tasks addressed by these systems include: evaluating expressions step-by-step, predicting the result of a program line by line, and identifying buggy code in a program. For instance, for a debugging task, a tutor asks the student to identify a bug and to explain why the code is erroneous. If the student does not solve the problem correctly, the tutor provides an explanation of the step-by-step execution of the program. The AI technique which has been deployed in these systems is the model-based reasoning approach. In model-based reasoning, a model of a C++ domain consists of the structure (the objects in the language) and behavior of the domain (the mechanisms of interaction). This model is used to simulate the correct behavior of an artifact in the domain, e.g., the expected behavior of C++ pointers. The correct behavior generated from the model is compared with the behavior predicted by the student for that artifact. The discrepancies between these two behaviors are used to hypothesize structural discrepancies in the mental model of the student which are used to generate feedback for the student.

2.5 Feedback-Based Approaches

Feedback on student solutions can be used in a specific tutoring approach, e.g., ADAPT [3] is able to provide feedback during example-based tutoring. There exists a variety of learning systems for programming which make use of feedback on student solutions as the only means for tutoring. Systems of this class assume that feedback would be useful to help students solve programming problems, and thus programming skills can be improved. To be able to analyze student solutions and give feedback, learning systems for programming require a domain model and appropriate error diagnosis techniques. Here, we discuss the AI techniques applied in learning systems which have shown to be effective in helping students: model-tracing, machine learning, libraries of plans and bugs, and a weighted constraint-based model.

The core of a *model-tracing* tutoring system is the *cognitive model* which consists of "ideal rules" which represent correct problem-solving steps of an expert and "buggy rules" which represent typical erroneous behaviors of students. When the student inputs a solution, the system monitors her action and generates a set of correct and buggy solution paths. Whenever a student's action can be recognized as belonging to a correct path, the student is allowed to go on. If the student's action deviates from the correct solution paths, the system generates instructions to guide the student towards a correct solution. Applying

this approach, Anderson and Reiser [11] developed an intelligent tutoring system for LISP. This tutor presents to the student a problem description containing highlighted identifiers for functions and parameters which have to be used in the implementation. To solve a programming problem, the student is provided with a structured editor which guides the student through a sequence of templates to be filled in. Applying the same approach, Sykes [12] developed JITS, an intelligent tutoring system for Java. JITS is intended to "intelligently" examine the student's submitted code and determines appropriate feedback with respect to the grammar of Java. For this purpose, production rules have been used to model the grammar of Java and to correct compiler errors. Since this system does not check semantic requirements which are specific for each programming problem, it does not require representations of problem-specific information.

Since the task of modeling "buggy rules" is laborious and time-consuming, Suarez and Sison applied a multi-strategy machine learning approach to automatically construct a library of Java errors which novice programmers often make [13]. Using this bug library, the authors developed a system called JavaBugs which is able to examine a small Java program. The process of identifying errors in the student solution takes place in two phases. First, the system uses Beam search to compare the student solution to a set of reference programs and to identify the most similar correct program to the student's solution. The solution strategy intended by the student to solve a programming problem is considered the student's intention. After the student's intention has been identified, differences between the reference program and the student solution are extracted. In the second phase, misconception definitions are learned based on the discrepancies identified in the first phase. These misconceptions are used to update the error library and to return appropriate feedback to the student.

Another class of systems uses a library of programming plans or bugs for modeling domain knowledge and diagnosing errors. Representatives for this class include, for example, PROUST [14], ELM-ART [15], and APROPOS2 [16]. The process of error diagnosis of these systems builds on the same principles: 1) modeling the domain knowledge using programming plans or algorithms and buggy rules, 2) identifying the student's intention, and 3) detecting errors using buggy rules. In PROUST, a tutoring system for Pascal, each programming problem is represented internally by a set of programming goals and data objects. A programming goal represents the requirements which must be satisfied, while data objects are manipulated by the program. A programming goal can be realized by alternative programming plans. A programming plan represents a way to implement a corresponding programming goal. In contrast to PROUST, APROPOS2 (a tutoring system for Prolog) uses the concept of algorithms to represent different ways for solving a Prolog problem instead of programming plans, because Prolog does not have keywords (the only keyword is ":-" which separates the head and the body of a Prolog clause) to define programming plans and to anchor program analysis like in the case of Pascal. In addition to modeling domain knowledge using programming plans or algorithms, common bugs are collected from empirical studies and represented as *buggy rules* or *buggy clauses*. While

