2015-04-23

Enhancing Personalization Within ASSISTments

Christopher Donnelly
Worcester Polytechnic Institute

ENHANCING PERSONALIZATION WITHIN ASSISTMENTS

by

Christopher John Donnelly

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

May 2015

APPROVED:

Professor Neil Heffernan, Advisor

Professor Mark Claypool

Abstract

ASSISTments is an online adaptive tutoring system with the ability to provide assistance to students in the form of hints and scaffolding. ASSISTments has many features to help students improve their knowledge. Researchers run studies in order to discover ways for students to learn better but ASSISTments is missing one major aspect for researchers: student level personalization. It is easy to create an assignment for a particular class or school but it would take much longer to create an assignment for each student and it would be difficult for the teacher to look through many assignment reports. One of the strongest code blocks in coding is the if-then; allowing the program to branch off to another set of code under certain circumstances. ASSISTments needed an if-then system in order for students to branch off to other parts of the assignment under certain circumstances. With this, researchers would be able to personalize assignments to give more help to lower knowledge students or allow students to get a choice of what kind of tutoring they would like to receive. With this idea in mind, the basic if-then structure was implemented into ASSISTments using problem or problem set correctness as the condition statement. Once the if-then system was created opportunities opened to create additional experiments and run studies in ASSISTments. The basic if-then was limited in using correctness only for its condition statement. This meant that a new if-then system would need to be implemented to include custom condition statements that allowed the researcher to have the assignment branch on any condition using all the information recorded in the assignment. While work was being done on the if-then system, research was being done and two papers were written on partial credit in ASSISTments. Partial credit was found out to be as accurate as knowledge tracing in determining student performance on the next problem. Once a partial credit algorithm was found, a study using if-then was analyzed. It was found that there was no statistically significant difference between students who were given a choice on their feedback and students who received no choice.

Acknowledgements

I acknowledge funding from NSF (1316736, 1252297, 1109483, 1031398, 0742503, 1440753), ONR's "STEM Grand Challenges," and IES (R305A120125, R305C100024). Thanks to Neil Heffernan, Korinn Ostrow, and David Magid.

Table of Contents

Table of Figures

Table of Tables

Chapter 1. Introduction

ASSISTments is an online adaptive tutoring system used by over 50,000 students around the world that is offered as a free service of Worcester Polytechnic Institute. The platform was created over a decade ago by Neil Heffernan. With the help of numerous graduate and undergraduate students, ASSISTments has evolved steadily since its inception. The user population has doubled each year and new features and preferences are consistently in development for teachers and students.

The ASSISTments platform uniquely provides assistance to students in the form of adaptive feedback. This can take many forms, from simply correctness feedback to tutoring strategies including hints and scaffolding problems that provide worked examples. It also provides teachers with powerful assessment through a variety of reports, allowing teachers to easily isolate where students are struggling and enhance classroom techniques using a data driven model.

Currently in ASSISTments there is no student level personalization in assignments. If a researcher or a teacher wanted to assign problem sets based on how well a student did they would need to separate these problem sets and use data driven rules to assign to the students a specific problem set. ASSISTments also does not support giving a student a choice during their assignment. If a student wanted to get video feedback versus text feedback that is not possible in the current system.

The next chapter in this paper will explain the process involved in adding personalization and choice to ASSISTments through an if-then system. The first section will talk about the tutor and how it works which will make explaining the if-then system much easier. In section 2.2, the basic if-then system will be discussed. The basic if-then system deals only with correctness and lays a foundation for the advanced system. Next, the advanced if-then system will be covered. This includes all the work in creating a custom language and grammar, along with the parser and the new if-then system itself. The advanced if-then system will allow for greater personalization and also open up a new infrastructure for researchers to use. Section 2.4 will cover that new infrastructure, which is variables being assigned and used in if statements. Researchers will be able to give variables their own values or read variables that are created by the ASSISTments system and use them in their if statements for even greater customization and personalization.

Chapters three and four discuss two papers written in parallel with the work done on the if-then structure. Chapter three discusses research done with partial credit and how it can be used to determine next problem correctness. Partial credit can show the difference between a student who does not know how to solve the problem and a student who made a minor mistake on the problem. Chapter four covers a grid search about finding the best partial credit algorithm. The best partial credit algorithm would give the students a score between 0 and 100, which correctly models student knowledge, instead of a correct/incorrect score, which is marked incorrect at the slightest mistake. Partial credit will be used in future analyses to determine differences between conditions; currently binary credit is the only thing used but partial credit will be able to provide more information.

The fifth chapter covers the studies currently running using the if-then system and continues on to analyze data obtained from another one of the if-then studies. The study analyzed is a choice study and shows how if-then can be used not only as a video check to remove students who cannot see video, but also a way to allow students to have a choice in what they get for feedback. There are no significant results dealing with choice versus no-choice yet as there are only sixty students to have completed the study. The analysis does show, at this point, that students need to complete more problems to finish the assignment when given choice compared to those without choice. In terms of binary score and partial credit score, students in the choice group scored lower than those in the no-choice group. This study shows only one use of the if-then system; there are over a hundred studies running using if-then right now. All of these studies are using if-then in a slightly different way.

Chapter 2. The If-Then Structure

2.1 The Tutor

The ASSISTments tutor is the application that runs when a student runs an assignment in ASSISTments. It is written in Java that is translated to javascript using the Google Web Toolkit (GWT). An assignment is an instantiation of a particular problem set. Each problem set contains problems and other problem sets. In its simplest form the tutor is a set of navigators and terminators working through the problem sets to finish the assignment. Each problem set has one navigator and terminator pair. The terminator tells the tutor when this particular problem set is finished and if this is the main problem set then the tutor tells the student the assignment is finished. If it is not the main problem set then the parent problem set's navigator is told to move to the next problem or problem set. Problems and problem sets are represented with manifests in the tutor. A manifest contains a pointer to the actual problem or problem set along with properties that belong to that problem or problem set. Manifest will be used to talk about a problem or problem set throughout the rest of this paper so as not to repeat the phrase "problem or problem set" too many times. Navigators, in the tutor, decide which manifest to run. If the terminator is not finished then the navigator is told to go find another manifest from its list and the tutor will run that manifest. Running the manifest depends on whether its a problem or problem set. Running a problem set means creating a navigator and terminator for it and then having its navigator and terminator take control until that problem set is finished. Running a problem simply means printing the problem to the screen for the student to attempt. An example assignment would be a skillbuilder. Skillbuilders in ASSISTments are assignments that students complete in order to prove they know a specific skill. In order to complete a skillbuilder the student must get a certain number (usually three) of problems correct in a row. In the tutor this would be a RandomNavigator and a CorrectInARowTerminator. The navigator randomly selects problems from the list that belongs to this problem set while the terminator keeps a counter of how many problems the student has correct in a row. Once the student hits 3 problems the terminator tells the navigator it's all done and the tutor then alerts the student that the assignment is complete.

2.2 Basic If-Then Structure

The base if-then structure was created in collaboration with David Magid. This project included adding the if-then problem set to the builder along with adding the support for this type of problem set into the tutor. This new problem set was envisioned to work exactly like an if-then statement in coding. Figure 2-1 shows the basic workings of an if-then statement in code. The if-then problem set will consist of three children manifests. Each child manifest corresponds to a different portion of the if-then statement. The first child would be the condition statement and will evaluate to either true or false depending on whether or not the child was considered correct. The second child would be the then statement and will run if the condition is true; the third child will be the else statement and run if the condition is false.

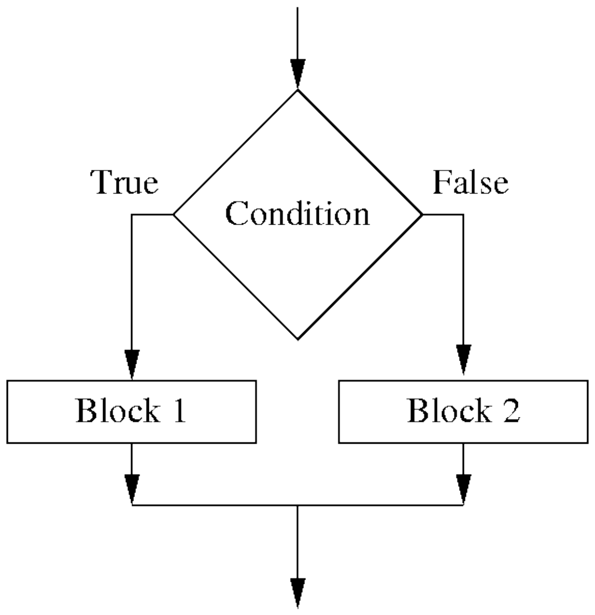


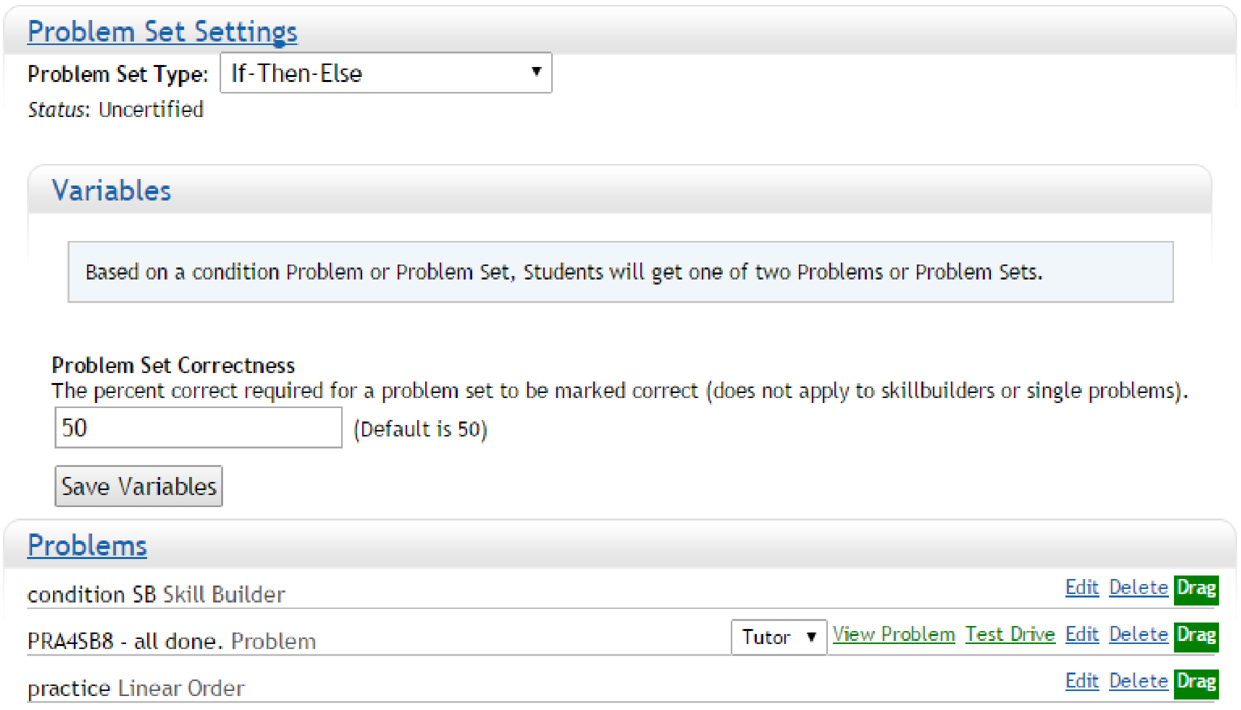
Figure 2-1. Workflow of an if-then statement in coding. The parser comes in and reads the condition statement. If the condition statement evaluates to true then the parser goes and runs code block 1. If the condition statement evaluates to false then the parser goes and runs code block 2.

Once the if-then problem set was designed the tutor had to be updated to actually understand how to run the problem set. Since the tutor needed to only complete two of the three children manifests (condition and one of the branches), the IfThenElseTerminator was a simple class that extended a current terminator and that told the tutor to only complete two items. The difficult part was creating the IfThenElseNavigator as this would have to run the condition manifest and then determine if it should evaluate to true or false and then run the correct branch manifest. If the condition manifest is only a problem then evaluating true or false is easy as that is whether or not the student got the problem correct. ASSISTments already records the problem's correctness and the only thing that had to be done was get that correctness to the navigator. Unfortunately the idea of correctness in a problem set did not exist until now. It was eventually decided that researchers may all have a different idea of correctness for a problem set so a setting was created in the builder. If the problem set is a skillbuilder, correct is determined by whether or not the student mastered the skillbuilder (got X right in a row). If it is any other problem set then correct is determined by the percentage of problems the student got correct out of the total number of problems the student has seen. The condition is marked correct (true) if this percentage is greater than the percentage from the builder setting. Table 2-1 summarizes how and when the if-then condition statement is marked true or false.

Table 2-1. Basic if-then condition statement matrix. This table shows the requirements for a condition statement to be evaluated as true or false. There are only three different manifests that could end up being the condition: a problem, a skillbuilder problem set, and any other problem set.

Manifest type	True	False
Problem	Correct	Incorrect
Skillbuilder	Mastered (X right in a row)	Did not master
Other Problem Sets	Correct/Total seen > Setting value	Correct/Total seen < Setting value

Now that correctness was determined, all that was left was getting the correctness to the navigator. The idea of FinishedConditions and a FinishedConditionService was then implemented. The FinishedConditionService is a service separate from the actual tutor that listens for student actions and updates FinishedConditions depending on certain actions. In this first iteration all the service cared about was the first student response action (when a student enters his/her first answer). The service then created a FinishedCondition for the problem the student is on and marked it as either correct or incorrect. A FinishedCondition is an object that contains data about a problem or problem set to be used at a later time. When a problem set is finished the service finds all the problem FinishedConditions that belong to that problem set and counts up all the correct and incorrect problems and stores those numbers in the problem set FinishedCondition. If the problem set was a skillbuilder then the FinishedCondition also contains whether or not the student mastered the skillbuilder. Now when the student finishes the condition manifest the IfThenElseNavigator goes to the FinishedConditionService and asks for the correct FinishedCondition and is able to determine which branch manifest to run. After it decides which manifest to run the terminator will complete since the student finished two items and then the if-then problem set is complete. The if-then problem set was then released to the public and is currently being used in numerous experiments. Figure 2-2 shows an example If-Then problem set. In this example there is a skillbuilder condition. If the student masters the skillbuilder they are given problem PRA4SB8 which is a simple placeholder problem telling them they have finished the assignment. If they do not master the skillbuilder, then they move onto the practice problem set. “Problem Set Correctness” is not used in this case because the condition is a skillbuilder but would decide on the ratio between correct and incorrect problems needed to mark a problem set as correct


Problem Set Settings

Problem Set Type: If-Then-Else

Status: Uncertified

Variables

Based on a condition Problem or Problem Set, Students will get one of two Problems or Problem Sets.

Problem Set Correctness
The percent correct required for a problem set to be marked correct (does not apply to skillbuilders or single problems).

50 (Default is 50)

Problems

condition SB Skill Builder

PRA4SB8 - all done. Problem

practice Linear Order

Figure 2-2. Basic layout of an if-then problem set.

