Modeling How Students Learn to Program

Chris Piech[1], Mehran Sahami[1], Daphne Koller[1], Stephen Cooper[1], Paulo Blikstein[2]
[1]Computer Science Department, [2]School of Education
Stanford University
Stanford, CA. 94305
{piech, sahami, koller, coopers}@cs.stanford.edu, paulob@stanford.edu

ABSTRACT

Despite the potential wealth of educational indicators expressed in a student's approach to homework assignments, *how* students arrive at their final solution is largely overlooked in university courses. In this paper we present a methodology which uses machine learning techniques to autonomously create a graphical model of how students in an introductory programming course progress through a homework assignment. We subsequently show that this model is predictive of which students will struggle with material presented later in the class.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education.

General Terms

Algorithms, Measurement, Experimentation, Languages

Keywords

Probabilistic Graphical Models, Hidden Markov Model, Program Dissimilarity Metric, Intelligent Tutor, Student Progress Model

1. INTRODUCTION

In analyzing student learning in introductory programming courses, there is a wealth of information not only in the final products (i.e., programs) that students submit for evaluation, but also in the development *path* they took to produce their programs. In traditional settings, the data on how students developed their programs over time is either not available or not analyzed. In this work, we show that temporal traces of the development paths taken by students in an introductory programming class can be mined to build graphical models that compactly capture the common major milestones in such development paths. More significantly, we show that these models also carry predictive capability in that the paths students take in these models are correlated with their future performance in the class.

We gathered and analyzed the development paths for students in Stanford's CS1 course, which begins with an initial assignment in Karel the Robot that is subsequently followed by five or six Java programming assignments. While students' final submitted solutions to course assignments can be used to identify which individuals need extra assistance or are struggling with the material, there are surprisingly a substantial number of novice programmers who do not grasp important core concepts, but are able to somehow still produce a fully functional final solution to the first programming assignment. As a result, the submitted work is devoid of any indication that the student actually needs help. Though the student's solution might not contain obvious warning signs of missed concepts, there tends to be evidence of such misunderstandings hidden in the development *path* by which the student arrived at his/her final solution.

It is easy to claim that understanding how students progress through an assignment allows educators to better identify students that need interventions. It is difficult to implement a process to record and analyze students' progress. Manual observation and analysis of students as they program raises privacy concerns, and it is tedious and difficult to personally interpret raw snapshots of student code over time, especially in large courses. Rather, we take an automated approach, developing machine learning techniques that can be applied to code snapshots captured periodically by an instrumented IDE. Our machine learning algorithm produces a finite state machine of development "milestones" that provide a high-level graphical view of student development paths through an assignment. Such graphical models help provide a better understanding of how novice programmers go about solving a problem. More significantly, these models allow us to then cluster students into groupings that are predictive of students' future performance. We applied this technique to one of the problems given as part of the first programming assignment in our CS1 class. This problem, dubbed "Checkerboard Karel", requires that Karel the Robot produce a checkerboard pattern of beepers in his world.

As we show later in this paper, the patterns that our machine learning algorithm found in *how* students solved the Checkerboard Karel problem were more informative at predicting how well students would perform on the class midterm than the grades students received on the assignment. We demonstrate that the algorithm captured a meaningful general trend in how students were solving this programming problem by using the model generated from student development traces in the spring offering of the course to predict student performance in the subsequent summer term. While our results initially focus on a programming problem in the limited domain of Karel the Robot, we show the more general applicability of our methodology by applying our algorithm to a Java assignment in which students write an open-ended version of the graphical arcade game Breakout (also known as Brick Breaker).

There are many potential applications for high-level representations of student progress in programming assignments. These include using such models to track the progress of new students and suggest possible interventions if it has been determined that the development path the student is on is not likely to lead to a positive outcome. Similarly, such information can be logged, compiled, and relayed to the course instructor to help provide a more accurate picture of the concepts in the course with which the students are truly struggling.

