Visualizing Live Collaboration in the Classroom with AMOEBA

Matthew Berland, University of Wisconsin–Madison, Madison, WI, mberland@wisc.edu
Carmen Petrick Smith, University of Vermont, Burlington, VT, carmen.smith@uvm.edu
Don Davis, University of Wisconsin–Madison, Madison, WI, dgdavis@wisc.edu

Abstract: Collaboration in computer science class is vital for supporting novices, but few tools support substantive collaboration during in-class programming. In this work, we describe the implementation of AMOEBA – a new system that helps computer science instructors support student collaboration through live visualization of similarities in students' programs. We explore a single, 40-minute session of students learning to program for the first time. In that session, the teacher used AMOEBA to pair up students with similar programs to work together successfully; paired students' programs both became more similar to their partner and, on average, scored more goals. We then discuss implications and future directions for AMOEBA, including ways that it could be adapted to better support collaboration.

Introduction

Live classroom collaboration on creative computer programming projects is hard to represent. There are currently no widespread commercial tools available that provide real time analyses and visualization of student collaboration to support teachers' facilitation of students learning to program. In this work, we describe the design and implementation of AMOEBA – a new system that helps computer science instructors support their students in collaborating to write code. In this paper, we explore how a teacher supported collaboration with AMOEBA in a single, 40-minute session with novice students learning to program.

The design hypothesis of AMOEBA is that by visualizing uncommon similarities between individual students' program code as they produce it, the teacher or facilitator can match students who might be able to help each other progress. This hypothesis is controversial, as the vast majority of research into program code similarity deals instead with preventing cheating (e.g., Lancaster & Culwin, 2004). Such work fails to account for significant research in learning sciences and computer science education that has repeatedly confirmed that working in concert with other students of similar ability level can accelerate learning and make understanding more robust (e.g., Nagappan et al., 2003). This paper is intended to serve as a proof-of-concept that such collaboration visualization systems can add value to a computer science class.

Collaboration in the Computer Science Classroom

In computer science (CS) education, there has been an increased call for promoting collaboration in the CS classroom (e.g., Teague & Roe, 2008), especially for underrepresented students (Hug, Thiry, & Tedford, 2011). Analyses of how this collaboration is realized and its benefits have been frequently documented (Preston, 2005). However, among the software tools available to CS educators, there are a preponderance of tools to catch plagiarists (see Lancaster & Culwin, 2004) and few (if any) to determine if students are collaborating, though the evidence needed might be quite similar. Researchers have identified varying benefits of CS collaboration arising from social, cognitive, and affective factors (e.g., Teague & Roe, 2008). Collaborative programming approaches (such as pair programming) have been shown to contribute to novice student success, improved program quality, improved problem solving performance, greater productivity, and improved CS matriculation and persistence (Nagappan et al., 2003). Furthermore, though the benefits of collaborative programming have been frequently documented, operationalization of collaborative programming tends to focus on more general aspects such as assessment and grouping strategies (Preston, 2005).

AMOEBA

In order to better leverage the benefits of collaboration in computer science classrooms, we developed AMOEBA. Though AMOEBA was initially designed for post hoc analyses of collaboration (Berland, Martin, Benton, & Petrick, 2012), we realized that AMOEBA could be an effective tool for teachers to use in real-time in their classrooms. AMOEBA works by showing the current status of students in the classroom, with a circle representing each student in the class on the screen (see Figure 1). The teacher can change the location of the circles so as to mimic the physical position of the students in the classroom. As students begin to write programs, an *edge* (connecting line) appears between any two students whose program code shares statistically uncommon elements (the *similarity* metric is described below). This indicates to the teacher that two students have a 'surprisingly unlikely' amount of similarity between their programs. The level of similarity that will create a line between two students can be changed dynamically by the teacher at any time by adjusting a slider ('connection-threshold' in Figure 1), and each edge is labeled with the most unlikely similar elements between the two students' programs. Furthermore, AMOEBA allows teachers to track their interactions with students.

When a teacher uses AMOEBA to inform an instructional decision (such as pairing two students with similar code), the teacher can click on the students' circles; these clicks record the students' usernames and the time of the interaction allowing the teacher to track which students were paired together at which times.