Once the basic if-then structure was released the next step was to create a more advanced version that would allow for more customization and did not only rely on correctness of a problem or problem set. Researchers would be able to write their own condition statements which allowed them to branch to either of the branch manifests on any condition they choose. The advanced version would not require all three children manifests and could access data from either a condition problem/problem set or from some variable map with user specific data or custom data set by the researcher. The first stage of the advanced if-then structure included building a parser to understand the custom condition statements the researchers write. That meant creating a language and a grammar for that language to help in building the parser and also guide researchers on how to construct their condition statements.

2.3 Advanced If Then

2.3.1 Create a language/grammar

To prevent confusion and allow for easily understandable condition statements, it was decided that the language will be written using a syntax like Java.

Example condition statements:
problem.getCorrectness() > 20
(pr.isCorrect() == false) && (pr.getHintRequestCount() < 2)

The language had been created and defined in an Extended Backus–Naur Form (EBNF) Grammar. Having a language defined in this form will help in not only explaining the language that researchers can use but also designing and coding the parser to understand the researchers conditions. EBNF is a set of notations used to express a context-free grammar and is an extension of the basic Backus-Naur Form. Backus-Naur Form (BNF) is a notation for context-free grammars for describing programming languages in a human readable format (Might). EBNF was chosen because of the expressions that are allowed to make rules smaller and easier to read. Figure 2-3 shows a subset of the rules from the created grammar. Refer to Appendix A for the full EBNF grammar.

```
start        = expr ;
expr         = rel_expr | prop_expr | “(“, expr, and, expr, “)” | “(“, expr, or, expr, “)” ;
rel_expr     = lhs, rel_op, rhs;
lhs          = “pr.”, p_method | “ps.”, ps_method;
rel_op       = “==” | “<” | “<=” | “>” | “>=” | “!=”;
rhs          = string | decimal | number | true | false;
```

Figure 2-3. This shortened grammar defines condition statements that access data from the FinishedCondition for the condition manifest. A handful of rules have been removed to save space but the full grammar can be found in APPENDIX A.

2.3.2 Creating the parser

After creating the grammar, research was done to find a program that exists that would read in an EBNF grammar and create a parser. JavaCC, an open source parser generator, was chosen to build the parser. A parser generator is a tool that reads a grammar specification and converts it to a program that can recognize matches to the grammar (JavaCC). JavaCC generators top-down (recursive descent) parsers which allows for the use of more general grammars and many other advantages over bottom-up parsers. Top-down parsers are able to parse any non-terminal in the grammar and pass values both up and down the parse tree (JavaCC Features). JavaCC reads in a config file that contains the grammar re-written using JavaCC's syntax along with extra Java code. Given that config file, JavaCC then builds the parser and outputs a set of java files that can be dropped into any project and used right away. One major difficulty in building the parser was that it needed to be small and also have all the source java code along with the class files. In order for GWT to translate all the java code into javascript, to be used in browsers, it needs to have all the java source code. Most other parser generators required their specific jar to be included in the project but those jar files never contained source code. JavaCC on the other hand, created Java files so that GWT would be able to translate everything into javascript without a problem. One problem did arise though; JavaCC used Java's built-in I/O classes to read and walk through the characters in the string it was parsing. This is a problem because in a browser there is no file system and no I/O. Fortunately, the newest beta release of JavaCC allowed for an option to build specifically for GWT (Ainsley). This particular build created a custom CharStream class to walk through the characters in the string being parsed and allowed the use of this parser in the tutor.

2.3.3 Adding custom condition

Once the parser was built the next step was to put it into the if-then structure and allow researchers to write their own condition statements. If-Then problem sets would still contain three children manifests but this time the condition would not be limited to whether the condition manfest was correct or not. The condition would be dependent on a custom statement written by the researchers. This meant editing how the second and third child was chosen and adding additional data to the FinishedConditions so there was more data than just a problem's correctness. The first thing that had to be done was to update FinishedConditions with more information. In creating the basic if-then structure, a FinishedConditionService was created in the tutor which listened for student actions (ProblemStarted, StudentResponse, ProblemFinished, etc.) and would create the FinishedCondition for that problem or problem set when the student finished the particular problem or problem set. Adding additional information required a bit of code refactoring but was fairly straight forward. The service had to be told to listen for more student actions and then record the data from each action into the FinishedCondition object. Figure 2-4 shows the actions created while a student works on a problem and how the old FinishedConditionService responded to these actions compared to the new FinishedConditionService.

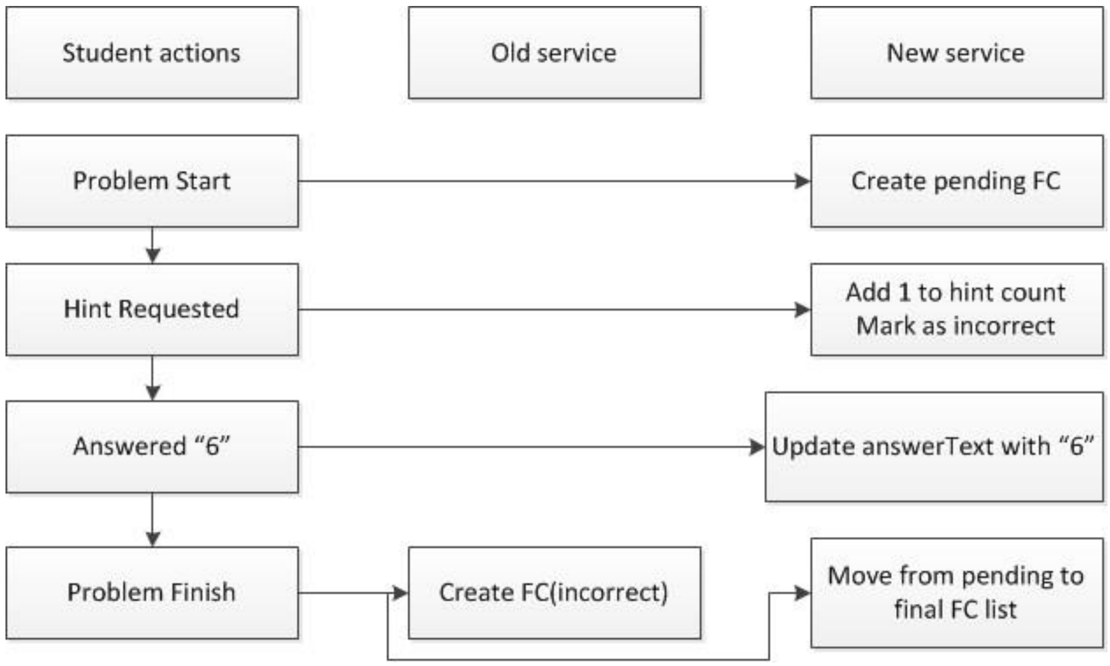


Figure 2-4. How the old FinishedConditionService worked versus the new service. In the new service, it would listen for each action and do something to the current FinishedCondition. In the old service, it would wait until the end to create the FinishedCondition.

Now that FinishedConditions were updated with new information, the if-then navigator had to be updated to use the new parser and the updated FinishedConditions in order to determine which of the last two children manifests to run. The FinishedConditions were setup with a set of functions that allowed outside objects (like the parser) to get access to the additional data stored. The parser was setup to accept specific function strings so that when it read those functions it would call off to the FinishedConditions accessor functions to retrieve the data and then compare the returned values to those in the rest of the condition.

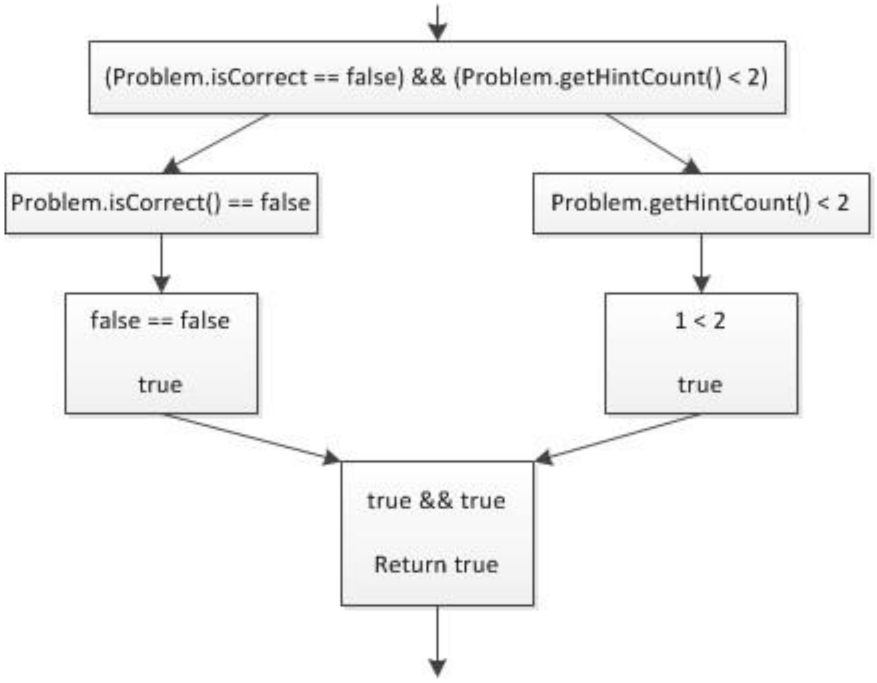

(Problem.isCorrect == false) && (Problem.getHintCount() < 2)
Problem.isCorrect() == false
Problem.getHintCount() < 2
false == false
true
1 < 2
true
true && true
Return true

Figure 2-5. An example condition statement being parsed.

Now, when a student finishes the condition manifest, the if-then navigator will pass the corresponding finishedCondition and the condition statement to the parser and the parser will return a true (run the second child manifest) or a false (run the third child manifest) depending on the condition statement. Figure 2-5 shows an example condition statement being parsed. The parser splits the main statement into two expressions and then sees the problem tokens. The problem token tells the parser to access the FinishedCondition and use those function calls to evaluate the expressions. After parsing it returns the values up the tree until it is left with either a true or false and then returns that to the if-then navigator.

2.4 Custom Variables

The original plan was to spend some time creating a prettier user interface (UI) for the builder when creating If-Then problem sets. The main idea was to remove the need for the researcher to write their condition statement and instead use building blocks they would drag and drop and to form the condition statement without running the risk of spelling or syntax errors. This idea was put aside to save time and to allow for more time to go toward including the user model and custom variables in the tutor and builder. As for the user model, this was pushed aside until after the custom variables were implemented because the user model information could be stored in the same manner as the custom variables and the parser would need to be updated to support both anyway.

The next goal was to allow researchers to add custom variable assignments into the builder and have the tutor keep track of them and the parser to have access to them. A couple of use cases for custom variables include, cross assignment variables and setting student knowledge. Cross assignment variables covers the idea that a variable was assigned a value during assignment one and that value will be used/read in an if statement in assignment two. An example would be if a student was part of an experiment and in their first assignment was put in condition group one. The variable would be assigned a value “condition group one” and that would then be used on the next assignment to either keep the student in condition group one or force the student into a different condition group. Setting student knowledge is as simple as assigning a variable to “low” or “high” knowledge and this variable is saved for later use in that assignment or use in a different assignment. Knowing a student’s knowledge can be helpful with deciding what set of problems to give the student.

Since time was running out most of the coding was done to create an infrastructure for these variables and then support for them was added. That meant that many of the use cases for custom variables weren’t fully implemented but the infrastructure exists to support the implementations when they are done. It was decided that variables could be assigned values at two points during the execution of a manifest in the tutor; before the manifest is run, and after. These variables assignments were deemed PreVariableAssignments (run before) and PostVariableAssignments (run after). This was believed to be the best way to give researchers any opportunity to use custom variables. Once the builder was setup to allow for custom variable assignments, the tutor had to be updated to read them in and store them for use in the parser. The tutor was fairly complicated as variables could be used for so many things in the future the variable assignments had to be run at the correct time. PostVariableAssignments were run once the problem or problem set had finished and was fairly straight forward but PreVariableAssignments were harder. ASSISTments has received requests to allow variables in problems and allow variables to override certain settings in the manifest (for example the number correct in a row needed to master a skillbuilder). With this in mind, PreVariableAssignments needed to be run at a specific time in order for their values to be ready in case they were to be used for this particular problem/problem set. After running tests and exploring more about the tutor it turns out that the variable assignments can be run like the FinishedConditionService. There is another service in the tutor called the StateService, which was used to store the variable values. This service also listens to specific student actions and when those actions come through, the service was modified to run the pre and post variable assignments. Since most of the tutor is event driven and javascript is a single threaded environment, it's possible to just wait for the student's actions to come through to the StateService to run the variable assignments because every service needs to process each action before any new action or event is processed. This means that the small handful of events that tell the tutor to move onto the next problem or problem set will not be run until after everything has finished processing the current student actions. We now have an infrastructure for assigning variables values and retrieving these values.

The last thing left is to give the parser access to these variables and allow the if-then problem sets to use them. This meant that the grammar would need to be changed and the parser would need to be updated. Thankfully JavaCC makes this extremely easy. This involved updating the grammar config file to expect variable tokens along with problem/problem set tokens and then rebuilding the parser. With the new parser, the if-then navigator had to pass a pointer to the StateService to the parser so when the parser reads a variable token it will go and ask the StateService for the value of that variable and continue on with parsing like it had done before.

Chapter 3. Improving Student Modeling Through Partial Credit and Problem Difficulty

This chapter starts discussing research that was done while work was being done on the if-then system. The paper in chapter three introduces the idea of partial credit and how it can be used to analyze data instead of the binary scores currently being used for problem correctness. The idea of partial credit will be expanded upon in chapter four and used in the analysis in chapter five.

Student modeling within intelligent tutoring systems is a task largely driven by binary models that predict student knowledge or next problem correctness (i.e., Knowledge Tracing (KT)). However, using a binary construct for student assessment often causes researchers to overlook the feedback innate to these platforms. The present study considers a novel method of tabling an algorithmically determined partial credit score and problem difficulty bin for each student's current problem to predict both binary and partial next problem correctness. This study was conducted using log files from ASSISTments, an adaptive mathematics tutor, from the 2012-2013 school year. The dataset consisted of 338,297 problem logs linked to 15,253 unique student identification numbers. Findings suggest that an efficiently tabled model considering partial credit and problem difficulty performs about as well as KT on binary predictions of next problem correctness. This method provides the groundwork for modifying KT in an attempt to optimize student modeling.

This chapter has been published at the following venue:
Ostrow, K., Donnelly, C., Adjei, S., & Heffernan, N. (2015). Improving Student Modeling Through Partial Credit and Problem Difficulty. In Russell, Woolf & Kiczales (Eds), Proceedings of the 2nd Annual Meeting of the ACM Conference on Learning at Scale. Vancouver, British Columbia, March 14-15. pp .11-20

3.1 INTRODUCTION

Modeling student learning within an intelligent tutoring system can be a daunting task. In order to make predictions about a student's knowledge or their next problem correctness, models must decipher noisy input and isolate only those features that define the probability of knowledge or learning. As such, designers of intelligent tutoring systems have largely relied on Knowledge Tracing (KT), as presented by Corbett & Anderson (1995), to model the probability of student learning at real time within popular systems such as Cognitive Tutor (Koedinger & Corbett, 2006). Other methods, such Performance Factors Analysis, seek to model learning when considering overlapping knowledge components (i.e., skills) and individualized student metrics (Paylik, Cen, & Koedinger, 2009), offering an alternative to KT in certain circumstances.