As mentioned previously, our machine learning algorithm autonomously produces a probabilistic finite state machine representation of how students in the class traversed through various "milestones" in the Karel assignment. The patterns that the machine learning algorithm finds provide insight into what programming motifs are common for students who would struggle later on in the course, and also provides a visualization of how the class as a whole approached the Karel assignment.

The main results presented in this paper are:

- the development of machine learning methods that build models of high-level student development pathways in programming assignments,
- the application of these methods to a large set of student trace data by which the algorithm is successfully able to autonomously extract novel features of a student's progress over the course of an assignment, and
- the use of these features to predict students' future performance in the class as measured by their midterm grades.

The novelty of this work stems from:

- the collection of a unique dataset of student development trace data,
- the presentation of an application of unsupervised machine learning concepts to a new domain, and
- the potential pedagogical insights that can be gained from the model generated by the machine learning algorithm. This research is particularly pertinent to large lecture-based classes and online courses.

2. RELATED WORK

This research expands upon previous attempts to find a symbolic representation of student progress. Reiser [19] made the argument that development of an autonomous system that could understand a student's current state as the student solves a programming problem would have profound educational implications.

Spohrer and Soloway [27] tried to represent how students learned to program through an investigation of the bugs in the students' code. Students' programs were examined as soon as their code cleanly compiled (and was thus devoid of syntax errors), and their bugs identified and categorized. The decision to analyze student bugs as soon as their code compiled cleanly was reasonable, given that it would not have been possible to analyze all intermediate versions of students' code as the analysis was done by hand. Their strategy was limited in that it would not be useful for trying to analyze the students who solve a problem one part at a time (those initial clean compiles would not include much of the overall solution), and they did not observe the progression of student code development throughout the assignment.

There has been work on studying students' progress at a more fine-grained level, by focusing on specific programming language constructs. These constructs include variables [23], conditionals [10], looping [25], and methods [12]. The assumption in many of these studies is that student progress can be understood through difficulties with specific programming constructs.

Many researchers have attempted to determine students' mental models of computers and computing (see [13] as an interesting early example), using various qualitative techniques such as phenomenography [1, 3]. Because such studies tend to be in-depth and time-consuming, the number of participants tends to be quite small.

The task of constructing a dynamic student model has had a resurgence with the introduction of artificial intelligence algorithms. In a paper on coached problem solving using Bayesian networks, Conati [5] demonstrated the potential of using an expert crafted graphical model for tutoring students learning physics. However, in Conati's work, as in many instances of supervised learning applied to constructing a dynamic student model, it is noted that supervised learning is limited by the laborious process of expert graphical model construction and the lack of transferability of these expert generated models from one program to another.

Recent research has used automated log analysis analyze student programs, especially trying to distinguish novices and experts. Blikstein [30, 31, 32] used thousands of time-stamped snapshots of students' code and found markedly diverse strategies between experienced and novice programmers. By mining snapshots from code repositories, Berland and Martin [29] found that novice students' developed successful program code by following one of two progressions: planner and tinkerer. Planners found success by carefully structuring programs over time, and tinkerers found success by accreting programs over time. Students were generally unsuccessful if they didn't follow one of those paths.

This common limitation in the state of the art for dynamic student modeling highlights the need for an unsupervised (i.e., fully autonomous) approach. However, despite the apparent utility of a fully autonomous system, little work has been done to apply unsupervised learning algorithms.

3. DATA COLLECTION

Over the summer of 2010 we modified the Integrated Development Environment (IDE)—Eclipse—used by students in Stanford's CS1 course so that it would log snapshots, a complete version of the student's program at that point in time, every time a student compiles a project (which the students must do before they can run their program) and commits that snapshot to a local *git* repository. When the student submits the final version of their assignment, they can elect (opt-in) to also submit the full git repository of their progress (i.e., code snapshots) in developing their solution. For this investigation we analyzed data from two assignments, Checkerboard Karel and Breakout, described below.