PROUST contains only buggy rules, ELM-ART adds two more types: good and sub-optimal rules which are used to comment on good programs and less efficient programs (with respect to computing resources or time), respectively. After the domain model has been specified by means of programming plans (or algorithms) and buggy rules, systems of this class perform error diagnosis in two steps: identifying the student's intention and detecting errors. The process of identifying student's intention starts by using pre-specified programming plans (or algorithms) to generate a variety of different ways of implementations for a programming goal. Then, it continues to derive hypotheses about the programming plan (or algorithm) the student may have used to satisfy each goal. If the hypothesized programming plan (or algorithm) matches the student program, the goal is implemented correctly. Otherwise, the system looks up the database of buggy rules to explain the discrepancies between the student solution and the hypothesized programming plan (or algorithm).

Instead of using programmings/algorithms and buggy rules for modeling a domain, Le and Menzel [17] used a semantic table and a set of weighted constraints to develop a weighted constraint-based model for a tutoring system (INCOM) for Prolog. The semantic table is used to model alternative solution strategies and to represent generalized components required for each solution strategy. Constraints are used to check the semantic correctness of a student solution with respect to the requirements specified in the semantic table and to examine general well-formedness conditions for a solution. Constraints' weight values are primarily used to control the process of error diagnosis. In their approach, the process of diagnosing errors in a student solution consists of two steps which take place on two levels: hypotheses generation and hypotheses evaluation. First, on the *strategy level*, the system generates hypotheses about the student's intention by iteratively matching the student solution against multiple solution strategies specified in a semantic table. After each solution strategy has been matched, on the *implementation variant level* the process generates hypotheses about the student's implementation variant by matching components of the student solution against corresponding components of the selected solution strategy. The generated hypotheses are evaluated with respect to their plausibility by aggregating the weight value of violated constraints. As a consequence, the most plausible solution strategy intended by the student is determined.

All the AI techniques presented in this subsection share one common point: they are able to identify the student's intention first, then detect errors in student solutions. The model tracing technique diagnoses the student's intention by relating each step of the student's problem-solving with generated correct and erroneous solution paths. The approach in JavaBugs uses several reference programs to hypothesize the student's implemented strategy. Other systems use a library of programming plans/algorithms to recognize the student's intention. The weighted constraint-based model uses a semantic table, which is specified with different solution strategies, to hypothesize a solution strategy underlying a student solution.

2.6 Collaboration-Based Approaches

In comparison to other tutoring approaches (presented in previous subsections) which support individual learning, a collaboration-based tutoring approach assumes that learning can be supported through collaboration between several students.

HABIPRO [18] is a collaborative environment that supports students to develop good programming habits. The system provides four types of exercises: 1) finding a mistake in a program, 2) putting a program in the correct order, 3) predicting the result, and 4) completing a program. When a student group proposes an incorrect solution, the system shows four types of help: 1) giving ideas to the student how they can solve the problem, 2) showing and explaining the solution, 3) showing a similar example of the problem and its solution, and 4) displaying only the solution. The system has a simulated agent that is able to intervene in students' learning, to avoid off-topic situations, and to avoid passive students. This system exploits several AI techniques. First, it is able to guide students in conversations using natural language. Second, it uses knowledge representation techniques to model a group of students and an interaction model. The interaction model defines a set of patterns which represent possible characteristics of group interaction (e.g., the group prefers to look at the solution without seeing an explanation). During the collaboration, the simulated agent uses the group model to compare the current state of interaction to these patterns, and proposes actions such as withholding solutions until the students have tried to solve the problem. Third, the system exploits information from a student model which includes features that enable the system to reason about the student's collaborative behavior: e.g., frequency of interaction, type of interaction, level of knowledge, personal beliefs, and mistakes (individual misconceptions). The simulated agent makes use of information from both the student models and the group model to decide when and how to intervene in the collaboration.

