AIED 2013 Workshops Proceedings
Volume 9

The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)

Workshop Co-Chairs:

Nguyen-Thinh Le[1]
Kristy Elizabeth Boyer[2]
Beenish Chaudhry[3]
Barbara Di Eugenio[4]
Sharon I-Han Hsiao[5]
Leigh Ann Sudol-DeLyser[6]

[1]*Clausthal University of Technology, Germany*
[2]*North Carolina State University, USA*
[3]*Indiana University Bloomington, USA*
[4]*University of Illinois Chicago, USA*
[5]*Columbia University, USA*
[6]*New York University, USA*

https://sites.google.com/site/aiedcs2013/

Preface

The global economy increasingly depends upon Computer Science and Information Technology professionals to maintain and expand the infrastructure on which business, education, governments, and social networks rely. Demand is growing for a global workforce that is well versed and can easily adapt ever-increasing technology. For these reasons, there is increased recognition that computer science and informatics are becoming, and should become, part of a well-rounded education for every student. However, along with an increased number and diversity of students studying computing comes the need for more supported instruction and an expansion in pedagogical tools to be used with novices. The study of computer science often requires a large element of practice, often self-guided as homework or lab work. Practice as a significant component of the learning process calls for AI-supported tools to become an integral part of current course practices.

Designing and deploying AI techniques within computer science learning environments presents numerous challenges. First, computer science focuses largely on problem solving skills in a domain with an infinitely large problem space. Modeling possible problem solving strategies of experts and novices requires techniques that address many types of unique but correct solutions to problems. In addition, there is growing need to support affective and motivational aspects of computer science learning, to address widespread attrition of students from the discipline. AIED researchers are poised to make great strides in building intelligent, highly effective AI-supported learning environments and educational tools for computer science and information technology. Spurred by the growing need for intelligent learning environments that support computer science and information technology, this workshop will provide a timely opportunity to present emerging research results along these lines.

June, 2013
Nguyen-Thinh Le, Kristy Elizabeth Boyer, Beenish Chaudhry,
Barbara Di Eugenio, Sharon I-Han Hsiao, and Leigh Ann Sudol-DeLyser

Program Committee

Table of Contents

Sequential Patterns of Affective States of Novice Programmers

Nigel Bosch[1] and Sidney D'Mello[1,2]

Departments of Computer Science[1] and Psychology[2], University of Notre Dame
Notre Dame, IN 46556, USA
{pbosch1, sdmello}@nd.edu

Abstract. We explore the sequences of affective states that students experience during their first encounter with computer programming. We conducted a study where 29 students with no prior programming experience completed various programming exercises by entering, testing, and running code. Affect was measured using a retrospective affect judgment protocol in which participants annotated videos of their interaction immediately after the programming session. We examined sequences of affective states and found that the sequences Flow/Engagement ↔ Confusion and Confusion ↔ Frustration occurred more than expected by chance, which aligns with a theoretical model of affect during complex learning. The likelihoods of some of these frequent transitions varied with the availability of instructional scaffolds and correlated with performance outcomes in both expected but also surprising ways. We discuss the implications and potential applications of our findings for affect-sensitive computer programming education systems.

Keywords: affect, computer programming, computerized learning, sequences

1 Introduction

Given the unusually high attrition rate of computer science (CS) majors in the U.S. [1], efforts have been made to increase the supply of competent computer programmers through computerized education, rather than relying on traditional classroom education. Some research in this area focuses on the behaviors of computer programming students in order to provide more effective computerized tutoring and personalized feedback [2]. In fact, over 25 years ago researchers were exploring the possibility of exploiting artificial intelligence techniques to provide customized tutoring experiences for students in the LISP language [3]. This trend has continued, as evidenced by a number of intelligent tutoring systems (ITSs) that offer adaptive support in the domain of computer programming (e.g. [4–6]).

One somewhat neglected area in the field is the systematic monitoring of the affective states that arise over the course of learning computer programming and the impact of these states on retention and learning outcomes. The focus on affect is motivated by considerable research which has indicated that affect continually operates throughout a learning episode and different affective states differentially impact per-

formance outcomes [7]. Some initial work has found that affective states, such as confusion and frustration, occur frequently during computer programming sessions [8, 9] and these states are correlated with student performance [10].

The realization of the important role of affect in learning has led some researchers to develop learning environments that adaptively respond to affective states in addition to cognitive states (see [11] for a review). Previous research has shown that affect sensitivity can make a measurable improvement on the performance of students in other domains such as computer literacy and conceptual physics [12, 13]. Applying this approach to computer programming education by identifying the affective states of students could yield similarly effective results, leading to more effective systems.

Before it will be possible for an affect-sensitive intelligent tutoring system to be successful in the computer programming domain, more research is needed to determine at a fine-grained level what affective states students experience and how affect interacts and arises from the students' behaviors. Previous work has collected affective data at a somewhat coarse-grained level in a variety of computer programming education contexts. [10] collected affect using two human observers, and were able to draw conclusions about what affective states led to improved performance on a computer programming exam. [14] induced affect in experienced programmers using video stimuli, and found that speed and performance on a coding and debugging test could be increased with high-arousal video clips.

In our previous work [15], we examined the affect of 29 novice programmers at 20-second intervals as they solved introductory exercises on fundamentals of computer programing. We found that flow/engagement, confusion, frustration, and boredom dominated the affect of novice programmers when they were not in a neutral state. We found that boredom and confusion were negatively correlated with performance, while the flow/engagement state positively predicted performance. This paper continues this line of research by exploring transitions between affective states.

Specifically, we test a theoretical model on affect dynamics that has been proposed for a range of complex learning tasks [16]. This theoretical model (Fig. 1) posits four affective states that are crucial to the learning process: flow/engagement, confusion, frustration, and boredom. The model predicts an important interplay between confusion and flow/engagement, whereby a learner in the state of flow/engagement may encounter an impasse and become confused. From the state of confusion, if an impasse is resolved the learner will return to the state of flow/engagement, having learned more deeply. This is in line with other research which has shown that confusion helps learning when impasses are resolved [17]. On the other hand, when the source of the confusion is never resolved, the learner will become frustrated, and eventually bored if the frustration persists.

Researchers have found some support for this theoretical model of affective dynamics in learning contexts such as learning computer literacy with AutoTutor [16], unsupervised academic research [18], and narrative learning environments [19]. We expect the theoretical model to apply to computer programming as well.

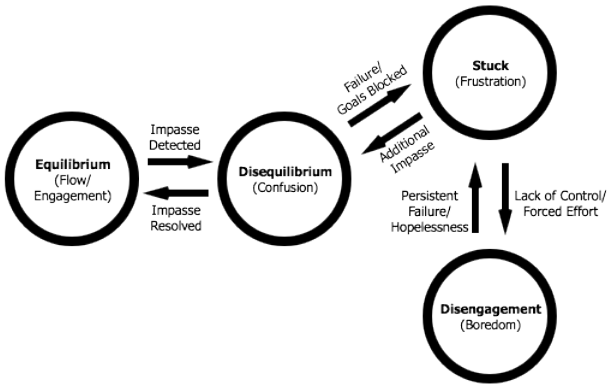


Fig. 1. Theoretical model of affect transitions.

We posit that encountering unfamiliar concepts, syntax and runtime errors, and other impasses can cause confusion in a computer programmer. When those impasses are resolved, the programmer will be better equipped to anticipate and handle such impasses in the future, having learned something. Alternatively, if the impasses persist, programmers may become frustrated and eventually disengage, entering a state of boredom in which it is difficult to learn.

To explore the applicability of this model to the domain of novice computer programming, this paper focuses on answering the following research questions: 1) what transitions occur frequently between affective states? 2) how are instructional scaffolds related to affect transitions? and 3) are affective transitions predictive of learning outcomes? These questions were investigated by analyzing affect data collected in a previous study [15] where 29 novice programmers learned the basics of computer programming over the course of a 40-minute learning session with a computerized environment, as described in more detail below.

2 Methods

Participants were 29 undergraduate students with no prior programming experience. They were asked to complete exercises in a computerized learning environment designed to teach programming fundamentals in the Python language. Participants solved exercises by entering, testing, and submitting code through a graphical user interface. Submissions were judged automatically by testing predetermined input and output values, whereupon participants received minimal feedback about the correctness of their submission. For correct submissions they would move on to the next exercise, but otherwise would be required to continue working on the same exercise.

The exercises in this study were designed in such a way that participants would likely encounter some unknown, potentially confusing concepts in each exercise. In this manner we elicited emotional reactions similar to real-world situations where computer programmers face problems with no predefined solutions and must experiment and explore to find correct solutions. Participants could use hints, which would gradually explain these impasses and allow participants to move on in order to pre-

vent becoming permanently stuck on an exercise. However, participants were free to use or ignore hints as they pleased.

Exercises were divided into two main phases. In the first phase (scaffolding), participants had hints and other explanations available and worked on gradually more difficult exercises for 25 minutes. Performance in the scaffolding phase was determined by granting one point for each exercise solved and one point for each hint that was not used in the solved exercises. Following that was the second phase (fadeout), in which they had 5 minutes to work on a debugging exercise, and 10 minutes to work on another programming exercise with no hints. In this study we will not consider the debugging exercise because it was only 5 minutes long. Performance was determined by two human judges who examined each participant's code, determined the number of lines matching lines in the correct solution, and resolved their discrepancies.

Finally, we used a retrospective affect judgment protocol to assess student affect after they completed the 40-minute programming session [20]. Participants viewed video of their face and on-screen activity side by side, and were polled at various points to report the affective state they had felt most at the polling point. The temporal locations for polling were chosen to correspond with interactions and periods of no activity such that each participant had 100 points at which to rate their affect, with a minimum of 20 seconds between each point. Participants provided judgments on 13 emotions, including basic emotions (anger, disgust, fear, sadness, surprise, happiness), learning-centered emotions (anxiety, boredom, frustration, flow/engagement, curiosity, confusion/uncertainty) and neutral (no apparent feeling).The most frequent affective states, reported in [15], were flow/engagement (23%), confusion (22%), frustration (14%), and boredom (12%), a finding that offers some initial support for the theoretical model discussed in the Introduction.

3 Results and Discussion

We used a previously developed transition likelihood metric to compute the likelihood of the occurrence of each transition relative to chance [21].

$$L(Current \rightarrow Next) = \frac{\Pr(Next|Current) - \Pr(Next)}{1 - \Pr(Next)} \quad (1)$$

This likelihood metric determines the conditional probability of a particular affective state (*next*), given the current affective state. The probability is then normalized to account for the overall likelihood of the *next* state occurring. If the affective transition occurs as expected by chance, the numerator is 0 and so likelihood is as well. Thus we can discover affective state transitions that occurred more ($L > 0$) or less ($L < 0$) frequently than expected by chance alone.

Before computing L scores we removed transitions that occurred from one state to the same state. For example, a sequence of affective states such as *confusion, frustration, frustration, boredom* would be reduced to *confusion, frustration, boredom*. This was done because our focus in this paper is on the transitions between different affective states, rather than on the persistence of each affective state [16, 18]. Furthermore,

although transition likelihoods between all 13 states (plus neutral) were computed, the present paper focuses on transitions between states specified in the theoretical model (boredom, confusion, flow/engagement, and frustration), which also happen to be the most frequent affective states.

What transitions occur frequently between affective states? We found the transitions that occurred significantly more than chance ($L = 0$) by computing affect transition likelihoods for individual participants and then comparing each likelihood to zero (chance) with a two-tailed one-sample *t-test.* Significant ($p < .05$) and marginally significant ($p < .10$) transitions are shown in Figure 2 and are aligned with the theoretical model on affect dynamics.

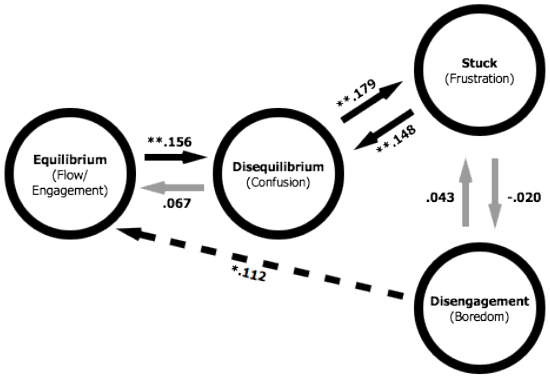


Fig 2. Frequently observed affective state transitions. Edge labels are mean likelihoods of affective state transitions. Grey arrows represent transitions that were predicted by the theoretical model but were not significant. The dashed arrow represents a transition that was marginally significant but not predicted. *$p < .10$, **$p < .05$

Three of the predicted transitions, Flow/Engagement → Confusion, Confusion → Frustration, and Frustration → Confusion, were significant and matched the theoretical model. Confusion → Flow/Engagement was in the expected direction and approached significance ($p = .108$), while Boredom → Frustration was in the expected direction but not significant. The Frustration → Boredom transition was not in the expected direction and was also not significant. Hence, with the exception of the Frustration ↔ Boredom links, there was support for four out of the six transitions espoused by the theoretical model. This suggests that the components of the model related to the experience of successful (Flow/Engagement ↔ Confusion) and unsuccessful (Confusion ↔ Frustration links) resolution of impasses were confirmed. Therefore, the present data provide partial support for the model.

The Boredom → Flow/Engagement transition, which occurred at marginally significant levels ($p = .091$), was not predicted by the theoretical model. It is possible that the nature of our computerized learning environment encouraged this transition more than expected. This might be due to the fast-paced nature of the learning session, which included 18 exercises and an in-depth programming task in a short 40-minute session. Furthermore, participants had some control over the learning environment in that they could use bottom-out hints to move to the next exercise instead of being forced to wallow in their boredom. The previous study that tested this model used a learning environment (AutoTutor) that did not provide any control over the learning activity, which might explain the presence of Frustration → Boredom (dis-

engaging from being stuck) and Boredom → Frustration (being frustrated due to forced effort) links in the earlier data [16].

How are instructional scaffolds related to affect transitions? To answer this question we looked at the differences between the scaffolding and fadeout phases of the study, as previously described. We discarded the first 5 minutes of the scaffolding phase to allow for a "warm-up" period during which participants were acclimating to the learning environment. We also discarded the 25 to 30 minutes portion, which was the debugging task in the fadeout phase. The debugging task was significantly different from the problem-solving nature of the coding portions, and so we excluded it from the current analysis to increase homogeneity. Differences between likelihoods of the five significant or marginally significant transitions from Figure 2 were investigated with paired samples *t*-test (see Table 1).

Table 1. Means and standard deviations (in parentheses) for common transitions in the scaffolding phase (5-25 minutes) and the coding portion of the fadeout phase (30-40 minutes).

Transition	Scaffolding	Fadeout Coding	N
Flow/Engagement → Confusion	**.115 (.308)	**.354 (.432)	20
Confusion → Flow/Engagement	.101(.241)	.029(.331)	27
Confusion → Frustration	.105 (.276)	.184 (.416)	27
Frustration → Confusion	.047 (.258)	.116 (.445)	21
Boredom → Flow/Engagement	.096 (.166)	.226 (.356)	14

*$p < .10$, **$p < .05$

The likelihood of participants transitioning from flow/engagement to confusion was significantly higher in the fadeout phase compared to the scaffolding phase. This may be attributed to the fact that participants have hints and explanations in the scaffolding phase, so in the event of a confusing impasse, a hint may be helpful in resolving the impasse, thereby allowing participants to return to a state of flow/engagement. With no such hints, confused participants may become more frustrated in the fadeout phase, as evidenced by a trend in this direction. This finding is as expected from the theoretical model, which states that confusion can lead to frustration when goals are blocked and the student has limited coping potential (e.g. being unable to progress on an exercise in this case).

Although not significant, there also appears to be an increase in the Boredom → Flow/Engagement affect transition in the fadeout phase. It is possible that too much readily available assistance prevents students from re-engaging on their own.

Are affective transitions predictive of learning outcomes? To determine what affective state transitions were linked to performance on the programming task, we correlated the likelihood of affect transitions with the performance metrics described in the Methods. In previous work we found correlations between performance and the proportions of affective states experienced by students [15]. Hence, when examining the correlations between affect transitions and performance, partial correlations were used to control for the proportions of the affective states in the transitions.

Table 2 lists correlations between frequent transitions and performance. These include correlations between affect transitions in the scaffolding phase with performance in the scaffolding phase (Scaffolding column) and transitions in the fadeout phase with performance in the fadeout coding phase (Fadeout Coding 1). We also correlated transitions in the scaffolding phase with performance in the fadeout coding phase (Fadeout Coding 2). This allows us to examine if affect transitions experienced during scaffolded learning were related to future performance when learning scaffolds were removed. Due to the small sample size, in addition to discussing significant correlations, we also consider non-significant correlations approaching 0.2 or larger to be meaningful because these might be significant with a bigger sample. These correlations are bolded in the table.

Table 2. Correlations between affect transitions and performance.

Transition	**Scaffolding**	**Fadeout Coding 1**	**Fadeout Coding 2**
Flow/Engagement → Confusion	.046	-.094	-.098
Confusion → Flow/Engagement	**-.274**	**-.256**	***-.365**
Confusion → Frustration	.114	****.499**	****.424**
Frustration → Confusion	***-.368**	.051	**-.275**
Boredom → Flow/Engagement	-.034	.050	-.063

*$p < .10$, **$p < .05$

The correlations were illuminating in a number of respects. The Confusion → Flow/Engagement transition correlated *negatively* with performance. This is contrary to the theoretical model which would predict a positive correlation to the extent that confused learners return to a state of flow/engagement by resolving troublesome impasses with effortful problem solving. It is possible that students who frequently experienced this transition were doing so by taking advantages of hints as opposed to resolving impasses on their own. This would explain the negative correlation between Confusion → Flow/Engagement and performance.

To investigate this possibility we correlated hint usage in the scaffolding phase with the Confusion → Flow/Engagement transition, controlling for the proportion of confusion and flow/engagement. The number of hints used in the scaffolding phase correlated positively, though not significantly, with the Confusion → Flow/Engagement transition in the scaffolding phase ($r = .297$) and the fadeout coding phase ($r = .282$). Additionally, hint usage correlated negatively with score in the scaffolding phase ($r = -.202$) and the fadeout coding phase ($r = -.506$). This indicates that students using hints tended to experience the Confusion → Flow/Engagement transition more (as expected) but this hindered rather than helped learning because students were not investing the cognitive effort to resolve impasses on their own.

Similarly, the correlation between Confusion → Frustration and performance is inconsistent with the theoretical model, which would predict a negative relationship between these variables. This unexpected correlation could also be explained on the basis of hint usage. Specifically, the number of hints used in the scaffolding phase

correlated negatively, though not significantly, with the Confusion → Frustration transition in the scaffolding phase (r = -.258) and the fadeout coding phase (r = -.171). This finding suggests that although hints can alleviate the Confusion → Frustration transition, learning improved when students are able to resolve impasses on their own, which is consistent with the theoretical model.

Finally, the correlation between Frustration → Confusion was in the expected direction. The Frustration → Confusion transition occurs when a student experience additional impasses while in the state of frustration. This transition is reflective of hopeless confusion, which is expected to be negatively correlated with performance, as revealed in the data.