Checkerboard Karel: Karel the Robot is used in the first week of CS1 to teach students basic program flow and decomposition. The particular variant of the Karel programming language we use is a Java-based language [20], which most notably does not include variables or parameters. In this assignment the students were asked to make Karel the Robot place beepers in a checkerboard fashion with an alternating pattern of beepers and no beepers, filling up the whole world. The full solution needs to work on any sized Karel world. It is hard to get an algorithm to work on worlds with an odd number of columns, particularly so if there is only one column. At this point in the course most students struggle with understanding nested while loops and how to identify pre and post conditions for their methods and loops.

Breakout: This assignment asks students to implement a classic arcade game [16]. The students need to write an animation loop, incorporate mouse events and keep track of game state. It is the third assignment given in CS1 and the first large programming project written in Java using the ACM graphics library [28].

We collected repositories from N = 370 Karel assignments (238 from spring 2011 and 132 from summer 2011). For each student we had on average 159 snapshots of them programming checkerboard Karel with a standard deviation of 82 snapshots. Each snapshot was time-stamped and could be run through a simulator to record errors or to test functionality. We also analyzed Breakout repositories from N = 205 students all of which were from winter 2011 where the snapshot count per student had a mean of 269, and a variance of 153. To protect the privacy of our students we removed students' names from all snapshots. We chose to capture snapshots when the student compiled/saved as we thought this would be the best interpretation of a "unit" of work.

4. DATA ANALYSIS

4.1 Program Distance Metric

For the machine learning algorithm to build a model of how students progress through an assignment, it needs to compare two programs against one other and determine the extent to which the two pieces of code should be considered similar. Specifically, the algorithm used requires that we calculate a real number that accurately reflects the degree to which two programs are dissimilar. We considered three algorithms for calculating dissimilarity:

1. Bag of Words Difference: We built histograms of the different key words used in a program and used the Euclidean distance between two histograms as a naïve measure of the dissimilarity. This is similar to distance measures of text commonly used in information retrieval systems [22].

2. Application Program Interface (API) Call Dissimilarity: We ran each program with standard inputs and recorded the resulting sequence of API calls. We used Needleman-Wunsch global DNA alignment [14] to measure the difference between the lists of API calls generated by the two programs. Intuitively, we think of the sequence of API calls made in the program to be the "DNA" of that program, and we are comparing the DNA of two programs to determine their dissimilarity. We note that we modified the Needleman-Wunsch algorithm to allow for gap penalties which varied based on the API call that was matched with the gap.

API calls are a particularly good representation of Karel programs because the Karel programs do not store variables—and as a result an assignment's entire functionality is expressed in terms of simple API calls (e.g., turnLeft, move, etc,). This metric is more difficult to implement for full Java programs. For example, Breakout makes graphics API calls but with parameters that change the nature of the API call. This makes it more difficult to create a single token that fully captures the effect of any call.

3. Abstract Syntax Tree (AST) Change Severity: We built AST representations of both programs and calculated the summation of Evolizer abstract change severities as described by Gall *et al* [9]. The Evolizer change severity score calculates the minimum number of rotations, insertions and deletions that need to be applied to the AST of one program to transform it into the AST of another program. It uses an analysis of each syntax tree to weight how significantly an edit changes the AST.

All of our dissimilarity metrics were normalized by sum of the distances between the programs and the "starter" code (the small amount of code students may have been given initially as part of their assignment framework). To evaluate each metric we had a group of five advanced computer science students with teaching experience (whom we subsequently refer to as "experts") label a set of snapshot pairs as either similar or different and measured how well the distance metrics could replicate the expert labels.

Table 1. Distance Metric Evaluation

Metric	*Percent Accuracy*
1. Bag of Words	55%
2. API Calls	86%
3. AST Change	75%

The experts were asked to assess similarity based on a rubric having them identify major and minor stylistic and functional differences between pairs of programs. We selected 90 pairs of programs, capturing a spectrum of similar, dissimilar and slightly similar code. For the checkerboard Karel assignment the API Call Dissimilarity performed best (see Table 1) with an accuracy of 86% ($p < 10^{-8}$ relative to chance). The set of pairs mislabeled by the Bag of Words and AST Change Severity largely overlapped with the set of pairs where the human experts disagreed.

