Calculating Probabilistic Distance to Solution in a Complex Problem Solving Domain

Leigh Ann Sudol
Carnegie Mellon University
Pittsburgh, PA
leighann@cmu.edu

Kelly Rivers
Carnegie Mellon University
Pittsburgh, PA
krivers@andrew.cmu.edu

Thomas K. Harris
Tutor Technologies
Pittsburgh, PA
thomas@tutortechnologies.com

ABSTRACT

In complex problem solving domains, correct solutions are often comprised of a combination of individual components. Students usually go through several attempts, each attempt reflecting an individual solution state that can be observed during practice. Classic metrics to measure student performance over time rely on counting the number of submissions or focusing on time taken to complete the problem correctly. These metrics are not robust to the correction of errors that may increase problem solving time, and do not reflect topical misunderstandings on the part of the student. In this paper we propose a metric to measure the probabilistic distance between an observed student solution and a correct solution. Students working in an online programming environment completed four practice problems. Their submissions were then evaluated against a model of the algorithmic components necessary for a correct solution. A Markov Model was used to generate a problem state graph. Our proposed Probabilistic Distance to Solution (PDS) metric was applied to the graph to determine the distance, in program states, from an observed program model to the model of a correct solution. Results indicate that the PDS is useful in determining if an edit or student path is (a) typical of students who have mastered content, and (b) productive in progressing toward a solution. We offer implementation details of PDS and implications for future work based upon current observations.

1. INTRODUCTION

Modern data mining and classification techniques allow for increasingly complex solution spaces to be automatically modeled and assessed. For example, automated essay grading[10], mathematical proofs[1], and even complex computer programs[3] can be analyzed for completeness and scored. In these complex problem spaces, novices will often attempt several unique edits and approaches in order to create a finished correct solution. Intelligent Tutoring Systems (ITS) can be used to provide feedback for these attempts based on the scoring criteria used for the assessment. Although the models, feedback strategies and mechanisms used by each domain vary, it is still important for researchers to assess students' progress and make comparisons between research conditions in order to refine and improve such tutoring systems.

In complex problem solving spaces, such as natural language production or computer programming, students may make edits or submit attempts that are not directly related to the specific learning outcomes of the tutoring task[4]. For example, in computer programming, a student may struggle with a compilation error, such as having a parenthesis out of place, which is not reflective of their understanding of the desired learning goal. Students may also produce submissions that progress through multiple skills, creating a complex path to solution, with many possible states[8]. We propose that normal indicators of student performance within tutoring activities, such as time to completion or number of submissions, are too coarse-grained to distinguish between conceptual misunderstanding and syntactical or parsing mistakes that take time and multiple submissions to debug and correct.

In this paper we propose a new metric, Probabilistic Distance to Solution (PDS) and describe its implementation in assessing student progress on an introductory programming assignment. We then apply this metric to a dataset and highlight cases where PDS offers additional insight into misconceptions and problem solving paths.

2. PERFORMANCE METRICS

Within-tutor measures of performance are sometimes used instead of running pretests and posttests outside of the tutor, when the creator of the tutor wants more immediate learning feedback. Some within-tutor metrics have already been created and used effectively in tutors; for example, number of submissions and amount of time taken to get to a correct state were used in a system focused on improving math scores[2]. These metrics are not as effective in complex problem solving domains, however, due to the variety of strategies used to solve problems and the difficulty of merging them[5].

Other Intelligent Tutoring Systems use constraint-based modeling to determine how well a program matches up to the expectations of the problem; for example, Mitrovic built an ITS for SQL that used over six hundred constraints to provide accurate and useful hints to students[7]. Le and Menzel also describe techniques for building constraint-based tutors in 'ill-defined domains', similar to the complex domains we describe[6]. However, both of these approaches require that the author of the ITS generate the constraints by hand, which becomes very time-consuming when applied to a broad domain (such as programming). The metric we propose aims to improve on these models by examining more fine-grained aspects of the problem states in a potentially automatable way[1].

3. THE DATA

During the fall of 2011 and winter of 2012, eighteen participants solved four programming problems in an online tutor[9] for computer science. The problems were presented in the same order for each student so that comparison's between students' performances on problem X could be made without the confound of which problems preceded it. Each submission recorded data on the program's text, start and finish time, whether the attempt was successful, and the feedback that was given. Participants' submissions were then evaluated under the model described above and were coded appropriately. Participants generated 354 submissions with 63 observed distinct model states.

3.1 Feature Space

For each program submission, the participant's code was translated to a vector of binary features representing the inclusion of semantic program features, correctness outcomes, and compilation success. The semantic program features focuses on the inclusion of algorithmic components such as a looping structure, decision structure, use of the loop control variable in an array access, and inclusion of a return statement. The data collected represents a model of programming as a time series within a high dimensional binary space. The algorithms required by the four problems were similar in terms of features required. This model of required algorithmic elements was based upon the thesis proposal of the first author, and employs identifying the use of semantic structures as well as correctness testing using JUnit.

