Discovering Patterns in Student Activity on Programming Assignments

Anthony Allevato[1] and Stephen H. Edwards[2]

Abstract – In our introductory computer programming courses, students submit their code to an automated grading system for assessment, and they are allowed to make an unlimited number of attempts before the assignment is closed. This sequence of submissions can be viewed as time series data where each attempt gives rise to one or more data points, or events, that describe either an action that the student has taken or a result of those actions. Frequent episode mining is a data mining technique that lets us discover frequently occurring patterns in time series data. We apply this technique with data generated during one of our courses in order to better understand the behaviors of our students during the time that they work on a programming assignment. This paper presents an examination of the trends that we discovered in our student data.

Keywords: computer science education, data mining, introductory programming

INTRODUCTION

Instructors of computer science courses have long held beliefs about the kind of behavior that leads students to perform well on programming assignments and the kind that leads them to perform poorly. Intuition tells us that students who begin work early should achieve higher scores than those who wait until the last minute to start an assignment. Likewise, in courses such as ours where we place a strong emphasis on test-driven development, we would expect that students who perform greater testing of their code would produce code with fewer defects than students who test less.

Students at our institution submit their programming assignments to an automated grading system called Web-CAT [3], and we have compiled a large set of data spanning ten semesters of CS1, CS2, and CS3 course offerings. Many of our previous investigations of this data [1, 2] have attempted to find correlations between individual metrics such as code size, amount of testing, or time started, and the grade earned on the assignment. The work presented in this paper is our first attempt at a different approach, which uses a data mining technique known as *frequent episode mining* [4, 5] to find common patterns of behavior that occur across different views of the entire submission set for a course, such as students who performed well vs. students who performed poorly, or activity that occurred early during the time the assignment was available vs. activity that occurred close to the due date.

DESCRIPTION OF THE EVENT SPACE

Integrated into Web-CAT is a reporting tool that allows us to extract scores, code metrics, and other data about every submission that a student makes to the system. Each submission is a snapshot of a student's development process on an assignment. In order to use frequent episode mining techniques on our data set, we need to codify this sequence of submissions into a sequence of "events" that characterize that process by examining properties of each submission as well as the degree of changes between adjacent submissions.

---

[1] Department of Computer Science, 114 McBryde Hall (0106), Virginia Tech, Blacksburg VA 24061, allevato@vt.edu

[2] Department of Computer Science, 114 McBryde Hall (0106), Virginia Tech, Blacksburg VA 24061, edwards@cs.vt.edu

For each assignment in the course, we transform each student's sequence of submissions $S = \langle s_1, s_2, \ldots, s_n \rangle$ into a sequence of events $E = \langle e_1, e_2, \ldots, e_m \rangle$. Depending on the changes that a student has made to his or her code, a single submission can generate multiple events.

Table 1 describes the event space that we initially defined for our experimentation. *Test code* refers to the portion of code that belongs to unit tests that students have written to test their solution. Non-test code refers to the remaining solution code in their submissions. *Correctness score* is defined by our grading system as the product of three values: (1) the percentage of a student's own test cases that passed, (2) the percentage of a student's solution code that was covered by his or her own tests, and (3) the percentage of the instructor's reference tests that passed when executed against the student's solution. *Tool score* is a value obtained by executing static analysis tools on the student's code to evaluate code style and documentation. *Automated score* is the sum of the previous two values.

The metric-type events are induced based on the amount of change in a code metric from one submission to the next. For some of these events, such as those regarding the number of lines of code in the submission, absolute thresholds are used to determine whether a change is significant enough to be considered. In the case of the code coverage ratios, a slightly different approach is used. First, code coverage is not considered significant unless it is above 80%; our empirical analysis indicates that values much lower than 80% are infrequent and a result of a student who is simply not testing their code, or has other more severe problems in their solution. Above 80%, we consider an