Despite the popularity of KT and PFA, the standard models rely on binary input to establish predictions of students' knowledge state or performance, failing to consider continuous metrics that would better individualize the model across students or skills. Expansion in the field educational data mining has since lead to a number of alternative or supplementary learning models. For instance, researchers have attempted to impart individualized prior knowledge nodes for each student (Pardos & Heffernan, 2010), to supplement KT with a flexible metric for item difficulty (Pardos & Heffernan, 2011), to ensemble various methods of binning student performance (i.e., partial credit) with standard KT models (Wang & Heffernan, 2011), and to consider the sequence of a student's actions within the tutor to help predict next problem correctness (Duong, Zhu, Wang, & Heffernan, 2013).

Without modifying KT or PFA directly, adding parameters to student learning models can be a limited approach. Tabling methods that quickly establish maximum likelihood probabilities have previously been used by Wang & Heffernan (2011,2013) to test and optimize various potential adaptations to KT. Following in this process, the present study uses a tabling method to lay the groundwork for future modifications to KT that will allow for predictions of next problem correctness using the partial credit score and difficulty estimate of the current item. While previous work has shown the benefit of ensembling tabling methods with KT, we hope to use the findings presented herein to modify KT directly, as it has previously been suggested that ensembling can be a rather sensitive approach (Gowda, Baker, Pardos, & Heffernan, 2011).

Perhaps standard learning models rely on binary correctness as measured by a student's first response at each skill opportunity (i.e., a sequence of correct and incorrect responses based on a student's first action within each problem) due to the complexity of accurately and universally defining an algorithm that validates partial credit scores within intelligent tutoring systems. Within the majority of current learning models, a student would be penalized with a score of zero for taking advantage of the tutoring that plays an integral role in these platforms. Yet the primary goal of most intelligent tutoring systems is not solely to assess student knowledge, but to simultaneously promote student learning through adaptive feedback, making binary correctness a stale concept. Students often require multiple attempts to solve a problem or request system feedback for guidance, thus assigning value to the concept of partial credit. Attali and Powers (2010) suggested the benefits of considering partial credit when predicting learning outcomes in adaptive environments, as evidenced by the modification of standardized tests to allow partial credit when predicting GRE scores.

Within ASSISTments, an adaptive mathematics tutor, a naïve model of partial credit scoring was previously established by Wang and Heffernan (2011), termed the “Assistance” Model. This method calculated maximum likelihood probabilities for next problem correctness using a twelve-parameter table built from binning students’ hint usage and attempt count. In this manner, the authors used system features to indirectly gauge a partial credit metric that would help predict binary performance.

The present study provides methodological evidence that student modeling can be enhanced through the use of algorithmically derived partial credit scores and a binned metric of problem difficulty. We first use tabling method (a probabilistic approach employing maximum likelihood estimations) that considers the partial credit score of the current problem to predict both binary and partial next problem correctness. We also establish a more complex prediction table that considers both partial credit and problem difficulty. Through this novel concept, we hope to show that students can ultimately gain knowledge from a problem even if they fail to earn full credit. Our findings argue for the design of a modified KT model that is sensitive to a continuous measure of partial credit rather than binary input, and that isolates a known level of problem difficulty for each question. We seek to answer the following research questions:

1. Does an algorithmically determined partial credit score outperform binary metrics when used to predict next problem correctness?
2. Does a binned metric of current problem difficulty (e.g., Low, Medium, or High difficulty) provide a valid prediction of next problem correctness?
3. Can current problem difficulty supplement partial credit score to outperform similar modeling techniques?

3.2 DATASET

The dataset used for this analysis was compiled from problem logs from the ASSISTments platform during the 2012-2013 school year. The original file included roughly 1.5 million rows of problem level data (i.e., each row detailed all logged actions for one problem for one student). For this study, we chose to analyze only the top ten most densely populated knowledge components. Attributes of these skills are further explained in Table 3-1. The dataset examined here has been made publicly available at (Ostrow, L@S 2015 Submission, 2014).

In order to properly calculate partial credit, approximately 5,000 rows were removed due to a lack of logged end time, meaning that these problems had never been properly completed. Using the platform's current grading method, which is based on the students' first response, these logs carry binary correctness scores. However, as the problem was ultimately considered incomplete, partial credit could not be determined with certainty and the logs were therefore excluded from analysis. Further, the analysis presented herein reports only on main problems. Scaffolding problems, a feedback style within the ASSISTments platform typically used to break a problem down into steps or to provide worked examples, were excluded from the final dataset. The decision to work with main problems was based in part on the justification made by Pardos & Heffernan (2011) when using a similar dataset from the ASSISTments platform. As scaffolding problems are guided, they offer a less accurate view of skill knowledge and skew performance data within an opportunity based analysis. An analysis of the remaining dataset revealed that only 0.3% of first actions were scaffold requests, further supporting the intuition that the removal of scaffolding data was appropriate.

Due to the time constraints involved in running multiple models with five fold cross-validation (explained further in the Compared Models and Model Testing and Training sections), we chose to restrict the dataset to a maximum of 15 opportunities per student per skill. This reduced the dataset by 46,680 rows, primarily removing students who were excessively struggling and those gaming the system; the majority of students were unaffected by this refinement.

The resulting dataset consisted of 338,297 problem logs representative of 15,253 unique student identification numbers. On average, each student identifier linked to approximately 3.3 skills. Further exploration of this dataset revealed that it was comprised of 7,363 unique problems. A total of 3,787 unique assignments were made by 417 teachers spanning 231 schools. The skill content ranged from grades 6-8 as shown in Table 3-1. The majority of logged problems (over 90%) were completed by students who 'mastered' or finished the full assignment from which the problem originated.

Three types of questions were represented in the dataset. The majority of problems logged, 84.3%, were 'mathematical expressions,' a problem type that accepts any answer that is mathematically equivalent to the correct answer (i.e., answers of 1/2 and 0.5 are both accurate). In contrast, 12.5% of problems logged were 'fill-in,' a problem type that requires the student to input an *exact* string matching the preset correct response (i.e., if 1/2 was the preset answer, 0.5 would be incorrect). The remaining 3.2% of problems logged in the dataset were 'multiple choice,' featuring two or more answers available for selection.

Table 3-1Skill details and distribution in resulting dataset

Skill ID	Definition	Grade Level	# Logs	% Resulting
277	Addition and Subtraction of Integers	7	44,731	13.2
311	Equation Solving with Two or Fewer Steps	7	44,005	13.0
280	Addition and Subtraction of Fractions	6	42,550	12.6
276	Multiplication and Division of Positive	6	37,033	10.9
47	Conversion of Fractions, Decimals, and	6	32,741	9.7

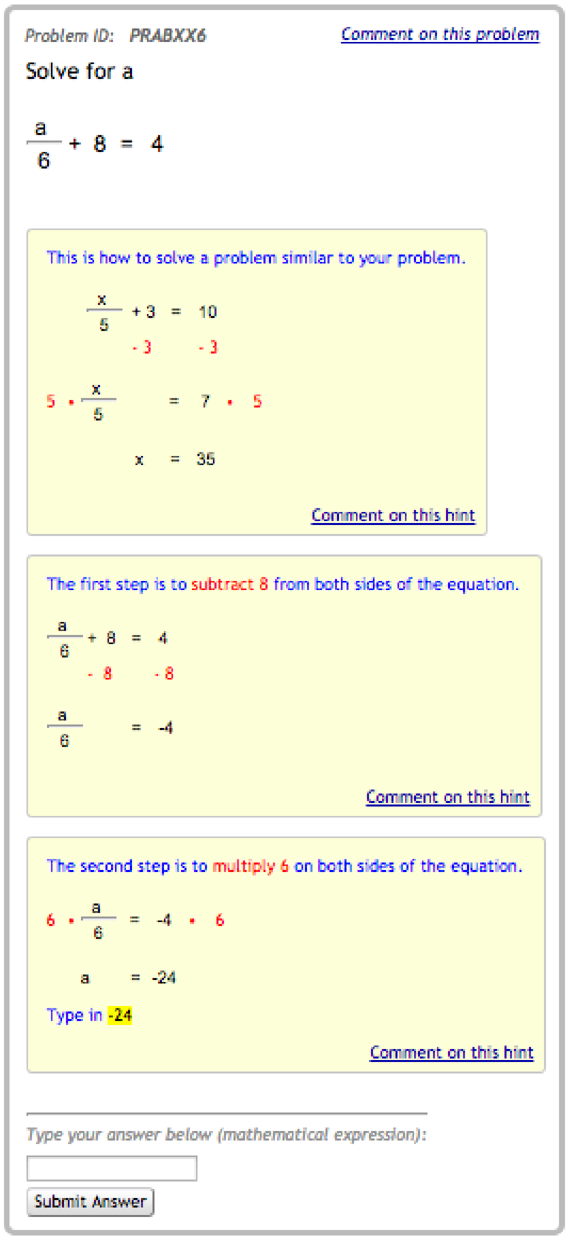
67	Multiplication of Fractions	6	31,716	9.4
61	Division of Fractions	6	28,809	8.5
278	Addition and Subtraction of Positive Decimals	6	27,301	8.1
310	Order of Operations	8	25,132	7.4
79	Proportions	7	24,279	7.2

Further assessment of students' responses provided insight into their first actions, attempt counts, and hint usage. For 95.5% of logged problems, the student's first action was to make an answer attempt. Using ASSISTments' current scoring scheme, these attempts would receive binary scores of either correct (1) or incorrect (0). Within this subgroup of logged problems, 24% of the problems were marked as incorrect while 76% were marked as correct. This suggests that a partial credit metric could provide benefit for approximately one quarter of attempted questions. Of the remaining logged problems, 4.2% represented first action hint requests, and 0.3% represented first action scaffolding requests.

Given that partial credit scores for the present study are algorithmically derived from an assessment of the student's attempt count and hint usage for each logged problem, these variables were examined thoroughly. Analysis of attempt counts across logged problems revealed a minimum of 0 and a maximum of 496, with a mean of 1.47 and a standard deviation of 2.23. For logs that were marked as incorrect based on first action, mean attempts rose to 2.70 with a standard deviation of 3.99. Within the full dataset, students made a total of 496,533 attempts.

Hint counts were also analyzed across logged problems and compared to the total number of hints available for each problem. Each problem had at least one hint, usually serving as the bottom out hint (i.e., it provided the answer). The average number of hints available per problem was 3.38, with a standard deviation of 0.88. The majority of problems had three hints (38.9%) or four hints (33.4%), with the maximum number of hints available in any problem topping off at seven. Across all logged problems, a total of 1,090,225 hints were available. Of the available hints, students only used a total of 167,371, or roughly 15.4%. The average number of hints used was 0.49 with a standard deviation of 1.20. For problem logs in which students answered incorrectly on their first attempt, 55.8% of available hints were utilized. Information particular to the bottom out hint showed that within problems initially answered incorrectly, only 14.5% of students proceeded to the bottom out hint. Thus, when struggling, the majority of students used the adaptive feedback inherent to the tutoring system in an appropriate manner. This provides further evidence for consideration of valid partial credit metrics.

Figure 3-1 provides a screenshot of a typical problem within the ASSISTments tutor. Specifically, this problem is a representation of the second most densely populated skill in the 2012-2013 ASSISTments log file: “Equation Solving with Two or Fewer Steps.” This skill is exemplified, rather than highlighting the top skill, “Addition and Subtraction of Integers,” as the problem provides a more robust example of the system’s tutoring feedback. As shown in Figure 3-1, the student is presented with the equation and asked to solve for the missing variable. He or she can make an attempt to solve the problem, or may ask for the first of three hints. The hints increase in specificity, in an attempt to guide the student without providing excess assistance. The first hint shown in Figure 3-1 provides a worked example of a similar problem solving for the missing variable, x. If the student is unable to proceed using only the worked example, he or she can request the second and third hints as needed. The third hint in Figure 3-1 is the bottom out hint; it provides the correct answer (“-24”) in an attempt to keep the student from getting stuck in the assignment, as it is not possible to skip problems and return at a later point as one can with traditional bookwork.



Problem ID: PRABXX6 Comment on this problem

Solve for a

$$\frac{a}{6} + 8 = 4$$

This is how to solve a problem similar to your problem.

$$\frac{x}{5} + 3 = 10$$
$$-3 \quad -3$$
$$5 \cdot \frac{x}{5} = 7 \cdot 5$$
$$x = 35$$

Comment on this hint

The first step is to subtract 8 from both sides of the equation.

$$\frac{a}{6} + 8 = 4$$
$$-8 \quad -8$$
$$\frac{a}{6} = -4$$

Comment on this hint

The second step is to multiply 6 on both sides of the equation.

$$6 \cdot \frac{a}{6} = -4 \cdot 6$$
$$a = -24$$

Type in -24

Comment on this hint

Type your answer below (mathematical expression):

Figure 3-1 An example problem featuring three hints for the skill “Equation Solving with Two or Fewer Steps”

3.3 Compared Models

The following subsections explain the design and brief history (when appropriate) of the five models compared in the current study. All five models are primarily designed to predict binary next problem correctness. For permitting models, we present predictions of partial credit next problem correctness using continuous probabilities for additional consideration.

3.3.1 Partial Credit Predicting Next Problem Correctness

A naïve partial credit algorithm was derived by the ASSISTments design team in hopes of providing the system with partial credit scoring capabilities based on students' attempt count and feedback usage. Scores were determined subjectively based on teacher input and a conceptual understanding of how students typically behave within the tutoring platform. For this study, the algorithm was altered slightly to consider multiple problem types and to account for the students' first action. For instance, if a student asked for tutoring feedback without making an attempt to solve the problem, we felt that a larger penalty was merited.

The resulting algorithm used to define partial credit scores is depicted in Figure 3-2. Rather than establishing a deduction method on a per hint or per attempt basis as shown in previous work (Wang & Heffernan, 2011), the algorithm presented in Figure 3-2 places each logged problem into one of five partial credit bins (0, .03, 0.6, 0.7, 0.8, 1.0) by considering the logged data pertaining to first response type (attempt = 0, hint request = 1, scaffold request = 2), attempt count, and hint count.

For example, if a student makes only one attempt and is correct without requiring feedback, they earn full credit (a score of 1). This is similar to the notion of binary correctness on first response that ASSISTments currently employs. However, in the current method, all other first actions equate to an incorrect answer (i.e., requesting a hint or scaffold, or making a first attempt that is incorrect would both earn the student a score of 0).

As shown in Figure 3-2, after ruling out a 'correct' response, the partial credit algorithm considers whether the student requested a scaffold before even making an attempt. This behavior would suggest that either the student was not actually trying to answer the problem, or that he or she was struggling conceptually. Thus, rather than earning no credit, the student is only discounted to a score of 0.6.

```
IF type = algebra OR type = fill_in
    IF attempt = 1 AND correct = 1 AND hint_count
= 0
        THEN 1
    ELSIF first_action = 2
        THEN .6
    ELSIF attempt < 3 AND hint_count = 0
        THEN .8
    ELSIF (attempt <= 3 AND hint=0)
        OR (hint_count = 1 AND bottom_hint != 1)
        THEN .7
    ELSIF (attempt < 5 AND bottom_hint != 1)
        OR (hint_count > 1 AND bottom_hint != 1)
        THEN .3
    ELSE 0

IF type = multiple_choice
    IF correct = 1
        THEN 1
    ELSE 0
```

Figure 3-2 Algorithm used to determine Partial Credit score based on first response, attempt count, and hint usage

Regardless of the student's first action, if he or she uses less than three attempts and does not request any hints, they earn slightly more, with a score of 0.8. The next bin is marked by students who have three or fewer attempts and have not used a hint, or those who have asked for only one hint and were not provided the answer (i.e., if a student's first action is to request a hint that is not the bottom out hint, they would fall into this bin). These students earn a score of 0.7. If the student can solve the problem within 5 attempts without seeing the bottom out hint, or if he or she uses multiple hints without ultimately reaching the bottom out hint, their partial credit score is 0.3. Finally, for students who use five or more attempts, or for those that see the answer, the problem is marked incorrect (a score of 0).