While the distance metrics seemed to accurately measure the dissimilarity between code snapshots, they are biased towards assigning low dissimilarity score to snapshots from the same student. To account for this bias, we modified all of our algorithms to never use the dissimilarity value computed from two snapshots that originated from the same student.

The distance metric used to build our model was a weighted sum of the AST Change metric and a set of API Dissimilarity scores (each generated by running the programs with a different input world). We built a Support Vector Machine [6] trained to classify the human labeled data using the different distance metrics as features. The values used to weight the distance measures in our composite dissimilarity score were the weights assigned to each distance measure by the Support Vector Machine.

4.2 Modeling Progress

The first step in our student modeling process was to learn a high level representation of how each student progressed through the checkerboard Karel assignment. To learn this representation we modeled a student's progress as a Hidden Markov Model (HMM) [17]. The HMM we used (see Figure 1) proposes that at each snapshot, a student is in a "high-level milestone," referred to as a *state*. While we cannot directly observe the state (it is a *latent* variable), we can observe the source code of the snapshot, which is a noisy sensor of the latent variable. Example states could be "the student has just started" or "the student has gotten Karel to checker all worlds except for worlds with a single column." Note that these states need not be explicitly labeled in the model. They are autonomously induced by our learning algorithm given the student trace data provided. The HMM is parameterized by the probabilities of a student going from one state to another and the probability that a given snapshot came from a particular milestone. A HMM is a relevant model of student progress because programming is generally an incremental process. The milestone where a student is at a given time is not independent of the milestone that he/she was at in the previous time step. The HMM explicitly captures this dependency.

Modeling student progress as a HMM makes the over-simplifying Markov assumption that the future state of a student is independent of past states that the student was in given that we know the individual's current state. While this assumption is not entirely correct (a student's current state alone does not capture what they learned from experience), this statistical simplification is still useful for finding patterns and making predictions while maintaining algorithmic tractability. Similarly this HMM assumes a relatively constant rate of change between observations. Realizing that a commit/save is a coarse representation of a unit of work, and that there are notably different patterns in commit rates among different students, we smooth out the difference in commit rates using dynamic time warping [18] to better match the underlying assumptions of the model.

Learning a HMM is identical to learning a finite state machine (FSM) of how students transition through the high-level milestones. Each state from the HMM becomes a node in the FSM and the weight of a directed edge from one node to another provides the probability of transitioning from one state to the next.

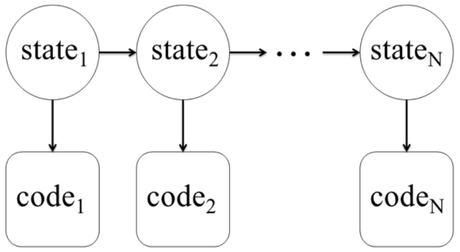


Figure 1. The program's Hidden Markov Model of state transitions for a given student. The node "code$_t$" denotes the code snapshot of the student at time *t*, and the node "state$_t$" denotes the high-level milestone that the student is in at time *t*. N is the number of snapshots for the student.

In order to learn the HMM we must identify three variables:

1. The finite set of high-level $States = \{s_1 \cdots s_k\}$ or milestones that a student could be in. A state is defined by a set of snapshots where all the snapshots in the set came from the same milestone.
2. The transition probability, $P\left(State^{(t+1)} \middle| State^{(t)}\right)$, of being in a state given the state you were in in the previous unit of time.
3. The emission probability, $P\left(Code^{(t)} \middle| State^{(t)}\right)$, of seeing a specific snapshot given that you are in a particular state. To calculate the emission probability we interpreted each of the states as emitting snapshots with normally distributed dissimilarities. In other words, given the dissimilarity between a particular snapshot of student code and a state's "representative" snapshot, we can calculate the probability that the student snapshot came from a given state using a Normal distribution based on the dissimilarity.

While it would be possible to learn all three variables in the HMM in a single Expectation Maximization (EM) algorithm [8], for computational ease the process is divided into two phases: learning the assignment states and learning the transition and emission probabilities.