Table 1: Types of Events in the Web-CAT Event Space

Category	Events
UNIVERSE events describe the state of the universe in which the students are working. They are time-based events that represent fixed points relative to the due date.	• *N* days before the assignment due date (*N* = 5, 4, 3, 2, 1) • assignment due date • *N* days after the assignment due date (*N* = 1, 2, 3)
SUBMISSION events describe how far along a student is in his or her submission sequence.	• Student has made his/her first submission • Student has made his/her first submission that compiled without errors • Student has made *N*% of their eventual total number of submissions for this assignment (*N* = 25, 50, 75) • Student has made his/her final submission
METRIC events describe changes that a student has made to his or her code, based on code metrics that are computed during grading.	• # of non-commented lines of (test, non-test) code (increased, decreased) by 5 or more lines • # of commented lines of (test, non-test) code (increased, decreased) by 5 or more lines • # of functions/methods in non-test code (increased, decreased) by 1 or more • # of test cases (increased, decreased) by 1 or more • Cyclomatic complexity number [6] of non-test code (increased, decreased) by 5 or more • Ratio of (methods, statements, conditionals) in non-test code (increased, decreased) significantly
OUTCOME events describe a student's score on a particular submission.	• (Correctness, tools, automated) score started at (A, B, C, D, F) • (Correctness, tools, automated) score increased to (A, B, C, D) • (Correctness, tools, automated) score decreased to (B, C, D, F) • (Correctness, tools, automated) score at the end of the assignment was (A, B, C, D, F)

increase in code coverage to be significant if the increase is equal to half of the remaining distance to 100%. In other words, 90% to 95% is significant, but 80% to 85% is not. The basis for this decision is that it is more difficult for students to obtain those final coverage points when they're very close to 100% than it is for them to gain one or two points from a lower rate.

MINING FREQUENT EPISODES

Given event sequences for each student who submitted to an assignment, we wish to mine episodes that occur frequently across all of these sequences. A *(serial) episode* is a subsequence of events, not necessarily adjacent in the original sequence, that occur near each other. (In a general discussion of frequent episode mining, there are other types of episodes, such as parallel episodes. For our purposes we are only concerned with serial episodes, and will henceforth drop the "serial" qualifier.) The degree of proximity required for two events to be considered "near each other" can be controlled based on the needs of the problem; we use the constraint that two events must occur within a day of each other.

The episode mining algorithm works iteratively as follows. First, for each event the total number of occurrences is calculated across all of the event sequences. The most frequent of these events (determined by the number of occurrences being greater than a threshold of our choosing) are retained; these are the *frequent episodes of length 1*. Next, these episodes of length 1 are combined to form episodes of length 2, and those that do not appear frequently enough are again sifted out. This process continues until a length n is reached such that there are no episodes of length n that appear more frequently than the specified threshold.

A single submission to an assignment may generate multiple events (for example, the number of test cases going up at the same time that code coverage increases), and would therefore have the same timestamp. We discovered that if two events have the same timestamp, they will not combine in the next step of the algorithm, and this might cause us to miss relationships between code metrics that change during the same submission. To work around this issue, we add a small fractional offset determined by the event type to each event occurrence so that every timestamp is guaranteed to be unique.

Due to the combinatorial nature of the algorithm, the size of the event space being used also greatly affects performance. Our initial event space in Table 1 proved to be too large to mine our data set in a sufficient amount of time. We pared it down in the following ways to make it tractable. All of the "universe" events were removed; if we were interested in looking at episodes that occur before or after a certain point in time, we simply ran the algorithm on that subset of events instead. All of the "outcome" events were also removed. As above, we ran the algorithm on subsets of events that we were interested in (for example, to compare students who earned an A/B on the assignment to those who earned a C/D/F). Lastly, the "ratio of methods covered" and "ratio of statements covered" metrics were removed; only "ratio of conditionals covered" was retained, as this is the most detailed metric and the one that students' scores were based on.

Therefore, our final event space consisted of 16 "metric" events and six "submission" events. In some of the examinations that we will discuss below, we found it useful to further limit the event space to just the "metric" events and exclude the "submission" events.

Once the final event space was established, various subsets of the data were mined for frequent episodes, and the following sections describe three scenarios that we examined. In each case, we isolated the frequent episodes of length n for the largest n that had a significant number of episodes; this was typically the second-to-last set of episodes in each output, as the set of longest episodes would only consist of one or two that met the frequency threshold. We then constructed digraphs from these episodes in order to more clearly visualize the results. A digraph contains a vertex that corresponds to each event type in the data set, and there is a directed edge between events A and B if and only if there is an episode in which B appears immediately after A. It follows that each episode forms a path in the digraph. The converse, however, is not true; not all paths in the digraph correspond to episodes. The weight of an edge in the digraph is the cumulative frequency that its events appear across all episodes, so a pair of events that appears in multiple episodes will have higher weight than a pair that appears fewer times. In the graphical representations of these digraphs presented below, darker and thicker lines represent edges with higher weights. While the existence of a path does not necessarily guarantee the existence of an episode containing that event sequence, heavily weighted paths do strongly suggest the presence of such episodes. In order to present the

results clearly, the graphs below use abbreviations for the event types, the explanations of which are given in Table 2.