For multiple-choice questions the algorithm reverts to binary correctness because this type of problem does not usually provide feedback and guessing can be far more prevalent and consequential. Thus, if a student fails to get the correct answer on their first attempt, he or she receives a score of 0. This method was employed to keep the problem type from gaining an unfair advantage within the dataset. For instance, using the algorithm applied to other problem types, a student guessing through a multiple-choice problem with only four responses would still receive a score of 0.3.

The full algorithm was run across the dataset and partial credit scores were obtained for each logged problem. These partial credit scores were then used to define a pivot table to predict averages for both binary and partial next problem correctness, using maximum likelihood estimation. Results are presented in Table 3-2. For all parameter Tables, the number of logged problems falling into respective bins is depicted by sample size, *n*. The distribution of the data suggests that slight improvements could be made to the partial credit algorithm as few students fell into the 0.6 bin. Of all available 'next problem' data, only 14.7% of logs had partial credit values between 0 and 1. Thus, 85.3% of students would be insured by the platform's current method of binary correctness. This suggests that any significant finding among the models considered in the present study would be quite intriguing, as only a small portion of the sample is actually receiving the 'partial credit' treatment.

It should be noted that a potential problem inherent to this tabling method (apparent in all tabled models in the present study) is the inability to predict correctness on a student's first opportunity within a skill, as there is no preceding problem data. This essentially causes the loss of 49,990 rows of data representing first problem predictions. Thus, sample sizes in Tables 3-2, 3-3, and 3-4 total 288,307 logs rather than 338,297.

Table 3-2 Parameters for predicting Binary and Partial Next Problem Correctness from current problem Partial Credit

Partial Credit	n	Binary	Partial
0	45,735	0.5062	0.5634
0.3	6,471	0.5902	0.7438
0.6	940	0.3660	0.7948
0.7	12,077	0.6921	0.8396
0.8	22,797	0.7085	0.8668
1	200,287	0.8050	0.8785

3.3.2 Problem Difficulty Predicting Next Problem Correctness

A continuous metric of problem difficulty was calculated by retrieving data from all problems logged in the platform that were created before August 2012 (i.e., prior to the first timestamp in the modeling dataset). For each unique problem, all existing logs were averaged and a percentage of correct responses were determined. The resulting value offers an inverse metric of the problem's difficulty level. For instance, a problem on which students averaged 80% on all previous opportunities would not be considered very difficult. This metric was then binned into Low, Medium, and High difficulties by defining Medium difficulty as scores falling within +/- 0.5 standard deviations from the mean. Considering the inverse nature of the metric, High difficulty problems therefore had continuous values *below* this cut off, and Low difficulty problems had continuous values *above* this cutoff.

The bins for current problem difficulty were used in a maximum likelihood probability table to predict averages for both binary and partial scores for next problem correctness. Resulting parameters are presented in Table 3-3.

Table 3-3 Parameters predicting Binary and Partial Next Problem Correctness from current problem Difficulty

Difficulty	**n**	**Binary**	**Partial**
Low	91,712	0.7764	0.8465
Medium	107,901	0.7452	0.8297
High	88,694	0.6928	0.7895

3.3.3 Partial Credit and Problem Difficulty Predicting Next Problem Correctness

Based on the definitions of partial credit and problem difficulty defined in the singular models above, our goal was to create a novel model that used a tabling approach to consider partial credit together with problem difficulty to make predictions about next problem correctness. For each logged problem, partial credit score and problem difficulty were referenced to determine parameters for both binary and partial credit next problem correctness. Resulting probabilities are presented in Table 3-4.

Table 3-4 Parameters predicting Binary and Partial Next Problem Correctness from Partial Credit and Problem Difficulty

		High			**Medium**			**Low**	
Partial Credit	**n**	**Binary**	**Partial**	**n**	**Binary**	**Partial**	**n**	**Binary**	**Partial**
0	8,357	0.5130	0.5621	16,307	0.5027	0.5622	21,071	0.5062	0.5650
0.3	1,107	0.6035	0.7401	2,332	0.6017	0.7548	3,032	0.5766	0.7367
0.6	29	0.5902	0.8508	236	0.3388	0.7897	675	0.3661	0.7943
0.7	2,829	0.6971	0.8288	4,888	0.6987	0.8463	4,360	0.6816	0.8391
0.8	5,094	0.7770	0.8753	8,342	0.7354	0.8712	9,361	0.6473	0.8581
1	74,296	0.8116	0.8787	75,796	0.8072	0.8841	50,195	0.7921	0.8697

3.3.4 Knowledge Tracing

Knowledge Tracing (KT) is perhaps the most common method for modeling student performance. The standard KT model (Corbett & Anderson, 1995) has successfully proven itself as the basis for modeling student knowledge within intelligent tutoring systems (Koedinger & Corbett, 2006) and thereby serves as a stable comparison for new work.

As shown in Figure 3-3, the standard model of KT is a Bayesian Network comprised of four learned parameters. Two parameters represent student knowledge (*prior knowledge* and *learn rate*) and two parameters represent student performance (*guess rate* and *slip rate*). The standard KT model is binary in that skills can only be in a 'learned' or 'unlearned' state, and questions can only be 'correct' or 'incorrect.' The model is updated with each skill opportunity based on the student's performance by using the following equation as defined by Corbett & Anderson (1995):

$$p(L_n) = p(L_{n-1}|evidence) + \left(1 - p(L_{n-1}|evidence)\right) * p(T)$$

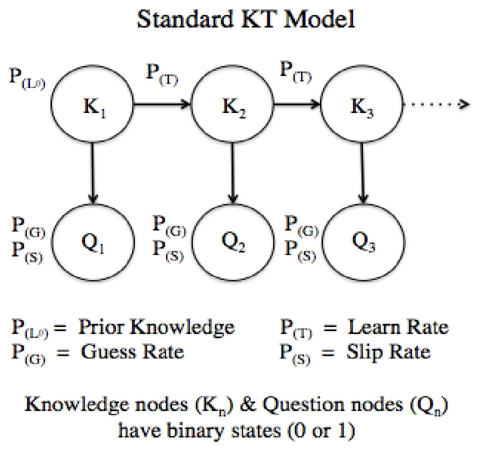


Figure 3-3 The standard Knowledge Tracing model with all learned parameters and nodes explained

Forgetting does not factor into the standard KT model when observing individual skills, as *guess* and *slip* parameters are thought to account for incorrect answers within the students' sequence of opportunities. For further information regarding the details of KT, refer to (Corbett & Anderson, 1995).

For this study, KT analysis was performed using the Bayes Net Toolbox (BNT), a popular open-source code for fitting directed graphical models within MATLAB (Murphy, 2001).

3.3.5 Performance Factors Analysis

Performance Factors Analysis (PFA) was proposed as an alternative to KT by Pavlik, Cen, and Koedinger (2009). The method can model problems with multiple skills and has been shown to accurately model and select practice within adaptive systems. PFA was derived from Learning Factors Analysis (LFA), an approach that considers a parameter for student ability, a parameter for the skill's difficulty, and a learning rate for each skill. While PFA still considers skill difficulty, β, the model improves upon LFA by considering the frequency of both correct and incorrect answers in a student's response pattern, rather than simply assessing the frequency of skill practice. Thus, PFA predictions are updated with each skill opportunity based on a cumulative history of the student's successes (weighted by γ) and failures (weighted by ρ), as depicted in the following equation defined by Pavlik, Cen, and Koedinger (2009):

$$m(i, j \in KCs, s, f) = \sum_{j \in KCs} (\beta_j + \gamma_j s_{i,j} + \rho_j f_{i,j})$$

The log-likelihood (m) attained through this equation can then be passed through an exponential function to find the probability that the student will get the item correct. This model suggests that learning is defined by more than just skill practice, and that performance is strongly tied to skill acquisition.

For this study, PFA was performed using unpublished code within MATLAB. With properly formatted data, the analysis can also be performed using logistic regression in common statistical packages like IBM's SPSS.

3.4 Model Training And Testing

Five-fold cross validation was used to train and test each model. In order to perform five-fold cross validation within our tabled models, the dataset was divided using a modulo operation on each student's unique identification number. Thus, for every student in the file, student id mod 5 was called, returning a remainder falling into bins from 0 to 4, thereby assigning students to folds. The distribution of the resulting folds was roughly equivalent, as shown in Table 3-5. With 15,253 unique student identification numbers in the dataset, the largest fold had 3,082 student ids and the smallest fold had 2,996 student ids, leaving a range of 86 and a standard deviation of 33.7.

Within each iteration of the cross-validation process, the model was trained on approximately 80% of the data and tested on the 20% that had been held out. Thus, when trained on folds 1, 2, 3, and 4 (80% of the data) the model would impart predictions on fold 0 (the held out 20%). In this manner, for each tabling method described in Section 3, table parameters were learned using four training folds and predictions were made on the held out fold. The process was repeated for all folds, thus resulting in five probability tables for each prediction type (i.e., five 'training' tables for partial credit predicting binary next problem correctness). Using an extensive formula in Microsoft Excel, the predicted averages were then applied back to each logged problem respective of test fold. For predictions of binary next problem correctness, rather than arbitrarily selecting a cutoff point for classifying binary correctness (e.g., simply using values greater than 0.5 to convey '1'), we instead subtracted the prediction directly from the actual binary result. Thus, when predicting next problem correctness using partial credit alone, if the next problem is actually correct using binary standard, the resulting residual is calculated as: 1.0000 - 0.7085 = 0.2915. In this manner, residuals were calculated for each log entry in each test fold that contained data for next problem correctness.

A similar method of five fold cross-validation was coded into the KT and PFA analyses within MATLAB. Without modification, KT and PFA are not intended to accurately predict partial credit next question correctness, and as such we have saved these analyses for future work.

Table 3-5 Distribution of data across five folds

Fold	Unique Students	# Logs	% Dataset
0	3074	67,715	20.0
1	3046	68,081	20.1
2	3082	68,739	20.3
3	3054	67,996	20.1
4	2996	65,766	19.4

To compare our tabled models with KT and PFA, slight modifications were made to the standard modeling procedures. Unlike tabling, these models carry the benefit of being able to predict performance on a student's first opportunity within a skill. Based on a 'prior knowledge' parameter, KT is able to predict the student's initial knowledge state, K_1, and therefore their performance on the first question, Q_1. Similarly, the equation for PFA defaults a prediction of the skill's difficulty parameter, β, as the student's initial state. These values essentially define a baseline for the student's knowledge, prior to any practice. Thus, in order to provide a fair comparison to tabled models, these first opportunity predictions were removed by shifting predictions to align with our 'next problem' analysis. Within KT, all subsequent skill opportunities were predicted using Expectation Maximization, a standard method for parameter learning within KT. The model was supplied the following initial parameters as 'ground truths' to begin the hill climbing process: *prior knowledge* = 0.30, *learn rate* = 0.20, *forget rate* = 0.00, *guess rate* = 0.20, and *slip rate* = 0.08. Within PFA, all subsequent skill opportunities were predicted by updating the equation presented in the previous section. These modifications resulted in the same number of data points for each model, providing grounds for fair comparison of the models.

Further, as noted briefly in the Dataset section, all models were restricted to 15 predicted opportunities per student per skill. This method was chosen largely to reduce the computation time required to fit KT using five-fold cross validation on such an extensive dataset. By capping the opportunity count, analysis time was reduced to approximately 20 hours. Other models were far less time intensive, all taking under three hours to arrive at predictions. Setting this restriction also served to reduce potential skewing in student level analyses by removing outliers with extensive opportunity counts.

3.5 RESULTS

All models were compared using the fit statistics of RMSE, R^2, AUC, and model accuracy. As the tabled models were not restricted to binary input, fit statistics were also found for consideration of modeling partial credit next problem correctness.

For each model, these statistics were found at the problem log level, the skill level, and the student level where merited. These statistics were then averaged across the level of analysis, resulting in the findings presented in Table 3-6, Table 3-7, and Table 3-8, respectively. Thus, at the problem log level, fit statistics were determined overall for the 288,307 predictions, without consideration of student or skill before being averaged across all problems. At the skill level, ten sets of fit statistics were determined (one set for each skill), which were then averaged across skills. At the student level, 15,253 sets of fit statistics were determined (one set for each student), which were then averaged across students. The latter two procedures were intended to properly weight skill and students based on their contribution to the dataset, thereby improving measures of model fit.

Student level statistics of RMSE were calculated based on all predictions. However, it should be noted that measures of R^2, AUC, and model accuracy could not be calculated for students with less than three skill opportunities. This discrepancy should affect all models equally, and thus we provide these measures for comparison in Table 3-8 with the caveat that they should not be directly compared to measures of student level RMSE.

Table 3-6 Problem Level Average RMSE, R2, AUC, and Accuracy for Models Predicting Next Problem Correctness (NPC)

	Binary NPC				**Partial NPC**			
Model	*RMSE*	*R^2*	*AUC*	*Accuracy*	*RMSE*	*R^2*	*AUC*	*Accuracy*
Partial Credit +	0.4241	0.0674	0.6365	0.7310	0.3326	0.1062	0.5395	0.7298
Partial Credit	0.4244	0.0660	0.6309	0.7309	0.3327	0.1060	0.5351	0.7298
Problem Difficulty	0.4379	0.0057	0.5464	0.7300	0.3511	0.0043	0.3953	0.7298
Knowledge Tracing	0.4240	0.0680	0.6621	0.7298	--	--	--	--
Performance Factors	0.4227	0.0738	0.6644	0.7485	--	--	--	--

Table 3-7 Skill Level Average RMSE, R2, AUC, and Accuracy for Models Predicting Next Problem Correctness (NPC)

	Binary NPC				**Partial NPC**			
Model	*RMSE*	*R^2*	*AUC*	*Accuracy*	*RMSE*	*R^2*	*AUC*	*Accuracy*
Partial Credit +	0.4224	0.0670	0.6300	0.7414	0.3284	0.1032	0.5130	0.7399
Partial Credit	0.4229	0.0656	0.6290	0.7414	0.3284	0.1031	0.5103	0.7399
Problem Difficulty	0.4364	0.0046	0.5323	0.7402	0.3473	0.0037	0.3560	0.7399
Knowledge Tracing	0.4225	0.0602	0.6500	0.7466	--	--	--	--
Performance Factors	0.4212	0.0664	0.6506	0.7499	--	--	--	--

Table 3-8 Student Level Average RMSE, R2, AUC, and Accuracy for Models Predicting Next Problem Correctness (NPC)

	Binary NPC				**Partial NPC**			
Model	*RMSE**	*R^2*	*AUC*	*Accuracy*	*RMSE**	*R^2*	*AUC*	*Accuracy*
Partial Credit +	0.3864	0.1027	0.5431	0.7684	0.2702	0.1108	0.3593	0.7674
Partial Credit	0.3866	0.0994	0.5392	0.7683	0.2701	0.1057	0.3619	0.7674
Problem Difficulty	0.4064	0.0829	0.5219	0.7676	0.2941	0.0851	0.3145	0.7674
Knowledge Tracing	0.3897	0.1057	0.4425	0.7729	--	--	--	--
Performance Factors	0.3882	0.0970	0.5003	0.7754	--	--	--	--

*R^2, AUC, and Accuracy are reported with less data than RMSE due to the nature of student level data.