To compute the set of high-level states (the different milestones of the assignment), we sampled two thousand snapshots chosen from the Karel training dataset (distributed evenly over students and evenly over time) and clustered the sample using K-Medioids [11]. K-Medioids is a variation of K-Means clustering where centroids are represented by the median code snapshot instead of being a numerical average of the set of examples in the cluster, as it is not possible to construct a synthetic "average" code snapshot. In a similar vein to the Buckshot algorithm [7], we initialized K-Mediods using cluster labels from Hierarchical Agglomerative Clustering.

Once we had established the set of states (i.e., milestones) of an assignment, we used an EM algorithm to simultaneously compute both the transition and emission probabilities in the state diagram. To initialize the algorithm, we calculate the probabilities under the assumption that all state variables are independent of one another. In the expectation step of EM, the best guess at the progress of a student through the HMM was made using the forward-backwards algorithm [17]. The EM algorithm resulted in a probabilistic assignment to the state variables for *each* student at each point in time, and also provided estimates for parameters to the HMM reflecting the state diagram induced from data over all students in the class.

4.3 Finding Patterns in Paths

The final step in our algorithm was to find patterns in how the students were transitioning through the HMM. To find these patterns we clustered the paths that students took through the HMM. To cluster the paths we ran the K-Means algorithm over the space of HMM, using a method developed by Smyth [24]. For each student we constructed a HMM to represent their state transitions—these *per student* models also incorporated prior probabilities based on the HMM developed using data from the whole class. We measured dissimilarity between two students as the symmetric (i.e., averaged) probability that student A's trajectory could be produced by student B's HMM and vice versa. Using this distance measure, we then clustered all the student paths to create groupings of students based on the characteristics of their entire development path for the assignment.

5. RESULTS

Clustering on a sample of 2000 random snapshots from the training set returned a group of well-defined snapshot clusters (see Figure 2). The value of K that maximized silhouette score (a measure of how natural the clustering was) was 26 clusters. A visual inspection of these clusters confirmed that snapshots which clustered together were functionally similar pieces of code.

When we repeated this process with a different set of random snapshots, 25 of the 27 clusters were the same, indicating that the results were quite stable and that the patterns found were not due to chance. Moreover, a manual examination of the states of the induced HMM revealed that they made intuitive sense. The class-wide state machine showed that there were several "sink" states—milestones where the students clearly had serious functional problems. Interestingly, once a student transitioned to such a state, the student had a high probability of remaining there through several code updates. For each "sink" state, the state machine showed how significant the sink state was and what transition most students took to get out of that state. Such revelations can be quite useful for determining how to appropriately aid students who may have encountered significant difficulties in programming an assignment, as revealed by the trajectory taken in the student's development path.

In addition to finding patterns in students' development progress, which can be indicative of a student's need for help (and the means for providing such help), we also wanted to determine the predictive power of such data mining in the early identification of students who may have more difficulty with computing generally. To this end, we sought to measure the extent to which student development trajectories on their first assignment in the course could be used to determine the performance of those students on the midterm exam (generally 3 to 4 weeks later in the term).

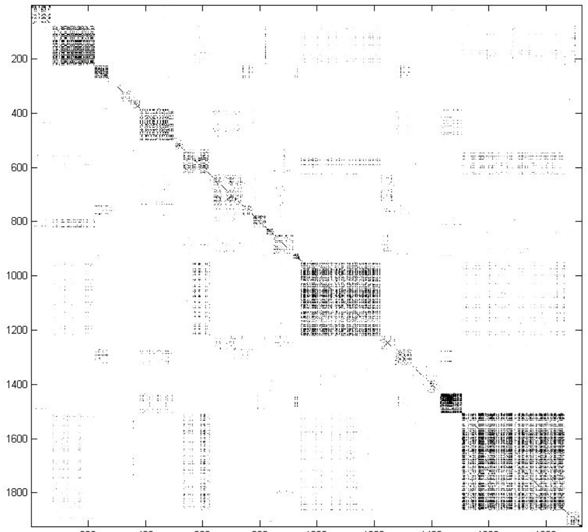
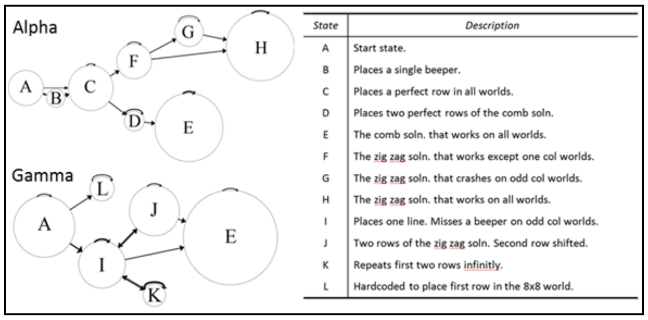