The data that we examined is the work of 102 students in the third programming course in our computer science program, which is an introduction to C++ (after two semesters of Java). We looked at two assignments from this course: the first assignment, a small interpreter that required some basic class design; and the final assignment, a general tree implementation.

Scenario 1: A/B Students vs. C/D/F Students

In order to look for differences in behavior between students who ended up scoring well on an assignment compared to those who scored poorly, we partition the data into two subsets: events leading up to students receiving an A or B, and events leading up to students receiving a C, D, or F. Frequent episodes were mined from each subset, and the digraphs are shown in Figure 1 (for the A/B students) and Figure 2 (for the C/D/F students).

In both figures, we see that the path (MTHN+, CMPN+, MTHT+, CVCN+) occurs with very high frequency, so both the high scoring students and low scoring students are frequently increasing the number of methods and complexity of their code, as well as testing more which then results in an increase in their coverage score. However, we see a striking frequently occurring path in the C/D/F graph: (MTHN–, CMPN–, MTHT–). This is the result of students both removing entire methods from their solution, which lowers its complexity, and then removing entire test cases as well (which could be test cases that tested methods that were removed). The fact that we see C/D/F students frequently removing entire methods instead of merely decreasing the number of lines of solution code might indicate that the student has deficiencies not only in their implementation but also in their design, which can be more difficult to resolve.

We performed the same partitioning and mining on the submissions for the final assignment in the course. These results are shown in Figure 3 (A/B students) and Figure 4 (C/D/F students). In this case the full path (MTHN+, CMPN+, MTHT+, CVCN+) seen previously in both sets of students only appears in the A/B subset on this assignment. Among C/D/F students, only the (MTHN+, CMPN+) prefix appears frequently, which indicates that these students, compared to the A/B students, are testing their code less.

Scenario 2: Early Activity vs. Late Activity on an Assignment

The next case examines the type of activity that occurs several days before an assignment is due and compares that to activity that occurs on or after the due date. We use the first assignment in the course again as an example. Figure 5 shows frequent episodes that were discovered in activity four days before the assignment was due or earlier.

Table 2. Definitions of the event abbreviations used in the figures.

Abbreviation	Description
NCLT+/–	# of non-commented lines of test code increased/decreased
CLT+/–	# of commented lines of test code increased/decreased
MTHT+/–	# of test cases (test methods) increased/decreased
NCLN+/–	# of non-commented lines of non-test code increased/decreased
CLN+/–	# of commented lines of non-test code increased/decreased
MTHN+/–	# of non-test methods increased/decreased
CMPN+/–	Cyclomatic complexity number of non-test code increased/decreased
CVCN+/–	Ratio of conditionals covered increased/decreased
SUB00	Student made first submission
SUBCM	Student made first submission that compiled without errors
SUB25	Student has made 25% of his/her eventual total number of submissions
SUB50	Student has made 50% of his/her eventual total number of submissions
SUB75	Student has made 75% of his/her eventual total number of submissions
SUBFN	Student has made his/her final submission

Figure 6 shows frequent episodes that were discovered in activity on the due date and later (students in this course were permitted to work on an assignment after its due date, with a penalty applied to their final score).

The event space for this scenario includes the submission events as well as the metric events in order to determine when students are reaching certain milestones in their submission chain relative to the due date.

Among the frequent episodes discovered in the early submissions, the strong edge between SUB00 and SUBCM is one that we expect; students should be working on their solution for a not-insignificant amount of time before making their first submission to Web-CAT, so their first submission should almost always be their first compiling submission, barring any differences in each environment that the student may not have accounted for. Aside from this, we see some interesting frequent "inversions" in the code metrics. The edge (NCLT–, CLT+) indicates that non-commented lines of test code decreased at the same time or just before commented lines of test code increased. Similarly, (NCLN–, CLN+) shows non-commented lines of solution code decreasing when commented lines of solution code are increasing, and (NCLN+, CLN–) shows non-commented lines of solution increasing when commented lines of code are decreasing. The values of non-commented lines vs. commented lines are absolute and independent (that is, they are not ratios), which means that these changes are a result of direct student activity on both metrics and not one change affecting the other.

Since this graph shows submissions four days out and earlier, we are seeing the development process at its earliest stages. One possible explanation then for the first two inversions is that students wrote some solution code early on and discovered that it did not produce the correct results when they submitted it for grading. These students may have removed the code that caused the problem (NCLN–) and then better documented what they found (CLN+). The same could be said for the similar inversion in test code. As for the inversion (NCLN+, CLN–), it may be the case that students used comments in their code as placeholders for the logic that they were expected to implement and then replaced those comments with their solution implementation later on. A closer examination of the actual submitted source code would be required to see if these hypotheses hold. Lastly, we see that a noticeable number of students have made at least 25% of their eventual number of submissions this early in the lifetime of the assignment, as evidenced by the edge coming into SUB25.