4 General Discussion

Previous research has shown that some affective states are conducive to learning in the context of computer programming education while others hinder learning. Flow/engagement is correlated with higher performance, while confusion and boredom are correlated with poorer performance [10, 15]. Transitions between affective states are thus important because they provide insight into how students enter into an affective state. Affect-sensitive ITSs for computer programming may be able to use this information to better predict affect, intervening when appropriate to encourage the flow/engagement state and minimize the incidence of boredom and frustration.

We found that the presence or absence of instructional scaffolds were related the affect transitions experienced by students, especially the Flow/Engagement → Confusion transition. Our findings show that this transition is related to the presence of hints, a strategy which might be useful in future affect-sensitive ITS design for computer programming students. Similarly, we found that instructional scaffolds were related to the Boredom → Flow/Engagement transition, which is not part of the theoretical model. Future work on ITS design might also need to take into account this effect and moderate the availability of scaffolds to promote this affect transition.

The affect transitions that we found partially follow the predictions of the theoretical model. Impasses commonly arise in computer programming, particularly for novices, when they encounter learning situations with which they are unfamiliar. New programming language keywords, concepts, and error messages present students with impasses that must be resolved before the student will be able to continue. Unresolved impasses can lead to frustration and eventually boredom. The alignment between the theoretical model and the present data demonstrates the model's applicability and predictive power in the context of learning computer programming.

That being said, not all of the affect transitions we found matched predictions of the theoretical model. This includes lack of data to support the predicted Frustration → Boredom and Boredom → Frustration transitions and the presence of an unexpected Boredom → Flow/Engagement transition. Limitations with this study are likely responsible for some of these mismatches. The sample size was small, so it is possible that increased participation in the study might confirm some of these expected transitions. In particular, the Boredom → Frustration transition was in the predicted

direction but not significant in our current sample. Additionally, we exclusively focused on affect, but ignored the intermediate events that trigger particular affective states (e.g., system feedback, hint requests, etc.). We plan to further explore our data by incorporating these interaction events as possible triggers for the observed transitions between affective states. This will allow us to more deeply understand why some of the predicted transitions did not occur (e.g., Frustration → Boredom) and some unexpected transitions did (e.g., Boredom → Flow/Engagement).

It is also possible that some aspects of the model might need refinement. In particular there appears to be an important relationship between Confusion → Frustration transitions, Confusion → Flow/Engagement transitions, performance, and hint usage. While hints may allow students to move past impasses and re-enter a state of flow/engagement, they may lead to an illusion of impasse resolution, which is not useful for learning. Conversely, resolving impasses without relying on external hints might lead a confused learner to momentarily experience frustration, but ultimately improve learning. Future work that increases sample size and specificity of the data (i.e., simultaneously modeling dynamics of affect and interaction events) will allow us to further explore the interaction of hints with the theoretical model, and is expected to yield a deeper understanding of affect dynamics during complex learning.

Acknowledgment. This research was supported by the National Science Foundation (NSF) (ITR 0325428, HCC 0834847, DRL 1235958). Any opinions, findings and conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

1. Haungs, M., Clark, C., Clements, J., Janzen, D.: Improving first-year success and retention through interest-based CS0 courses. Proceedings of the 43rd ACM technical symposium on Computer Science Education. pp. 589–594. ACM, New York, NY, USA (2012).
2. Fossati, D., Di Eugenio, B., Brown, C.W., Ohlsson, S., Cosejo, D.G., Chen, L.: Supporting Computer Science Curriculum: Exploring and Learning Linked Lists with iList. IEEE Transactions on Learning Technologies. 2, 107–120 (2009).
3. Anderson, J.R., Skwarecki, E.: The automated tutoring of introductory computer programming. Communications of the ACM. 29, 842–849 (1986).
4. Brusilovsky, P., Sosnovsky, S., Yudelson, M.V., Lee, D.H., Zadorozhny, V., Zhou, X.: Learning SQL programming with interactive tools: From integration to personalization. ACM Transactions on Computing Education. 9, 19:1–19:15 (2010).
5. Cheung, R., Wan, C., Cheng, C.: An ontology-based framework for personalized adaptive learning. Advances in Web-Based Learning–ICWL 2010. pp. 52–61. Springer, Berlin Heidelberg (2010).
6. Hsiao, I.-H., Sosnovsky, S., Brusilovsky, P.: Guiding students to the right questions: adaptive navigation support in an E-Learning system for Java programming. Journal of Computer Assisted Learning. 26, 270–283 (2010).
7. Pekrun, R.: The impact of emotions on learning and achievement: Towards a theory of cognitive/motivational mediators. Applied Psychology. 41, 359–376 (1992).

8. Grafsgaard, J.F., Fulton, R.M., Boyer, K.E., Wiebe, E.N., Lester, J.C.: Multimodal analysis of the implicit affective channel in computer-mediated textual communication. Proceedings of the 14th ACM international conference on Multimodal interaction. pp. 145–152. ACM, New York, NY, USA (2012).
9. Lee, D.M.C., Rodrigo, M.M.T., Baker, R.S.J. d, Sugay, J.O., Coronel, A.: Exploring the relationship between novice programmer confusion and achievement. In: D'Mello, S., Graesser, A., Schuller, B., and Martin, J.C. (eds.) Affective Computing and Intelligent Interaction. pp. 175–184. Springer, Berlin Heidelberg (2011).
10. Rodrigo, M.M.T., Baker, R.S.J. d, Jadud, M.C., Amarra, A.C.M., Dy, T., Espejo-Lahoz, M.B.V., Lim, S.A.L., Pascua, S.A.M.S., Sugay, J.O., Tabanao, E.S.: Affective and behavioral predictors of novice programmer achievement. SIGCSE Bulletin. 41, 156–160 (2009).
11. D'Mello, S., Graesser, A.: Feeling, thinking, and computing with affect-aware learning technologies. In: Calvo, R.A., D'Mello, S., Gratch, J., and Kappas, A. (eds.) Handbook of Affective Computing. Oxford University Press (in press).
12. D'Mello, S., Lehman, B., Sullins, J., Daigle, R., Combs, R., Vogt, K., Perkins, L., Graesser, A.: A time for emoting: When affect-sensitivity is and isn't effective at promoting deep learning. In: Aleven, V., Kay, J., and Mostow, J. (eds.) Intelligent Tutoring Systems. pp. 245–254. Springer, Berlin Heidelberg (2010).
13. Forbes-Riley, K., Litman, D.: Benefits and challenges of real-time uncertainty detection and adaptation in a spoken dialogue computer tutor. Speech Communication. 53, 1115–1136 (2011).
14. Khan, I.A., Hierons, R.M., Brinkman, W.P.: Mood independent programming. Proceedings of the 14th European Conference on Cognitive Ergonomics: Invent! Explore! pp. 28–31. ACM, New York, NY, USA (2007).
15. Bosch, N., D'Mello, S., Mills, C.: What Emotions Do Novices Experience During their First Computer Programming Learning Session? Proceedings of the 16th International Conference on Artificial Intelligence in Education (AIED 2013) (in press).
16. D'Mello, S., Graesser, A.: Dynamics of affective states during complex learning. Learning and Instruction. 22, 145–157 (2012).
17. D'Mello, S., Lehman, B., Pekrun, R., Graesser, A.: Confusion can be beneficial for learning. Learning and Instruction. (in press).
18. Inventado, P.S., Legaspi, R., Cabredo, R., Numao, M.: Student learning behavior in an unsupervised learning environment. Proceedings of the 20th International Conference on Computers in Education. pp. 730–737. National Institute of Education, Singapore (2012).
19. McQuiggan, S.W., Robison, J.L., Lester, J.C.: Affective transitions in narrative-centered learning environments. In: Woolf, B.P., Aïmeur, E., Nkambou, R., and Lajoie, S. (eds.) Intelligent Tutoring Systems. pp. 490–499. Springer, Berlin Heidelberg (2008).
20. Rosenberg, E.L., Ekman, P.: Coherence between expressive and experiential systems in emotion. Cognition & Emotion. 8, 201–229 (1994).
21. D'Mello, S., Taylor, R.S., Graesser, A.: Monitoring affective trajectories during complex learning. Proceedings of the 29th annual meeting of the cognitive science society. pp. 203–208. Cognitive Science Society, Austin, TX (2007).

Towards Deeper Understanding of Syntactic Concepts in Programming

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

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

Abstract. Syntactic mistakes and misconceptions in programming can have a negative impact on students' learning gains, and thus require particular attention in order to help students learn programming. In this paper, we propose embedding a discourse on syntactic issues and student's misconceptions into a dialogue between a student and an intelligent tutor. Based on compiler (error) messages, the approach aims to determine the cause for the error a student made (carelessness, misconception, or lack of knowledge) by requesting explanations for the violated syntactic construct. Depending on that cause, the proposed system adapts dialogue behaviours to student's needs by asking her to reflect on her knowledge in a self-explanation process, providing error-specific explanations, and enabling her to fix the error herself. This approach is designed to encourage students to develop a deeper understanding of syntactic concepts in programming.

Keywords: intelligent tutoring systems, programming, dialogue-based tutoring

1 Introduction

Programming is a useful skill and is related to several fields of study as economy, science, or information technology. Thus, teaching basics of programming is part of many curricula in universities and higher education. Programming is often taught bottom-up: First, syntactic aspects and low-level concepts are presented to students (e. g. variable declarations, IF, WHILE constructs, ... in the object-oriented programming paradigm). Then, iteratively higher-level concepts are taught (e. g. methods, recursion, usage of libraries, ...). Learning a programming language, however, cannot be approached theoretically only. It requires a lot of practice for correct understanding of abstract concepts (technical expertise) as well as logical and algorithmic thinking in order to map real-world problems to program code. Studies [8, 17] and our own teaching experiences have shown that studying programming is not an easy task and many students already experience (serious) difficulties with the basics: writing syntactically correct programs which can be processed by a compiler.

Source code is the basis for all programs, since without it algorithms cannot be executed and tested. Here, testing does not only mean testing done by

students themselves. Often tutorial and/or submission systems [7, 18] are used by lecture advisors in order to optimize their workflow and to provide students some further testing opportunities. These tests often focus on the algorithms, check program outputs given a specific input and require runnable source code.

Creating correct source code requires good knowledge and strict observance of the syntax and basic constructs of the programming language. Yet, students often use an integrated development environment (IDE) from the very beginning. Here, code templates and also possible solutions for syntactic errors are offered. Based on our experience over several years of teaching a course on "Foundations of programming" in which Java is introduced and used as a main programming language, we suppose that these features (code templates provided by an IDE) possibly hinder learning and deeper understanding: Novice programmers seem to use these features and suggestions (which are actually addressed to people who already internalized the main syntactic and semantic concepts of programming) blindly. As a result, students are often not able to write programs on their own (e. g. on paper) and do not understand the cause of errors.

In this paper, we propose a new tutoring approach which initiates a dialogue-based discourse between a student and an intelligent tutor in case of a syntactic error. The intelligent tutor aims at detecting a possible lack of knowledge or an existing misconception as well as suggesting further readings and correcting the misconception, respectively. The remainder of this paper is organized as follows: First, in Section 2, we give an overview of the state of the art of intelligent learning systems in programming. In Section 3, we then describe our approach in more detail, illustrate an exemplary discourse, and characterize possible approaches for an implementation. Finally, we discuss our approach in Section 4, draw a conclusion and point out future work in Section 5.

2 Intelligent Learning Systems in Programming

In recent years, Intelligent Tutoring Systems (ITSs) have found their way increasingly into classrooms, university courses, military training and professional education, and have been successfully applied to help humans learn in various domains such as algebra [10], intercultural competence [16], or astronaut training [1]. Constraint-based and cognitive tutor systems are the most established concepts to build ITSs, and have shown to have a positive impact on learning [14]. In the domain of programming, several approaches have been successfully applied to intelligently support teaching of programming skills using artificial intelligence (AI) techniques. In previous work [12], we reviewed AI-supported tutoring approaches for programming: example-based, simulation-based, collaboration-based, dialogue-based, program analysis-based, and feedback-based approaches.

Several approaches for building ITSs in the domain of programming are based on information provided by compilers. The Expresso tool [6] supports students in identifying and correcting Java programming errors by interpreting Java compiler error messages and providing feedback to students based on these messages. JECA is a Java error correcting algorithm which can be used in Intelligent Tutoring Systems in order to help students find and correct mistakes [19]. The

corresponding system prompted learner whether or not the system shall automatically correct found errors. Coull and colleagues [3] suggested error solutions to learners based on compiler messages by parsing these messages and comparing them to a database. These approaches aim to support learners in finding and correcting syntactic errors without explicitly explaining these issues, and, thus, did not ensure that a learner internalizes the underlying concept. HelpMeOut [5], however, is a recommender system based on compiler messages and runtime exceptions which formulated queries to a database containing error-specific information in order to recommend explanations for students' mistakes. The underlying database could be extended by users' input generated via peer interactions. This approach did not allow a discourse in order to determine student's knowledge or to correct possible misconceptions in student's application of knowledge, but provides solutions to students without encouraging students' learning. In our approach, we propose a dialogue-based discourse between a student and a tutor which aims at identifying the cause of the syntactic error, and at ensuring that the student gains a deeper understanding of the underlying syntactic concept she violated.

3 Solution Proposal

Programmers need to master syntactic and semantic rules of a programming language. Using integrated development environments such as Eclipse or Netbeans supports experienced programmers in finding and correcting careless mistakes and typos, and thus help them to efficiently focus on semantic issues. Novice programmers, however, who are still learning a programming language and, thus, are probably not entirely familiar with the syntactic concepts might be overwhelmed by messages provided by compilers. Interpreting error messages and correcting mistakes based on these messages can be a frustrating part of programming for those learners. IDEs, indeed, help them finding and correcting an error, but also impede learner's learning if learners follow IDEs' suggestion without reflecting on these hints and understanding why an error occurred.

How well programmers are able to find and correct syntactic mistakes strongly depends on the quality of messages and hints provided by compilers or IDEs [2, 13, 15]. Following previous work in the field of intelligent supporting systems for programming, we propose to provide guidance to novice programmers based on compiler (error) messages in order to help them master syntactic issues of programming languages. Instead of enriching compiler messages, we aim to determine student's knowledge about a specific violated syntactic construct. Depending on a student's level of knowledge, we propose to adapt the system's learning support to student's individual needs. For this, we distinguish three causes for syntactic errors:

E1 Errors caused by carelessness,
E2 Errors caused by lack of knowledge,
E3 Errors caused by misconceptions.

In order to determine which one of the three causes applies to a specific error, we propose to initiate a discourse between the learner and an intelligent

tutor (shown in Figure 1). Information provided by a compiler can be used to identify an erroneous part and the syntactic concept the student violated in order to lead the discourse to corresponding syntactic aspects. Embedded in dialogues and backed up by a knowledge database, the tutor first aims to determine whether or not the student is able to explain the underlying concept of the violated statement or syntactic expression. Our approach requires a knowledge base of the most typical errors of students. For this purpose, we used data collected in the submission system GATE [18]. We used GATE in our introductory Java teaching courses since 2009. This system supports the whole workflow from task creation, file submission, (limited) automated feedback for students to grading. We analyzed and categorized 435 compiler outputs of failed Java code compilations of student solutions: The ten most common syntax errors according to the compiler outputs (covering 70 % of all errors) are missing or superfluous braces (56 cases), usage of missing classes (e. g. based on an incomplete upload; 45), mismatching class names (according to the file name; 37), usage of undeclared variables (35), problems with if-constructs (23), usage of incompatible types (21), method definitions within other methods (primarily within the main method; 19), usage of undeclared methods (18), missing return statements in methods (14), and problems with SWITCH statements (12).

Just as experienced programmers also novice programmers make mistakes which are caused by carelessness (**E1**, e. g. a typo). In this case students are able to correctly and completely explain the concepts. The tutor then confirms the student's correct explanation, and students are able to fix the error without any further help. Errors caused by lacks of knowledge or misconception in the application of the knowledge, however, require special attention. This is the case if the student is not able to correctly and/or completely explain the underlying concept of a statement or syntactic expression which was violated. Then the tutor is not able to recognize student's explanation and distinguishes whether

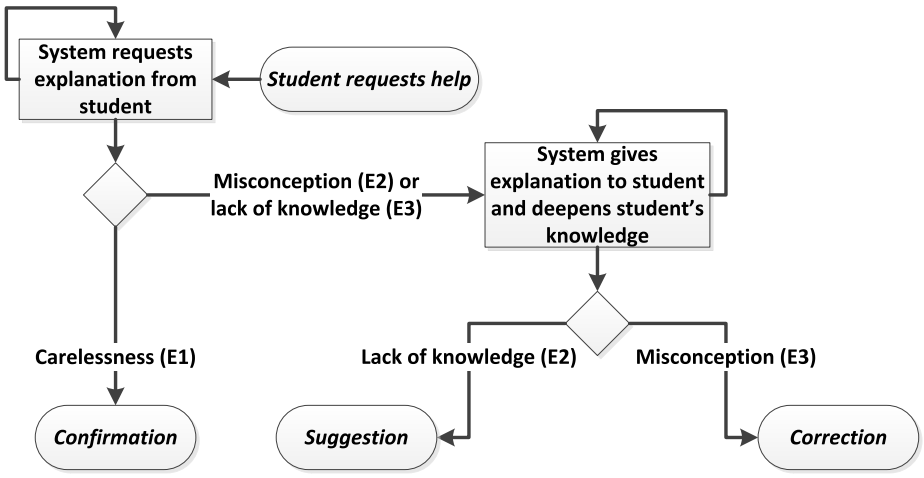


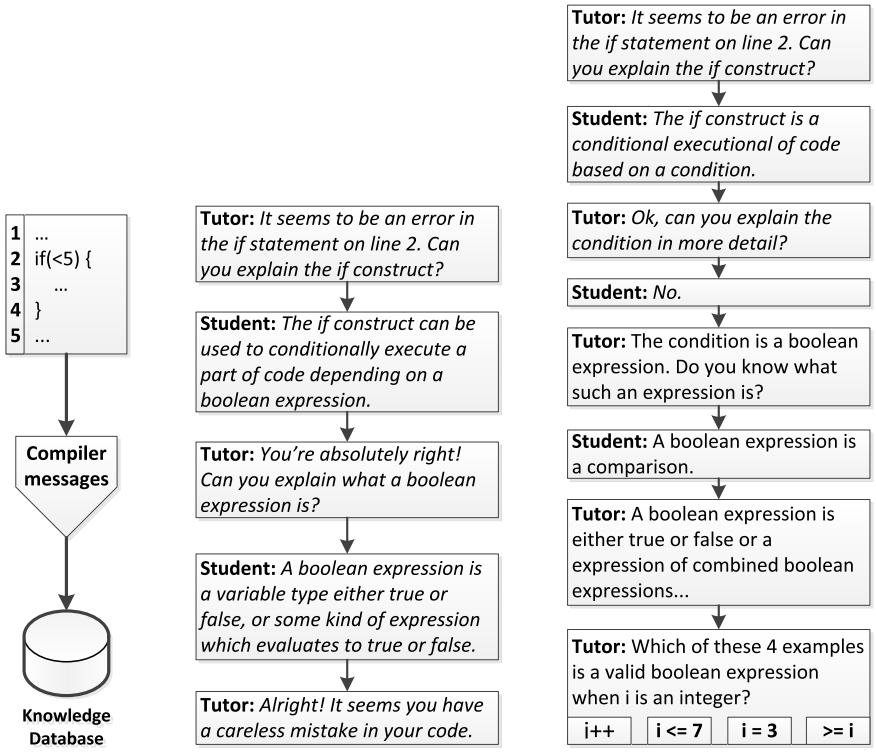
Fig. 1: Dialogue-based identification of cause for syntactic error

a lack of knowledge or a misconception caused the error by requesting further explanations from students. If a lack of knowledge is detected (**E2**), the tutor then suggests how to correct the error or points to the part of a (video) lecture explaining the violated concept. In the other case, if a misconception is detected (**E3**), the tutor changes its role in the discourse in order to revise the student's wrong and/or incomplete explanation. In this error-specific dialogue, the tutor then tries to explain the underlying concept the student violated. Therefore, the tutor could then provide step-by-step explanations using the knowledge base. To evaluate student understanding of single steps of explanations, the tutor could ask the student to confirm whether or not she understood the explanation, to ask her to complete/correct incomplete/erroneous examples covering the underlying syntactic concept, or to assess student's knowledge in question and answer manner.