Figure 2. Dissimilarity matrix for clustering of 2000 snapshots. Each row and column in the matrix represents a snapshot and the entry at row i, column j represents how similar snapshot i and j are (dark means more similar)



State	*Description*
A	Start state.
B	Places a single beeper.
C	Places a perfect row in all worlds.
D	Places two perfect rows of the comb soln.
E	The comb soln. that works on all worlds.
F	The zig zag soln. that works except one col worlds.
G	The zig zag soln. that crashes on odd col worlds.
H	The zig zag soln. that works on all worlds.
I	Places one line. Misses a beeper on odd col worlds.
J	Two rows of the zig zag soln. Second row shifted.
K	Repeats first two rows infinitly.
L	Hardcoded to place first row in the 8x8 world.

Figure 3. Visualization of finite state machines for Alpha and Gamma clusters of students.

Table 2. Delineation of Alpha, Beta and Gamma clusters

Metric	*Alpha*	*Beta*	*Gamma*
Num Students (count)	84	108	46
Midterm score (percent)	$\mu = 73.3$, $\sigma = 20.0$	$\mu = 71.4$, $\sigma = 26.2$	$\mu = 65.4$, $\sigma = 23.2$
Time (days)	$\mu = 8.2$, $\sigma = 2.0$	$\mu = 9.1$, $\sigma = 1.8$	$\mu = 9.3$, $\sigma = 1.9$
Karel score (percent)	$\mu = 91.2$, $\sigma = 7.0$	$\mu = 88.6$, $\sigma = 6.9$	$\mu = 88.3$, $\sigma = 7.6$

We clustered students' development paths into three groups (see Table 2), hereafter referred to as *alpha*, *beta* and *gamma*. The group that a student was clustered into was indeed predictive of the student's midterm grade. The distinction between the *alpha* and *gamma* groups was particularly large, having a mean midterm score difference of 7.9%. The difference in midterm performance between the *alpha* and *gamma* groups is statistically significant, yielding a *p* value of 0.04 using a two-tailed t-test.

A visualization of the finite state machine for the *alpha* group versus that for the *gamma* group (Figure 3) shows that there are clear differences in both the types of states that the two groups visit and the pattern of how students transition between states. Qualitatively, the *gamma* group can be described as getting stuck into several sink states and then making a direct transition from a semi-working program to a fully functional program. The *alpha* group seems to make smaller but steadily positively accretive steps towards the solution.

Seeking to understand the generality of such models in their application to future (out of sample) students, we classified the students who took the class in the *summer* quarter into the *alpha*, *beta* and *gamma* groups, which were induced from students taking the class during the prior quarter. The *alpha* group (N = 30) had midterm scores $\mu = 69.7$, $\sigma = 14.1$ and the *gamma* group (N = 42) had midterm scores $\mu = 63.3$, $\sigma = 15.4$. Again, we found a statistically significant difference between the mean midterm scores of these groups as a two-tailed t-test yielded a *p* value of 0.08. This result shows that the induced models are quite robust, as they capture patterns in development paths that are not specific to a single class of students, but generalize across student development behavior between classes (that also had different instructors). We believe this generality of the induced models has the potential to not only provide predictive capability in helping address student difficulties in programming, but also yielding deeper insights regarding fundamental misunderstandings of programming that transcend the details of one particular class.