3.6 Discussion

The fit statistics for both the problem log and skill level generalizations paint very similar pictures of the relative success of our tabling method. The combined Partial Credit and Problem Difficulty model performs about as well as KT at both levels of analysis. At these levels, PFA appears to be the ‘best’ model for predicting binary next problem correctness, showing the lowest RMSE and highest AUC and model accuracy. However, we feel that a simple tabling method that can be performed with extreme efficiency yet still meets the standards of KT is well worth discussion.

Our first research question, “Does an algorithmically determined partial credit score outperform binary metrics when used to predict next problem correctness?” was answered with mixed results for binary predictions. Considering problem log level analysis, while KT and PFA attained fit statistics relative to those accepted in the field, our tabling method for partial credit considered alone only slightly underperformed these standards (RMSE = 0.4244, R^2 = 0.0660, AUC = 0.6309, Accuracy = 0.7309). However, when considering student level analysis, our partial credit tabling method outperformed both KT and PFA in terms of RMSE and AUC. To confirm that these findings were significantly different, we used a two-tailed paired samples t-test for RMSE comparison at both the student level and skill level. RMSEs obtained using our tabling method with partial credit alone were significantly different from those found using KT at the student level, $t = 5.65$, $p < .001$, but were not significantly different at the skill level, $t = -1.65$, $p = 0.133$. Thus, it is difficult to tell if this finding is truly significant.

Our second research question, “Does a binned metric of current problem difficulty (e.g., Low, Medium, or High difficulty) provide a valid prediction of next problem correctness?” was answered by assessing the “Problem Difficulty” model. When taken alone, problem difficulty is not very helpful in predicting next problem correctness. This was the worst performing model across all granularities of analysis. A paired samples t-test was again used to compare student level and skill level RMSEs to those observed using the KT model. RMSEs obtained using our tabling method for Problem Difficulty were significantly worse than those found using KT at the student level, $t = -41.27$, $p < .001$, as well as those found using KT at the skill level, $t = -9.93$, $p < .001$. Of the tabled models, this model was also the lowest performing model when considering predictions of partial next credit correctness, drastically underperforming models that considered current problem partial credit score. Thus, we argue that problem difficulty alone is a poor metric for modeling student performance.

Our final research question, "Can current problem difficulty supplement partial credit score to outperform similar modeling techniques?" was answered by assessing the fit statistics for the combined "Partial Credit + Problem Difficulty" model. At the student level, this model outperformed both KT and PFA on predictions of binary next problem correctness as measured by RMSE (0.3864) and AUC (0.5431). This finding was significant using a two-tailed paired samples t-test comparing student level RMSEs, $t = 6.50$, $p < .001$, but was not significant when considering skill level RMSEs, $t = -1.34$, $p = 0.214$. Despite the low performance of the Problem Difficulty model, this combined model consistently outperformed partial credit when modeled alone, suggesting possible mediation effects. Using a paired t-test comparison, this difference was significant at the student level, $t = -4.55$, $p < .001$, but was not significantly reliable at the skill level, $t = -1.03$, $p = .310$. As such, it is difficult to quantify the potentially negative impact of considering problem difficulty when using partial credit to model next problem correctness.

Model fit indices for the prediction of partial credit scores for next problem correctness are provided for further consideration, but do not specifically link to our research questions. Drastic improvements in model fit suggest that intelligent tutoring systems should incorporate partial credit scoring as it has the potential to enhance the precision of student modeling. In the current study, these findings cannot be compared to standard KT and PFA models that utilize binary input and essentially predict binary performance. Future research will incorporate modifying these models to predict continuous partial credit metrics, thus allowing for further comparison.

3.7 Contribution

The results from the present study suggest that considering partial credit for each skill opportunity can enhance the accuracy of student modeling. While the concept of using a tabling method to establish partial credit metrics that predict binary correctness is not novel (Wang & Heffernan, 2011), tabling a model based on algorithmically determined partial credit is, to the best of our awareness, a unique approach. This method was shown to perform about as well as KT when predicting binary next problem correctness. We feel that this finding still provides a significant contribution to the field, as KT is far more computationally expensive. Our KT analysis took approximately 20 hours to run, while all tabling methods were conducted by hand in less than three hours. While this is impressive in and of itself, the second author was then able to implement the tabling method presented here within the ASSISTments test database, arriving at a replication of our predictions in less than two minutes. If automated in such a manner, our Partial Credit + Problem Difficulty model could predict next problem performance on par with KT in approximately one 600th of the time. This increase in efficiency could prove essential for intelligent tutoring systems that currently incorporate KT models to adaptively control student skill practice.

Further, the partial credit model was novel in its ability to predict partial credit scores for next problem correctness, thereby enhancing model fit even further. In future research, we hope to modify the standard KT and PFA models to allow for the prediction of continuous variables for comparison. We also anticipate directly comparing our partial credit model to the “Assistance” Model established in previous research (Wang & Heffernan, 2011). The “Assistance” Model cited a clear cut, albeit subjective, method for the provision of partial credit scores. As the tabling technique employed made predictions on a continuous scale rather than by binning partial credit as we have shown in the present study, comparison was not presently possible without ensembling our findings with standard KT measures (Wang & Heffernan, 2011). However, as alternating ensembling techniques lead to inconsistent results (Gowda, Baker, Pardos, & Heffernan, 2011), we argue for direct modifications within KT that will allow the model to learn partial credit scores at each opportunity and to gauge a student’s knowledge state on a continuum. A similar model was previously suggested by Pardos & Heffernan (2012), but to our knowledge has never been implemented. Thus, the present study lays the groundwork for future research in modifying KT.

The assessment of models considering problem difficulty also provides a contribution to the modeling literature. It seems intuitive that problem difficulty should influence a students’ ability to answer the current problem correctly, and that it likely influences their knowledge state and next problem correctness. The findings here suggest that problem difficulty alone, when binned into generic groups of Low, Medium, and High difficulty, does not provide accurate models of next problem correctness. However, problem difficulty appeared to enhance modeling when coupled with partial credit in comparison to partial credit modeled alone, although this difference was not shown to be significant. Still, we believe that some measure of problem difficulty is important to consider when modeling student learning. Future research should investigate using a continuous metric or designing an alternative binning approach for this feature. Future work should also consider devising an approach to remedy the issue of being unable to predict a student’s first opportunity within a skill when using tabled models. Possible solutions include per student estimates of prior knowledge based on performance on other skills within the tutor, or simply implementing problem difficulty as a measure of likelihood for accuracy.

Despite the impressive performance of our partial credit model, we retain skepticism in regards to the subjective nature of our partial credit algorithm. As multiple arbitrary partial credit models have now been designed to assess log data from the ASSISTments platform (Wang & Heffernan, 2011), we argue for the design of a data driven algorithm that considers and compares a myriad of logged features. Future work will examine a grid search of possible hint and attempt penalties to examine the sensitivity of the approach described herein. The data files of intelligent tutoring system are rich with information pertaining to students’ actions, including the time required for first response, their sequence of actions within each problem, and the specific misconceptions that are driving incorrect responses. These features may provide critical information for the scoring of partial credit. When considering the approach used in the present study, using an algorithm to establish partial credit scores prior to tabling provides the leeway for tabled models to consider these additional features. Future research could easily replicate similar models, combining partial credit with novel features for additional exploration of the observed effect.

Chapter 4. Optimizing Partial Credit Algorithms to Predict Student Performance

This chapter covers another research paper written during the creation of the if-then system. This paper expands on the idea of partial credit in the hopes of finding an optimal algorithm for calculating partial credit that best shows students' knowledge and using that score to predict students' performance.

As adaptive tutoring systems grow increasingly popular for the completion of classwork and homework, it is crucial to assess the manner in which students are scored within these platforms. The majority of systems, including ASSISTments, return the binary correctness of a student's first attempt at solving each problem. Yet for many teachers, partial credit is a valuable practice when common wrong answers, especially in the presence of effort, deserve acknowledgement. We present a grid search to analyze 441 partial credit models within ASSISTments in an attempt to optimize per unit penalization weights for hints and attempts. For each model, algorithmically determined partial credit scores are used to bin problem performance within a maximum likelihood table, using partial credit to predict binary correctness on the next question. An optimal range for penalization is discussed and limitations are considered.

4.1 INTRODUCTION

Adaptive tutoring systems provide rich feedback and an interactive learning environment in which students can excel, while teachers can maintain data-driven classrooms, using the systems as powerful assessment tools. Simultaneously, these platforms have opened the door for researchers to conduct minimally invasive educational research at scale while offering new opportunities for student modeling. Still, they are commonly restricted to measuring performance through binary correctness at the problem level. Arguably the most popular form of student modeling within computerized learning environments, Knowledge Tracing, is rooted in the binary correctness of each opportunity or problem a student experiences within a given skill (Corbett & Anderson, 1995). Knowledge Tracing (KT) drives the mastery-learning component of renowned tutoring systems including the Cognitive Tutor series, allowing for real time predictions of student knowledge, skill mastery, or next problem correctness (Koedinger & Corbett, 2006). Similar modeling methods consider variables that extend beyond correctness but rarely escape the binary nature of the construct, including Item Response Theory (Drasgow & Hulin, 1990) and Performance Factors Analysis (Pavlik, Cen, & Koedinger, 2009). By restricting input to a single binary metric across questions, these modeling techniques fail to consider a continuous metric that is commonplace for many teachers: partial credit.

Partial credit scoring used within adaptive tutoring systems could provide more individualized prediction and thus establish models with better fit. It is likely that binary correctness has remained the default for learning models due to the inherent difficulty of defining a universal algorithm to generalize partial credit scoring across platforms. Additional onus may fall on users' familiarity with current system protocol; students tend to avoid using system feedback regardless of the benefits it may provide as requesting feedback results in score penalization. However, the primary goal of these platforms is to promote student learning rather than simply offering assessment, and thus binary correctness becomes a weakness.

The present study considers data from ASSISTments, an online adaptive tutoring system that provides assistance and assessment to over 50,000 users around the world as a free service of Worcester Polytechnic Institute. Researchers have previously used ASSISTments data to modify student-modeling techniques in a variety of ways including through student level individualization (Pardos & Heffernan, 2010), item level individualization (Pardos & Heffernan, 2011), and the sequence of student response attempts (Duong, Zhu, Wang, & Heffernan, 2013). Previous work has also shown that naïve algorithms and maximum likelihood tabling methods that consider hints and attempts to predict next problem correctness can be successful in establishing partial credit models meant to supplement KT (Wang & Heffernan, 2011; Wang & Heffernan, 2013). More recently, algorithmically derived partial credit scoring has resulted in stand-alone tabled models relying on data from only the most recent question, showing goodness of fit measures on par with KT at lower processing costs (Ostrow, Donnelly, Adjei, Heffernan, 2015). However, we hypothesize that some conceptualizations of partial credit may lead to better predictive models than others. Rather than subjectively defining tables or algorithms, a data driven approach should be considered. Thus, considering student performance within the ASSISTments platform, the current study employs a grid search on per unit penalizations of hints and attempts to ask:

1. Based on penalties for hints and attempts dealt per unit, is it possible to algorithmically define partial credit scoring that optimizes the prediction of next problem correctness?
2. Does the optimal model of partial credit differ across different granularities of dataset analysis?

Establishing an optimal partial credit metric within ASSISTments would allow teachers to gain more accurate assessment of student knowledge and learning, while allowing students to alter their approach to system usage to take advantage of adaptive feedback. The optimization of partial credit scoring would also enhance student-modeling techniques and offer a new approach to answering complex questions within the domain of educational data mining.

4.2 DATA

The ASSISTments dataset used for the present study is comprised solely of assignments known as Skill Builders. This type of problem set requires students to correctly answer three consecutive questions to complete their assignment. Questions are randomly pulled from a large pool of skill content and are typically presented with tutoring feedback, most commonly in the form of hints. The dataset has been de-identified and is available at (Ostrow, 2014) for further investigation.

The dataset used in the present study is a compilation of Skill Builder data from the 2012-2013 school year, containing data for 866,862 solved problems. Data recorded includes a student's performance on the problem (i.e., binary correctness, hint count, attempt count), variables that identify the problem itself (i.e., problem type, unique problem identification number) and information pertaining to the assignment housing the problem (i.e., unique identifiers for assignments, skill type, teachers, and schools). This dataset was representative of 120 unique skills and 24,912 unique problems, solved by 20,206 students.

On average, students made 1.53 attempts per problem (SD = 15.08). The minimum number of attempts was 0 (i.e., a student who opened the problem and then left the tutor), while the maximum number of attempts was a daunting 12,246 (i.e., a student who hit enter repeatedly for a prolonged period of time, likely out of frustration or boredom). Students made a total of 1,324,226 attempts across all problems. The majority of problems (74.9%) had just one logged attempt per student (typically correct answers), while 15.1% of problems carried only two logged attempts.

Hint usage among all students averaged 0.61 hints per problem (SD = 1.29). The minimum number of hints used was 0 (i.e., no feedback requested), while the maximum number of hints used was 10. Interestingly, the maximum number of hints available for any particular problem was 7. Thus, a handful of students who logged more than 7 hints were accessing the tutor in multiple browser windows (i.e., cheating). On average there were 3.22 hints available per problem (SD = 0.89). The majority of problems contained 3 hints (44.6%), 4 hints (28.9%) or 2 hints (18.2%). Although there were 2,768,299 hints available across all problems, students only used 529,394 hints, or approximately 19% of available feedback. Bottom out hints, or those providing the problem's solution, were only used on 146,742 (16.9%) of problems.

Additional analysis was performed on the 261,787 problems that students answered incorrectly out of the original 866,862 problems solved. Within this subset of data, students made an average of 2.75 attempts per problem (SD = 27.40). Students also used an average of 2.02 hints (SD = 1.63). This subset of problems had 860,131 total hints available, of which students used 528,644 hints (61.5%).

Hint usage would likely increase if partial credit scoring was implemented within the ASSISTments platform. Binary first attempt scoring has created an environment in many classrooms where students are afraid to use hints although they would benefit from feedback, as they know they will receive no credit. Further, the dataset suggests that once students are marked wrong, they are more likely to jump through all available hints and seek out the answer (56% of incorrect first attempts led to bottom out hinting). This reflects another substantial downfall in the system's current protocol; once the risk has passed, so has the drive to learn. The implementation of partial credit scoring has the potential to alleviate this misuse.

4.3 METHODS

The present study presents an extensive grid search of potential per hint and per attempt penalizations. The full dataset described above was used to algorithmically define partial credit scores based on a per unit penalization scale ranging from 0 to 1 in increments of 0.05 for both variables. Thus, for each problem in the dataset, 441 potential partial credit scores were established based on each possible combination of per unit penalization. For instance, in a model in which each attempt earned a penalization of 0.05, and each hint earned a penalization of 0.1, a student who made three attempts and used one hint would receive a penalty of 0.25 ((3x0.05) + (1x0.1)), effectively scoring 75% on that problem. This process was used to score each problem in the dataset for each possible penalty combination, with a minimum per problem score of 0 set as the floor (students could not receive negative scores). This method was similar to that presented by Wang & Heffernan in the Assistance Model (2011) which established a tabling method to calculate probabilities of next problem correctness based on combinations of hints and attempts that resulted in twelve possible bins or parameters.