3 Discussion

We summarize the tutoring approaches and AI techniques presented in the previous section in Table 1. From this table we can learn three things. First, we have found that most learning systems have been developed applying the feedback-based tutoring approach in comparison to other categories of tutoring. This does not mean that the feedback-based tutoring approach is more effective than others. Rather, the feedback-based tutoring approach can be regarded as the first attempt to building intelligent learning systems. Indeed, we can deploy any AI technique for building feedback-based tutoring systems to support other categories of tutoring as shown in case of ADAPT. That is, feedback on student solutions can be used as a means within different tutoring scenarios. Second, from the last column of the table we can identify that AI techniques, which have been deployed in different tutoring approaches, serve three purposes: to support adaptive navigation, to analyze student solutions, and to enable a conversation with students. In addition, we can also recognize that AI techniques developed

for feedback-based tutoring approaches are able to identify the student's intention underlying the student solution. This is necessary to generate feedback on student solutions, because there are often different ways for solving a programming problem. If feedback returned to the student does not correspond to her intention, feedback might be misleading. Hence, the first step of diagnosing errors in a student solution is identifying the student's intention. The third lesson we have learned from the survey above is that most systems support individual learning. HABIPRO is the only system (to our best knowledge) which has been designed to support collaborative programming activities.

Table 1. AI-supported Tutoring Approaches

Type	**System**	**AI Support**	
		Techniques	**Function**
Example	NavEx	concept-based mechanism	adaptive navigation
	ADAPT	Prolog schemata	solution analysis
	Chang's	template-based	solution analysis
Simulation	SICAS	test cases	solution analysis
	OOP-AMIN	test cases	solution analysis
	PESEN	test cases	solution analysis
	WADEin	expression evaluation	solution analysis
Dialogue	PROPL	natural language processing	conversation
Program Analysis	Kumar's	model-based reasoning	solution analysis
Feedback	LISP-Tutor	model-tracing	solution analysis
	JITS	model-tracing	solution analysis
	JavaBugs	machine learning	solution analysis
	PROUST	programming plans & buggy rules	solution analysis
	APROPOS2	algorithms & buggy rules	solution analysis
	ELM-ART	algorithms & buggy, good, sub-optimal rules	solution analysis
	INCOM	semantic table & weighted constraints	solution analysis
Collaboration	HABIPRO	natural language processing	conversation

4 Intelligent Peer Learning Support

One of the lessons we have learned from the survey above is that most tutoring approaches have been developed to support individual learning. Yet, in the context of solving programming problems, collaboration and peer learning have been shown to be beneficial [19], [20], [21].

From a course "Foundations of programming" in which primarily the programming language Java is used, we have collected thousands of student solutions over three winter semesters (2009-2012). During the semester, every two weeks students were given an exercise sheet. For each task, students could achieve maximal 10 points. Students were asked to submit their solutions in pairs or alone using the GATE submission system [22]. By comparing the score of pair submissions

with the score of single submissions we have learned that the paired students achieved higher scores than single submissions: there were 2485 submissions in total (1522 single, 963 pair submissions); the average score of single submissions and of pair submissions were 6.59 and 6.91 points, respectively (p=0.008). This suggests that pair submissions correlate with higher achievement. This can be explained by the reason that solving programming problems in pairs, students could contribute complementarily, and thus a solution for a given problem could be found easier by a pair of students than by individual learners.