3.2 Two similar solutions

Figure 1 illustrates two participants' paths from an empty start state to a correct finished state. State A represents code that has all of the correct algorithmic components, is compilable, but does not return the appropriate value. State B represents code that has all of the correct algorithmic components, but does not compile, so it cannot check the final return state. Participant 5 corrected a sign error (step 2), then a syntax error (step 3), and finally another sign error (step 4), which resulted in a correct solution. Participant 12, on the other hand, initially submitted code that did not contain a return statement, indicating a misunderstanding about how information is communicated back from the function. State C represents an observed model state where three of the features are marked incorrect for the submission. The participant then made a small change resulting in the same model state for the code (step 2), then added a return statement based on a compile message (step 3), and finally fixed another compile error and submitted a correct solution (step 4). Although these two participants have the same number of submissions, the reasons for and the nature of the submissions are very different and expose a misconception by Participant 12.

Because each submit may represent multiple edits or steps in the problem solving process, simply counting the number of submits as a measure of errors across steps is not informative enough to express the difference between students who make errors with regard to the learning goals of the activity, and errors that do not inform measures of desired learning outcomes.

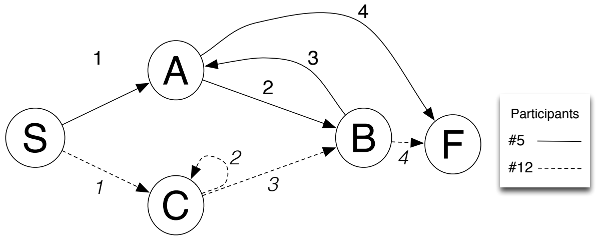


Figure 1: Comparing two student paths - Problem 2

3.3 Traditional Measures of Performance

Students tended to reduce overall time taken to solve problems as they moved through the set of problems (see Table 1, where SD is the standard deviation for the indicated problem). Students were performing think-aloud protocols while completing the problems, making the time to submit slightly exaggerated due to verbalization.

Problem #		Mean/Median	Min	Max	SD
# of Submits	1	4.06 / 3	1	14	3.13
	2	7.44 / 4	1	49	11.35
	3	4.56 / 2	1	23	5.71
	4	3.61 / 3	1	9	2.45
Overall Time in Seconds	1	625 / 452	93	2121	510
	2	571 / 364	134	1795	490
	3	437 / 331	103	1438	349
	4	363 / 315	53	1199	279

Table 1: Traditional Statistics for Student Data

The ability level of individual participants varied greatly, with some participants submitting final solutions with minimal modifications from their first attempt, while other participants struggled and progressed through multiple incorrect model states before arriving at a correct solution. Individual participants were consistent in their performance across problems either doing well or struggling with all of them. The traditional measurement metrics can be used to separate participants into two groups: high performers (students who were able to quickly solve the problems), and low performers (students who needed time and several attempts to get a problem right).

Of the seven students requiring more than six submissions to solve at least one problem, only two averaged fewer than six submissions per problem. These seven students also tended to take more than 500 seconds (8.33 minutes) overall to solve their problems, apart from the two mentioned above, who have individual outliers above that line. This disjoint grouping suggests that we can subdivide the low performing group into students who performed uniformly badly and students who struggled only with a specific problem.

The eleven high performers all averaged four or fewer submissions to reach a correct answer, and all clustered under an average of 400 seconds (6.66 minutes), with the exception of one student who took nearly eighteen minutes to finish the first problem, but only needed to submit once.

[1] Automation of this model's generation will be done in upcoming work, as proposed in [9].

4. PROBABILISTIC DISTANCE TO SOLUTION

In order to draw generalizations about how program states correspond to student performance and other latent factors such as learning, we aggregated all student submission paths for each problem into a network (see Figure 2). The network nodes $S_1 \cdots S_{n-1}$ are possible program states with an end state node E, and the edges are the observed transitions between states.

For each node, we use our observations to compute a Maximum Liklihood Estimate (MLE) transition probabilities to every other node. Given the number of observed transitions from state x to state y ($T_{x,y}$), we estimate the probability of being in state y at time t, with the MLE:

$$\hat{p}(S_y(t)) = P(S_y(t)|S_x(t-1)) = \frac{T_{x,y}}{\sum\limits_i T_{x,i}} \tag{1}$$

This is a equivalent to a Markov chain estimate with a 1-state history.[2]

By modeling each edge as a transition probability and distance between states, we use a set of linear equations to calculate a mean distance from each state to the end (successful completion) state. With