For each of the 441 partial credit models, a maximum likelihood tabling method was employed using five fold cross validation. Within each model, a modulo operation was used on each student's unique identification number to assign students to one of five folds. Note that this method results in folds that all represent approximately 20% of students in the dataset. Maximum likelihood probabilities for next problem correctness were then calculated for each partial credit score within each model. Table 4-1 presents an average of test fold probabilities for the model in which each attempt hint are penalized 0.1. This model is depicted as the penalization structure results in eleven clean and understandable bins for partial credit scores. For instance, a student using two attempts and one hint would be penalized 0.3, thus falling into the score bin of 0.7 (PC Score). Following through with this example, based on 11,174 logged problems that fit this scoring structure, the average of known binary performance on the following problem (performance at time *t* +1) was 0.599. This value becomes the prediction for next problem correctness.

Using the maximum likelihood probabilities for next problem correctness within each test fold as predicted values, residuals were then calculated by subtracting predictions directly from actual next problem binary correctness (i.e., 1 – 0.725 = 0.275; 0 –0.571 = -0.571). This approach was used rather than selecting an arbitrary cutoff point to classify a prediction as correct or incorrect in the binary sense (i.e., values greater than or equal to 0.6 serve as predictions of correctness) because it reduced the potential for researcher bias.

Table 4-1 Probabilities averaged across test folds for the model in which the penalization per hint and per attempt is 0.1

PC Score	**n**	**Max. Likelihood NPC**
0	149,504	0.467
0.1	422	0.571
0.2	685	0.581
0.3	1,055	0.578
0.4	1,784	0.574

0.5	3,442	0.583
0.6	6,623	0.585
0.7	11,174	0.599
0.8	18,679	0.662
0.9	49,972	0.725
1.0	476,523	0.802

4.4 RESULTS

For each model, residuals were used to calculate RMSE, R^2 & AUC at three levels of granularity: problem level, student level, and skill level. Heat maps are presented here only for RMSE, as the other metrics established almost identical maps. Metrics representing greater model fit are depicted using the purple end of the spectrum, while those representing poor fit are represented using the red end of the spectrum. Further, a series of ANOVAs are used to compare each set of models within the same penalization level for attempts and hints. For instance, the 21 models in which attempt penalty was set to 0.2 were compared to all other sets of attempt penalty models to investigate significant differences occurred across penalties. This method was used rather than attempting to compare each model with all other models using paired samples t-tests, as the resulting 194,481 analyses (441^2) would greatly inflate the rate of Type I error without vast and unrealistic correction strategies.

Initial analysis was performed at the problem level; residuals were calculated for each row of data that contained next problem correctness metrics and goodness of fit measures were averaged across the dataset. Each metric followed a similar structure in which low attempt penalties appear to result in better fitting models, while hint penalty does not appear to be significant. Thus, partial credit scoring algorithms using lower penalties for attempts were better at predicting next problem performance, as depicted in Figure 4-1. The ANOVA results depicted in Table 4-2 suggest that differences in attempt penalty models were significant. Thus, the set of models with per attempt penalties of 0.1 differed significantly from the set of models with per attempt penalties of 0.8. Differences among hint penalty models were not reliably significant. Figure 4-1 also suggests that the current binary scoring protocol used by ASSISTments results in predictive models that are inadequate. First attempt binary correctnes is the equivalent of the model in which per attempt and per hint penalty are both set to 1, or the upper right corner of each heatmap). This model resulted in consistently poor fit metrics, suggesting that modeling techniques such as KT should use continuous or binned partial credit values as input as they enhance next problem prediction ability. It has not yet been investigated how this alteration would change the prediction of other variables commonly predicted through KT, such as latent student knowledge or mastery.

Student level analysis was undertaken using a subset of the original data file. At this granularity, goodness of fit metrics were calculated for each student and averaged across students to obtain final metrics for each of the 441 models. As the ASSISTments system measures completion of a Skill Builder as three consecutive correct answers, a number of high performing students had limited opportunity counts within skills. For students with too few data points, it was not possible to calculate R^2 and AUC given the nature of these metrics. Therefore, student level analysis incorporated 7,429 students from the original dataset, or 651,849 problem logs. Answering our second research question, it appears as though the region of optimal partial credit values observed at the problem level remains consistent at the student level, as shown in Figure

4-2. ANOVA results depicted in Table 4-2 show reliably significant differences across attempt penalty models but not across hint penalty models.

Table 4-2 ANOVA results for groups of attempt and hint penalty models at each level of analysis

			Attempt Penalty			Hint Penalty		
Level	*Min*	*Max*	*F*	*p*	*R^2*	*F*	*p*	*R^2*
Problem								
RMSE	.430	.435	302.70	.000	.935	0.95	.519	.043
AUC	.626	.655	295.46	.000	.934	1.14	.304	.052
R^2	.070	.091	304.34	.000	.935	0.95	.525	.043
Student								
RMSE	.424	.429	222.49	.000	.914	1.34	.149	.060
AUC	.578	.593	208.19	.000	.908	1.42	.106	.063
R^2	.096	.110	374.52	.000	.947	0.80	.715	.037
Skill								
RMSE	.423	.429	517.85	.000	.961	0.55	.944	.026
AUC	.624	.647	250.17	.000	.923	0.72	.805	.033
R^2	.073	.090	510.96	.000	.961	0.49	.971	.023

Note. For all models, df = (20, 420).

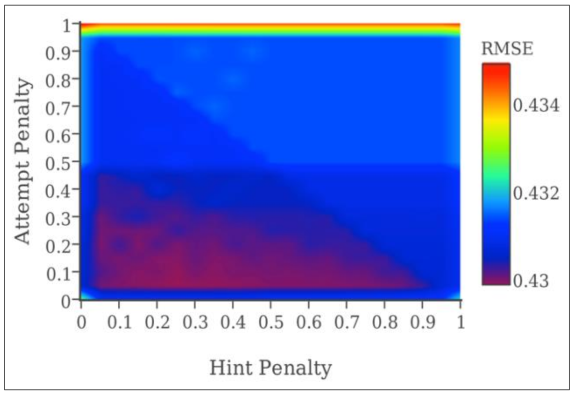


Figure 4-1 Problem Level RMSE

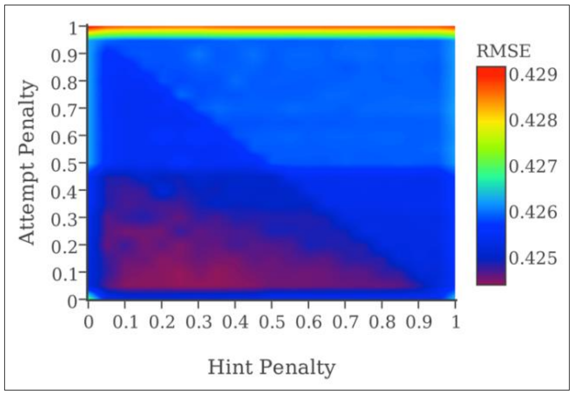


Figure 4-2 Student Level RMSE

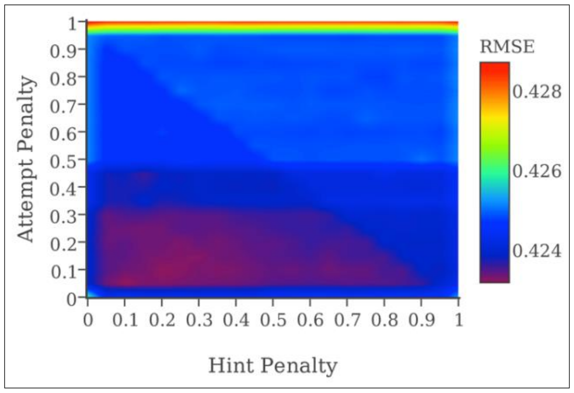


Figure 4-3 Skill Level RMSE

Skill level analysis was also undertaken using a subset of the original data file. One skill did not have enough data based on a low number of users and high mastery within those users, and was excluded from skill level analysis, resulting in a file with 119 skills. At this granularity, goodness of fit metrics were calculated for each skill and averaged across all skills to obtain final metrics for each of the 441 models. Results are depicted in Figure 4-3. The heat map shows that the region of optimal penalization has grown more concise, showing optimal fit among models with low per hint and per attempt penalties (< 0.3). ANOVA results depicted in Table 4-2 suggest reliably significant differences in all metrics across attempt penalty models but not across hint penalty models.

Post-hoc analyses were conducted on ANOVA results using multiple comparisons to examine significant differences between attempt penalty and hint penalty model groups when considering problem level AUC. Using a Bonferroni correction to reduce Type I error, this process resulted in a series of significance estimates for penalty group comparisons (i.e., all models where attempt penalty is 0.1 compared to all models where attempt penalty is 0.3 results in a non-significant difference, p = 0.88). Results suggest that models close in penalty are less likely to differ significantly than models with greater difference in penalty. For instance, models with an attempt penalty of 0.1 are significantly different than those with an attempt penalty of 0.4, but are not significantly different than those with an attempt penalty of 0.2. This information can be used to help select optimal partial credit penalizations, as it may be more motivating and productive for students to receive smaller penalizations. Such information would allow systems like ASSISTments to define a range of possible penalizations that can then be refined by the teacher, providing all users with a greater sense of control.

4.5 DISCUSSION

The initial findings of a grid search on partial credit penalization through per unit hint and attempt docking suggest that the implementation of partial credit within adaptive tutoring systems can be established using a data driven approach that will ultimately produce stronger predictive models of student performance while ultimately enhancing the way these systems are used by students and teachers.

Our first research question was answered with a resounding "Yes," certain algorithmically derived combinations of partial credit penalization are better than others when used to predict next problem performance. Optimal partial credit models were visible in heat maps spanning three levels of data granularity and remained relatively consistent across granularities, thus answering our second research question. ANOVAs revealed that differences in attempt penalty models were consistently significant across dataset granularities, while differences in hint penalty models were not reliable. This finding is likely due to the fact that hint usage is lower and less distributed than attempt count across problems in the dataset, and it is possible that this finding would diminish in a system that more readily promoted the use of tutoring feedback without penalization, or a system already employing partial credit scoring.

The partial credit models that we define here as optimal, based on their ability to predict next problem performance, were models with per hint and per attempt penalties of 0.3 or less. Additional analyses revealed that at the problem level, there should be no reliable difference in predictive ability of a model penalizing 0.3 per attempt from a model penalizing 0.1 per attempt, with variable hint penalization. This finding suggests that less penalization is just as effective, offering an opportunity to consider student motivation and affect when defining a partial credit algorithm. This grid search also revealed that partial credit metrics outperform binary metrics when predicting next problem performance, as previously shown in (Ostrow, Donnelly, Adjei, & Heffernan, 2015). Thus, it is possible to improve prediction of student performance within adaptive tutoring systems simply by implementing partial credit scoring. It should also be noted that a leading limitation of the approach presented here is that we have only been predicting next problem correctness, rather than latent variables such as skill mastery or student knowledge. It is possible that optimizing partial credit would also provide benefits for the prediction of latent effects, but further research is necessary in this domain.

Chapter 5. Studies Using If-Then

5.1 IQP Studies

With the completion of the basic if-then system, two groups of undergraduate students started doing their IQP (Interactive Qualifying Project) within ASSISTments to test the new if-then. These experiments are being run to beta-test the If-Then system and have groundwork setup for the next stage. Table 5-1 shows the studies created by the IQP students and gives a small explanation of each study. Korinn Ostrow, another student of Neil Heffernan's, is overseeing these IQP studies and will be running more once the next stage of the If-Then system is complete.

Table 5-1 IQP Studies running in 2014-2015 using If-Then.

SetID	CommonCore	Content	Description
PSARRVW	2.OA.A.1	Story Problems	Tests whether students have access to video. Students who pass the video check are routed to one of 5 conditions testing motivational and content videos
PSASA67 PSARKHS	2.MD.C.7 3.OA.D.8	Elapsed Time Multistep Word Problems	Students who fail the video check only receive a text assignment. Student who pass the video check are randomly assigned to either text or video feedback. Those in the video condition receive videos made by other students
PSAR9Y9 PSASDZY PSARRCY PSASA4B	3.OA.A.4-1 3.OA.A.4-2 3.G.A.2 2.MD.C.8	Solve for Unknown - Mult Solve for Unknown - Div Partition Shape into Equal Areas Coin Values	Students are randomly assigned to either adaptive or non-adaptive conditions. In the non-adaptive condition they receive a hard Skill Builder (SB). In the adaptive condition they get a 'pretest' question. Those who answer correctly are routed to a hard SB, while those who answer incorrectly are routed to an easy SB and then a hard SB.
PSARB8Q PSARMW3 PSARZ7W	3.NF.A.2.B 3.NF.A.3.D 3.NBT.A.1	Number Line Fractions: Same Numer/Denom Rounding 10s and 100s	Students are randomly assigned to one of four conditions: generic survey (control), an encouraging video from President Obama, an encouraging video from a peer, or false choice of challenge. All students then get a difficult problem.

PSARZ8B	3.NBT.A.3	Multiplying by Multiples of 10	Students' confidence is gauged. They are then randomly assigned to control or experimental conditions. In the control, students undertake a traditional SB. In the experimental condition students are shown the skill and asked to choose their mastery level (default 3, 2, or 4 questions for mastery). If they choose 3 they are provided a traditional SB. If they request more or less they are routed to a SB with the proper settings. Students' confidence is gauged again.
PSASD3S PSASDZH PSARZXV PSARZX2	3.NF.A.1 3.NF.A.2.B 3.NBT.A.2 3.OA.C.7-2 3.MD.A.2 3.MD.C.7	Understand A/B Number Line Adding & Subtracting Divide and Measure Liquid Volumes Divide and Measure Masses Area of a Rectangle	Students' confidence is gauged. They then take a traditional SB. Upon completion they are randomly assigned to either a challenge choice, or a generic survey question (control). All students then receive two additional problems.

Korinn originally created a choice study that did not use the If-Then system and lead to the creation of the If-Then system and this thesis project (Ostrow, Heffernan, In Press). The new choice study using the If-Then system was simpler to build and easier to explain to other researchers. The original study had a strange structure that took advantage of a loophole in ASSISTments. In ASSISTments, skillbuilders pay attention to the correctness of all descendent problems; while other problem sets only paid attention to direct children problems or problem sets. This meant researchers could put skillbuilders within skillbuilders in order to trick the system into doing what they wanted. Since the study was using a loophole, it wasn't easy to convey the structure to other researchers and was prone to errors by the builder and some students got errors from the tutor itself while running the assignment. With the If-Then system, building the new study was straight forward and did not require skillbuilders to contain anything other than just problems. This made explaining to researchers easier and students did not receive errors while doing their work. Figure 5-1 shows the design for the new choice study. The Choice branch used the If-Then system to ask each student if they wanted text or video feedback and the student's answer would decide which branch (Video or Text) to go down. The next section of this chapter will talk about the analysis of the choice study using the If-Then.