The graph in Figure 6 of activity on or after the due date is considerably noisier, since it does not distinguish between activity by students finishing the project at this time and activity by those just starting. The most shocking (and disappointing) finding in this graph is the edge (SUB00, SUBCM) that strongly dominates every other edge in the graph; even on the final day of the assignment and beyond, the most frequently occurring pair of events is that which describes a student making his or her first compiling submission.

Scenario 3: Activity on First Assignment vs. Activity on Final Assignment

Our last case aims to see if there are significant changes in the most frequent behavior that we see at the start of the course and at the end of the course, in order to determine if students' programming habits are changing in a noticeable way. Figure 7 shows frequent episodes that were mined from all submissions to the first assignment in the course, and Figure 8 shows frequent episodes mined from all submissions to the final assignment.

Our intuition tells us that there might be strong differences in the frequent paths that occur in the first assignment graph and the final assignment graph, as students develop (and hopefully improve) their programming habits as they progress through the course. This, however, does not appear to be the case with this data set; the same edges dominate the graph in both cases. Two of the most dominating paths are those that we saw in Scenario 1: the (MTHN+, CMPN+, MTHT+, CVCN+) path that reflected increasing code complexity and testing, and the (MTHN–, CMPN–, MTHT–) path that we hypothesized was indicative of design problems. It is interesting to note that this second path is strong in Figure 8, which represents all submissions to the final assignment regardless of final grade, when it was not strong in the partitioned A/B and C/D/F graphs on that assignment. This leads us to believe that the path occurred with roughly equal frequency in each partition, but less frequently than other event sequences that overshadowed it, and only the union of those occurrences was significant enough to appear on the larger graph. This is unlike the first assignment in the course, where the "design problems" path was much stronger for students in the C/D/F subset.

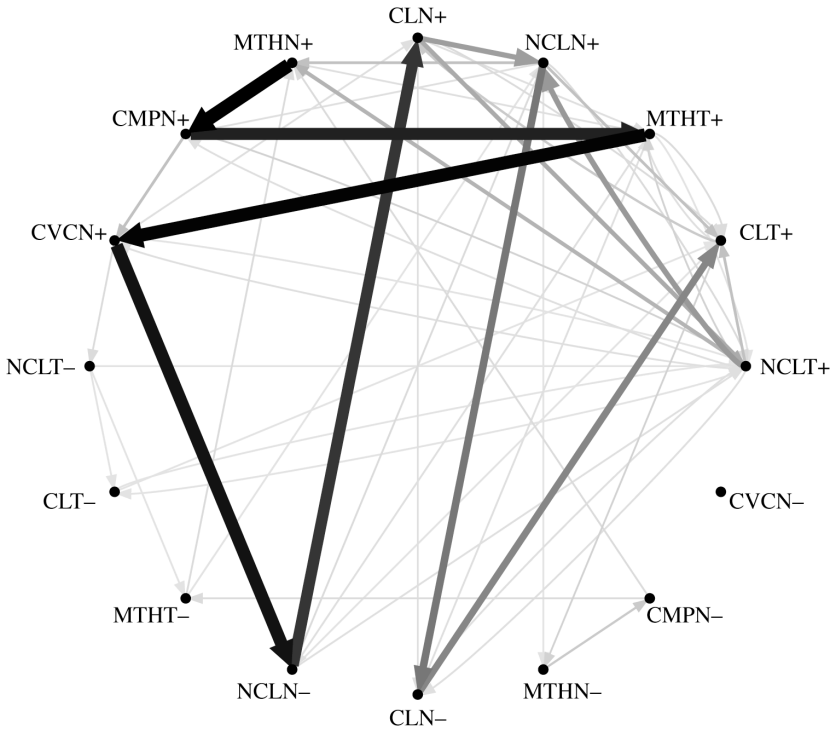

Figure 1. Frequently occurring pairs of events among submissions to the first programming assignment by students who scored A/B.