Identifying the need of supporting solve programming problems in pairs, in this paper, we propose an approach for intelligent pair learning support which can take place when two students work together. Researchers agreed that the programming process consists of the following phases: 1) problem understanding, 2) planning, 3) design, and 4) coding (implementation in a specific programming language). After reviewing literature of learning systems for programming, to our best knowledge, only SOLVEIT [23] is able to support all four phases. However, SOLVEIT does not provide feedback to the student. We propose to support pair learning during these four phases. In the first phase, the system poses questions to ask a pair of students to analyze a problem statement, e.g., "Can you identify given information in the problem statement?" A list of keywords which represent given information in a problem statement can be associated with each problem. Using this list, the system can give hints to the students. In the planning phase, the system asks a pair of students to plan a solution strategy, e.g., "Which strategy can be used to solve this problem?" or "What kind of loop can be used to solve this problem?" Similarly to the first phase, to each problem, a list of alternative solution strategies can be stored which are used to propose to the student. PROPL [9] is able to support this phase applying the dialogue-based tutoring approach. In the design phase, the system asks the pair of students to transform the chosen solution strategy into an algorithm which can be represented, e.g., in activity diagrams. Required components of an activity diagram can be modeled using one of the AI techniques presented in the previous section in order to check the solution of the pair of students. In the fourth phase, the system can ask the students to transform the designed algorithm into code of a specific programming language, e.g., "How can we code the input action which reads the value of a variable?" Again, we can apply one of the AI techniques introduced above to examine the semantic correctness of the student solution.

We need to identify and to combine strengths and weaknesses of individual learners in order to strengthen the collaboration between students and to help learners overcome weaknesses. For this purpose, models for individual learners need to be developed. User models can be used to parametrize group learning [24] as well as to build learning groups intelligently [25].

In order to build learner models, a system which is intended to support pair learning needs to distinguish contributions between different learners. Contrary to recognizing individual learners using unique identifiers (e.g., user name and password), pair learning systems should be able to distinguish learners (who are working in collaboration) by identifying contributions of each peer in order to

associate individual progress of a learner. Modeling characteristics of learners and groups could then be used to adapt a pair learning supporting system to individual and group needs. For example, recognizing learners' inactivity could lead to a stimulation by the system. How can we realize a pair learning supporting system technically? We might apply visual recognition techniques in order to distinguish different learners or exploit speech recognition techniques in order to recognize activity and to assign contributions to learners.

5 Conclusion and Future Work

We have reviewed various tutoring approaches which are supported by AI techniques for the domain of programming. We have learned that most existing learning systems aim at analyzing student's solutions and providing feedback which serve as an important means for improving programming skills. We identified that AI techniques, which have been deployed in different tutoring approaches, serve three purposes: to support adaptive navigation, to analyze student solutions, and to enable a conversation with students. Especially, AI techniques which have been applied for feedback-based tutoring approaches, are able to identify the student's intention before detecting errors. The third lesson we have learned is that most systems support individual learning. This is a research gap, because collaborative learning is beneficial for solving programming problems. Hence, we propose in this paper an approach for pair learning which is intended to coach students along typical phases of programming and is supported by a conversational approach. In the future, we plan to implement this approach using a conversational agent.

References

1. Radosevic, D., Lovrencic, A., Orehovacki, T.: New Approaches and Tools in Teaching Programming. In: Central European Conference on Information and Intelligent Systems (2009)
2. Yudelson, M., Brusilovsky, P.: NavEx: Providing Navigation Support for Adaptive Browsing of Annotated Code Examples. In: The 12th International Conference on AI in Education, pp. 710–717. IOS Press (2005)
3. Gegg-Harrison, T.S.: Exploiting Program Schemata in a Prolog Tutoring System. Phd Thesis, Duke University, Durham, North Carolina 27708-0129 (1993)
4. Chang, K.E., Chiao, B.C., Chen, S.W., Hsiao, R.S.: A Programming Learning System for Beginners - A Completion Strategy Approach. Journal IEEE Transactions on Education 43(2), 211–220 (1997)
5. Gomes, A., Mendes, A.J.: SICAS: Interactive system for algorithm development and simulation. In: Computers and Education - Towards an Interconnected Society, pp. 159–166 (2001)
6. Esteves, M., Mendes, A.: A simulation tool to help learning of object oriented programming basics. In: FIE 34th Annual on Frontiers in Edu., vol. 2, pp. F4C7-12 (2004)