In summary, we propose a dialogue-based intelligent tutor which initially interprets compiler (error) messages in order to identify the syntactic concept the student violated. Based on the compiler information, the tutor initiates a discourse with the student where it determines the cause of the error (**E1**, **E2** or **E3**). In a deeper examination of student's knowledge, the tutor uses a knowledge base in order to impart and deepen the concept which the syntactic error corresponds to. The tutor uses a computational model that is capable of automatically evaluating student's responses on tutor's questions. The goal is to correct misconceptions or to suggest further readings in order to fill lacks of knowledge and enable students to fix their mistakes in their own this way. In Section 3.2, we explain how such a model can be implemented.

3.1 Exemplary Dialogue-Based Discourse

In the above, we introduced typical syntactic errors that were made by students who attended a course on "Foundations of programming". The dataset contained students' exercise submissions of one of our introductory Java courses. To illustrate our approach (described in Section 3), we discuss a dialogue-based discourse exemplary for one of those typical errors (see Figure 2). A typical error that often occurred in students' submissions was that the implementation of a condition statement (IF construct) did not match the underlying syntactic concept. In the first dialogue (shown in Figure 2b), the tutor asks the student to explain the IF construct and, because it is part of an if-statement, what a boolean expression is. Here, the student is able to explain both concepts, and thus the mistake seems to have been caused by carelessness and the tutor confirms the student's explanations. In the second dialogue (shown in Figure 2c), the student gives an incomplete explanation on tutor's request. The tutor, consequently, asks the student to explain the condition in more detail which the student is not able to do. At that point, the tutor switches from requesting to providing explanations, and aims at deepening student's knowledge. Finally, the tutor aims at evaluating whether the student understood its explanations by asking a multiple-choice-question. Depending on the student's answer, the tutor can then assess whether the error was caused by a misconception or lack of knowledge. In the one case, the student is able to correctly respond to tutor's



(b) Dialogue 1 (c) Dialogue 2

Fig. 2: Dialogue-based discourse between student and intelligent tutor

question which indicates a misconception that could be corrected during the discourse. In the other case, the student is not able to correctly respond to the tutor's question which indicates lack of knowledge. Here, the tutor might suggest the student to repeat appropriate lecture(s)/exercise(s) in order to acquire the necessary knowledge.

3.2 Technical Implementation

In the dialogue-based approach proposed in this paper we need to distinguish two types of student's answers. The first one consists of explanations about a concept upon request of the system, and the second one includes short answers on error-specific examples and questions.

In order to understand a student's explanation on a programming concept we either provide her options to be chosen or allow her to express the explanation in a free form. In the first case, the system can understand the student's explanation by associating each template with a classifier of the error type. For example, in order to determine whether the student has made an error in the IF condition statement by carelessness, by misconception or lack of knowledge,

we can ask the student to explain this concept and provide her with three possible answers: 1) *The IF construct can be used to conditionally execute a part of code depending on a boolean expression.*, 2) *The IF construct can be used to express factual implications, or hypothetical situations and their consequences.*, 3) *I have no idea.* Obviously, the first answer is correct and the second answer is a misconception because students might refer the IF construct of a programming language (e. g., Java) to the IF used in conditional sentences in the English language. The third option indicates that the student has lack of the condition concept. This approach seems to be easy to implement, but requires a list of typical misconceptions of students. If we allow the student to express an explanation in a free form, the challenge is to understand possible multi-sentential explanations. In order to deal with this problem Jordan and colleagues [9] suggested to process explanations through two steps: 1) single sentence analysis, which outputs a first-order predicate logic representation, and 2) then assessing the correctness and completeness of these representations with respect to nodes in correct and buggy chains of reasoning.

In order to understand short answers on error-specific examples and questions, we can apply the form-filling approach for initiating dialogues. That is, for each question/example, correct answers can be anticipated and authored in the dialogue system. This approach is commonly used in several tutoring systems, e. g., the dialogue-based EER-Tutor [20], PROPL [11], AUTOTUTOR [4]. In addition to the form-filling approach, the Latent Semantic Analysis technique can also be deployed to check the correctness in the natural language student's answer by determining which concepts are present in a student's utterance (e. g., AUTOTUTOR).

4 Discussion

Our approach relies on the compiler's output. So, ambiguity of compiler messages is a crucial issue (also for students). The standard Java compiler works by following a greedy policy which causes that errors are reported for the first position in the source code where the compiler recognized a mismatch despite the fact that the cause of the error might lie somewhere else. There are also different parsers that use other policies and are capable of providing more specific feedback (e. g. the parser of the Eclipse IDE). Taking the code fragment "int i : 5;", e.g., the standard Java compiler outputs that it expects a ";" instead of the colon. The Eclipse compiler, however, outputs, that the colon is wrong and suggests that the programmer might have wanted to use the equal character "=". This difference in the compilers becomes even more manifest for lines where an opening brace is included. If there is an error in this line before the brace, the whole line is ignored by the standard Java compiler and a superfluous closing brace is reported at the end of the source code. Here, using a better parser (or even a custom parser) could improve error recognition regarding the position of the error and the syntactic principles violated by the programmer. Additional and more detailed information can help to cover more syntactic issues and to apply a more sophisticated discourse between learners and a dialogue-based tutor.

Generally, it is sufficient for our approach that a compiler reports the correct line and the affected basic structure of an error (e. g. If-statement), since our approach does not aim for directly solving the error, but supporting the students to fix the mistake on their own. This, however, requires a good knowledge base of the basic structures about a programming language.

5 Conclusion and Future Work

In this paper, we proposed a dialogue-based approach interpreting compiler (error) messages in order to determine syntactic errors students made, and thus to adapt the behaviour of the intelligent tutor to the individual needs of students depending on three causes of errors (carelessness, lack of knowledge, or misconception). Our proposed system initiates a dialogue asking for explanations of the violated syntactic construct and determines which cause applies for the affected violated construct. Then the proposed approach adapts dialogue behaviours to student's needs confirming correct knowledge or providing error-specific explanations. We argued that this method works better than just presenting error messages or suggestions for fixing an error, because it encourages students to reflect on their knowledge in a self-explanation process and finally enables them to fix the errors themselves.

In future, we plan to implement our approach and test it with students in an introductory programming course. Initially, we will apply self-explanation in human-tutored exercises in order to gather dialogues which can be used to build a model for our approach.

References

[1] K. Belghith, R. Nkambou, F. Kabanza, and L. Hartman. An intelligent simulator for telerobotics training. *IEEE Transactions on Learning Technologies*, 5(1):11–19, 2012.

[2] B. Boulay and I. Matthew. Fatal error in pass zero: How not to confuse novices. In G. Veer, M. Tauber, T. Green, and P. Gorny, editors, *Readings on Cognitive Ergonomics Mind and Computers*, volume 178 of *Lecture Notes in Computer Science*, pages 132–141. Springer Berlin Heidelberg, 1984.

[3] N. Coull, I. Duncan, J. Archibald, and G. Lund. Helping Novice Programmers Interpret Compiler Error Messages. In *Proceedings of the 4th Annual LTSN-ICS Conference*, pages 26–28. National University of Ireland, Galway, Aug. 2003.

[4] A. Graesser, N. K. Person, and D. Harter. Teaching Tactics and Dialog in AutoTutor. *International Journal of Artificial Intelligence in Education*, 12:257–279, 2001.

[5] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[6] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE '03, pages 153–156, New York, NY, USA, 2003. ACM.

[7] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[8] T. Jenkins. A participative approach to teaching programming. In *Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education*, ITiCSE '98, pages 125–129, New York, NY, USA, 1998. ACM.

[9] P. W. Jordan, M. Makatchev, U. Pappuswamy, K. VanLehn, and P. L. Albacete. A natural language tutorial dialogue system for physics. In G. Sutcliffe and R. Goebel, editors, *FLAIRS Conference*, pages 521–526. AAAI Press, 2006.

[10] K. R. Koedinger, J. R. Anderson, W. H. Hadley, and M. A. Mark. Intelligent tutoring goes to school in the big city. *International Journal of AI in Education*, 8:30–43, 1997.

[11] H. C. Lane and K. VanLehn. A dialogue-based tutoring system for beginning programming. In V. Barr and Z. Markov, editors, *FLAIRS Conference*, pages 449–454. AAAI Press, 2004.

[12] N. T. Le, S. Strickroth, S. Gross, and N. Pinkwart. A review of AI-supported tutoring approaches for learning programming. In *Accepted for the International Conference on Computer Science, Applied Mathematics and Applications 2013, Warsaw, Poland*. Springer Verlag, 2013.

[13] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 499–504, New York, NY, USA, 2011. ACM.

[14] A. Mitrovic, K. Koedinger, and B. Martin. A comparative analysis of cognitive tutoring and constraint-based modeling. In P. Brusilovsky, A. Corbett, and F. Rosis, editors, *User Modeling 2003*, volume 2702 of *Lecture Notes in Computer Science*, pages 313–322. Springer Berlin Heidelberg, 2003.

[15] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. Compiler error messages: what can help novices? In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, pages 168–172, New York, NY, USA, 2008. ACM.

[16] A. Ogan, V. Aleven, and C. Jones. Advancing development of intercultural competence through supporting predictions in narrative video. *International Journal of AI in Education*, 19(3):267–288, Aug. 2009.

[17] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[18] S. Strickroth, H. Olivier, and N. Pinkwart. Das GATE-System: Qualitätssteigerung durch Selbsttests für Studenten bei der Onlineabgabe von Übungsaufgaben? In *DeLFI 2011: Die 9. e-Learning Fachtagung Informatik*, number P-188 in GI Lecture Notes in Informatics, pages 115 – 126. GI, 2011.

[19] E. R. Sykes and F. Franek. Presenting jeca: a java error correcting algorithm for the java intelligent tutoring system. In *Proceedings of the IASTED Conference on Advances in Computer science*, 2004.

[20] A. Weerasinghe, A. Mitrovic, and B. Martin. Towards individualized dialogue support for ill-defined domains. *International Journal of AI in Education*, 19(4):357–379, Dec. 2009.

An Intelligent Tutoring System for Teaching FOL Equivalence

Foteini Grivokostopoulou, Isidoros Perikos, Ioannis Hatzilygeroudis

University of Patras, Department of Computer Engineering &Informatics, 26500, Hellas (Greece)

{grivokwst,perikos,ihatz @ceid.upatras.gr}

Abstract. In this paper, we present an intelligent tutoring system developed to assist students in learning logic. The system helps students to learn how to construct equivalent formulas in first order logic (FOL), a basic knowledge representation language. Manipulating logic formulas is a cognitively complex and error prone task for the students to deeply understand. The system assists students to learn to manipulate and create logically equivalent formulas in a step-based process. During the process the system provides guidance and feedback of various types in an intelligent way based on user's behavior. Evaluation of the system has shown quite satisfactory results as far as its usability and learning capabilities are concerned.

Keywords: Intelligent Tutoring System, Teaching Logic, First Order Logic, Logic Equivalence

1 Introduction

The advent of the Web has changed the way that educational material and learning procedures are delivered to the students. It provides a new platform that connects students with educational resources which is growing rapidly worldwide giving new possibilities to students and tutors and offering better, cheaper and more efficient and intensive learning processes. ITSs constitute a popular type of educational systems and are becoming a fundamental mean of education delivery. Their main characteristic is that they provide instructions and feedback tailored to the learners and perform their tasks mainly based on Artificial Intelligence methods. The teacher's role is also changing and is moving from the face-to-face knowledge transmission agent to the specialist who designs the course and guides and supervises the student's learning process [10]. ITSs have been used with great success in many challenging domains to offer individualized learning to the students and have demonstrated remarkable success in helping students learn challenging content and strategies [18].

Logic is considered to be an important domain for the students to learn, but also a very hard domain to master. Many tutors acknowledge that AI and logic course contains complex topics which are difficult for the students to grasp. Knowledge Repre-

sentation & Reasoning (KR&R) is a fundamental topic of Logic. A basic KR&R language is First-Order Logic (FOL), the main representative of logic-based representation languages, which is part of almost any introductory AI course and textbook. So, teaching FOL as a KR&R language is a vital aspect. Teaching and learning FOL as KR&R vehicle includes many aspects. During an AI course the student's learn to translate Natural Language (NL) text into FOL, a process also called *formalization*. A NL sentence is converted into a FOL formula, which conveys the sentence's meaning and semantics and can be used in several logic processes, such as inference and equivalency creation. Equivalency is a fundamental topic in logic. It characterizes two or more representations in a language that convey the same meaning and have the same semantics. Manipulating FOL formulas is considered to be a hard, cognitive complex and error prone process for the students to deeply understand and implement. In this paper, we present an intelligent tutoring system developed to assist students in learning logic and more specifically to help students learn how to construct equivalent formulas in FOL. The system provides interactive guidance and various types of feedback to the students.

The rest of the paper is structured as follows. Section 2 presents related work. Section 3 presents the logic equivalences in FOL. Section 4 presents the system architecture and analyzes its functionality. Section 5 presents the logic equivalent learning. More specifically describes the learning scenarios, the student's interaction and the feedback provided by the system. Section 6 presents the evaluation studies conducted and the results gathered in real classroom conditions. Finally, Section 7 concludes our work and provides directions for future work.

2 Related work

There are various systems created for teaching for helping in teaching logic [8] [19]. However, most of them deal with how to construct formal proofs, mainly using natural deduction. Logic Tutor [1] is an intelligent tutoring system (ITS) for learning formal proofs in propositional logic (PL) based on natural deduction. As an intelligent system, it adapts to the needs of the students via keeping user models. In [4], an intelligent tutoring system is developed for teaching how to construct propositional proofs and visualize student proof approaches to help teachers to identify error prone areas of the students. All the above systems, although deal with learning and/or teaching logic, they are not concerned with how to use FOL as a KR&R language.

KRRT [2] is a web–based system the main goal of which is helping students to learn FOL as a KR&R language. The student gives his/her FOL proposal sentence and the system checks its syntax and whether is the correct one. NLtoFOL [7] is a web-based system developed to assist students in learning to convert NL sentences into FOL. The student can select a NL sentence and interactively convert it in a step based approach into the corresponding FOL. In [6], we deal with teaching the FOL to CF (Clause Form) conversion, via a web-based interactive system. It provides a step-by-step guidance and help during that process. Organon [5] is a web-based tutor for basic logic courses and helps the students during practice exercises. All the above systems, although deal with learning (or teaching) logic, they do not deal with logic

equivalency and how to assist students to learn how to construct logically equivalent formulas. As far as we are aware of, there is only one system that claims doing the latter. It is called IDEAS [11] and deals with rewriting formulas from propositional logic into disjunctive normal form. A student is called to transform a formula by applying one transformation rule at a time. The system provides feedback to the student. Also, the system provides a tool [12] for proving equivalences between propositional logic formulas. However, it is restricted to propositional logic and does not deal with FOL.

3 Logical Equivalences in FOL

FOL is the most widely used logic-based knowledge representation formalism. Higher order logics are difficult to handle, whereas lower order logics, such as those based on propositional calculus, are expressively poor. FOL is a KR&R language used for representing knowledge in a knowledge base, in the form of logical formulas, which can be used for automatically making inferences. Logical formulas or sentences explicitly represent properties of or relations among entities of the world of a domain. In logic, two logical formulas *p* and *q* are logically equivalent if they have the same logical content. Logical equivalence between *p* and *q* is sometimes expressed as $p \leftrightarrow q$. Logical equivalence definition in FOL is the same as in propositional logic, with the addition of rules for formulas containing quantifiers. Table 1 presents rules of logical equivalence between FOL formulas.

Table 1. Rules of logical Equivalence for FOL

Equivalence	Name
$p \wedge T \leftrightarrow p$, $p \vee F \leftrightarrow p$	Identity Laws
$p \vee T \leftrightarrow T$, $p \wedge F \leftrightarrow F$	Domination Laws
$p \vee p \leftrightarrow p$, $p \wedge p \leftrightarrow p$	Idempotent Laws
$\neg(\neg p) \leftrightarrow p$	Double Negation Law
$p \vee q \leftrightarrow q \vee p$, $p \wedge q \leftrightarrow q \wedge p$	Commutative Laws
$(p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$, $(p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$	Associative Laws
$(p \Rightarrow q) \leftrightarrow (\neg p \vee q)$	Implication Elimination
$\neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$, $\neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$	De Morgan's Laws
$\forall x\, P(x) \leftrightarrow \neg \exists x \neg P(x)$, $\neg \exists x\, P(x) \leftrightarrow \forall x \neg P(x)$	De Morgan's FOL
$p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	Distribution Laws
$\forall x\, (P(x) \wedge Q(x)) \leftrightarrow \forall x\, P(x) \wedge \forall x Q(x)$ $\exists x\, (P(x) \vee Q(x)) \leftrightarrow \exists x\, P(x) \vee \exists x Q(x)$	Distribution Laws FOL

4 System Architecture and Function

The architecture of our system is depicted in Fig.1. It consists of five units: *Domain Model (DM), Student Model (SM), Student Interface (SI), Interface Configuration (IC)* and *Intelligent Data Unit (IDU).*

Domain Model (DM) contains knowledge related to the subject to be taught as well as the actual teaching material. It focuses on assisting students to learn how to create FOL-equivalent formulas and so syntax of FOL and equivalence rules constrains are stored in the domain model.

Student Model (SM) unit is used to record and store student related information. Also contains the system's beliefs regarding the student's knowledge of the domain and additional information about the user, such as personal information and characteristics. SM enables the system to adapt its behavior and its pedagogical decisions to the individual student who uses it [3]. Also it sketches the cognitive process that happens in the student learning sessions.