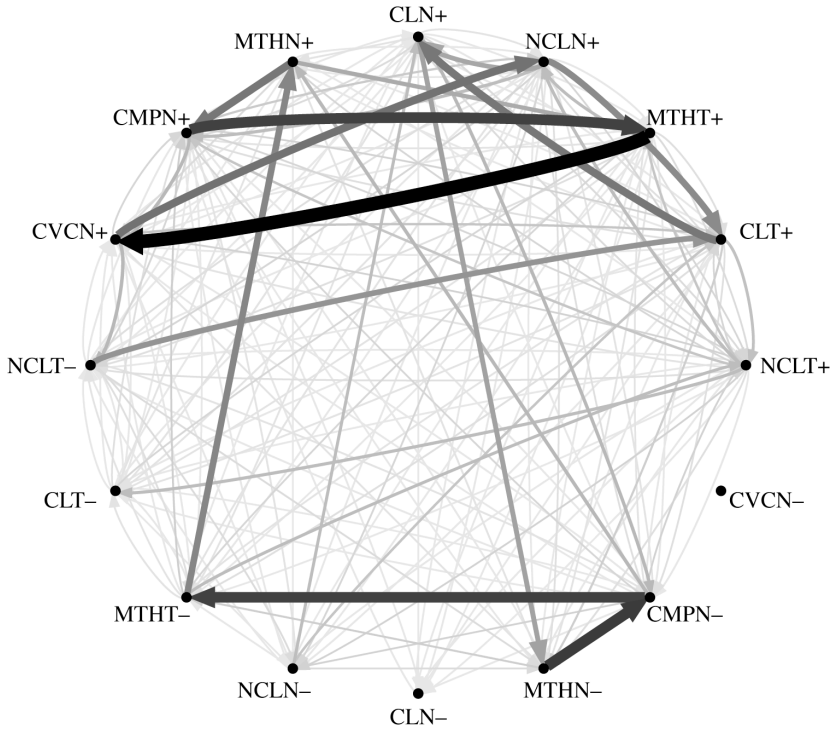

Figure 2. Frequently occurring pairs of events among submissions to the first programming assignment by students who scored C/D/F.

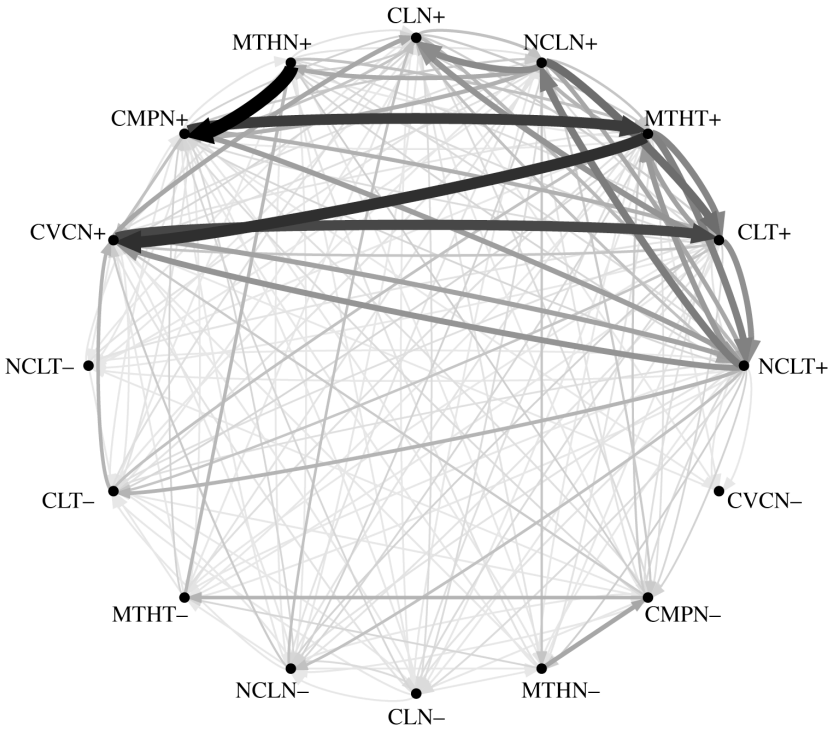

Figure 3. Frequently occurring pairs of events among submissions to the final programming assignment by students who scored A/B.