7. Mendes, A., Jordanova, N., Marcelino, M.: PENSEN - A visual programming environment to support initial programming learning. In: International Conference on Computer Systems and Technologies - CompSysTech 2005 (2005)
8. Brusilovsky, P., Spring, M.: Adaptive, Engaging, and Explanatory Visualization in a C Programming Course. In: World Conference on Educational Multimedia, Hypermedia and Telecommunications (ED-MEDIA), pp. 21–26 (2004)
9. Lane, H.C., Vanlehn, K.: Teaching the tacit knowledge of programming to novices with natural language tutoring. J. Computer Science Education 15, 183–201 (2005)
10. Kumar, A.N.: Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors. Journal Technology, Instruction, Cognition and Learning 4 (2006)
11. Anderson, J.R., Reiser, B.: The Lisp Tutor. Journal Byte 10, 159–175 (1985)
12. Sykes, E.R.: Qualitative Evaluation of the Java Intelligent Tutoring System. Journal of Systemics, Cyber (2006)
13. Suarez, M., Sison, R.C.: Automatic construction of a bug library for object-oriented novice java programmer errors. In: Woolf, B.P., Aïmeur, E., Nkambou, R., Lajoie, S. (eds.) ITS 2008. LNCS, vol. 5091, pp. 184–193. Springer, Heidelberg (2008)
14. Johnson, W.L.: Understanding and debugging novice programs. Journal Artificial Intelligence 42, 51–97 (1990)
15. Brusilovsky, P., Schwarz, E.W., Weber, G.: ELM-ART: An Intelligent Tutoring System on World Wide Web. In: Lesgold, A.M., Frasson, C., Gauthier, G. (eds.) ITS 1996. LNCS, vol. 1086, pp. 261–269. Springer, Heidelberg (1996)
16. Looi, C.-K.: Automatic Debugging of Prolog Programs in a Prolog Intelligent Tutoring System. Journal Instructional Science 20, 215–263 (1991)
17. Le, N.T., Menzel, W.: Using Weighted Constraints to Diagnose Errors in Logic Programming - The case of an ill-defined domain. J. of AI in Edu. 19, 381–400 (2009)
18. Vizcaíno, A.: A Simulated Student Can Improve Collaborative Learning. Journal AI in Education 15, 3–40 (2005)
19. Van Gorp, M.J., Grissom, S.: An Empirical Evaluation of Using Constructive Classroom Activities to Teach Introductory Programming. J. Computer Science Education 11(3), 247–260 (2001)
20. Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., Balik, S.: Improving the CS1 experience with pair programming. In: The SIGCSE Technical Symposium on CS Education, pp. 359–362. ACM (2003)
21. Wills, C.E., Deremer, D., McCauley, R.A., Null, L.: Studying the use of peer learning in the introductory computer science curriculum. J. Computer Science Education 9(2), 71–88 (1999)
22. Strickroth, S., Olivier, H., Pinkwart, N.: Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben? In: GI LNI, vol. P-188, pp. 115–126. Köllen Verlag (2010)
23. Deek, F.P., Ho, K.-W., Ramadhan, H.: A critical analysis and evaluation of web-based environments for program development. J. Internet and Higher Education 3, 223–269 (2000)
24. Hoppe, H.U.: The use of multiple student modeling to parameterize group learning. In: Int. Conference on AIED (1995)
25. Muehlenbrock, M.: Formation of Learning Groups by using Learner Profiles and Context Information. In: Int. Conference on AI in Education, pp. 507–514 (2005)