- $n-1$ non-terminal states $S_1 \cdots S_{n-1}$ and an end state E,
- and with each state S having transition probabilities $P_{s,1} \cdots P_{s,n-1}$ and $P_{s,e}$,
- and transition distances $D_{s,1} \cdots D_{s,n-1}$ and $D_{s,e}$,

a system of equations for the mean distance to the end state $d_e(S)$ is:

$$\begin{aligned} d_e(S_1) &= \left[\sum_{s=1}^{n-1} P_{1,s}(D_{1,s} + d_e(S_s))\right] + P_{1,e}D_{1,e} && (2)\\ d_e(S_2) &= \left[\sum_{s=1}^{n-1} P_{2,s}(D_{2,s} + d_e(S_s))\right] + P_{2,e}D_{2,e} && (3)\\ &\vdots && (4)\\ d_e(S_{n-1}) &= \left[\sum_{s=1}^{n-1} P_{n-1,s}(D_{n-1,s} + d_e(S_s))\right] \\ &\quad + P_{n-1,e}D_{n-1,e} && (5)\\ d_e(E) &= 0 && (6) \end{aligned}$$

For the case where we are interested only in the mean number of *submissions* to the end state, each $D_{x,y} = 1$, and the calculation simplifies to the system of dot products:

$$\begin{aligned} d_e(S_1) &= \vec{\boldsymbol{P}}_{\boldsymbol{1}} \bullet d_e(\vec{\boldsymbol{S}}) + 1 && (7)\\ d_e(S_2) &= \vec{\boldsymbol{P}}_{\boldsymbol{2}} \bullet d_e(\vec{\boldsymbol{S}}) + 1 && (8)\\ &\vdots && (9)\\ d_e(S_{n-1}) &= \vec{\boldsymbol{P}}_{\boldsymbol{n-1}} \bullet d_e(\vec{\boldsymbol{S}}) + 1 && (10)\\ d_e(E) &= 0 && (11) \end{aligned}$$

There are several variations on the algorithm that we would like to explore, given a larger dataset with more statical power. In particular, these variations include treating state-features differently, so that transitions between states that differ in particular features (e.g., compilability) will be able to indicate different latent variables of the students, (e.g., example concept understanding).

5. APPLYING PDS

The PDS metrics accompanied by the transition graph are rich sources of information about the paths that participants pursued in order to arrive at a correct solution. Figure 2 includes a table illustrating the observed model states in the binary vector, as well as the PDS for each state. In problem 4, S56 is the solution state and S0 the initial starting state. Before even evaluating the student paths, we can observe that an additional state, S63, was also a terminal state for a participant. This participant located a bug in the evaluation system that has since been corrected.[3]

By looking at the PDS combined with the Program State Graph (PSG) we can identify more productive edits by participants. For example one participant's first submission was observed as S55 (PDS 2.99), and next state was S57 (PDS 4.43). This edit would be less productive as it resulted in a transition to a state with a greater probabilistic number of submits required to obtain a correct solution.

With the possibility of including terms in the algorithm for syntactic but not model changes (i.e. two states that are identical except for a compilation error would not be counted as a full step), PDS can be adapted to focus on model state transitions that indicate misconceptions or other model-based goals of the data miner.

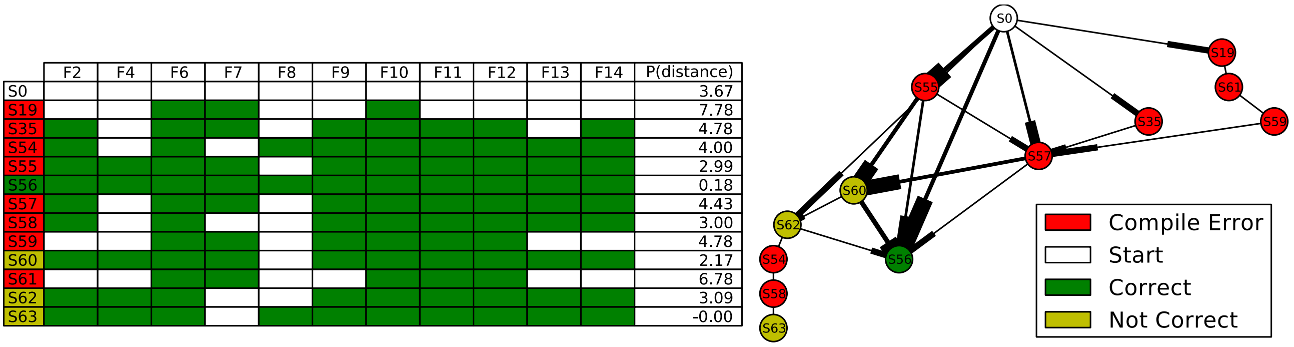
6. CONCLUSIONS

Within these early results, we have already identified model states on productive and unproductive PDS paths. Using the actual PDS values we can determine if a student is making a productive edit, engaging in either guessing behavior, or pursuing a misconception. An edit resulting in an observed state with a higher PDS than the prior submission indicates a move away from a correct answer.