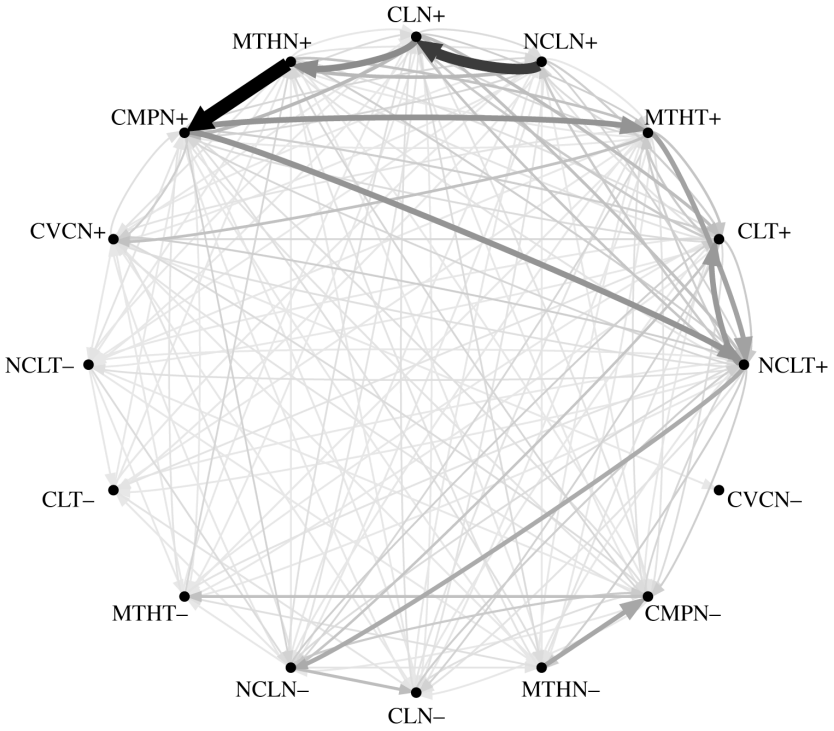

Figure 4. Frequently occurring pairs of events among submissions to the final programming assignment by students who scored C/D/F.

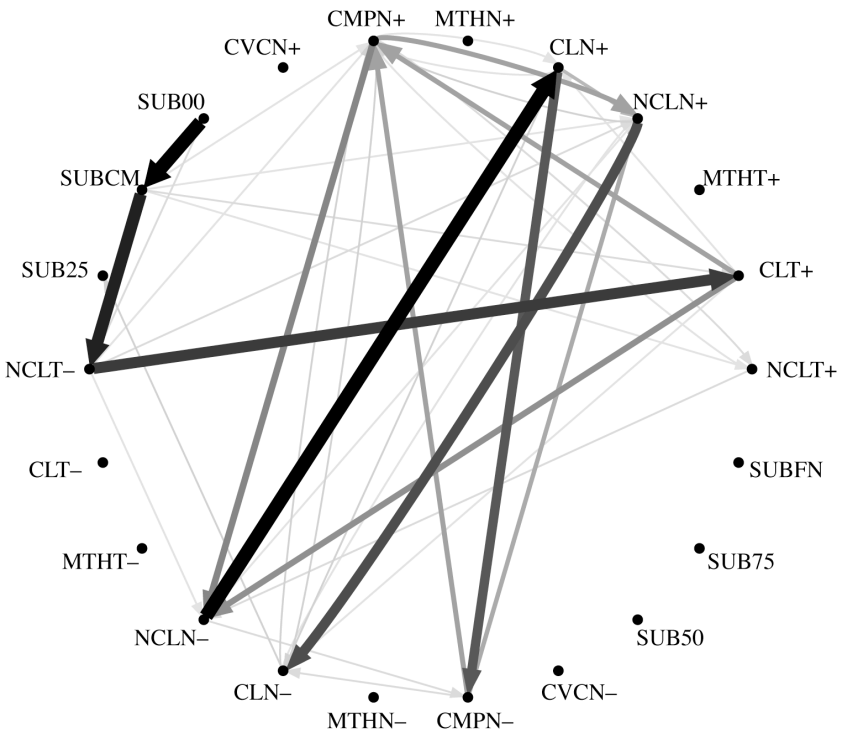

Figure 5. Frequently occurring pairs of events among all submissions made to the first programming assignment four days before it was due or earlier.