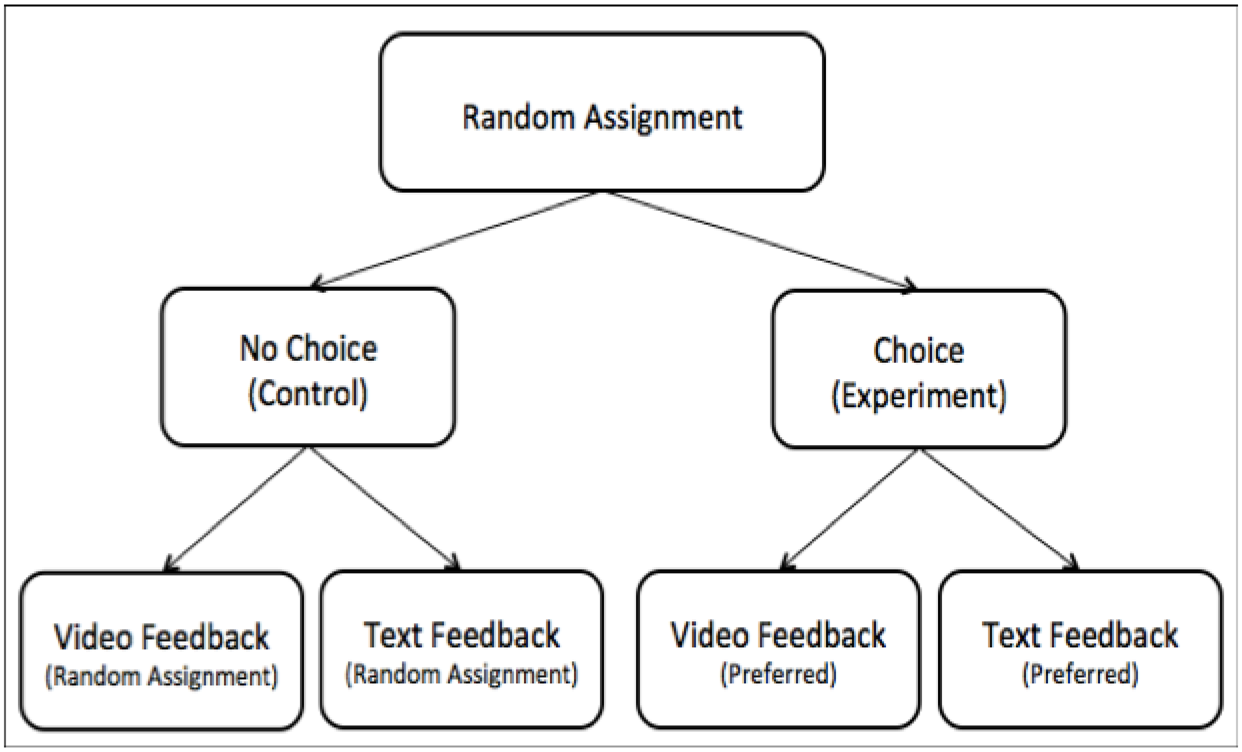

Random Assignment
No Choice (Control)
Choice (Experiment)
Video Feedback (Random Assignment)
Text Feedback (Random Assignment)
Video Feedback (Preferred)
Text Feedback (Preferred)

Figure 5-1 Design for the new choice study using If-Then.

5.2 Choice Study Analysis and If-Then Evaluation

The If-Then system is being used in over one hundred studies currently and that number is expected to rise as more researchers learn about it and with the release of the advanced if-then system. This study only shows one main use of the if-then system: choice. Since there are videos in this study, students who cannot view videos had to be removed from the study. It is not shown in the design in Figure 5-1, but using If-Then and a video in a problem, students need to type what the video tells them to type in order to pass the video check (the phrase is "wpi"). If the student gets the problem wrong they were unable to either see or hear the video and were removed from the study and pushed into an alternative skillbuilder. All the students who passed the video check were moved into the actual study designed in Figure 5-1.

5.2.1 Data

This choice study has only been running for a month (March 15th, 2015) since writing this paper, so the data is fairly small. Refer to Donnelly (2015) to find a link to the data as a csv file on google drive. 59 students have run this study and 6 of them had to be thrown out because they failed the video check. Out of the 53 students in the study, 22 were put into the no-choice section and 31 were put into the choice section. The distribution between choice and no-choice is not randomly evenly distributed like many other studies but is completely random which is why there seems to be slightly more students in the choice section. In the no-choice section, 12 students were given text feedback and 10 students were given video feedback. In the choice section, 21 students chose text feedback and 10 students chose video feedback. Partial Credit scores were calculated with the formula created in the paper from chapter 4:

$$Score = 1 - 0.1 * (num_{attempts} - 1) - 0.1 * num_{hints}$$

5.2.2 Results

Across all the variables, the effect of choice is examined for the full dataset on “choice” (n = 31) versus “no-choice” (n = 22). The findings are listed in Table 5-2, and explained more in the rest of this section. Since this is a feedback study, further examination was done to test the effect of text versus video feedback. In that examination, students who actually requested hints were considered (the “treated” group contained 5 in video and 8 in text). Feedback findings are listed in Table 5-3 and explained more in the rest of this section. Interactions between choice and feedback are considered with the treated sample and discussed, but should be taken with cautious because of the low number of students in each condition.

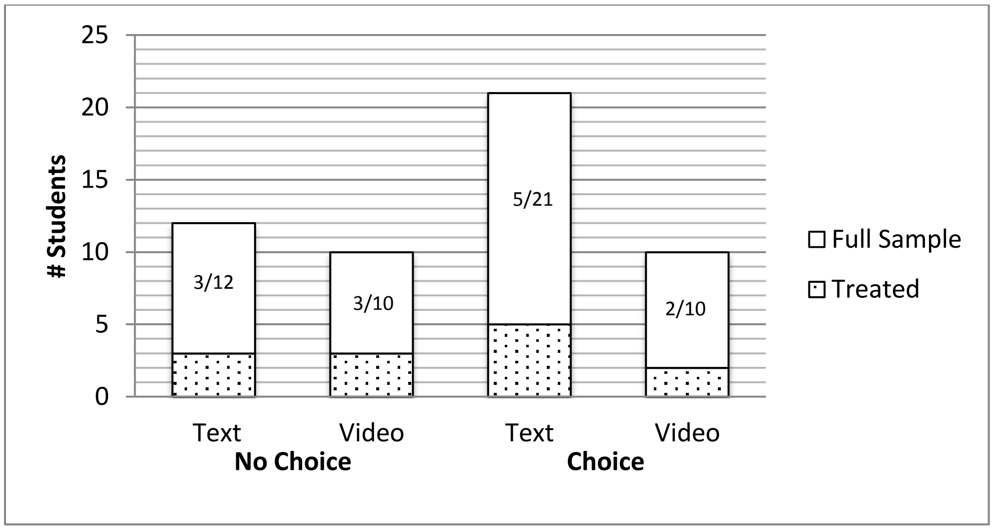


Figure 5-2 Distribution of Data

Examining the variables, for the effect of choice, shows that there is no significant difference between the choice and no-choice. Taking a closer look, it appears that students given a choice, on average, saw more problems than those that did not see choice. The students in the no-choice group had a higher binary score and higher partial credit score compared to the choice students. This study needs to run longer and collect more data but if the trend continues then that suggests students see more problems and get a lower binary or partial credit score when given a choice. Table 5-2 summarizes these findings below.

Table 5-2 Means, SDs, and ANOVA Results for Effects of Choice in Full Sample

	Choice *(n = 31)*	No Choice *(n = 22)*			
Dependent Variable	*M (SD)*	*M (SD)*	*F*	*p*	η^2
Problems Seen	4.16 (1.90)	3.91 (1.34)	0.286	0.595	0.006
Binary Score	0.84 (0.22)	0.89 (0.15)	0.965	0.330	0.019
Partial Credit Score	0.94 (0.12)	0.97 (0.05)	0.823	0.369	0.016

When examining the variables with the effect of feedback on the treated sample, problems seen and partial credit are not significantly different but binary score is significantly different between text and video feedback. Looking at binary score, students with video feedback scored significantly higher than those in the text group. Unfortunately when partial credit is looked at, the scores swap and the text group is doing slightly better. That seems to show that most of the students in the text group have a very high partial credit score just under the full score (0.9 versus 1.0). Looking at problems seen, students in the text group saw more problems than the video group. Table 5-3 summarizes these findings below.

Table 5-3 Means, SDs, and ANOVA results for Effects of Feedback in Treated Sample

	Video *(n = 5)*	Text *(n = 8)*			
Dependent Variable	*M (SD)*	*M (SD)*	*F*	*p*	η^2
Problems Seen	4.80 (0.84)	6.00 (2.00)	1.582	0.234	0.126
Binary Score	0.79 (0.04)	0.68 (0.09)	6.211	0.030	0.365
Partial Credit Score	0.91 (0.06)	0.92 (0.03)	0.268	0.615	0.024

Overall, it is difficult to collect much information from this data because of how small it is and the study needs to run longer in order to find any significant differences. However, if the trend were to continue as it looks now; Students see more problems when given a choice, and fewer problems when given video. The choice students tend to score lower than the no-choice students (both binary and partial credit). Binary and partial credit scores in the effects of feedback are very interesting. Binary is significantly different in favor of video (0.79 versus 0.68) and partial credit is not significantly different but is still in favor of text (0.91 versus 0.92).

Conclusion

ASSISTments originally had no personalization in assignments at a student level. The basic if-then system was then designed and implemented in order to fulfil this requirement. With the implementation of the if-then system came along the FinishedConditionService, which kept a list of FinishedConditions for each problem and problem set in the assignment. These FinishedConditions would contain information about each problem and problem set to be used by if-then and any other service that needed information. Once the basic if-then system was released, many opportunities opened up to add personalization and customization to assignments.

The next goal was to create a language and a grammar for the advanced if-then system. A language was chosen that was syntactically similar to Java but was limited to if statements. The grammar was then written using Extended Backus-Naur Form for easier understandability and the ability to translate it easily to a format for the parser generator to read. Building the parser was the next goal; JavaCC was chosen as the parser generator because it would generate Java files for GWT without too much trouble. The grammar was then translated into a config file for JavaCC to read and the parser was built.

The FinishedConditionService was updated to record additional information in each FinishedCondition for the new if-then system. The tutor was then updated to include the parser which would tell the tutor which child manifest to run based on the condition statement given to the parser. With the advanced if-then system developed, custom variables were the next addition to this project. The parser was easily updated to read the variable map provided by the StateService. The tutor itself was then updated to correctly assign variable values based on PreVariableAssignments and PostVariableAssignments on each manifest. With the completion of the custom variable infrastructure and basic implementation of variables, this project was finished and opened up many opportunities for future work and expansion.

During the creation of the if-then system, research on partial credit was done and two papers were written. The first paper, covered in Chapter three, introduced the idea of partial credit and using it to predict performance on the next problem. The paper showed that partial credit used in a tabling method is as accurate as running Knowledge Tracing (KT) and the tabling method is faster than KT. The second paper, covered in Chapter four, discussed a grid search method to determine the best partial credit algorithm. This paper showed that using a 0.1 attempt and hint weight gives the best next problem performance prediction. Each hint the student asks for and each attempt the student makes lowers their score by 0.1. With this partial credit algorithm, a study can be analyzed using the algorithm to see if there are any significant differences.

A study that used the if-then system was chosen to be analyzed and was used to evaluate the if-then system. This particular study had the most students at the time but was still not enough to demonstrate a significant difference between conditions. With the partial credit algorithm declared in chapter four, partial credit scores were calculated in the data. Using problem seen, binary credit, and partial credit as variables, ANOVA tables were created to compare choice versus no-choice and text versus video. In the entire dataset, choice versus no-choice had no significant difference according to any of the variables. With that said, students in the choice group saw slightly more problems than those in the no-choice group. They also had a slightly lower binary and partial credit score compared to the no-choice group. When looking at students who used feedback and comparing text versus video feedback, there was a significant difference between binary scores with the video group having a slightly higher binary score.

Even though it is not significantly different, it is interesting to point out that partial credit scores slightly favor the text group (0.91 versus 0.92). Students in the text group also saw slightly more problems than those in the video group. When this study has more time to run and gains more students (more than 60 overall and more than 13 that used feedback) another analysis can be run and can actually demonstrate a significant difference to see if choice is useful for students and whether text or video feedback is better.

The choice study analysis may not have proved much but it along with the list of other studies running shows how quickly researchers are willing to pick up the if-then system and use it in their experiments. The if-then system can be used for countless studies for things as simple as a video check or a choice study to many more complicated studies.

Future Work

Unfortunately there was not enough time to continue work on custom variables but the infrastructure exists to finish implementing variables and the user model data. There are many more things that could be implemented with the variable infrastructure. These include persisting variables and including the user model. Persisting variables is when the tutor sends the variable's value to the server to be saved in the database for later use or to allow researchers to read the variable values. Having these variables in the database allow for other assignments to read variables that have been declared in earlier assignments. Implementing the user model into the variable infrastructure would involve having the server keep a list of variables to be tracked per student and have this list sent down to the tutor on startup. Then when the tutor gets this list, the tutor needs to know to add this list to the StateService variables to be used later in the assignment. Some future additions to the variables could include: Storing answers or problem set state into variables, using variables values in problem bodies and overriding manifest properties. Saving answers and other state information into variables will allow that particular information to be used outside of just that problem/problem set which allows for even greater customization than what already exists. Overriding manifest properties will allow researchers to make problem sets smaller and just give each student a different set of properties. This will also allow students to request harder skillbuilders to gain more practice. Normally a skillbuilder requires the student get three problems correct in a row, but with property override the student can say they want to have to complete four or five in a row to make the assignment harder but also prove they know the material.

When coding an if statement, it is not required to have both a then clause and an else clause. Some future work on the if-then structure could include making the three children requirement optional. Since condition statements can use variables and don't need to use the condition problem or problem set then there are some condition statements where having that extra manifest is unnecessary. In the current system if you wanted to have an else clause that just finished the assignment or a condition statement with variables only then you have to include a dummy problem for the student to complete. If both the condition manifest and the else manifest were optional then creating if-then problem sets would be faster and easier to create and have less useless problems.

ASSISTments has an interesting system called scaffolding that is an alternative to hints with helping students work through problems. Hints are usually text based messages telling the student bits of information to help them with the problem and eventually giving the student the actual answer. Scaffolding works by either splitting up the current problem into smaller problems or by moving the student onto another problem that is very similar to the current one and helping the student through steps in that new problem. The system does not do any of this automatically; it is up to the builder of the problem to create these scaffold problems. Scaffolding is meant to work like a human tutor and help the student through the steps of the problem so the student can eventually figure out how to answer the original main question. Unfortunately this system is currently a bit buggy and running on old, fragile code. ASSISTments has recently begun discussing using the if-then infrastructure in place of the scaffolding system. In the current system, scaffolding activates when the student answers a question incorrectly. This falls into the if-then system very easily as the main problem is the condition manifest of the if-then problem set. If a student answers incorrectly then the if-then navigator would move the student onto one of the branch conditions and that would contain the scaffolding problem. This would allow a system that is known to be working run the scaffold problems and also remove a huge chunk of code that is written specifically for scaffolding.