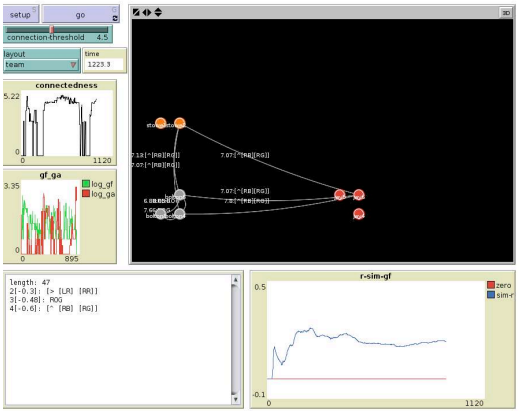

Figure 1. Screenshot of AMOEBA interface.

Methods

The participants in this study were 23 undergraduate students with no formal programming experience who were part of an educational technology course at a large public university. The study took place during a unit on mobile learning environments in which the students were testing and evaluating mobile learning applications.

Materials

IPRO is a mobile, social programming environment in which each student programs a virtual robot to play soccer with and against the other students. In this study, each student ran IPRO on an iPad. Within the IPRO environment, students are encouraged to gesture and physically mimic the actions of the agent they are programming, as this has been shown to benefit students' learning (Petrick, Berland, & Martin, 2011). Additionally, the mobile nature of the IPRO environment affords greater collaboration as two students working in separate areas of the room can physically move to share ideas, bringing their programs with them on their iPads. All activity by any student in IPRO is captured in the *log* – the primary source of data in this study.

Collaboration in an IPRO classroom builds upon students' "positive interdependence" (Preston, 2005), as each student works toward the visible goal of programming a robot to score (or block) goals both for herself and for her team. Moreover, the environment is intended to build upon special affordances of group interactions such as distributed cognition that may lead to higher quality programs (Petrick, Berland, & Martin, 2011). Building upon these and other best practices of CS collaboration, we sought to improve visualization and oversight of student interactions, which is a noted limitation of existing work (see Preston, 2005).

Procedures

This study was conducted during a 40-minute session as part of the students' normal class. Students were given a handout with a key to the IPRO 'primitives' (core elements). Then the classroom teacher, the second author of this paper, described the IPRO application and its use in computer science classes. A large screen displayed a series of simple IPRO programs, and the teacher led the students in physically enacting the movements of the robot, at which point each student logged into IPRO with a unique login ID. Students were assigned to one of 4 teams of 5-6 students each. The teacher demonstrated how to write basic IPRO code. Then the students were given time to work on their programs. Initially, most of the students worked individually.

While the class was programming, the teacher assisted with technical issues, but she did not provide any help writing code. During this time, the teacher also monitored AMOEBA. After approximately ten minutes of unsupervised coding time, the teacher asked pairs of students with high *similarity*, indicated by an AMOEBA *edge*, to work together. Each time the teacher directed two students to work with each other, she also marked it in AMOEBA by clicking on the circles representing those students. Of the 23 students in the class, the teacher directed 4 pairs of students to work together. The remaining 15 students were not asked to nor restricted from working together. At the end of the programming session, there were two matches in which the four different teams faced off against each other. The matches were projected on a large screen for the class to watch.

Measures

The *similarity* metric is based on Dunning's (1993) metric of *surprise* in computational linguistics. Given all sub-trees of the parse tree of two students' code, the *similarity* of two students is the calculated *surprise* of the maximally *surprising* isomorphic sub-tree of the two students normalized to the percent of the maximum

observed *surprise*. The metric is similar to *tf-idf* (Salton, 1989), which is the number of terms in common times the inverse frequency of that term in the corpus as a whole. Conceptually, this similarity metric is meaningful because two programs with identical code are solving a problem similarly (by definition). Program code, even in a limited language like IPRO, can vary enormously. In C++, there are infinite variations of source code that could produce 'hello world'. That said, (almost) every one of those programs will have 'int main(...)' at the beginning – that code is common to almost every C++ program ever written. By using a similarity metric that matches isomorphic code but discounts code provided by teachers, required code, or obvious code (such as 'int main(…)' in C++), we can find students who are using similar logic and approaches. Figure 1 (above) shows edges between students where the similarity between two programs is unlikely to happen by chance; the threshold for likelihood is dynamic and can be changed by the teacher at any time with a slider.