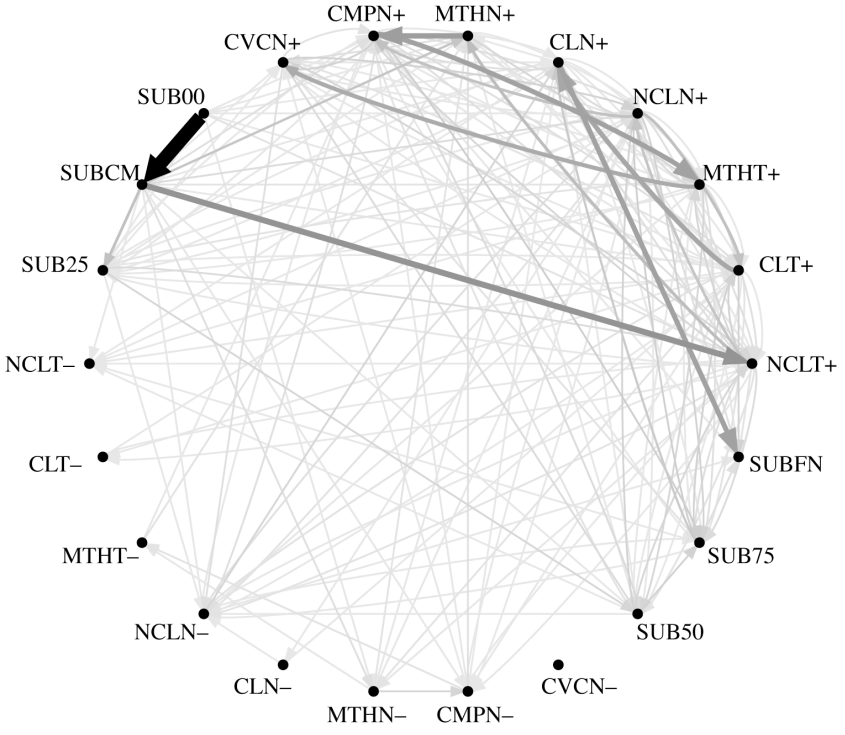

Figure 6. Frequently occurring pairs of events among all submissions made to the first programming assignment on the due date or later.

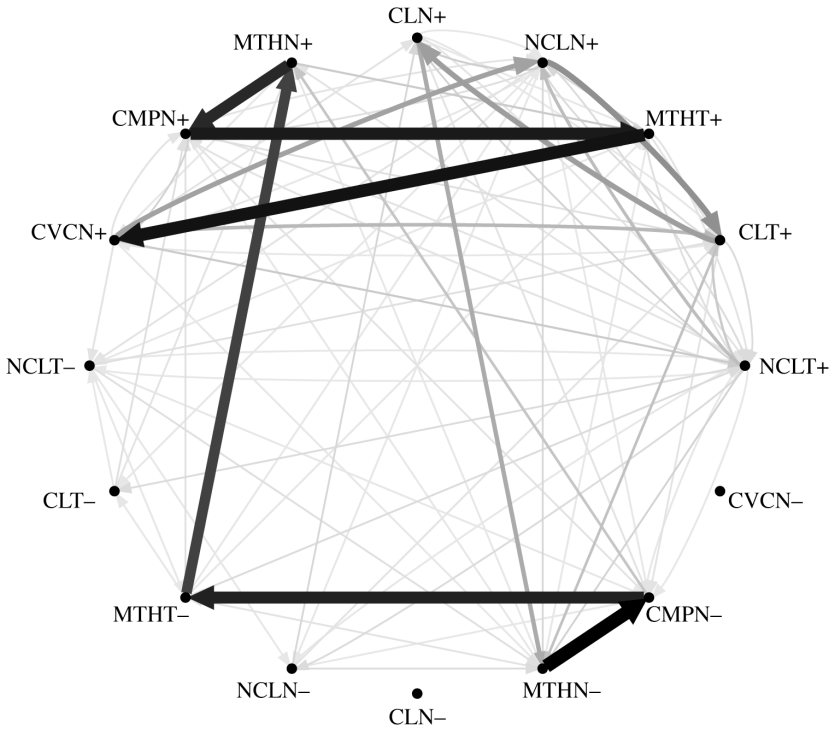

Figure 7. Frequently occurring pairs of events among all submissions made to the first programming assignment.