Student interface (SI) is the interactive part of the system. Through SI, a student initially subscribes to the system. During subscription, the required personal information, such as name, age, gender, year of study and email are stored. After subscription, the student can anytime access the system. SI is also responsible for configuring the interface to adapt to the needs of the specific session.

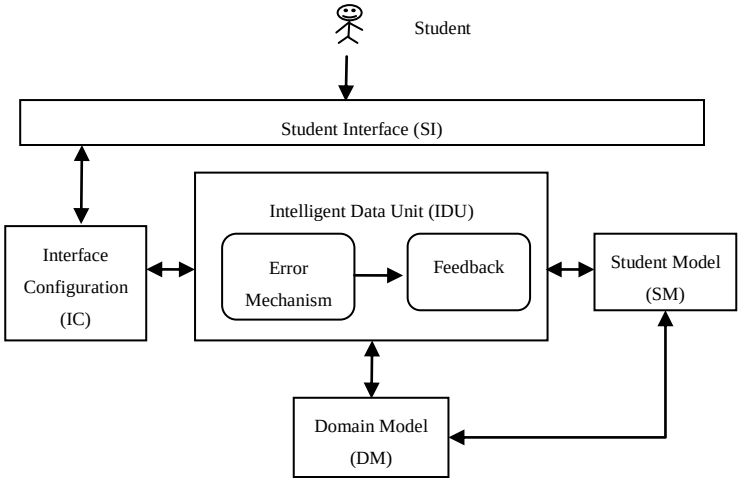


Fig.1. System architecture and its components.

Interface Configuration (IC) unit is responsible for configuring the student interface during the learning sessions, based on the guidelines given by the intelligent data unit. So, the student interface is dynamically re-configured to adapt to the needs of the specific session.

Intelligent Data Unit (IDU) interacts with IC and its main purpose is to provide guidance and feedback to the students and help during application of the logical equivalence rules. It is a rule-based system that based on the input data from user interface decides on which reconfigurations should be made to the user interface or which kind of interaction will be allowed or given to the user. It is also responsible for tracing user's mistakes and handling them in terms of appropriate feedback to the student.

IDU deals with a student's actions for each equivalence exercise as follows:

1. Let the student select an equivalence rule to apply to the FOL formula
2. Check if the selected current equivalence rule can be applied.
   - If it can, allow the student to insert his/her answer to the current rule and go to 2.
   - Otherwise, inform the student that the selected rule is not applicable, provide proper feedback and allow select a new answer.
3. Check the student's answer (formula) to the selected rule
   - If it is correct, inform his/her and go to 1.
   - Otherwise: (a) Determine the error(s) made by the student. (b) Provide feedback based on the error(s) and the corresponding equivalence rule. (c) Allow the student to give a new answer for the selected rule and go to 1.

5 User interaction

The student interface of the system is dynamically reconfigured during a conversion process. After the student enters the system, he/she can select any of the existing FOL formulas/exercises and then starts its conversion into an equivalent formula. This process is made in a step-based approach where the student, at each step, has to select and implement a logic equivalence rule (see above, Table 1). At each step the student can request the system's assistance and feedback (which is based on student's actions and knowledge state). Initially, the student has to select a proper equivalence rule to implement. All the equivalence rules are presented at the working area of student's interface. The student can select a rule and apply it to the formula. If the rule cannot be applied, the system provides proper feedback messages notifying with the reason why it cannot be applied. In contrast, if the rule can be applied, a proper work area is created and the student can manipulate the formula and transform it by applying the selected rule. Then the student can submit the answer (FOL formula). After the student gives an answer, the system informs him/her whether the answer is correct or incorrect. If it is incorrect, the system performs an analysis of the student's answer to find and recognize the errors made by the student. After that the student can submit the new formula derived by the rule application.

As an example, consider the FOL formula "$(\forall x)\sim likes(x,snow) \Rightarrow \sim skier(x)$". Initially the student selects to apply the *implication elimination* of equivalency as illustrated in Fig. 2. The system analyses the formula and recognizes that the selected law can be applied. So, proper configurations are made on the interface and the student can insert his/her answer, which is the equivalent formula derived from the application of the rule. After the student submits his/her answer, the system analyzes it and recognizes that the implication is not removed correctly and generates the proper feedback message(s). The feedback messages are linked to the help button and the student can look at them by clicking on it.

5.1 Feedback

The behavior of the system is modeled to consist of two (feedback) loops, the inner and the outer loop respectively [16]. The main role of the inner loop is to provide

feedback to the student as a reaction to his/her actions during an exercise, whereas the role of the outer loop is to select the next exercise corresponding to the student's knowledge state. The inner loop of the system is responsible for analyzing the student's answer and provides the proper feedback messages. The feedback provided, in order to enhance its effectiveness, refers to different levels of verification and elaboration. Verification concerns the confirmation whether a student's process is correct or not, while the elaboration can address the answer and related topics, discuss particular error(s) and guide the student towards the correct answer [15].

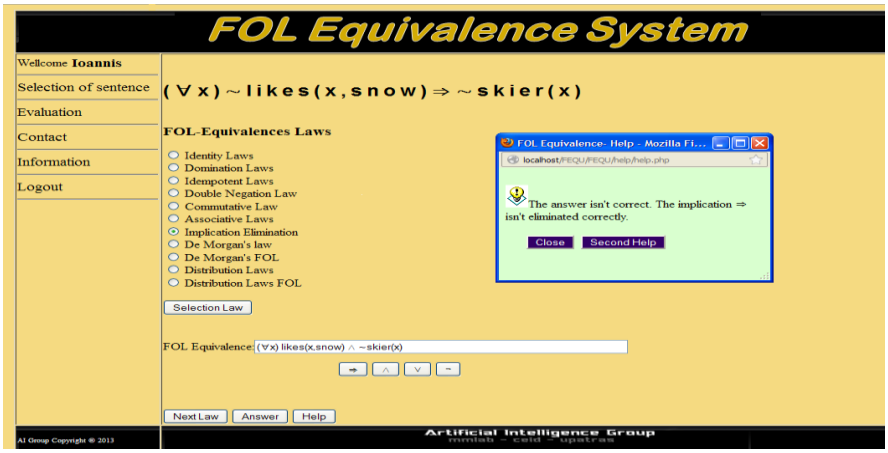


Fig.2. Student Interface

The categories and the types of feedback developed are based on combinations of the classifications of feedback presented in [13] and [16]. So, the main types of feedback offered to the students by the system are the following.

- ***Minimal feedback.*** The system informs the student if the answer is correct or not.
- ***Error-specific feedback.*** When a student's answer is incorrect the system provides the proper feedback based on the errors made, indicating what makes the answer incorrect and the reason why it does it.
- ***Procedural feedback.*** The system can provide a student with hints on what has to do to correct a wrong answer and also what to do next.
- ***Bottom-out hints.*** The system can decide to give the correct answer of a step to the student. This can be done after a student's request or after constantly failure rates and circumstances.
- ***Knowledge on meta-cognition.*** The system analyzes a student's interactions and behavior and can provide meta-cognitive guiding and hints.

The system implements an incremental assistance delivery. Initially, after a student's incorrect action, starts by delivering minimal feedback, just noticing that there are errors and inconsistencies in the student's action. Error-specific feedback is offered after a student's erroneous action. Research has shown that student's motivation

for understanding and learning is enhanced when errors are made [9] and the delivery of proper feedback can help the students get a deeper understanding and revise misconceptions. While a student is striving to specify the correct action, the system scales up its assistance till the delivery of the correct action/answer. Providing the correct answer in logic exercises-procedures are consider an important part of the system's assistance. Indeed, student knowledge and performance can be improved significantly after receiving knowledge of correct response feedback, indicating the correct answer [17]. The system never gives unsolicited hints to the student. If the student's answer is incorrect, the proper feedback messages are available (linked) via the help button. So, the student can get those messages on demand, by clicking on the help button. The pedagogical assumption indicates that when the student has the control of the timing of the help provided by the system, there is a greater likelihood that the help messages are received at the right time and therefore be more effective for knowledge construction [14].

6 Evaluation

We conducted an evaluation study of the system during the AI course in the fall semester of the academic year 2011-2012 at our Department. 100 undergraduate students from those enrolled in the course participated in the evaluation study. The students had already attended the lectures covering the relevant logic concepts. The methodology selected to evaluate the system is a pre-test/post-test, experimental/control group one, where the control group used a traditional teaching approach. The students were divided into two groups of 50 students each one, of balanced gender, which were named group A and group B respectively. Group A was selected to act as the experimental group and group B as the control group. Group A (experimental) did some homework through the system, whereas Group B (control) did the homework without using the system and then submit the answers to the tutor and discuss them with him.

Initially, all students took a pre-test on logical equivalence concept. The test included 15 FOL formulas-exercises and the students were asked to provide equivalent FOL formulas. After that, the students of group B were given access to the system and were asked to study for a week aiming at one 20 minutes session per day. After that intervention, the students of both groups took a final post-test including 15 FOL formulas-exercises. The two tests consisted of exercises of similar difficulty level and the score ranged from 0 to 100.

In order to analyze students' performance, an independent *t*-test was used on the pre-test. The mean and standard deviations of the pre-test were 45.18 and 14.73 for the experimental group, and 47.34 and 14.01 for the control group. As the *p*-value (Significant level) was $0.567 > 0.05$ and $t = 0.46$, it can be inferred that those two groups did not significantly differ prior to the experiment. That is, the two groups of students had statistically equivalent abilities before the experiment. In Table 2 and Table 3 the descriptive statistics and the t-test results from assessment of students' learning performance are presented. The results revealed that the mean value of the

pre-test of the experimental group is higher than the mean value of the pre-test of the control group. The Levene's test confirmed the equality of variances of the control and experimental groups for pre-test ($F = 0.330$, $p = 0.567$) and post-test ($F = 3.016$, $p = 0.086$). Also the t-test result ($p=0.000 < .05$) shows a significant difference between the two groups. Thus, it implies that the students in the experimental group got a deeper understanding in manipulating FOL formulas and created correctly equivalent formulas for more FOL formulas exercises than the control group.

Table 2. Descriptive Statistics of Pre-test and Post-test

	Group	N	Mean	SD	SE
Pre-Test	Group A	50	45.18	14.73	2.08
	Group B	50	47.34	14.01	1.98
Post-Test	Group A	50	51.74	18.17	2.57
	Group B	50	71.56	15.43	2.18

Table 3. t-test results

Equality of variance		**F-test for variance**		**t-Test for mean**			
		F	Sig.	t	df	Sig.(2-tailed)	MD
Pre-Test	Equal	0.33	0.567	-0.751	98	0.454	-2.16
	Unequal			-0.751	97.756	0.454	-2.16
Post-Test	Equal	3.016	0.086	-5.879	98	0.000	-19.8
	Unequal			-5.879	95.49	0.000	-19.8

In the second part of the evaluation study, the students of group B, who had used the system, were asked to fill in a questionnaire. The questionnaire was made to provide both qualitative and quantitative data. It included questions for evaluating the usability of the system, asking for the students' experience and their opinions about the impact of system in learning and understanding logical equivalence. The questionnaire consisted of nine questions and the results are presented in Table 4. Questions Q1-Q6 were based on a Likert scale (1: not at all, 5: very much). Questions 7-8 were open type questions and concerned strong and weak points of the system or problems faced and also improvements that can be made to the system. Finally, question 9 was about spent time to cope with the system and had three possible answers: less than 15 min, 15-30 min and more than 30 min. Their answers show that 72% of the students needed less than fifteen minutes and only 12% of them needed more than 30 min.

Table 4.Questionnaire Results.

		ANSWERS (%)				
Q	QUESTIONS	1	2	3	4	5
1	How you rate your overall experience?	0	0	20	28	52

2	How much the system did assisted you to learn logical equivalence?	0	0	18	32	50
3	How helpful was the feedback provided?	0	4	12	36	48
4	Did you find the interface of the system helpful?	0	0	28	36	36
5	When stuck, did the system provide enough help so that you could fix the problem(s)	0	2	14	34	50
6	Do you feel more confident in dealing with logical equivalence transformations?	0	4	16	38	42

The students' answers to Q1-Q6 indicate that the majority of the them enjoyed interacting with the system and 82% of them believe that the system helped them in learning FOL equivalences. Also, 84% of them found the feedback provided by the system very useful and that assisted them in manipulating FOL formulas and creating equivalent ones.

7 Conclusions and Future Work

Logic is acknowledged by tutors to be a hard domain for students to grasp and deeply understand. It contains complex cognitive processes and students face many difficulties to understand and correctly implement them. Manipulating FOL formulas and transforming them into equivalent forms is a fundamental topic in logic, but also hard and error prone for students.

In this paper, we introduce an intelligent tutoring system developed to help students in learning how to deal with FOL equivalent formulas. It provides the student an interactive way to manipulate FOL formulas and transform them into equivalent form(s) by applying equivalence rules (or chain of rules) or proper combinations of them. The student, at each stage of the transformation, gets proper guidance and feedback by the system on his/her actions. Regarding the usefulness of the system, the reactions of the students were very encouraging. An evaluation study was conducted to test the system impact on student's learning. The results revealed that the experimental group outperformed the control group significantly on the post-test exercises. According to the results, the students of the experimental group got a deeper understanding of the logical transformations and significantly enhanced their knowledge. Moreover, the system helped the students to improve their logic conceptual understanding and also to increase their confidence in handling equivalence.

However there are some points that the system could be improved. A direction for future research would be the development of an automatic assessment mechanism to assess the student's performance during the learning interaction with the system. This could help the system better adapt to the student.

Acknowledgements

This work was supported by the Research Committee of the University of Patras, Greece, Program "KARATHEODORIS", project No C901.

References

1. Abraham, D., Crawford, L., Lesta, L., Merceron, A.,Yacef, K.: The Logic Tutor: A multimedia presentation. Electronic Journal of Computer-Enhanced Learning, (2001)
2. Alonso, J.A., Aranda, G.A., Martin-Matceos, F.J.: KRRT: Knowledge Representation and Reasoning Tutor. In: Moreno Diaz, R., Pichler, F., Quesada Arencibia, A. (eds.) EUROCAST LNCS, vol. 4739, pp.400–407, Springer, Heidelberg (2007)
3. Brusilovsky, P.: Student model centered architecture for intelligent learning environment. In Proc. of Fourth International Conference on User Modeling, Hyannis, MA, pp.31-36 (1994)
4. Croy, M., Barnes, T., Stamper, J.: Towards an Intelligent Tutoring System for propositional proof construction. In: Brey, P., Briggle, A., Waelbers, K. (eds.) European Computing and Philosophy Conference, pp.145–155,Amsterdam (2007)
5. Dostálová, L., Lang.J.: Organon - the web-tutor for basic logic courses Logic Journal of the IGPL 15(4), pp.305-311, (2007)
6. Grivokostopoulou, F., Perikos I., Hatzilygeroudis I.: A Web-based Interactive System for Learning FOL to CF conversion In Proc. of the IADIS International Conference e-Learning 2012, Lisbon, Portugal, pp.287-294, (2012)
7. Hatzilygeroudis, I., Perikos, I.: A web-Based Interactive System for Learning NL to FOL Conversion. New Directions in Intelligent Interactive Multimedia Systems and Services-2 Studies in Computational Intelligence, vol. 226, pp.297-307 (2009)
8. Hendriks, M., Kaliszyk, C., van Raamsdonk, F., and Wiedijk, F.: Teaching logic using a state-of-the-art proof assistant. Acta Didactica Napocensia, 3(2): 35-48, (2010)
9. Hirashima, T., Horiguchi, T., Kashihara, A., Toyoda, J.:Error-Based Simulation for Error-Visualization and Its Management. J. of Artificial Intelligence in Education, vol.9, pp.17-31 (1998).
10. Huertas, A.: Teaching and Learning Logic in a Virtual Learning Environment. Logic Journal of the IGP 15(4), pp.321–331 (2007)
11. Lodder, J., Passier, H., Stuurman, S.:Using IDEAS in teaching logic, lessons learned. Computer Science and Software Engineering, In Proc. of International Conference Computer Science and Software Engineering, vol. 5, pp.553–556 (2008).
12. Lodder, J., Heeren, B.: A teaching tool for proving equivalences between logical formulae. In Third International Congress on Tools for Teaching Logic, pp.66-80, (2011).
13. Narciss, S.: Feedback strategies for interactive learning tasks. In J. M. Spector, M. D. Merrill, J. J. G. Van Merrienboer, M. P. Driscoll (Eds.), Handbook of research on educational communications and technology 3rd ed., pp.125-143,(2008)
14. Renkl, A., Atkinson, R. K., Maier, U. H., Staley, R.:From example study to problem solving: Smooth transitions help learning. J. of Experimental Education,vol.70, pp.293–315, (2002).
15. Shute, V.J. :Focus on formative feedback, Review of Educational Research, Vol. 78, No. 1, pp.153–189 (2008).
16. VanLehn, K.: The behavior of tutoring systems. International Journal of Artificial Intelligence in Education, 16,pp.227-265
17. Wang, S-L., Wu, P-Y.: The role of feedback and self-efficacy on Web-based learning: the social cognitive perspective, Computers & Education vol.51 pp.1589-1598(2008).
18. Woolf, B. P. :Building intelligent interactive tutors: Student-centered strategies for revolutionizing e-learning. Burlington MA: Morgan Kaufman Publishers (2009).
19. Sieg, W., Scheines, R.: Computer Environments for Proof Construction. Interactive Learning Environments 4(2), pp. 159–169 (1994).

Informing the Design of a Game-Based Learning Environment for Computer Science: A Pilot Study on Engagement and Collaborative Dialogue

Fernando J. Rodríguez, Natalie D. Kerby, Kristy Elizabeth Boyer

Department of Computer Science, North Carolina State University, Raleigh, NC 27695
{fjrodri3,ndkerby,keboyer}@ncsu.edu

Abstract. Game-based learning environments hold great promise for supporting computer science learning. The ENGAGE project is building a game-based learning environment for middle school computational thinking and computer science principles, situated within mathematics and science curricula. This paper reports on a pilot study of the ENGAGE curriculum and gameplay elements, in which pairs of middle school students collaborated to solve game-based computer science problems. Their collaborative behaviors and dialogue were recorded with video cameras. The analysis reported here focuses on nonverbal indicators of disengagement during the collaborative problem solving, and explores the dialogue moves used by a more engaged learner to repair a partner's disengagement. Finally, we discuss the implications of these findings for designing a game-based learning environment that supports collaboration for computer science.

Keywords: Engagement, Collaboration, Dialogue, Game-Based Learning.

1 Introduction

Supporting engagement within computer science (CS) education is a central challenge for designers of CS learning environments. More broadly, engagement is a subject of increasing attention within the AI in Education community. A growing body of empirical findings has revealed the importance of supporting learner engagement. Particular forms of disengagement have been associated with decreased learning, both overall and with respect to local learning outcomes within spoken dialogue tutoring systems [1, 2]. Targeted interventions can positively impact engagement; for example, metacognitive support may influence students to spend more time on subsequent problems, and integrating student performance measures into a tutoring system allows them to reflect on their overall performance [3]. A promising approach to support engagement involves adding game elements to intelligent tutors or other learning environments [4, 5] or creating game-based learning environments with engaging narratives [6]; both approaches have been shown to increase student performance and enjoyment in general. However, even with these effective systems, some disengaged

behaviors are negatively associated with learning, and the relationships between engagement and learning are not fully understood.