We used the same algorithm, with the AST dissimilarity metric instead of the API dissimilarity metric, to build a model of how students progressed through Breakout. Interestingly there was a more articulated trajectory that most students followed—this could be as a result of clear objectives laid out in the assignment handout or it could reflect that the students at this point in the course are more mature programmers. Clustering of the paths that students took through the assignment resulted in two groups of students that had a 6.9 difference of means for their midterm scores, with a student TTest score of 0.06. Since the Breakout samples were collected in a different quarter than the Karel samples, we could not test the extent to which the groups of students discovered through analyzing in breakout correlated to the groups of students discovered through the Karel assignment.

6. DISCUSSION

The process of building the Karel state machine provided several pedagogical insights into the introductory computer science class. We found that by analyzing students' development paths, rather than simply their final grade, on an assignment, we discovered a higher correlation with students' understanding of the material, as reflected by their midterm scores. We believe the underlying reason for this phenomenon is that there is greater variability in student development paths than their final program outcomes. The development path provides important information regarding students' understanding of concepts beyond simply their final product. We seek to explore this issue in further work to better understand the degree to which data from student development paths reflects measurable understanding of programming concepts. We believe this line of investigation is not only quite promising with respect to providing a data-driven methodology for improving programming instruction and pedagogy, but also for gaining deeper insight into learning in general.

The success of the model also implies more subtle insights. The construction of the Karel state machine relied on two substantial assumptions: that a dynamic time-warped commit reflected a unit of work and that in generating the state transitions the future is independent of the past given the present. Building the assignment state machine based on these assumptions does not prove the assumptions are correct. It does demonstrate a tractable scheme for model construction that is fine-grained enough to capture important trends while not providing enough degrees of freedom in the model to overfit the training data.

Our results also show the generality of patterns found in student programming paths. For example, we analyzed programming paths by students in CS1 in two different quarters, taught by different instructors, yet observed similar patterns of student development paths in both courses. Such observations lead us to believe that there are further insights regarding program development that can be gleaned from our models. Moreover, this generality indicates that such models may be more broadly applicable as a means for predicting the sorts of interventions that would most benefit novice programmers and provide guidance as to when such interventions would be useful to provide. As reflected in our empirical results showing the correlation between development paths from the first assignment in CS1 and students' midterm scores, we believe that these models can be successful for identifying students in need of early intervention.

This model could potentially provide further insight into whether or not students need interventions, as many schools offer various open and closed laboratory opportunities to students who may need help. By understanding what students have tried and their projected path, this model provides an indication whether students need help at the time that they seek it, or if a student should ponder the problem a bit more without external help.

7. REFERENCES

[1] Booth, S. 1992. Learning to program: A phenomenographic perspective. Gothenburg, Sweden: Acta Universitatis Gothoburgensis.

[2] Bowman, M., Debray, S. K., & Peterson, L. L. 1993. Reasoning about naming systems. *ACM Trans. Program. Lang. Syst.* 15, 5 (Nov. 1993), 795-825.

[3] Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M., & Stoodley, I. 2004. Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university. *Journal of Information Technology Education*, 3, 143-160.

[4] Brusilovsky, Peter. 2000. Adaptive hypermedia: From intelligent tutoring systems to web-based education. LNCS.

[5] Conati, C., Gertner, A. S., VanLehn, K., & Druzdzel, M. J. 1997. On-line student modeling for coached problem solving using Bayesian networks. *Proceedings of the 6th Int'l Conference on User Modeling (UM-96)*, 231-242.

[6] Cristianini, N. & Shawe-Taylor, J. 2000. An introduction to support vector machines and other kernel-based learning methods. Cambridge University Press.

[7] Cutting, D., Karger, D., Pedersen, J., & Tukey, J. 1992. Scatter/gather: A cluster-based approach to browsing large document collections. *Proc. 15th SIGIR*, 1992.

[8] Dempster, A.P., Laird, N.M., & Rubin, D.B. 1977. Maximum likelihood from incomplete data via the em algorithm. *J. of the Royal Statistical Society B*, 39 (1): 1–38.