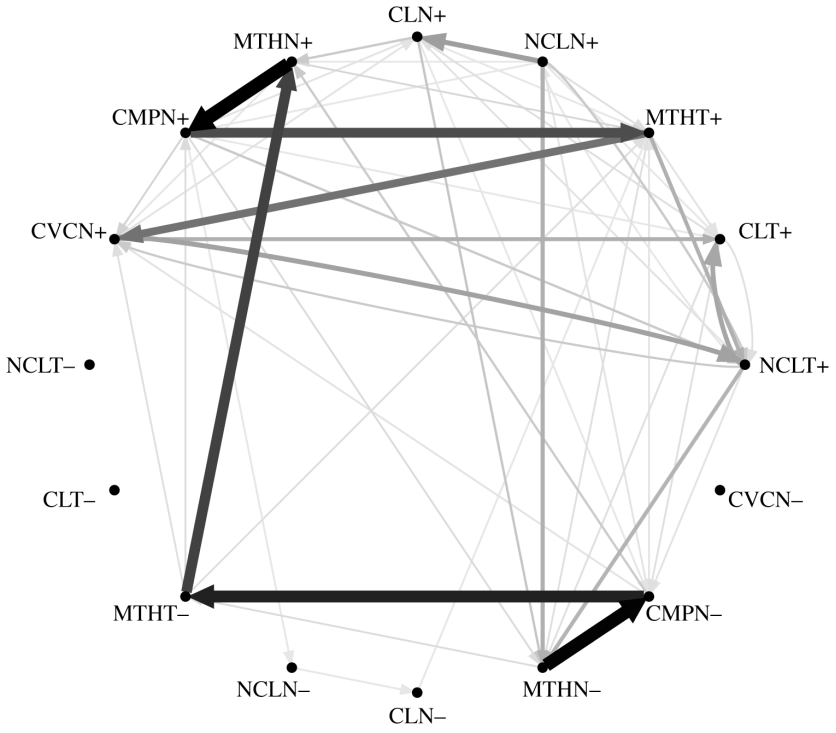

Figure 8. Frequently occurring pairs of events among all submissions made to the final programming assignment.

SUMMARY

This has been our first foray into the use of this particular data mining technique to examine student activity data on programming assignments. In some cases, the results discovered by examining the graphs above are compatible with the intuition that we develop as educators by observing students over the years. Students who perform well on programming assignments predominantly make increasing changes to their solution and testing, as if they start with a clear concept of the path to take to the end result and are incrementally working toward it. Students who performed poorly on the assignments, on the other hand, tended to make more decreasing changes as they struggled between implementing new parts of their solution and repairing those parts when they failed.

In other cases, the interpretation of the graphs defies our expectations, such as the hope that students' behaviors would differ on the first assignment in the course and the last assignment. While it is possible that their programming habits did improve or change in other ways, they were not reflected in the results shown here.

Many of the parameters in our experimentation can be chosen in different ways so there is a great deal of future work that can be done here to determine what the best choices might be. Some questions that we might ask include: what other types of events could we define in our event space to codify student activity? Should the inter-event time constraint (such that two events must occur within a day of each other) be tightened? Is partitioning the data beforehand the best way to distinguish frequent episodes in one data set vs. another (for example, episodes that result in an A/B grade and episodes that result in a C/D/F grade), or should we let the algorithm run longer on the union of these with event sequences that are terminated by an outcome event representing the grade? Are there other episode mining algorithms entirely that can be used on data of this nature? It is our hope that posing student activity data as this kind of data mining problem will lead to a better understanding of student programming habits.

ACKNOWLEDGMENTS

We would like to thank Dr. Naren Ramakrishnan and Debprakash Patnaik for their assistance in discussing how our data set can be analyzed with frequent episode mining and for the development of the tools that implement the algorithms that we used.

REFERENCES

[1] Allevato, Anthony, Matthew Thornton, Stephen H. Edwards, and Manuel A. Pérez-Quiñones, "Mining data from an automated grading and testing system by adding rich reporting capabilities," *Educational Data Mining 2008: 1st International Conference on Educational Data Mining, Proceedings*, Montreal, Quebec, Canada, 2008, pp. 167–176.

[2] Edwards, Stephen, Jason Snyder, Manuel Pérez-Quiñones, Anthony Allevato, Dongkwan Kim, and Betsy Tretola. "Comparing effective and ineffective behaviors of student programmers," *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, Berkeley, CA, 2009, pp. 3–14.

[3] Edwards, Stephen H., "Using test-driven development in the classroom: providing students with automatic, concrete feedback on performance," *Proc. Int'l Conf. Education and Information Systems: Technologies and Applications (EISTA '03)*, August 2003.

[4] Laxman, Srivatsan, P.S. Sastry, and K.P. Unnikrishnan, "Discovering frequent episodes and learning hidden Markov models: a formal connection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 11, November 2005, pp. 1505–1517.

[5] Mannila, Heikki, Hannu Toivonen, and A. Inkeri Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, no. 3, September 1997, pp. 259–289.

[6] McCabe, Thomas, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, 1976, pp. 308–320.

Anthony Allevato

Anthony Allevato received the BS and MS degrees in computer science from Virginia Tech and is currently a PhD candidate there. His research interests include computer science education, programming languages, and digital image processing.

Stephen H. Edwards

Stephen H. Edwards received the BS degree in electrical engineering from the California Institute of Technology, and the MS and PhD degrees in computer and information science from the Ohio State University. He is currently an associate professor in the Department of Computer Science at Virginia Tech. His research interests include software engineering, reuse, component-based development, automated testing, formal methods, and programming languages.