References

Ainsley, C. (2014, April 14). Building parsers for the web with JavaCC & GWT. Retrieved March 30, 2015, from The Shiz: http://consoliii.blogspot.co.uk/2014/04/creating-gwt-compatible-parser-using.html

Attali, Y. & Powers, D. (2010). Immediate feedback and opportunity to revise answers to open-end questions. *Educational and Psychological Measures*, 70 (1), 22-35.

Corbett, A.T., Anderson, J.R. (1995). Knowledge Tracing: Modeling the Acquisition of Procedural Knowledge. *User Modeling and User-Adapted Interaction*, 4: 253-278.

Donnelly, C. (2015). If-Then Study Dataset. Retrieved 4/9/15, http://goo.gl/cZGUpQ

Drasgow, F. & Hulin, C.L. (1990). Item response theory. In M.D. Dunnette & L.M. Hough (Eds.), Handbook of Industrial and Organizational Psychology, Vol. 1, pp 577-636. Palo Alto, CA: Consulting Psychologists Press.

Duong, H.D., Zhu, L., Wang, Y., & Heffernan, N.T. (2013). A Prediction Model Uses the Sequence of Attempts and Hints to Better Predict Knowledge: Better to Attempt the Problem First, Rather Than Ask for a Hint. In S. D'Mello, R. Calvo, & A. Olney (Eds.) Proceedings of the 6th International Conference on Educational Data Mining. Memphis, TN. 316-317.

Gowda, S.M., Baker, R.S.J.D, Pardos, Z., & Heffernan, N.T. (2011). The Sum is Greater than the Parts: Ensembling Student Knowledge Models in ASSISTments. Proceedings of the KDD 2011 Workshop on KDD in Educational Data.

JavaCC. (2015). Retrieved March 23, 2015, from Java Compiler Compiler (JavaCC) - The Java Parser Generator: https://javacc.java.net/

JavaCC Features. (2015). Retrieved March 23, 2015, from Java Compiler Compiler (JavaCC) - The Java Parser Generator: https://javacc.java.net/doc/features.html

Koedinger, K.R. & Corbett, A.T. (2006). Cognitive tutors: Technology bringing learning science to the classroom. In K. Sawyer (Ed.), *The Cambridge handbook of the learning sciences* (61-78). New York: Cambridge University Press.

Might, M. (2014). The language of languages. Retrieved December 8, 2014, from The language of languages: http://matt.might.net/articles/grammars-bnf-ebnf/

Murphy, K. (2001). The Bayes Net Toolbox for MATLAB. Computing Science and Statistics, 33(2), 1024-1034.

Ostrow, K. (2014). L@S 2015 Submission: Dataset. Retrieved 10/14/14, http://tiny.cc/LaS2015Submission

Ostrow, K. (2014). Optimizing Partial Credit Data. Accessed 12/8/14. https://tiny.cc/OptimizingPartialCredit

Ostrow, K., Donnelly, C., Adjei, S. & Heffernan, N. (2015). Improving Student Modeling Through Partial Credit and Problem Difficulty. In Russell, Woolf & Kiczales (Eds.), Proceedings of the 2nd ACM Conf on L@S. pp. 11-20.

Ostrow, K. S. & Heffernan, N. T. (In Press). The Role of Student Choice Within Adaptive Tutoring. . To be included in Conati, C., Heffernan, N., Mitrovic, A., & Verdejo, M. (Eds.) Proceedings of the 17th International Conference for Artificial Intelligence in Educations (AIED). Madrid, Spain. pp. forthcoming.

Pardos, Z.A. & Heffernan, N.T. (2010). Modeling Individualization in a Bayesian Networks Implementation of Knowledge Tracing. In Proceedings of the 18th International Conference on User Modeling, Adaptation and Personalization. 255-266.

Pardos, Z.A., & Heffernan, N.T. (2011). KT-IDEM: Introducing Item Difficulty to the Knowledge Tracing Model. In Joseph A. Konstan et al. (Eds.): UMAP 2011, LNCS 6787, 243-254.

Pardos, Z.A. & Heffernan, N.T. (2012). Tutor Modeling vs. Student Modeling. Proceedings of the Twenty-Fifth International Florida Artificial Intelligence Research Society Conference, 420-425.

Pavlik, P.I., Cen, H., Koedinger, K.R. (2009). Performance Factors Analysis - A New Alternative to Knowledge Tracing. In: Proceedings of the 14th International Conference on Artificial Intelligence in Education, Brighton, UK, 531-538.

Wang, Y. & Heffernan, N.T. (2011). The "Assistance" Model: Leveraging How Many Hints and Attempts a Student Needs. The 24th International FLAIRS Conference.

Wang, Y. & Heffernan, N. (2013). Extending Knowledge Tracing to Allow Partial Credit: Using Continuous versus Binary Nodes. In K. Yacef et al. (Eds.) AIED 2013, LNAI 7926, 181-188.

APPENDIX A - Full EBNF Grammar

{...} = (...)* in regex

start	= expr ;
expr	= rel_expr | prop_expr | “(“, expr, “&&”, expr, “)” | “(“, expr, “||”, expr, “)” ;
rel_expr	= lhs, rel_op, rhs;
lhs	= “pr.”, p_method | “ps.”, ps_method;
rel_op	= “==” | “<” | “<=” | “>” | “>=” | “!=”;
rhs	= string | decimal | number | true | false;
prop_expr	= object, rel_op, rhs;
object	= letter, {char};
true	= “true” (case insensitive);
false	= “false” (case insensitive);
digit	= “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”;
number	= [“-”], digit, {digit};
decimal	= number, “.”, digit, {digit};
letter	= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ; (case insensitive)
char	= letter | number | “_”;
string	= “‘“, char, {char}, “‘“ | “““, char, {char}, “““ (any string of chars surround by either double or single quotes)
p_method	= share_method | “getFirstResponseTime()” | “getAnswerText()” | “getHintRequestCount()”;
ps_method	= share_method (no problem set only methods yet)
share_method	= “isCorrect()” | “getCorrectness()”

APPENDIX B - JavaCC Config File

The tokens in this file will appear different from the grammar above because JavaCC provided some additional shorthand to shorten some rules. Some rules also had to be slightly changed to optimize the parser.

```
options {
  static = false;
  JAVA_UNICODE_ESCAPE = true;
  JDK_VERSION = "1.5";
  JAVA_TEMPLATE_TYPE = "modern";
}

PARSER_BEGIN(BoolParserImpl)

public class BoolParserImpl implements BoolParser {

  boolean parse_value;
  FinishedConditions fc;
  FinishedConditionsService fcs;
  StateService ss;
  Manifest fcManifest;
  boolean isProblemSet;

  /** Constructor. */
  @Inject
  public BoolParserImpl(FinishedConditionsService fcs, StateService ss) {
      this(new StringProvider(""));
    this.fcs = fcs;
    this.ss = ss;
  }

  @Override
  public boolean getParseValue(){
    return parse_value;
  }

  @Override
  public void setQuery(String dsl)
  {
    ReInit(dsl);
  }

  @Override
  public void setFinishedCondition(FinishedConditions fc)
  {
    this.fc = fc;
  }

  @Override
  public void setFCManifest(Manifest fcManifest)
  {
    this.fcManifest = fcManifest;
    isProblemSet = fcManifest.isProblemSet();
  }
}
PARSER_END(BoolParserImpl)




/** Skip these characters, they are considered "white space" **/
SKIP :
{
    " "
    | "\t"
    | "\r"
    | "\n"
}

/** Our constant tokens that the parser will use **/
TOKEN:
{
     < AND : "&&" >
   | < OR :  "||" >
   | < EQUALS : "==" >
   | < LESS : "<" >
   | < LESSEQUAL : "<=" >
   | < GREAT : ">" >
   | < GREATEQUAL : ">=" >
   | < NOTEQUAL : "!=" >
   | < LPAREN : "(" >
   | < RPAREN : ")" >
   | < PERIOD : "." >
   | < PROBLEM: "problem" >
   | < PR: "pr" >
   | < PROBLEM_SET: "problem_set" >
   | < PS: "ps" >
}

TOKEN [IGNORE_CASE] :
{
    < TRUE: "true" >
   | < FALSE: "false" >
}

/** Problem set Find methods */
TOKEN :
{
  < PS_FIND_CHILD : "getChild" >
}

/** Method tokens */
TOKEN :
{
    < M_CORRECT : "isCorrect()" >
   | < M_CORRECTNESS : "getCorrectness()" >
```

```
    |  < M_RESPONSE : "getFirstResponseTime()" >
    |  < M_ANSWERTEXT : "getAnswerText()" >
    |  < M_HINTCOUNT : "getHintRequestCount()" >
}




/** String tokens */
TOKEN :
{
    < OBJECT : <LETTER> (<CHARACTER>)* >
  |  < NUMBER : ("-")? (["0"-"9"])+ >
  |  < DECIMAL : <NUMBER> "." (["0"-"9"])+ >
  |  < LETTER : (["A"-"Z","a"-"z"]) >
  |  < CHARACTER : (["A"-"Z","a"-"z","0"-"9","_"])+ >
  |   < STRING: "\"" <CHARACTER> "\"" | "'" <CHARACTER> "'" >
}

/** Top level */
void parse() :
{}
{
    parse_value=expression()
}

/** An expression is either
 *    - A relation_expression
 *    - (expression) || (expression)
 *    - (expression) && (expression)
 */
boolean expression() :
{
  boolean ret1,ret2;
  Token comparator;
  boolean ret_val;
}
{
  (
    (
      ret_val = rel_expr()
    )

    |

    (
      ret_val = property_expr()
    )

    |

    (
      <LPAREN>
      ret1 = expression()
      <RPAREN>
      (comparator=<AND> | comparator=<OR>)
      <LPAREN>
      ret2 = expression()
      <RPAREN>
    )



    {
      if(comparator.kind == AND){
        ret_val = ret1 && ret2;
      }else if(comparator.kind == OR){
        ret_val = ret1 || ret2;
      }else{
        ret_val = false; //Should never hit here
      }
    }
  )
  {
    return ret_val;
  }
}

/** A relation_expression has
 * left-side  operator   right-size
 * ex:  problem.correctness == 1
 */
boolean rel_expr() :
{
    Token right;
  Pair<Token,FinishedConditions> left;
  Token method;
  Token op;
  boolean ret = false;
  FinishedConditions testfc, left_fc;
}
{
  (
    //left hand side
    left=lhs()
    // operator
    op=rel_op()
    // right hand side
    right=rhs()
  )
  //LOGIC
  {
    method = left.getFirst();

    left_fc = left.getSecond();
    if(left_fc == null){
      //If this is not a problem_set.find set of methods then use the condition fc
      testfc = fc;
    }
    else
    {
      //This is a problem_set.find set of methods so we want to use the fc that was
returned by the service
      testfc = left_fc;
    }



    switch(method.kind){
      case M_CORRECT:
        ret = compareBoolean(testfc.isCorrect(), op, parseBoolean(right));
        break;
      case M_CORRECTNESS:
        ret = compareInt(testfc.getCorrectness(), op, right);
        break;
      case M_RESPONSE:
        ret = compareLong(testfc.getFirstResponseTime(), op, right);
        break;
      case M_ANSWERTEXT:
        ret = compareString(testfc.getAnswerText(), op, right);
        break;
    }

    return ret;
  }
}

Token p_method() :
{
  Token m;
}
{
  (
    m = <M_RESPONSE>
    |   m = <M_HINTCOUNT>
    |   m = <M_ANSWERTEXT>
  )
  {
    return m;
  }
}

Token ps_method() :
{
  Token m;
}
{

  (
    m = <M_TESTPSMETHOD>
  )
  {
    return m;
  }
```

```
}

Token share_method() :
{
  Token m;
}
{


  (
    m = <M_CORRECT>
    |   m = <M_CORRECTNESS>
  )
  {
    return m;
  }
}


FinishedConditions ps_p_find_method(String manKey) :
{
  String key = manKey;
  Pair<String,FinishedConditions> pair;
  List<Integer> childIntList = new ArrayList<Integer>();
  Token f,arg;
}
{
  f = <PS_FIND_CHILD>
  <LPAREN>
  arg = <NUMBER>
  <RPAREN>
  <PERIOD>
  {
    childIntList.add(Integer.parseInt(arg.image));
  }
  (
    (
      f = <PS_FIND_CHILD>
      <LPAREN>
      arg = <NUMBER>
      <RPAREN>
      <PERIOD>
    )
    {
      childIntList.add(Integer.parseInt(arg.image));
    }
  )*
  {
    pair = fcs.findChildFinishedCondition(key, childIntList);
    return pair.getSecond();
  }
}
```

```
Pair<Token,FinishedConditions> lhs() :
{
  Token m = null;
  FinishedConditions fc = null;

}



{
  (
    (
      <PROBLEM> | <PR>
      {
        if(isProblemSet){
          throw new ParseException("This condition is a problem set not a problem");
        }
      }
      <PERIOD>
      (
        m = p_method()
        | m = share_method()
      )
    )
    |
    (
      <PROBLEM_SET> | < PS >
      {
        if(!isProblemSet){
          throw new ParseException("This condition is a problem not a problem set");
        }
      }
      <PERIOD>
      (
        (
          m = ps_method()
          | m = share_method()
        )
        |
        (
          fc = ps_p_find_method(fcManifest.getKey())
          (
            m = p_method()
            |  m = ps_method()
            |  m = share_method()
          )
        )
      )
    )
  )
  {
    return new Pair<Token,FinishedConditions>(m,fc);
  }
}
```

```
/** A relation operator is any of the following:
  *  - <
  *  - <=
  *  - >
  *  - >=
  *  - ==
  *  - !=
  */
Token rel_op() :
{
  Token op;
}
{
  (
  op=<EQUALS>
  |  op=<LESS>
  |  op=<LESSEQUAL>
  |  op=<GREAT>
  |  op=<GREATEQUAL>
  |  op=<NOTEQUAL>
  )
  {
    return op;
  }
}

/** the right-side is any of the following:
  *  - String: any set of characters, numbers, or symbols wrapped in double quotes ""
  *  - Number: any number (with an optional "-"  in front)
  *  - True
  *  - False
  */
Token rhs() :
{
  Token right;
}
{
  (
  right=<STRING>
  | right= <DECIMAL>
  | right=<NUMBER>
  | right=<TRUE>
  | right=<FALSE>
  )
  {
    return right;
  }
}

boolean property_expr() :
{
    Token right;
  Token left;
  Token method;
  Token op;
  boolean ret = false;
  String property_value;
  String property_type;
}
{



  (
    //left hand side
    left=<OBJECT>
    // operator
    op=rel_op()
    // right hand side
    right=rhs()
  )
  //LOGIC
  {

    property_value = ss.getProperty(left.image);

    if(property_value == null)
    {
      throw new ParseException("Property value: " + left.image + " does not exist in 
the properties map.");
    }

    switch(right.kind)
    {
      case TRUE:
        ret = compareBoolean(true, op, parseBoolean(right));
      case FALSE:
        ret = compareBoolean(false, op, parseBoolean(right));
        break;
      case NUMBER:
        ret = compareInt(Integer.parseInt(property_value), op, right);
        break;
      case DECIMAL:
        ret = compareDouble(Double.parseDouble(property_value), op, right);
        break;
      case STRING:
        ret = compareString(property_value, op, right);
        break;
      default:
        throw new ParseException("not sure how to determine property type for 
property_value " + property_value + " (property: " + left.image + ")");

    }

    return ret;
  }
}
```