Every time a student made any change to their program, the log registered a new *program state*. We evaluated changes to students' programs over time by analyzing differences in program states over time. *Program length* is determined for each program state by counting the number of primitives in the state of the program, which is similar to the number of lines of code in other programming languages.

We measured the *program quality* (PQ) of each program state of each student's program by running the state of the program in a simulator with various scenarios for approximately 10,000 turns per program state. The simulator then generates a number for "goals scored by the robot" (GF) and a number for "goals scored against the robot" (GA). PQ is calculated by subtracting GA from GF (PQ = GF - GA).

Results

In reporting results, we primarily compared paired students with non-paired students starting from the approximate time point when the pairing began. Additionally, we compared the level of similarity of program states of paired-students to their respective partners before and after the students were paired. Figure 2 shows average values of GA, GF, length, and similarity for both non-paired (left) and paired (right) students over time.

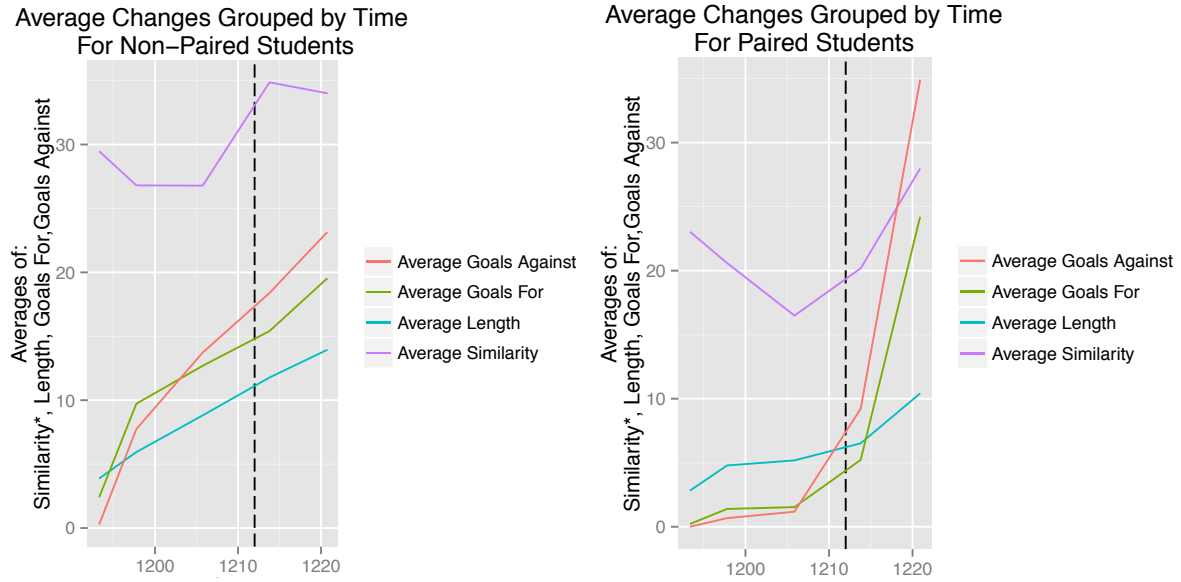

Figure 2. Average change of program descriptors over time for non-paired (left) and paired (right) students. The dotted vertical line marks the time at which the teacher paired students.

Similarity. The mean similarity scores for paired students to their partner before being paired was 19% and after being paired, it rose to 27%. The mean similarity score of all scores for non-paired students to one another before pairing time was 27%. These mean scores increased for both paired and non-paired students (as per Table 1). A closer examination of similarity in relation to time indicates a moderate to strong correlation of similarity to partner for paired students (r=0.42); whereas there is no significant correlation of time to overall similarity to one another (r=-0.16) for non-paired students. In other words, after two students were paired together, their programs tended to become more similar.

Length. Non-paired students tended to write longer programs than paired students (See Table 1). Among paired students, there was a large significant correlation between time and program length (r=0.43) and a large correlation among non-paired students as well (r=0.70). Thus, program length tended to increase over time for both paired and non-paired students.