[9] Gall, Harald et al. 2009. Change analysis with evolizer and changedistiller, *Software, IEEE*.

[10] Hoc, J-M. 1984. Do we really have conditional statements in our brains? *Proceedings of the 2nd European Conference on Readings on Cognitive Ergonomics - Mind and Computers*, G. van der Veer, M. Tauber, T. Green, and P. Gorny (Eds.), Springer-Verlag, London, UK, 92-101.

[11] Kaufman, L. and Rousseeuw, P.J. 1990. Finding groups in data: An introduction to cluster analysis, Wiley.

[12] Kessler, C. & Anderson, J. 1988. Learning flow of control: Recursive and iterative procedures. In [26], 229-260.

[13] Kurland, D. & Pea, R. 1983. Children's mental models of recursive logo programs. *Proceedings of the 5th Annual Conference of the Cognitive Science Society*, NY, 1-5.

[14] Needleman S. & Wunsch C. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*.

[15] Paine, Carina. 2001. How students learn to program: Observations of practical tasks completed. *Proceedings of the IEEE International Conference on Advanced Learning Technologies (ICALT '01)*.

[16] Parlante, N.,Wolfman, S., McCann, L., Roberts, E., Nevison, C., Motil, J., Cain, J., & Reges, S. 2006. Nifty assignments. *SIGCSE Bull.* 38, 1 (March 2006), 562-563.

[17] Rabiner, L.R., & Juang, B,H. 1986. Introduction to hidden markov models. *IEEE ASSP Magazine*, 3(1):4–16, 1986.

[18] Rabiner, L.R., & Juang, B,H. 1993. Fundamentals of speech recognition. Prentice-Hall, Inc.

[19] Reiser , B., Anderson , J., Farrell, R. 1985. Dynamic student modelling in an intelligent tutor for LISP programming, *Proceedings of the 9th int'l joint conference on AI*, 8-14.

[20] Roberts, Eric. Karel Learns Java. Available from: http://www.stanford.edu/class/cs106a/cs106a_spring11/book/karel-the-robot-learns-java.pdf Accessed 9/1/2011.

[21] Sagar, Tobias et al. 2006. Detecting similar java classes using tree algorithms. *Proceedings of the 2006 international workshop on mining software repositories (MSR '06)*.

[22] Salton, G., Wong, A., & Yang, C. S. 1975. A vector space model for automatic indexing. *CACM*, 18: 613-620.

[23] Samurcay, R. 1988.The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In [26], 161-178.

[24] Smyth, P. 1997. Clustering sequences with hidden markov models. Advances in Neural Information Processing Systems, volume 9, 648-654. The MIT Press.

[25] Soloway,E., Bonar,J., Ehrlich,K. 1983. Cognitive strategies and looping constructs: an empirical study. *CACM*, 26, 11, 853-860.

[26] Soloway E. & Spohrer, J. 1988. Studying the novice programmer. L. Erlbaum Assoc. Inc., Hillsdale, NJ, USA.

[27] Spohrer, J. & Soloway, E. 1986. Analyzing the high frequency bugs in novice programs. *Papers presented at the first workshop on empirical studies of programmers*, E. Soloway and S. Iyengar (Eds.). Ablex Publishing, Norwood, NJ, USA, 230-251.

[28] ACM Graphics library. Available from: http://www-cs-faculty.stanford.edu/~eroberts/jtf/ Accessed 9/1/2011.

[29] Berland, M. & Martin, T. 2011. Clusters and patterns of novice programmers. *AERA*, New Orleans, LA.

[30] Blikstein, P. & Worsley, M. (2011). Learning analytics: Assessing constructionist learning using machine learning. *AERA*, New Orleans, LA.

[31] Blikstein, P. (2011). Using learning analytics to assess students' behavior in open-ended programming tasks. *Proc. of the Learning Analytics Knowledge Conference*, Banff.

[32] Blikstein, P. 2008. An Atom is known by the company it keeps. Unpublished PhD. dissertation, Northwestern University, Evanston, IL.