These values can be invaluable to tutor designers as they seek to develop feedback and support tools for complex solution domains. Within computer programming tutors, the PDS could offer implications for more-than-compiler support, and perhaps even prompt the introduction of a similar worked example or code comprehension problem highlighting the incorrect features of the model.

[2] We believe that the student's state transitions will be better represented by a higher-order Markov process; however our current data set is too small to provide appropriate power for more than a first-order analysis.

[3] The bug was identified as a part of the think aloud protocol, however the PDS and Student Program States Graph would have identified the bug for the tutor designers as well.



	F2	F4	F6	F7	F8	F9	F10	F11	F12	F13	F14	P(distance)
S0												3.67
S19			🟩	🟩			🟩					7.78
S35	🟩		🟩	🟩		🟩	🟩	🟩	🟩		🟩	4.78
S54	🟩		🟩		🟩	🟩	🟩	🟩	🟩	🟩	🟩	4.00
S55	🟩	🟩	🟩	🟩		🟩	🟩	🟩	🟩	🟩	🟩	2.99
S56	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	0.18
S57	🟩		🟩	🟩		🟩	🟩	🟩	🟩	🟩	🟩	4.43
S58	🟩		🟩			🟩	🟩	🟩	🟩	🟩	🟩	3.00
S59			🟩	🟩		🟩	🟩	🟩	🟩			4.78
S60	🟩	🟩	🟩	🟩		🟩	🟩	🟩	🟩	🟩	🟩	2.17
S61			🟩	🟩			🟩	🟩	🟩			6.78
S62	🟩	🟩	🟩			🟩	🟩	🟩	🟩	🟩	🟩	3.09
S63	🟩	🟩	🟩		🟩	🟩	🟩	🟩	🟩	🟩	🟩	-0.00

Figure 2: Student program states for problem 4. Table columns Fx are binary program features. Table rows Sx are observed combinations of those features in program submissions. Only observed states (nodes) and transitions (edges) are shown. Node self-transitions also exist in the model, but are not shown here. The thicknesses of the edges are proportional to the log of observed transitions in the data. The lengths of the edges are arbitrary and do not relate to the model.

Although tested against data from a computer programming dataset, the authors believe that the PDS metric could be valuable across many domains with complex solutions demonstrating multiple skills. Future work is planned to use the PDS metric on a larger dataset to extract common paths and evaluate differences between tutoring conditions.

7. ACKNOWLEDGEMENTS

This work was supported through the Program for Interdisciplinary Education Research (PIER) at Carnegie Mellon University, funded through Grant R305B040063 to Carnegie Mellon University, from the Institute of Education Sciences, US Department of Education. The opinions expressed are those of the authors and do not represent the views of the Institute or the US Department of Education. We would also like to thank Dr. Christy McGuire of Tutor Technologies for her assistance in preparing this manuscript.

8. REFERENCES

[1] Tiffany Barnes and John C. Stamper. Automatic Hint Generation for Logic Proof Tutoring Using Historical Data. *Educational Technology & Society*, 13(1):3–12, 2010.

[2] Mingyu Feng, Neil T. Heffernan, and Kenneth R. Koedinger. Predicting State Test Scores Better with Intelligent Tutoring Systems: Developing Metrics to Measure Assistance Required. In *Proceedings of the 8th International Conference on Intelligent Tutoring Systems*, pages 31–40, 2006.

[3] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pages 86–93, 2010.

[4] Matthew C. Jadud. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, 15(1):25–40, 2005.

[5] H. Chad Lane and Kurt VanLehn. Intention-Based Scoring: An Approach to Measuring Success at Solving the Composition Problem. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 373–377, 2005.

[6] Nguyen-Thinh Le and Wolfgang Menzel. Using Constraint-Based Modelling to Describe the Solution Space of Ill-defined Problems in Logic Programming. In *Advances in Web Based Learning (ICSL 2007)*, pages 367–379, 2007.

[7] Antonija Mitrovic. An Intelligent SQL Tutor on the Web. *International Journal of Artificial Intelligence in Education*, 13(2-4):173–197, 2003.

[8] James Spohrer, Elliot Soloway, and Edgar Pope. A goal/plan analysis of buggy pascal programs. *Human Computer Interaction*, 1:463–207, 1985.

[9] Leigh Ann Sudol. Deepening Students' Understanding of Algorithms: Effects of Problem Context and Feedback Regarding Algorithmic Abstraction. *Carnegie Mellon Thesis Proposal*, 2011.

[10] Salvatore Valenti, Francesca Neri, and Alessandro Cucchiarelli. An Overview of Current Research on Automated Essay Grading. *Journal of Information Technology Education*, 2:319–330, 2003.