Program Quality, Goals For and Goals Against. The paired students had lower average program quality than the non-paired students (Table 1). The paired students had a higher average GF and a higher average GA; that means that the programs that they were writing both scored goals more frequently and were

scored against more frequently – better offense, worse defense. Both paired and non-paired students exhibited similar trends among PQ, GF, and GA. The PQ for both groups was negatively correlated over time (r_{paired}=-0.21; $r_{nonpaired}$=-0.19), and both groups exhibited small correlations between time and GF (r_{paired}=0.26; $r_{nonpaired}$=0.21). Both paired and non-paired students' programs exhibited small correlations between GA and time (r_{paired}=0.25; $r_{nonpaired}$=0.22).

Table 1: Averages of Similarity, Length, Program Quality (PQ), Goals For (GF), and Goals Against (GA) across students' programs, based on whether they were paired or not.

	Similarity Before Pairing	Similarity After Pairing	Length	PQ	GF	GA
Paired Students	19%	26%	9.3	-18.06	38.05	56.11
Non-Paired Students	27%	34%	13.2	-4.96	26.77	31.47

Discussion

Overall, we found that the instructor had little difficulty using AMOEBA in the classroom, though she had not used it before. AMOEBA has very few "moving parts" – it does not require oversight. Quickly, and without training, the teacher was able to use the software to identify students with similar programs to match them together. Simply, our data suggest that the experiment was successful in prompting the teacher to encourage collaboration, and that collaboration was successful in helping students write better program code. The conclusion that paired students actually worked together is almost too obvious to state as a claim – it is a core assumption of innumerable classroom teachers. However, evidence exists that being conscious of how and when people are sharing can aid in collaboration (Hug et al., 2011), and, by providing a visualization and record, AMOEBA can prompt that reflection.

Limitations and Future Directions

We cannot make especially strong claims about the effects of collaboration or its impact on students' code; this study is not designed to be broadly generalizable. However, our findings, when combined with a wealth of history and theory about collaboration in computational learning, suggest that this work serves as a model for how we can support student collaboration *in situ* through simple, timely visualizations of data. That said, AMOEBA is too simple – it provides little information to the teacher about the quality of student programs. This lack of data may cause problems, as the teacher could pair two struggling students who are both struggling in the same way. The next version of AMOEBA will allow the teacher to visualize student similarity across a variety of metrics – structural similarity, program length, program quality, etc. In the interest of simplicity, however, we will keep buttons and choices to a minimum. Last, AMOEBA prompted questions of what similarities and differences best support peer programming and collaboration. Future work will investigate various teaching and pairing strategies, such as matching partners whose code varies widely in quality.

References

Berland, M., Martin, T., Benton, T., & Petrick, C. (2012). AMOEBA: Mining how students learn to program together. *Proceedings of the 10th Int. Conf. of the Learning Sciences* (ICLS 2012). Sydney, AUS.

Goel, S., & Kathuria, V. (2010). A novel approach for collaborative pair programming. *Journal of Information Technology Education*, *9*, 183–196.

Hug, S., Thiry, H., & Tedford, P. (2011). Learning to love computer science: Peer leaders gain teaching skill, co mmunicative ability and content knowledge in the CS classroom. *Proceedings of the 42nd ACM Symposium on Computer Science Education* (SIGCSE-11). New York, USA.

Lancaster, T., & Culwin, F. (2004). A comparison of source code plagiarism detection engines. *Computer Science Education*, *14*(2), 101–112.

Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., & Balik, S. (2003). Improving the CS1 experience with pair programming. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE-2003). Reno, USA.

Petrick, C., Berland, M., & Martin, T. (2011). Allocentrism and computational thinking. *Proceedings of the 9th International Conference on Computer-Supported Collaborative Learning* (CSCL-11). Hong Kong.

Preston, D. (2005). Pair programming as a model of collaborative learning: A review of the research. *J. Comput. Small Coll.*, *20*(4), 39–45.

Salton, G. (1989). *Automatic text processing: the transformation, analysis and retrieval of information by computer*. Reading, MA: Addison Wesley.

Teague, D. M., & Roe, P. (2008). Collaborative learning: Towards a solution for novice programmers. *Proceedings of the 10th Conference on Australasian Computing Education 78*. Wollongong, AUS.