Collaboration is another promising approach for supporting engagement and can be combined with game-based learning environments [7]. Results have demonstrated the importance of well-timed help for collaborators [8] and the promise of pedagogical agents that support self-explanation [9]. This study considers collaboration in the problem-solving domain of computer science, where a combination of hints and collaboration support may be particularly helpful [10]. However, many questions remain regarding the best sources and types of engagement support in this context.

Game-based learning environments for teaching computer science have started to become popular in recent years. The CodeSpells game [11] aims to teach middle school students how to program in the Java programming language. The ENGAGE project aims to develop, implement, and evaluate a narrative-centered, game-based learning environment that will be deployed in middle school for teaching computer science principles. The game is designed to be played collaboratively by pairs of students. Presently, the project is in its design and implementation phase, conducting iterative refinement and piloting of curriculum and gameplay elements. During this process, we aim to extract valuable lessons about how middle school students collaborate to solve computer science problems and how this collaboration can be supported within an intelligent game-based learning environment.

This paper reports on a pilot study of the ENGAGE curriculum and simulated gameplay elements. In this study, pairs of students collaborated and their collaborative behaviors and dialogue were recorded with video cameras. Nonverbal indicators of disengagement were annotated manually across the videos. We report an analysis of these disengagement behaviors by students' collaborative role, and explore the dialogue moves used by a more engaged learner to repair a partner's disengagement.

2 ENGAGE Game Based Learning Environment

The main goals of the ENGAGE project, which is currently in its design and implementation phase, are to create a highly engaging educational tool for teaching computer science to middle school students, contribute to research on the effectiveness of game-based learning, and investigate its potential to broaden participation of underrepresented groups in computer science. During the first year of the project, the first draft of the curriculum to be used within the environment was developed. The curriculum is based on the CS Principles course under development by the College Board [12] with the goal of shifting focus from a specific programming language (Java, in the case of the existing AP Computer Science course) to the broader picture of computer science concepts. The CS Principles curriculum emphasizes seven *big ideas*:

Table 1. CS Principles focused evidence statement examples

CS Principles Number	Evidence Statement
6b	Explanation of how number bases, including binary and decimal, are used for reasoning about digital data
13a	Explanation of how computer programs are used to process information to gain insight and knowledge
18b	Explanation of how an algorithm is represented in natural language, pseudo-code, or a visual or textual programming language
24a	Use of an iterative process to develop a correct program
30c	Explanation of how cryptography is essential to many models of cybersecurity

1. Computer science is a creative process
2. Abstractions can reduce unimportant details and focus on relevant ones
3. Big data can be analyzed using various techniques in order to create a new understanding or refine existing knowledge
4. Algorithms are a sequence of steps used to solve a problem and can be applied to structurally similar problems
5. These algorithms can be automated using a programming language
6. The Internet has revolutionized communication and collaboration
7. Computer science has an impact on the entire world

Through an iterative process, we selected a subset of the CS Principles curriculum by analyzing the evidence statements [13] for suitability within a game-based learning environment and for appropriateness for the middle school audience. Additional validation of this curriculum will be undertaken by middle school teachers during an upcoming summer institute and through pilot testing. An example of learning objectives to be implemented as game-based learning activities is shown in Table 1.

The setting of the game is an underwater research facility that has been taken over by a rogue scientist. Students take on the role of a computer scientist sent to investigate the situation, reconnect the station's network, and thwart the villain's plot by solving various computer science puzzles in the form of programming tasks. There are two main gameplay mechanics: players can move around in the 3D environment in a similar manner to many 3D platforming games (Fig. 1a), and different devices within the environment can be programmed using a visual programming interface. Players can drag "blocks" that represent programming functions and stack them together to create a program (Fig. 1b).[1] By programming these devices in certain ways, players can manipulate the environment and solve each in-game area's puzzle and move on to the next task. The game sections are divided into four main levels:

[1] This drag-and-drop programming language with blocks is closely modeled after and inspired by Scratch [21], but for compatibility reasons, a customized programming environment is being created for the ENGAGE game.

- *Tutorial*: Students are introduced to the game environment and shown how to use the controls for both gameplay mechanics. They are also given an overview of basic programming concepts (sequences of statements, loops, and conditionals), as well as the concept of broadcasting (sending signals from one device to another).
- *Digital World*: The puzzles in this level involve binary numbers, and students must convert these binary numbers into an understandable form (decimal, text, color image, etc.) in order to solve them and progress. The conceptual objective of this level is that computers communicate in binary and that the meaning behind a binary sequence depends on its interpretation.
- *Cybersecurity*: Before the students can reconnect the station's network, they must establish proper cybersecurity measures so that their communications are not compromised by the villain. In this level, students learn about cryptography and various encryption techniques in order to ensure a safe network connection.
- *Big Data*: The research station that students must restore had been studying different aspects of the undersea environment, including the pollution of the water and how it has affected the life forms that inhabit the area. Students must try to reason about this data by performing basic analyses and creating visual models that are embodied in the 3D environment, which will enable students to progress to the final level.

a) Game environment b) Programming interface

Fig. 1. Engage screenshots

3 Pilot Study

The pilot study was conducted within a computer science elective course for middle school students (ages 11 to 14) at a charter middle school. Students attended 4 sessions lasting between 90 and 120 minutes long, facilitated by members of the ENGAGE project team. Each of the sessions resulted in a corpus of data, though only one of these, which involves solving a binary puzzle within a visual programming environment, is analyzed within this paper. This paper focuses on the final session of the course, in which students worked in pairs to solve three game-based tasks using Build Your Own Blocks, a drag-and-drop visual programming language [14]. This activity simulates programming within ENGAGE, whose programming environment is still under development. Student participants included 18 males and 2 females, though

the female pair was absent on the day the present corpus was collected. This gender disparity is an intrinsic problem in many technology electives and is an important consideration of the ENGAGE project, which will examine differential outcomes for students from underrepresented groups using the game-based learning environment.

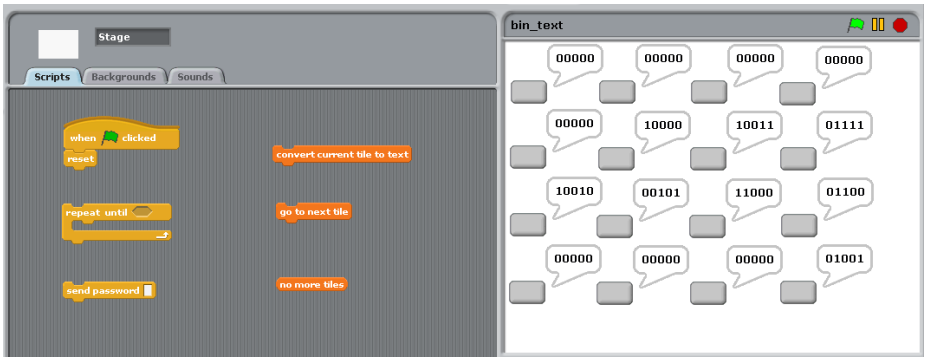


Fig. 2. Visual programming interface for final problem in pilot study

The exercises the students solved involved converting binary numbers into decimal numbers and textual characters. The first problem asked the students to write a program that would convert the given binary numbers into decimal numbers and highlight the cells that contained even numbers. The next problem was identical to the first except that the students had to highlight the prime numbers instead. The final and most challenging problem asked students to convert the binary numbers into textual characters, manually decipher a password, and input this password into their program (Fig. 2). All subroutines corresponding to binary conversions, highlighting blocks, number comparisons, and password entry were provided as blocks that students could use within their own programs. This design choice was made to abstract some complex implementation from students so they could focus on planning and implementing the steps to solve their problem.

For the duration of the exercise, students collaborated in pairs and took turns controlling the keyboard and mouse on a single computer [7]. In computer science education, this paradigm is referred to as pair programming: the *driver* actively creates the solution, while the *navigator* provides feedback [15] (Fig. 3). Research has shown that pair programming can provide many benefits to college-level students taking introductory programming courses, especially those with little to no prior programming experience. A review by Preston [16] highlights some benefits: students create higher-quality programs as a result of the communication of ideas between partners; they can achieve a better understanding of programming by supporting each other through the exercise; and although the activity is collaborative, individual test scores and course performance are also improved.

Students were asked by a researcher to switch roles every six minutes. When a pair would finish a task, they were asked to raise their hands and wait for a researcher to verify their program solution. If it was correct, the researcher would verbally describe the next exercise and set up the programming interface; if not, the researcher would

provide general feedback on the proposed solution. Students were also allowed to raise their hands if they had any questions for the researcher regarding the programming task. The allotted time to complete all three programming tasks was 40 minutes, with two pairs completing them sooner.

4 Video Corpus and Disengagement Annotation

During this pilot study, video was recorded for all nine pairs of students using a tripod-mounted digital camera recording at 640x480 resolution and 30 frames per second. The nine videos were divided into 5-minute segments to facilitate annotation and analysis. Of the total 65 segments, 25 were randomly selected for annotation and serve as the basis for the results presented here (a subset was necessary due to the time requirement of manual annotation, in this case approximately 8 minutes per minute of video). Each segment was manually annotated by a judge for student disengagement by observing for one of three signs of disengagement. First, *posture* was considered to indicate disengagement when gross postural shifts clearly suggested that the student was attending to something other than the programming task; this exaggerated disengaged posture was often accompanied by other indications of disengagement such as off-task speech (Fig. 3b). Another indicator of disengagement was averted *gaze*, which commonly accompanied the other two signs but could occur independently. Finally, students would sometimes engage in off-task *dialogue* with their partners, or even with other students in the classroom. It is important to note that we do not equate off-task behavior with disengagement; there were instances in which students continued to work on the learning task while holding off-task conversations with their partners. Sabourin et al. [17] show that off-task behavior can be a way for students to cope with negative emotions, such as confusion or frustration. Likewise, student disengagement does not necessarily imply off-task behavior. Disengagement in this context is defined primarily as focusing attention on something other than the learning task; identifying cognitive and affective states underlying the disengagement is left to future analyses.

To annotate the videos, each human judge would watch until disengagement was observed by either of the two collaborators, paused the video, annotated the start time of the disengagement event, then continued and annotated the end time, returning to previous points of the video as needed. Judges thus marked episodes of disengagement, as well as who appeared to facilitate re-engagement: did the student shift her attention back to the programming task by herself?; did the student's partner ask for her assistance?; or did an instructor need to arrive to provide feedback or clarify any questions? In order to establish reliability of this annotation scheme, 12 of the 65 video segments were randomly selected and assigned to two judges, and the tagged segments were discretized into one-second intervals in which each student was classified as either engaged or disengaged by each judge. The Kappa for disengagement was 0.59 (87.25% agreement). In other words, approximately 87.25% of the time, both judges applied the same engagement tag. Adjusting for chance (that is, students were more likely to be engaged than disengaged at a given point) the

Kappa agreement statistic was 0.59, indicating fair agreement [18]. For the events on which both judges agreed that disengagement had occurred, the tag for who facilitated re-engagement resulted in 78.57% agreement, or a Kappa of 0.60, indicating moderate agreement.

a) Engaged pair b) Disengaged pair

Fig. 3. Collaborative setup

5 Results

Overall, student drivers spent an average of 16.4% of their time disengaged (st. dev.=16.6%), compared to a much higher 42.6% for navigators (st. dev.=24.1%). This is not surprising, since drivers are more actively engaged in the programming activity. Across both roles, out of the student re-engagement events, 76.8% were annotated as self re-engagements, with the remaining events corresponding to an external source of re-engagement (either partner or instructor). However, the collaborative role plays an important part in self re-engagement: drivers had an 87.7% probability of self re-engaging, while navigators had a lower 68.7% probability of self re-engaging. These findings may indicate that repairing one's own disengaged state is more challenging for the collaborative partner who is not actively at the controls.

We examine instances in which annotators marked that the driver re-engaged a disengaged navigator through dialogue. There are 22 such instances. Four are questions addressed to the collaborative partner, such as, "OK, now where?" and "Do we delete this?" These questions re-engaged the navigator in part because attending to the speaker's questions is a social dialogue norm. Two utterances served as exclamations, e.g., "What the heck?" In these cases, the driver was expressing surprise with an event in the learning environment, which drew the disengaged student's attention back to the task. The remaining utterances were fragments, such as, "Pick up current tile…," though one utterance explicitly reminded the disengaged student about short time remaining, "So we only have a couple of minutes."

To examine these re-engagement events in context, two excerpts are considered (Table 2). In Excerpt A, the navigator gets stuck and raises his hand to ask for help when he notices the instructor is nearby, briefly becoming disengaged while his

partner continues to work on the exercise. The driver then turns to his partner and asks for feedback, causing the navigator to re-engage into the learning activity. In Excerpt B, the students had just received feedback on from the instructor when the navigator engages in off-topic dialogue with another team. Meanwhile, the driver makes a plan and then calls for the navigator's attention, re-engaging him.

Table 2. Dialogue excerpts featuring navigator disengagement

Timestamp	Role	Dialogue Excerpt A
19:25	**Navigator:**	OK, if prime, number is prime. Dang!
		[*Navigator notices instructor nearby, raises hand*]
19:34	**Navigator:**	Uh...
Disengaged		[*Navigator looks away from screen, leans back on seat*]
19:38	**Driver:**	OK, now where?
Re-engaged		[*Navigator points at program block*]
19:40	**Navigator:**	Put it there.
		Dialogue Excerpt B
		[Note: students are discussing '@' symbols]
26:01	**Navigator:**	OK, @'s. Do you want more @'s... (inaudible)
26:08	**Driver:**	One two three four five
Disengaged		[*Navigator looks away to talk to another student*]
26:14	**Driver:**	I have an idea. You (taps navigator's shoulder)
Re-engaged		
26:16	**Navigator:**	Me?

6 Discussion

These excerpts suggest that within a collaborative game-based learning environment for computer science, providing both students with a sense of control may be particularly important. Because it may be more difficult to stay engaged on a task if one is not actively participating in it, particularly for younger audiences, the issue of mutual participation is paramount within the learning environment. The narrative game-based learning framework may prove particularly suitable for addressing this challenge: drivers and navigators can be provided with separate responsibilities and even with complementary information so that the participation of both students is required to complete the game-based tasks. Examples include designing the algorithmic solution to the problem or performing some calculations relevant to the main task; Williams and Kessler [19] state that 90% of students surveyed about pair programming listed these as the tasks that the navigator typically assists with. These may, in turn, help the navigator experience a heightened sense of control, and thereby, engagement.

This pilot study demonstrated that because of strong social norms associated with human dialogue, strategic moves by a partner can serve to re-engage a student. Typically, drivers will ask their partners for feedback if they are unsure of their solution or if they are inexperienced programmers. In these cases, an active conversation between both students occurs, and both students are engaged. An intelligent game-based learning environment that senses disengagement may be able

to scaffold this type of dialogue in order to mitigate disengagement on the part of either student.

7 Conclusion and Future Work

Game-based learning environments hold great promise for supporting computer science education. The ENGAGE project is developing a game-based learning environment for middle school computer science, and we have presented results from an early pilot study for the curriculum and some simulated elements of gameplay in which students worked collaboratively in pairs to solve computer science problems. The results suggest that supporting engagement may be particularly important within a collaborative situation for the student who is not at the controls. Providing both students with an active role during gameplay, and scaffolding dialogue to re-engage a student who has become disengaged, are highly promising directions for intelligent game-based learning environments. Both of these interventions would be well supported within a narrative-centered, game-based learning environment framework.

There are several important directions for future work regarding engagement within game-based learning environments for computer science. First, the current study was very limited in sample size and diversity of participants, so expanding the scope of students considered is a key consideration. It is also important to examine the duration of engagement once re-engagement has occurred and the effectiveness of interventions with respect to longer-term engagement. Additionally, in contrast to the fully manual video annotation presented here, it would be beneficial to integrate automated methods of measuring disengagement, such as the ones presented by Arroyo and colleagues [20]. Finally, addressing issues of diversity and groupwise differences of re-engagement strategies is an essential direction in order to develop game-based learning environments that support engagement and effective learning for all students.

Acknowledgements. This paper relies on the contributions of the entire ENGAGE research team, including James Lester, Bradford Mott, Eric Wiebe, Rebecca Hardy, Wookhee Min, Kirby Culbertson, and Marc Russo. The authors wish to thank Joseph Grafsgaard and Alexandria Vail for their helpful contributions. This work is supported in part by NSF through grants CNS-1138497 and CNS-1042468. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the participants and do not necessarily represent the official views, opinions, or policy of the National Science Foundation.

References

1. Forbes-Riley, K., Litman, D.: When Does Disengagement Correlate with Learning in Spoken Dialog Computer Tutoring? Proceedings of AIED. pp. 81–89 (2011).
2. Cocea, M., Hershkovitz, A., Baker, R.S.J.: The Impact of Off-Task and Gaming Behaviors on Learning: Immediate or Aggregate? Proceedings of AIED. pp. 507–514 (2009).

3. Arroyo, I., Ferguson, K., Johns, J., Dragon, T., Meheranian, H., Fisher, D., Barto, A., Mahadevan, S., Woolf, B.P.: Repairing Disengagement With Non-Invasive Interventions. Proceedings of AIED. pp. 195–202 (2007).
4. Jackson, G.T., Dempsey, K.B., McNamara, D.S.: Short and Long Term Benefits of Enjoyment and Learning within a Serious Game. Proceedings of AIED. pp. 139–146 (2011).
5. Rai, D., Beck, J.E.: Math Learning Environment with Game-Like Elements: An Incremental Approach for Enhancing Student Engagement and Learning Effectiveness. Proceedings of ITS. pp. 90–100 (2012).
6. Rowe, J., Shores, L., Mott, B., Lester, J.C.: Integrating Learning, Problem Solving, and Engagement in Narrative-Centered Learning Environments. *IJAIED*. 21, 115–133 (2011).
7. Meluso, A., Zheng, M., Spires, H.A., Lester, J.C.: Enhancing 5th Graders' Science Content Knowledge and Self-Efficacy through Game-Based Learning. Computers & Education Journal. 59, 497–504 (2012).
8. Chaudhuri, S., Kumar, R., Howley, I., Rosé, C.P.: Engaging Collaborative Learners with Helping Agents. Proceedings of AIED. pp. 365–272 (2009).
9. Hayashi, Y.: On Pedagogical Effects of Learner-Support Agents. Proceedings of ITS. pp. 22–32 (2012).
10. Holland, J., Baghaei, N., Mathews, M., Mitrovic, A.: The Effects of Domain and Collaboration Feedback on Learning in a Collaborative Intelligent Tutoring System. Proceedings of AIED. pp. 469–471 (2011).
11. Esper, S., Foster, S.R., Griswold, W.G.: On the Nature of Fires and How to Spark Them When You're Not There. Proceedings of the SIGCSE Conference. pp. 305–310. ACM Press, New York, New York, USA (2013).
12. Stephenson, C., Wilson, C.: Reforming K-12 Computer Science Education… What Will Your Story Be? ACM Inroads. 3, 43–46 (2012).
13. The College Board: AP CS Principles: Learning Objectives and Evidence Statements, (2010).
14. Harvey, B., Mönig, J.: Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? Constructionism. pp. 1–10 (2010).
15. Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Miller, C., Balik, S., Yang, K.: Improving the CS1 Experience with Pair Programming. Proceedings of the SIGCSE Conference. pp. 359–362 (2003).
16. Preston, D.: Pair Programming as a Model of Collaborative Learning: A Review of the Research. Journal of Computing Sciences in Colleges. 20, 39–45 (2005).
17. Sabourin, J., Rowe, J.P., Mott, B.W., Lester, J.C.: When Off-Task is On-Task: The Affective Role of Off-Task Behavior in Narrative-Centered Learning Environments. Proceedings of AIED. pp. 523–536 (2011).
18. Landis, J.R., Koch, G.G.: The Measurement of Observer Agreement for Categorical Data. Biometrics. 33, 159–174 (1977).
19. Williams, L.A., Kessler, R.R.: All I Really Need to Know about Pair Programming I Learned in Kindergarten. Communications of the ACM. 5, 108–114 (1999).
20. Arroyo, I., Cooper, D.G., Burleson, W., Woolf, B.P., Muldner, K., Christopherson, R.: Emotion Sensors Go to School. Proceedings of AIED. pp. 17–24 (2009).
21. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y.: Scratch: Programming for All. Communications of the ACM. 52, 60–67 (2009).

When to Intervene: Toward a Markov Decision Process Dialogue Policy for Computer Science Tutoring

Christopher M. Mitchell, Kristy Elizabeth Boyer, and James C. Lester

Department of Computer Science, North Carolina State University,
Raleigh, North Carolina, USA
{cmmitch2, keboyer, lester}@ncsu.edu

Abstract. Designing dialogue systems that engage in rich tutorial dialogue has long been a goal of the intelligent tutoring systems community. A key challenge for these systems is determining when to intervene during student problem solving. Although intervention strategies have historically been hand-authored, utilizing machine learning to automatically acquire corpus-based intervention policies that maximize student learning holds great promise. To this end, this paper presents a Markov Decision Process (MDP) framework to learn when to intervene, capturing the most effective tutor turn-taking behaviors in a task-oriented learning environment with textual dialogue. This framework is developed as a part of the JavaTutor tutorial dialogue project and will contribute to data-driven development of a tutorial dialogue system for introductory computer science education.

Keywords: Tutorial Dialogue, Markov Decision Processes, Reinforcement Learning

1 Introduction

The effectiveness of tutorial dialogue has been widely established [1, 2]. Today's tutorial dialogue systems have been successful in producing learning gains as they support problem solving [3–5], encourage collaboration [6, 7], and adapt to student responses [8]. These systems have also been shown to be successful in implementing some affective adaptations of human tutors [5, 9]. Recent research into tutorial dialogue systems with unrestricted turn-taking has shown promise for simulating the natural tutorial dialogue interactions of a human tutor [7]. Recognizing and simulating the natural conversational turn-taking behavior of humans continues to be an area of active research [10–12], and there has recently been renewed interest in developing dialogue systems that harness unrestricted turn-taking paradigms [7, 13, 14].

The JavaTutor tutorial dialogue project aims to build a tutorial dialogue system with unrestricted turn-taking and rich natural language to support introductory computer science students. The overarching paradigm of this project is to automatically derive tutoring strategies using machine learning techniques applied to a corpus collected from an observational study of human-human tutoring. In

particular, the project focuses on how to devise tutorial strategies that deliver both cognitive and affective scaffolding in the most effective way. The project to date has seen the collection of a large corpus of tutorial dialogue featuring six repeated interactions with tutor-student pairs, accompanied by data on learning and attitude for each session as well as across the study [15–17]. This paper describes an important first step toward deriving tutorial dialogue policies automatically from the collected corpus in a way that does not simply mimic the behavior of human tutors, but seeks to identify the most effective tutorial strategies and implement those within the system's dialogue policy.

In recent years, reinforcement learning (RL) has proven useful for creating tutorial dialogue system policies in structured problem-solving interactions, such as what type of question to ask a student [18] and whether to elicit or tell the next step in the solution [19]. In order to harness the power of RL-based approaches within a tutorial dialogue system for computer science education, two important research problems must be addressed. First, a representation must be formulated in which student computer programming actions, which can occur continuously or in small bursts, can be segmented at an appropriate granularity and provided to the model. Second, because student dialogue moves, tutor dialogue moves, and student programming actions can occur in an interleaved manner with some overlapping each other, features to define the Markov Decision Process state space must be identified that preserve the rich, unrestricted turn-taking and mixed-initiative interaction to the greatest extent possible. In a first effort to address these challenges, this paper presents a novel application of RL-based approaches to the JavaTutor corpus of textual tutorial dialogue. In particular, the focus here is automatically learning when to intervene from this fixed corpus of human-human task-oriented tutorial dialogue with unrestricted turn-taking. The presented approach and policy results can inform data-driven development of tutorial systems for computer science education.

2 Human-Human Tutorial Dialogue Corpus

To date, the JavaTutor project has seen the collection of an extensive corpus of human-human tutoring. Between August 2011 and March 2012, 67 students interacted with experienced tutors through the Java Online Tutoring Environment (Figure 1). Students were drawn from a first-year engineering course on a voluntary basis. They earned partial course credit for their participation. Students who reported substantial programming experience in a pre-survey were excluded from the experienced-tutoring condition (and were instead placed in a peer-tutoring collaborative condition that is beyond the scope of this paper), since the target population of the JavaTutor tutorial dialogue system is students with no programming experience. Each student completed six tutoring sessions over a period of four weeks, and worked with the same tutor for all interactions. Each tutoring session was limited to forty minutes.

Seven tutors participated in the study. Their experience level ranged from multiple years' experience in one-on-one tutoring to one semester's experience as a teaching assistant or small group tutor. Gender distribution of the tutors was three female and

four male. Tutors were provided with printed learning objectives for each session and were reminded that they should seek to support the students' learning as well as motivational and emotional state. Also, because each subsequent tutoring session built on the completed computer program from the preceding session, tutors were encouraged to ensure that students completed the required components of the programming task within the allotted forty-minute time frame.

The overarching computer science problem-solving task was for students to create a text-based adventure game in which a player can explore scenes based on menu choices. In order to implement the adventure game, students learned a variety of programming concepts and constructs. This paper focuses on the first of the six tutoring sessions. The learning objectives covered in this first session included compiling and running code, writing comments, variable declaration, and system I/O. For each learning objective, there was a conceptual component and an applied component. For example, for the learning objective related to compiling code, the conceptual learning objective was for students to explain that compilation translates human-readable Java programs into machine-readable forms. The applied learning objective was for students to demonstrate that they can compile a program by pressing the "compile" button within the interface.

The Java Online Learning Environment, shown in Figure 1, supports textual dialogue between the human tutor and student. It also provides tutors with a real-time synchronized view of the student's workspace. The interface allows for logging events to a database with millisecond precision, making it straightforward to reconstruct the events of a session from these logs. There are two information channels between a tutor and a student. The first of these, the messaging pane, supports unrestricted textual dialogue between a tutor and a student, similar to common instant messaging applications. There are no restrictions placed on turn-taking, allowing either person to compose a message at any time. In addition, both students and tutors are notified when their partner is composing a message. The second information channel is the student's workspace. A tutor can see progress on the Java program written by the student in real-time, but the tutor is not able to edit the program directly. The Java programming environment is scaffolded for novices: it hides class declarations, method declarations, and import statements from the student, lowering the amount of complex syntax visible. Students effectively compose their programs within a main method "sandbox".

In order to measure the effectiveness of each session, students completed a pre-test at the beginning of each session and a post-test at the end of each session evaluating their knowledge of the material to be taught in that lesson. From these, we computed normalized learning gain using the following equation:

$$normalizedLearningGain = \begin{cases} \dfrac{posttest - pretest}{1 - pretest}, & posttest > pretest \\ \dfrac{posttest - pretest}{pretest}, & posttest \leq pretest \end{cases} \quad (1)$$

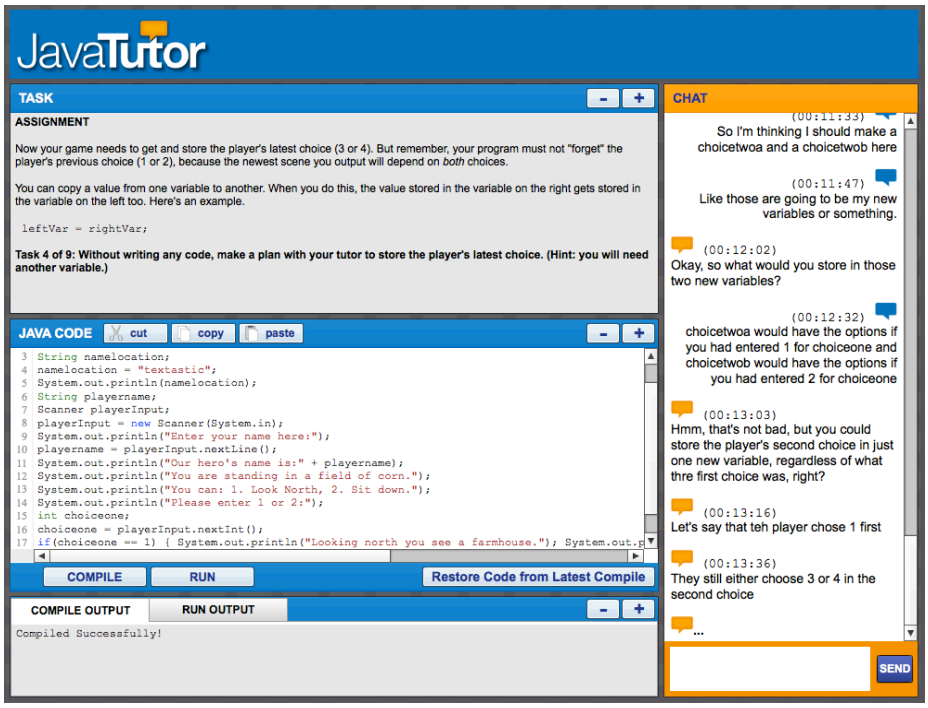


Fig. 1. A student's view of the JavaTutor human tutoring interface

This equation, adapted from Marx and Cummings [20] allows for the possibility of negative learning gain during a session, a phenomenon that occurred three times in the corpus. These normalized learning gain values can range from -1 to 1. In the present study normalized learning gains ranged from -0.29 to 1 (mean = 0.42; median = 0.45; st. dev. = 0.32). Students scored significantly higher on the post-test than the pre-test ($p < .001$).

3 Building the Markov Decision Process

The goal of the analysis presented here is to derive an effective tutorial intervention policy—*when* to intervene—from a fixed corpus of student-tutor interactions. From the tutors' perspective, the decision to intervene was made based on the state of the interaction as observed through the two information channels in the interface: the textual dialogue pane and the synchronized view of the student's workspace. In order to use a MDP framework to derive an effective intervention policy, we describe a representation of the interaction state as a collection of features from these information channels.

A Markov Decision Process is a model of a system in which a policy can be learned to maximize reward [21]. It consists of a set of states S, a set of actions A representing possible actions by an agent, a set of transition probabilities indicating

how likely it is for the model to transition to each state $s' \in S$ from each state $s \in S$ when the agent performs each action $a \in A$ in state s, and a reward function R that maps real values onto transitions and/or states, thus signifying their utility.

The goal of this analysis is to model tutor interventions during the task-completion process, so the possible actions for a tutor were to intervene (by composing and sending a message) or not to intervene. Hence, the set of actions is defined as $A = \{TutorMove, NoMove\}$. We chose three features to represent the state of the dialogue, with each feature taking on one of three possible values. These features, described in Table 1, combine as a triple to form the states of the MDP as (Current Student Action, Task Trajectory, Last Action). These three features were chosen because they succinctly represent the current state of the dialogue in terms of turn-taking information in the *Current Action* and *Last Action* features, while the recent behavior of the student is captured in the *Task Trajectory* and *Current Action* features. Thus, these features supply an agent with sufficient information to learn a basic intervention policy while relying only on automatically annotated features. By selecting a small state space and action space, we avoid data sparsity issues [22], thereby decreasing the likelihood of states being insufficiently explored in our corpus, and increasing the likelihood of producing a meaningful intervention policy.

Table 1. The features that define the states of the Markov Decision Process

Current Student Action	Task Trajectory	Last Action
• ***Task***: Working on the task • ***StudentDial***: Writing a message to the tutor • ***NoAction***: No current student action	• ***Closer***: Moving closer to the final correct solution • ***Farther***: Moving away from correct solution • ***NoChange***: Same distance from correct solution	• ***TutorDial***: Tutor message • ***StudentDial***: Student message • ***Task*****:** Student worked on the task

In addition, the model includes 3 more states: an *Initial* state, in which the model always begins, and two final states: one with reward +100 for students achieving higher-than-median normalized learning gain and one with reward -100 for the remaining students, following the conventions established in prior research into reinforcement learning for tutorial dialogue [18, 19].

Using these formalizations, one state was assigned to each of the log entries collected during the sessions and transition probabilities were computed between them when a tutor made an intervention (*TutorMove*) and when a tutor did not make an intervention (*NoMove*) based on the transition frequencies observed in the data. Any states that occurred less than once per session on average were combined into a single *LowFrequency* state, following the convention of prior work [23]. There were four states fitting this description: (*Task*, *Farther*, *StudentDial*), (*StudentDial*, *Farther*, *StudentDial*), (*StudentDial*, *Farther*, *Task*), and (*StudentDial*, *Farther*, *TutorDial*). Thus, the final MDP model contained 25 states requiring a tutorial intervention decision (23 states composed of feature combinations, the *LowFrequency* state, and the *Initial* state), and two final states.

The *Current Student Action* and *Last Action* features were relatively straightforward to assign to log entries by simply observing what a student was currently doing at that point in the session and observing what action had occurred most recently. The *Task Trajectory* feature was computed by discretizing the students' work on the task into chunks, which presents a substantial research question and design decision for supporting computer science learning. Historically, intelligent tutoring systems for computer science have utilized granularity at one extreme or the other. The smallest possible granularity is every keystroke, perhaps the earliest example of this being the Lisp tutor of Anderson and colleagues [24]. The largest granularity could arguably be to evaluate only when the student deems the artifact complete enough to manually submit for evaluation, which was the approach taken by another very early computer science tutor, Proust [25]. For the JavaTutor system, evaluating the student program more often than at the completion of tasks is essential to support dialogue, but an every-keystroke evaluation is too frequent due in part to algorithm runtime limitations. We define our task events as beginning when a student begins typing in the task pane and ending when a student has not typed in the task pane for at least 1.5 seconds. This threshold of 1.5 seconds was chosen empirically before model building to strike a balance between shorter thresholds, which resulted in frequent switching between "working on task" and "not working on task" states, and longer thresholds, which resulted in never leaving the "working on task" state.

After each task event (discretized as described above), a student's program was separated into tokens as defined by the Java compiler, and a token-level minimum edit distance was computed from that student's final solution for the lesson, tokenized in the same manner. Variable names, comments, and the contents of string literals were ignored in this edit distance calculation. The change in the edit distance from one chunk to the next determined the value of the *Task Trajectory* feature. Because the tutors were experienced in Java programming and had knowledge of the lesson structure, it is reasonable to assume that they were able to determine whether the student was moving farther or closer to the final solution. In this way, the edit distance algorithm provides a rough, automatically computable estimate of the tutors' assessment of student progress.

4 Policy Learning

The goal of this analysis is to learn a tutorial intervention policy—*when* to intervene—that reflects the most effective strategies within the corpus. In the MDP framework described above, this involves maximizing the learning gain reward. In order to learn this tutorial intervention policy, we used a policy iteration algorithm [21] on the MDP. For each iteration, this algorithm computes the expected reward in each state $s \in S$ when taking each action $a \in A$, based on the computed transition probabilities to other states and the expected rewards of those states from the previous iteration. Following the practice of prior work [13, 17], a discount factor of 0.9 was used to penalize delayed rewards (those requiring several state transitions to achieve) in favor of immediate rewards (those requiring few state transitions to achieve). The

policy iteration continues until convergence is reached; that is, until the change in expected reward for each state is less than some epsilon value between iterations. We used an epsilon of 10^{-7}, requiring 125 iterations to converge. The resulting policy is shown in Table 2.

Table 2. The learned tutorial intervention policy

State (*Current Action*, *Task Trajectory*, *Last Action*)	**Policy**
(Task, Closer, Task)	TutorMove
(Task, Closer, StudentDial)	TutorMove
(Task, Closer, TutorDial)	TutorMove
(Task, Farther, Task)	TutorMove
(Task, Farther, TutorDial)	TutorMove
(Task, NoChange, Task)	TutorMove
(Task, NoChange, StudentDial)	NoMove
(Task, NoChange, TutorDial)	TutorMove
(StudentDial, Closer, Task)	TutorMove
(StudentDial, Closer, StudentDial)	TutorMove
(StudentDial, Closer, TutorDial)	TutorMove
(StudentDial, NoChange, Task)	NoMove
(StudentDial, NoChange, StudentDial)	NoMove
(StudentDial, NoChange, TutorDial)	NoMove
(NoAction, Closer, Task)	TutorMove
(NoAction, Closer, StudentDial)	TutorMove
(NoAction, Closer, TutorDial)	NoMove
(NoAction, Farther, Task)	NoMove
(NoAction, Farther, StudentDial)	TutorMove
(NoAction, Farther, TutorDial)	NoMove
(NoAction, NoChange, Task)	TutorMove
(NoAction, NoChange, StudentDial)	NoMove
(NoAction, NoChange, TutorDial)	NoMove
Initial	TutorMove
LowFrequency	TutorMove

Some noteworthy patterns emerge in the intervention policy learned from the corpus. For example, in seven of the eight states where the student is actively engaged in task actions (*Task*, *, *), the policy recommends that the tutor make a dialogue move. An excerpt from the corpus illustrating this strategy in a high learning gain session is shown in Figure 2, on lines 2-4. An excerpt from a low learning gain session showing tutor non-intervention during task progress is shown in Figure 3. In addition, among the states in which no action is currently being taken by the student and the last action was a tutor message, i.e., matching the pattern (*NoAction*, *, *TutorDial*), we find that the policy recommends that a tutor not make another consecutive dialogue move, regardless of how well the student is progressing on the task. However, Figure 2 shows that high learning gains are possible without strictly following this particular recommendation. Additional discussion on these recommendations can be found in [26].

5 Conclusion and Future Work

Current tutorial dialogue systems are highly effective, and matching the effectiveness of the most effective tutors is a driving force of tutorial dialogue research. This paper

presents a step toward rich, adaptive dialogue for supporting computer science learning by introducing a representation of task-oriented dialogue with unrestricted turn-taking in a reinforcement learning framework and presenting initial results of an automatically learned policy for when to intervene. The presented approach will inform the development of the JavaTutor tutorial dialogue system, whose initial policies will be learned based on the fixed human-human corpus described here.

Event	Tutor action and state transition
1. *Student is declaring a String variable named "aStringVariable".*	*NoMove* ↓ (Task, NoChange, Task)
2. *Tutor starts typing a message*	*TutorMove*
3. *1.5 seconds elapse, task action is complete.*	↓
4. **Tutor message:** That works, but let's give the variable a more descriptive name	(NoAction, Closer, TutorDial)
5. *Tutor starts typing a message*	*TutorMove*
6. *Student starts typing a message*	↓
7. **Student message:** ok	
8. **Tutor message:** Usually, the variable's name tells us what data it has stored	(NoAction, Closer, TutorDial)

Fig. 2. An excerpt from a high learning gain session.

Event	Tutor action and state transition
1. *Student has just attempted to implement the programming code needed to complete the task, with no tutor intervention.*	*NoMove* ↓ (NoAction, Closer, Task)
2. *Student starts typing a message*	*NoMove* ↓ (StudentDial, Closer, Task)
3. **Student message:** not sure if this is right…	*NoMove* ↓ (NoAction, Closer, StudentDial)

Fig. 3. An excerpt from a low learning gain session.

Further exploring of the state space via simulation and utilizing a more expressive representation of state are highly promising directions for future work. Other directions for future work include undertaking a more fine-grained analysis of the timing of interventions, which could inform the development of more natural interactions, as well as allowing for more nuanced intervention strategies. Additionally, these models should be enhanced with a more expressive representation of both dialogue and task. It is hoped that these lines of investigation will yield highly effective machine-learned policies for tutorial dialogue systems and that tutorial dialogue systems for computer science will make this subject more accessible to students of all grade levels.

Acknowledgements

The JavaTutor project team includes Eric Wiebe, Bradford Mott, Eun Young Ha, Joseph Grafsgaard, Alok Baikadi, Megan Hardy, Mary Luong, Miles Smaxwell, Natalie Kerby, Robert Fulton, Caitlin Foster, Joseph Wiggins, and Denae Ford. This work is supported in part by the National Science Foundation through Grants DRL-1007962 and CNS-1042468. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the participants, and do not necessarily represent the official views, opinions, or policy of the National Science Foundation.

References

1. Bloom, B.: The 2 sigma problem: the search for methods of group instruction as effective as one-to-one tutoring. Educational Researcher. 13, 4–16 (1984).
2. VanLehn, K., Graesser, A.C., Jackson, G.T., Jordan, P., Olney, A., Rosé, C.P.: When are tutorial dialogues more effective than reading? Cognitive Science. 31, 3–62 (2007).
3. Evens, M.W., Michael, J.: One-on-One Tutoring by Humans and Computers. Lawrence Erlbaum Associates, Mahwah, New Jersey (2005).
4. Heffernan, N.T., Koedinger, K.: The design and formative analysis of a dialog-based tutor. Workshop on Tutorial Dialogue Systems. pp. 23–34 (2001).
5. Forbes-Riley, K., Litman, D.: Adapting to student uncertainty improves tutoring dialogues. Proceedings of the International Conference on Artificial Intelligence in Education. pp. 33–40 (2009).
6. Kersey, C., Di Eugenio, B., Jordan, P., Katz, S.: KSC-PaL: A peer learning agent that encourages students to take the initiative. Proceedings of the Fourth Workshop on Innovative Use of NLP for Building Educational Applications. pp. 55–63 (2009).
7. Kumar, R., Rosé, C.P.: Architecture for Building Conversational Agents that Support Collaborative Learning. IEEE Transactions on Learning. 4, 21–34 (2011).
8. Jackson, G.T., Person, N.K., Graesser, A.C.: Adaptive Tutorial Dialogue in AutoTutor. ITS 2004 Workshop Proceedings on Dialog-based Intelligent Tutoring Systems. pp. 9–13 (2004).
9. D'Mello, S., Graesser, A.: AutoTutor and affective autotutor: Learning by talking with cognitively and emotionally intelligent computers that talk back. ACM Transactions on Interactive Intelligent Systems. 2, (2012).
10. Jonsdottir, G.R., Thorisson, K.R., Nivel, E.: Learning Smooth, Human-Like Turntaking in Realtime Dialogue. Proceedings of the 8th International Conference on Intelligent Virtual Agents. pp. 162–175 (2008).
11. Ward, N.G., Fuentes, O., Vega, A.: Dialog Prediction for a General Model of Turn-Taking. Proceedings of the International Conference on Spoken Language Processing (2010).

12. Raux, A., Eskenazi, M.: Optimizing the turn-taking behavior of task-oriented spoken dialog systems. ACM Transactions on Speech and Language Processing. 9, 1–23 (2012).
13. Bohus, D., Horvitz, E.: Multiparty Turn Taking in Situated Dialog: Study, Lessons, and Directions. Proceedings of the 12th Annual Meeting of the Special Interest Group in Discourse and Dialogue. pp. 98–109 (2011).
14. Morbini, F., Forbell, E., DeVault, D., Sagae, K., Traum, D.R., Rizzo, A.A.: A Mixed-Initiative Conversational Dialogue System for Healthcare. Proceedings of the 13th Annual Meeting of the Special Interest Group in Discourse and Dialogue. pp. 137–139 (2012).
15. Mitchell, C.M., Boyer, K.E., Lester, J.C.: From strangers to partners: examining convergence within a longitudinal study of task-oriented dialogue. Proceedings of the 13th Annual SIGDIAL Meeting on Discourse and Dialogue. pp. 94–98 (2012).
16. Ha, E.Y., Grafsgaard, J.F., Mitchell, C.M., Boyer, K.E., Lester, J.C.: Combining verbal and nonverbal features to overcome the "information gap" in task-oriented dialogue. Proceedings of the 13th Annual SIGDIAL Meeting on Discourse and Dialogue. pp. 246–256 (2012).
17. Grafsgaard, J.F., Fulton, R., Boyer, K.E., Wiebe, E., Lester, J.C.: Multimodal analysis of the implicit affective channel in computer-mediated textual communication. to appear in Proceedings of the 14th ACM international conference on Multimodal Interaction (2012).
18. Tetreault, J.R., Litman, D.J.: A Reinforcement Learning approach to evaluating state representations in spoken dialogue systems. Speech Communication. 50, 683–696 (2008).
19. Chi, M., VanLehn, K., Litman, D.: Do micro-level tutorial decisions matter: applying reinforcement learning to induce pedagogical tutorial tactics. Proceedings of the International Conference on Intelligent Tutoring Systems. pp. 224–234. (2010).
20. Marx, J.D., Cummings, K.: Normalized change. American Journal of Physics. 75, 87–91 (2007).
21. Sutton, R., Barto, A.: Reinforcement Learning. MIT Press, Cambridge, MA (1998).
22. Singh, S., Litman, D., Kearns, M., Walker, M.: Optimizing Dialogue Management with Reinforcement Learning: Experiments with the NJFun System. Journal of Artificial Intelligence Research. 16, 105–133 (2002).
23. Tetreault, J.R., Litman, D.J.: Using Reinforcement Learning to Build a Better Model of Dialogue State. Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics. pp. 289–296 (2006).
24. Anderson, J.R., Boyle, C.F., Corbett, A.T., Lewis, M.W.: Cognitive modeling and intelligent tutoring. Artificial Intelligence. 42, 7–49 (1990).
25. Johnson, W.L., Soloway, E.: PROUST: Knowledge-based program understanding. ICSE '84: Proceedings of the 7th international conference on Software engineering. pp. 369–380 (1984).
26. Mitchell, C.M., Boyer, K.E., Lester, J.C.: A Markov Decision Process Model of Tutorial Intervention in Task-Oriented Dialogue. To appear in Proceedings of the 16th International Conference on Artificial Intelligence in Education (2013).

Automatic Generation of Programming Feedback: A Data-Driven Approach

Kelly Rivers and Kenneth R. Koedinger

Carnegie Mellon University

Abstract. Automatically generated feedback could improve the learning gains of novice programmers, especially for students who are in large classes where instructor time is limited. We propose a data-driven approach for automatic feedback generation which utilizes the program solution space to predict where a student is located within the set of many possible learning progressions and what their next steps should be. This paper describes the work we have done in implementing this approach and the challenges which arise when supporting ill-defined domains.

Keywords: automatic feedback generation; solution space; computer science education; intelligent tutoring systems

1 Introduction

In the field of learning science, feedback is known to be important in the process of helping students learn. In some cases, it is enough to tell a student whether they are right or wrong; in others, it is better to give more details on why a solution is incorrect, to guide the student towards fixing it. The latter approach may be especially effective for problems where the solution is complex, as it can be used to target specific problematic portions of the student's solution instead of throwing the entire attempt out. However, it is also more difficult and time-consuming to provide.

In computer science education, we have been able to give students a basic level of feedback on their programming assignments for a long time. At the most basic level, students can see whether their syntax is correct based on feedback from the compiler. Many teachers also provide automated assessment with their assignments, which gives the student more semantic information on whether or not their attempt successfully solved the problem. However, this feedback is limited; compiler messages are notoriously unhelpful, and automated assessment is usually grounded in test cases, which provide a black and white view of whether the student has succeeded. The burden falls on the instructors and teaching assistants (TAs) to explain to students why their program is failing, both in office hours and in grading. Unfortunately, instructor and TA time is limited, and it becomes nearly impossible to provide useful feedback when course sizes become larger and massive open online courses grow more common.

Given this situation, a helpful approach would be to develop a method for automatically generating more content-based and targeted feedback. An automatic approach could scale easily to large class sizes, and would hopefully be able to handle a large portion of the situations in which students get stuck. This would greatly reduce instructor grading time, letting them focus on the students who struggle the most. Such an approach is easier to hypothesize than it is to create, since student solutions are incredibly varied in both style and algorithmic approach and programming problems can become quite complex. An automatic feedback generation system would require knowledge of how far the student had progressed in solving the problem, what precisely was wrong with their current solution, and what constraints were required in the final, correct solutions.

In this paper, we propose a method for creating this automatic feedback by utilizing the information made available by large corpuses of previous student work. This data can tell us what the most common correct solutions are, which misconceptions normally occur, and which paths students most often take when fixing their bugs. As the approach is data-driven, it requires very little problem-specific input from the teacher, which makes it easily scalable and adaptable. We have made significant progress in implementing this approach and plan to soon begin testing it with real students in the field.

2 Solution Space Representation

Our method relies upon the use of *solution spaces*. A solution space is a graph representation of all the possible paths a student could take in order to get from the problem statement to a correct answer, where the nodes are candidate solutions and the edges are the actions used to move from one solution state to another. Solution spaces can be built up from student data by extracting students' learning progressions from their work and inserting them into the graph as a directed chain. Identical solutions can be combined, which will represent places where a student has multiple choices for the next step to take, each of which has a different likelihood of getting them to the next answer.

A solution space can technically become infinitely large (especially when one considers paths which do not lead to a correct solution), but in practice there are common paths which we expect the student to take. These include the learning progression that the problem creator originally intended, other progressions that instructors and teaching assistants favor, and paths that include any common misconceptions which instructors may have recognized in previous classes. If we can recognize when a student is on a common path (or recognize when the student has left the pack entirely) we can give them more targeted feedback on their work.

While considering the students' learning progressions, we need to decide at what level of granularity they should be created. We might consider very small deltas (character or token changes), or very large ones (save/compile points or submissions), depending on our needs. In our work we use larger deltas in order to examine the points at which students deliberately move from one state to the

next; every time a student saves, they are pausing in their stream of work and often checking to see what changes occur in their program's output. Of course, this approach cannot fully represent all of the work that a student does; we cannot see the writing they are doing offline or hear them talking out ideas with their TAs. These interactions will need to be inferred from the changes in the programs that the student writes if we decide to account for them.

It is simple to create a basic solution space, but making the space *usable* is a much more difficult task. Students use different variable names, indentations, and styles, and there are multitudes of ways for them to solve the same problem with the same general approach. In fact, we do not want to see two different students submitting exactly the same code– if they do, we might suspect them of cheating! But the solution space is of no use to us if we cannot locate new students inside of it. Therefore, we need to reduce the size of the solution space by combining all semantically equivalent program states into single nodes.

Many techniques have been developed already for reducing the size of the solution spaces of ill-defined problems. Some represent the solution states with sets of constraints [5], some use graph representations to strip away excess material [4], and others use transformations to simplify solution states [9, 8]. We subscribe to the third approach by transforming student programs into *canonical forms* with a set of normalizing program transformations. These transformations simplify, anonymize, and order the program's syntax without changing its semantics, mapping each individual program to a normalized version. All transformations are run over abstract syntax trees (ASTs), which are parse trees for programs that remove extraneous whitespace and comments. If two different programs map to the same canonical form, we know that they must be semantically equivalent, so this approach can safely be used to reduce the size of the solution space.

Example Let us consider a very simple programming problem from one of the first assignments of an introductory programming class. The program takes as input an integer between 0 and 51, where each integer maps to a playing card, and asks the student to return a string representation of that card. The four of diamonds would map to 15, as clubs come before diamonds; therefore, given an input of 15, the student would return "4D". This problem tests students' ability to use mod and div operators, as well as string indexing. One student's incorrect solution to this problem is shown in Figure 1.

```
def intToPlayingCard(value):
    faceValue = value%13
    #use remainder as an index to get the face
    face = "23456789YJQKA"[faceValue]
    suitValue = (value-faceValue)%4
    suit = "CDHS"[suitValue]
    return face+suit
```

Fig. 1. A student's attempt to solve the playing card problem.

To normalize this student's program, we first extract the abstract syntax tree from the student's code, as is partially shown in Figure 2. All of the student's variables are immediately anonymized; in this case, 'value' will become 'v0', 'faceValue' will be 'v1', etc. We then run all of our normalizing transformations over the program, to see which ones will have an effect; in this case, the only transformation used is *copy propagation*. This transformation reduces the list of five statements in the program's body to a single return statement by copying the value assigned to each variable into the place where the variable is used later on. Part of the resulting canonical form is displayed in Figure 2. The new tree is much smaller, but the program will have the same effect.

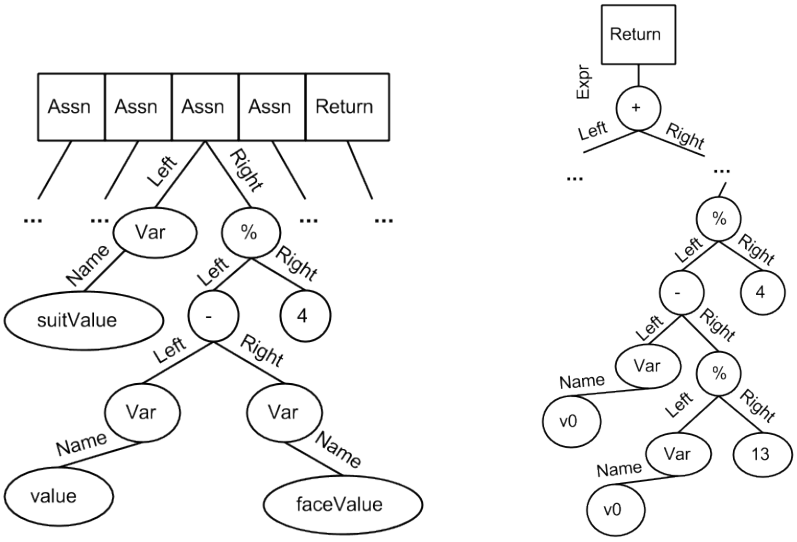


Fig. 2. Subparts of the student's ASTs, before (left) and after (right) normalization.

We have already implemented this method of solution space reduction and tested it with a dataset of final submissions from a collection of introductory programming problems. The method is quite effective, with the solution space size being reduced by slightly over 50% for the average problem [7]. However, we still find a long tail of singleton canonical forms existing in each problem's solution space, usually due to students who found strange, unexpected approaches or made unconventional mistakes. This long tail of unusual solutions adds another layer of complexity to the problem, as it decreases the likelihood that a new student solution will appear in the old solution space.

Our work so far has concentrated only on final student submissions, not on the paths students take while solving their problems. This could be seen as problematic, as we are not considering the different iterations a student might go through while working. However, our very early analysis of student learning progressions from a small dataset has indicated that students are not inclined to use incremental approaches. The students we observed wrote entire programs in single sittings, then debugged until they could get their code to perform correctly.

This suggests that our work using final program states may be close enough to the real, path-based solution space to successfully emulate it.

We note that the solution space is easiest to traverse and create when used on simple problems; as the required programs become longer, the number of individual states in the space drastically increases. We believe it may be possible to address this situation by breaking up larger problems into hierarchies of subproblems, each of which may map to a specific chunk of code. Then each subproblem can have its own solution space that may be examined separately from the other subproblems, and feedback can be assigned for each subproblem separately.

3 Feedback Generation

Once the solution space has been created, we need to consider how to generate feedback with it. The approach we have adopted is based on the Hint Factory [1], a logic tutor which uses prior data to give stuck students feedback on how to proceed. In the Hint Factory, each node of the solution space was the current state of the student's proof, and each edge was the next theorem to apply that would help the student move closer to the complete proof. The program used a Markov Decision Process to decide which next state to direct the student towards, optimizing for the fastest path to the solution.

Our approach borrows heavily from the Hint Factory, but also expands it. This is due to the ill-defined nature of solving programming problems, which specifies that different solutions can solve the same problem; this complicates several of the steps used in the original logic tutor. In this section we highlight three challenges that need to be addressed in applying the Hint Factory methodology to the domain of programming, and describe how to overcome each of them.

Other attempts have been made at automatic generation of feedback, both in the domain of programming and in more domain-general contexts. Some feedback methods rely on domain knowledge to create messages; Paquette et al.'s work on supporting domain-general feedback is an example of this [6]. Other methods rely instead on representative solutions, comparing the student's solution to the expected version. Examples here include Gerdes et al.'s related work on creating functional tutoring systems (which use instructor-submitted representative solutions) [2] and Gross et al.'s studies on using clustering to provide feedback (which, like our work, use correct student solutions) [3]. Though our work certainly draws on many of the elements used in these approaches, we explore the problem from a different angle in attempting to find entire paths to the closest solution (which might involve multiple steps), rather than jumping straight from the student's current state to the final solution. Whether this proves beneficial will remain to be seen in future studies.

3.1 Ordering of Program States

Our first challenge relates to the process of actually mapping out the suggested learning progressions for the student. Even after reducing the size of the solution space, there are still a large number of distinct solutions which are close yet not connected by previously-found learning paths. These close states can be helpful, as they provide more opportunities for students to switch between different paths while trying to reach the solution. Therefore, we need to connect each state to those closest to it, then determine which neighboring state will set the student on the best path to get to a final solution.

One obvious method for determining whether two states are close to each other would involve using tree edit distance, to determine how many changes needed to be made. However, this metric does not seem to work particularly well in practice; the weight of an edit is difficult to define, which makes comparing edits non-trivial. Instead, we propose the use of string edit distance (in this case, Levenshtein distance) to determine whether two programs are close to each other. To normalize the distances between states, we calculate the percentage similarity with $(l - distance)/l$ (where l is the length of the longer program); this ensures that shorter programs do not have an advantage over longer ones and results from different problems can easily be compared to each other. Once the distances between all programs have been calculated, a cut-off point can be determined that will separate close state pairs from far state pairs. Our early experimentation with this method shows that it is efficient on simple programs and produces pairs of close states for which we can generate artificial actions.

Once the solution space has been completely generated and connected, we need to consider how to find the best path from state A to state B. The algorithm for finding this will be naturally recursive in nature– the best path from A to B will be the best element of the set of paths S, where S is composed of paths from each neighbor of A to B. Paths which require fewer intermediate steps will be preferred, as they require the student to make less changes, but we also need to consider the distances between the program states. We can again use string edit distance to find these distances, or we can use the tree edits to look at the total number of individual changes required. Finally, we can use test cases to assign correctness parameters to each program state (as there are certainly some programs which are more incorrect than others); paths which gradually increase the number of test cases that a student passes may be considered more beneficial than paths which jump back and forth, as the latter paths may lead to discouragement and frustration in students.

Example In the previous section, we had found the canonical form for the student's solution; that form was labeled #22 in the set of all forms. As we were using a dataset of final submissions, we had no learning progressions to work with, we computed the normalized Levenshtein distance between each pair of states and connected those which had a percentage similarity of 90% or higher, thus creating a progression graph.

In Figure 3, we see that state #22 was connected to three possible next states: #4, #34, and #37. We know that #34 is incorrect, so it does not seem like a good choice; on the other hand, #4 and #37 are equally close to #22 and are both correct. State #37 had been reached by thirty students, while state #4 had only been reached by four; since #37 is more commonly used, it is probably the better target solution for the student.

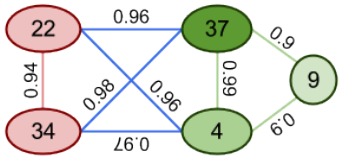


Fig. 3. The program state graph surrounding state #22. Red nodes are incorrect, green correct; a darker node indicates that more students used that approach.

3.2 Generating content deltas between states

Next, we face the challenge of determining how to extract the content of the feedback message from the solution space. The feedback that we give the student comes from the edge between the current and target states, where that edge represents the actions required to get from one state to the other. In well-defined domains, these actions are often simple and concrete, but they become more complex when the problems are less strictly specified.

Before, we used string distance to determine how similar two programs were, in order to find distances quickly and easily. Now that we need to know what the differences actually are, we use tree edits to find the additions, deletions, and changes required to turn one tree into another. It is moderately easy to compute these when comparing ordinary trees, but ASTs add an extra layer of complexity as there are certain nodes that hold lists of children (for example, the bodies of loops and conditionals), where one list can hold more nodes than another. To compare these nodes, we find the maximal ordered subset of children which appear in both lists; the leftover nodes can be considered changes.

After we have computed these edits, we can use them to generate feedback for the student in the traditional way. Cognitive tutors usually provide three levels of hints; we can use the same approach here, first providing the location of an error, then the token which is erroneous, and finally what the token needs to be changed to in order to fix the error. In cases where more than one edit needs to be made the edits can be provided to the student one at a time, so that the student has a chance to discover some of the problems on their own.

It may be possible to map certain edit patterns to higher-level feedback messages, giving students more conceptual feedback. Certain misconceptions and mistakes commonly appear in novice programs; accidental use of integer division and early returns inside of loops are two examples. If we can code the patterns that these errors commonly take (in these cases, division involving two integer values and return statements occurring in the last line of a loop's body), we can

provide higher-level static feedback messages that can be provided to students instead of telling them which values to change. This may help them recognize such common errors on their own in future tasks.

Example To generate the feedback message in our continuing example, we find the tree edits required to get from state #22 to state #37. These come in two parts: one a simple change, the other a more complex edit. Both are displayed in Figure 4. The first change is due to a typo in the string of card face values that the student is indexing (Y instead of T for ten); as the error occurs in a leaf node (a constant value), pointing it out and recommending a change is trivial. Such a feedback message might look like this: *In the return statement, the string "23456789YJQKA" should be "23456789TJQKA".*

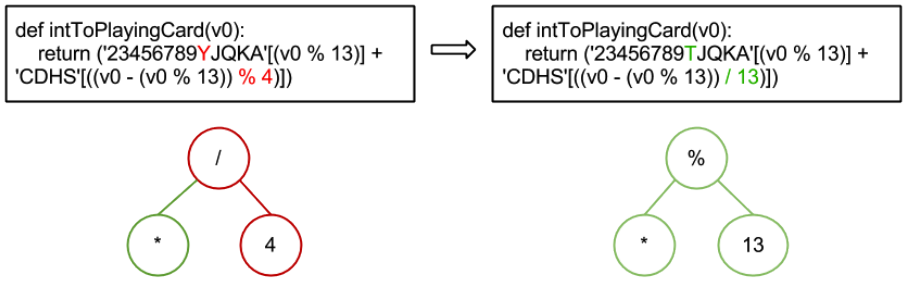


Fig. 4. The change found between the two programs, represented in text and tree format (with * representing a further subexpression).

The second error is due to a misconception about how to find the index of the correct suit value. In the problem statement, the integer card values mapped cards first by face value and then by suit; all integers from 0 to 12 would be clubs, 13 to 23 would be diamonds, etc. This is a step function, so the student should have used integer division to get the correct value. In a terrible twist of fate, this part of the student's code will actually work properly; $v0 - (v0\%13)$ returns the multiple of 13 portion of v0, and the first four multiples of 13 (0, 13, 26, and 39) each return the correct index value when modded by 4 (0, 1, 2, and 3). Still, it seems clear that the student is suffering from a missing piece of knowledge, as it would be much simpler to use the div operator.

In the AST, the two solutions match until they reach the value used by the string index node. At that point, one solution will use mod while the other uses div, and one uses a right operand of 4 while the other uses a right operand of 13. It's worth noting, however, that both use the same subexpression in the left operand; therefore, in creating feedback for the student, we can leave that part out. Here, the feedback message might be this: *In the right side of the addition in the return statement, use div instead of mod.* The further feedback on changing 4 to 13 could be provided if the student needed help again later.

3.3 Reversing deltas to regain content

Finally, we need to take the content of the feedback message which we created in the previous part and map it back to the student's original solution. If the student's solution was equivalent to the program state, this would be easy– however, because we had to normalize the student programs, we will need to map the program state back up to the individual student solution in order to create their personal feedback message.

In some situations, this will be easy. For example, it's possible that a student solution only had whitespace cleaned up and variable names anonymized; if this was the case, the location of the code would remain the same, and variable names could be changed in the feedback message easily. In many other cases, the only transformations applied would be ordering and propagation functions; for these, we can keep track of where each expression occurred in the original program, then map the code segments we care about back to their positions in the original code. Our running example falls into this "easy" category; even though the student's program looks very different from the canonical version, we only need to unroll the copy propagation to get the original positions back.

Other programs will present more difficulties. For example, any student program which has been reduced in size (perhaps through constant folding, or conditional simplification) might have a feedback expression which needs to be broken into individual pieces. One solution for this problem would be to record each transformation that is performed on a program, then backtrack through them when mapping feedback. Each transformation function can be paired with a corresponding "undo" function that will take the normalized program and a description of what was changed, then generate the original program.

Example All of the program transformations applied to the original student program in order to produce state #22 were copy propagations; each variable was copied down into each of its references and deleted, resulting in a single return statement. To undo the transformations, the expressions we want to give feedback on ('23456789YJQKA' and $(v0-(v0\%13))\%4$) must be mapped to the variables that replace them– face and suit (where suit is later mapped again to suitValue). We can then examine the variable assignment lines to find the original location in which the expression was used (see Figure 5), which maps the expressions to lines 2 and 4. The first expression's content is not modified, but the second changes into $(value - faceValue)\%4$. This change lies outside of the feedback that we are targeting, so it does not affect the message.

After the new locations have been found, the feedback messages are correspondingly updated by changing the location that the message refers to. In this case, the first feedback message would change to: In the **second line**, the string '23456789YJQKA' should be '23456789TJQKA'. The second would become: In the **fourth line**, use div instead of mod.

```
def intToPlayingCard(v0):
    return ('23456789YJQKA'[(v0 % 13)] +
'CDHS'[((v0 - (v0 % 13)) % 4)])
```

```
def intToPlayingCard(value):
    faceValue = value%13
    face = "23456789YJQKA"[faceValue]
    #use remainder as an index to get the face
    suitValue = (value-faceValue)%4
    suit = "CDHS"[suitValue]
    return face+suit
```

Fig. 5. A comparison of the canonical (left) and original (right) programs. The code snippets we need to give feedback on are highlighted.

4 Conclusion

The approach we have described utilizes the concept of solution spaces to determine where a new student is in their problem-solving process, then determines what feedback to provide by traversing the space to find the nearest correct solution. Representing the solution space has been implemented and tested, but generating feedback is still in progress; future work will determine how often it is possible to provide a student with truly useful and usable feedback.

Acknowledgements. This work was supported in part by Graduate Training Grant awarded to Carnegie Mellon University by the Department of Education (# R305B090023).

References

1. Barnes, Tiffany, and John Stamper. "Toward automatic hint generation for logic proof tutoring using historical student data." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2008.
2. Gerdes, Alex, Johan Jeuring, and Bastiaan Heeren. "An interactive functional programming tutor." Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. ACM, 2012.
3. Gross, Sebastian, et al. "Cluster based feedback provision strategies in intelligent tutoring systems." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
4. Jin, Wei, et al. "Program representation for automatic hint generation for a data-driven novice programming tutor." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
5. Le, Nguyen-Thinh, and Wolfgang Menzel. "Using constraint-based modelling to describe the solution space of ill-defined problems in logic programming." Advances in Web Based LearningICWL 2007. Springer Berlin Heidelberg, 2008. 367-379.
6. Paquette, Luc, et al. "Automating next-step hints generation using ASTUS." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
7. Rivers, Kelly, and Kenneth R. Koedinger. "A canonicalizing model for building programming tutors." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
8. Weragama, Dinesha, and Jim Reye. "Design of a knowledge base to teach programming." Intelligent Tutoring Systems. Springer Berlin Heidelberg, 2012.
9. Xu, Songwen, and Yam San Chee. "Transformation-based diagnosis of student programs for programming tutoring systems." Software Engineering, IEEE Transactions on 29.4 (2003): 360-384.

JavaParser: A Fine-Grain Concept Indexing Tool for Java Problems

Roya Hosseini, Peter Brusilovsky

University of Pittsburgh, Pittsburgh, USA
{roh38,peterb}@pitt.edu

Abstract. Multi-concept nature of problems in the domain of programming languages requires fine-grained indexing which is critical for sequencing purposes. In this paper, we propose an approach for extracting this set of concepts in a reliable automated way using JavaParser tool. To demonstrate the importance of fine-grained sequencing, we provide an example showing how this information can be used for problem sequencing during exam preparation.

Keywords: indexing, sequencing, parser, java programming

1 Introduction

One of the oldest functions performed by adaptive educational systems is guiding students to most appropriate educational problems at any time of their learning process. In classic ICAI and ITS system this function was known as task sequencing [1; 6]. In modern hypermedia-based systems it is more often referred as navigation support. The intelligent decision mechanism behind these approaches is typically based on a domain model that decomposes the domain into a set of knowledge units. This domain model serves as a basis of student overlay model and as a dictionary to index educational problems or tasks. Considering the learning goal and the current state of student knowledge reflected by the student model, various sequencing approaches are able to determine which task is currently the most appropriate.

An important aspect of this decision process is the granularity of the domain model and the related granularity of task indexing. In general, the finer are the elements of the domain model and the more precise is task indexing, the better precision could be potentially offered by the sequencing algorithm in determining the best task to solve. However, fine-grained domain models that dissect a domain into many dozens to many hundreds of knowledge units are much harder to develop and to use for indexing. As a result, many adaptive educational systems use relatively coarse-grained models where a knowledge unit corresponds to a considerably-sized topic of learning material, sometimes even a whole lecture.. With these coarse-grain models, each task is usually indexed with just 1-3 topics. In particular, this approach is used by the majority of adaptive systems in the area of programming [2; 4; 5; 7].

Our past experience with adaptive hypermedia systems for programming [2; 4] demonstrated that adaptive navigation support based on coarse grain problem indexing is surprisingly effective way to guide students over their coursework, yet it doesn't work well in special cases such as remediation or exam preparation. In these

special situations students might have a reasonable overall content understanding (i.e., coarse-grain student model registers good knowledge), while still possessing some knowledge gaps and misconceptions that could be only registered using a finer-grain student model.. In this situation only a fine-grain indexing and sequencing is able to suggest learning tasks that can address these gaps and misconceptions.

To demonstrate the importance of fine-grained indexing, we can look at an example of a system called *Knowledge Maximizer* [3] that uses fine-grain concept-level problem indexing to identify gaps in user knowledge for exam preparation. This system assumes a student already did considerable amount of work and the goal is to help her define gaps in knowledge and try to fix that holes as soon as possible. Fig. 1 represents the Knowledge Maximizer interface. The question with the highest rank is shown first. User can navigate the ranked list of questions using navigation buttons at the top. Right side of the panel shows the list of fine-grained concepts covered by the question. The color next to each concept visualizes the student's current knowledge level (from red to green). Evaluation results confirm that using fine-grained indexing in Knowledge Maximizer has positive effect on students' performance and also shorten the time for exam preparation.

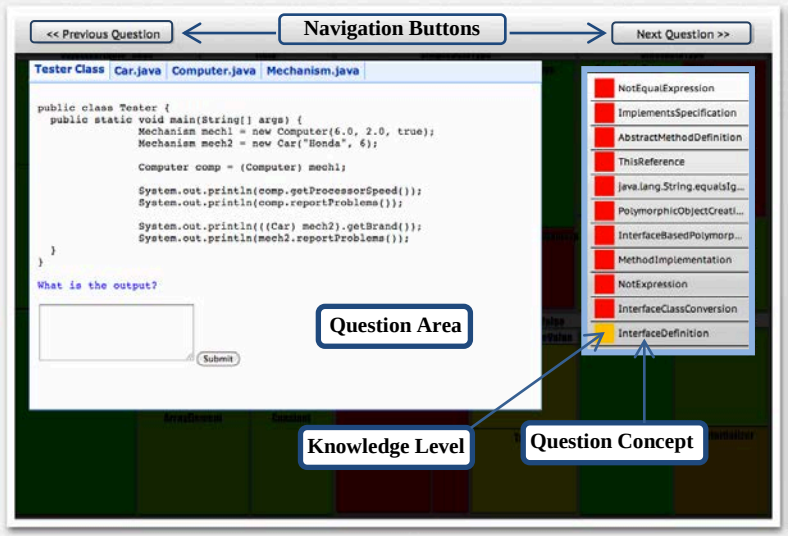


Fig. 1. The Knowledge Maximizer interface.

The problem with finer-grain indexing, such as used by the Knowledge Maximizer is the high cost of indexing. While fine-grain domain model has to be developed just once, the indexing process has to be repeated for any new question. Given that most complex questions used by the system include over 90 concepts each, the high cost of indexing effectively prevents an expansion of the body of problems. To resolve this problem, we developed an automatic approach for fine-grained indexing for programming problems in Java based on program parsing. This approach is presented in the following section.

2 Java Parser

Java parser is a tool that we developed to index Java programs with concepts of Java ontology developed by our group (http://www.sis.pitt.edu/~paws/ont/java.owl). This tool provides the user with semi-automated indexing support during developing new learning materials for the Java Programming Language course. This parser is developed using the Eclipse Abstract Syntax Tree framework. This framework generates an Abstract Syntax Tree (AST) that entirely represents the program source. AST consists of several nodes each containing some information known as *structural properties*. For example, Fig. 2 shows structural properties for the following method declaration:

```
public void start(BundleContext context) throws Exception {
        super.start(context);
}
```

Fig. 2. Structural properties of a method declaration

Table 1. Sample of JavaParser output

Source	Output
`public void start(BundleContext context) throws Exception { super.start(context); }`	Super Method Invocation, Public Method Declaration, Exception, Formal Method Parameter, Single Variable Declaration, Void

After building the tree using *Eclipse AST API*, the parser performs a semantic analyzed using the information in each node. This information is used to identify fine-grained indexes for the source program. Table 1 shows the output concepts of *JavaParser* for the code fragment mentioned above. Note that the goal of the parser is

to detect the lowest level ontology concepts behind the code since the upper level concepts can be deduced using ontology link propagation. For example, as you see in Table 1, parser detects “void” and “main” ignoring upper-level concept of “modifier”.

We compared the accuracy of *JavaParser* with manual indexing for 103 Java problems and found out that our parser was able to index 93% of the manually indexed concepts. Therefore, automatic parser can replace time-consuming process of manual indexing with a high precision and open the way to community-driven problem authoring and targeted expansion of the body of problems.

3 Conclusion

Having fine-grained indexing for programming problems is necessary for better sequencing of learning materials for students; however, the cost of manual fine-grained indexing is prohibitively high. In this paper, we presented a fine grained indexing approach and tool for automatic indexing of Java problems. We also showed an application of fine-grained problem indexing during exam preparation where small size of knowledge units is critical for finding sequence of problems that fills the gaps in student knowledge. Results show that proposed automatic indexing tool can offer the quality of indexing that is comparable with manual indexing by expert for a fraction of its cost.

References

1. Brusilovsky, P.: A framework for intelligent knowledge sequencing and task sequencing. In: Proc. of Second International Conference on Intelligent Tutoring Systems, ITS'92. Springer-Verlag (1992) 499-506
2. Brusilovsky, P., Sosnovsky, S., Yudelson, M.: Addictive links: The motivational value of adaptive link annotation. New Review of Hypermedia and Multimedia 15, 1 (2009) 97-118
3. Hosseini, R., Brusilovsky, P., Guerra, J.: Knowledge Maximizer: Concept-based Adaptive Problem Sequencing for Exam Preparation. In: Proc. of the 16th International Conference on Artificial Intelligence in Education. (2013) In Press
4. Hsiao, I.-H., Sosnovsky, S., Brusilovsky, P.: Guiding students to the right questions: adaptive navigation support in an E-Learning system for Java programming. Journal of Computer Assisted Learning 26, 4 (2010) 270-283
5. Kavcic, A.: Fuzzy User Modeling for Adaptation in Educational Hypermedia. IEEE Transactions on Systems, Man, and Cybernetics 34, 4 (2004) 439-449
6. McArthur, D., Stasz, C., Hotta, J., Peter, O., Burdorf, C.: Skill-oriented task sequencing in an intelligent tutor for basic algebra. Instructional Science 17, 4 (1988) 281-307
7. Vesin, B., Ivanović, M., Klašnja-Milićević A., Budimac, Z.: Protus 2.0: Ontology-based semantic recommendation in programming tutoring system. Expert Systems with Applications 39, 15 (2012) 12229-12246