AbstractTutor: Increasing Algorithm Implementation Expertise for Novices Through Algorithmic Feedback

Leigh Ann Sudol-DeLyser

CMU-CS-14-145

December 17, 2014

School of Computer Science
Program in Interdisciplinary Education Research
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported through the Program in Interdisciplinary Education Research (PIER) at Carnegie Mellon University, funded through Grant R305B040063 to Carnegie Mellon University, from the Institute for Education Sciences, US Department of Education. The opinions expressed are those of the author and do not represent the views of the Institute or the US Department of Education.

Keywords: Computer Science Education, Pedagogical IDE, Algorithmic Abstraction, Novice Feedback

For all novices who thought it was their fault.

Abstract

The translation of algorithms and abstractions to formalisms, most often code, is a fundamental task of a computer scientist. Both novices and experts use development environments to provide feedback about the code they have written as a part of the iterative process of solving a problem. Almost all of the environments available were designed for the expert and his or her feedback needs while writing code. In this thesis, I begin with an analysis of the feedback needs of the novice, and discuss the need for explicit feedback focused on the algorithmic components novices should use when practicing the construction of simple array algorithms. As a proof of concept, I pilot test a model of algorithmic components and feedback for four easily generalizable problems using think aloud protocols. After validating the model and feedback, and showing initial gains after practice, I discuss the implementation details of AbstractTutor, a pedagogical IDE using static analysis techniques to give pre-compilation feedback to students using the system. The implementation details include a series of case studies highlighting the successful evaluation of student code, and discussing the few situations where student mistakes produce code that is temporarily unable to be correctly parsed by the system. To increase the granularity, accuracy, and specificity of analysis, I present two metrics for evaluating student code from a body of submissions. These metrics represent a new way of thinking about novice progression through a coding problem, and allow analysis to focus on individual learning components within the larger algorithm students are constructing. Finally, I detail the result of two online, multi-institutional studies where novice programmers use the AbstractTutor system. Students who received the algorithmic feedback were more likely to make productive edits, and less likely to repeat errors on subsequent problems. This thesis offers contributions to the learning sciences, computer science, cognitive psychology and computer science education.

Acknowledgments

This thesis, and my PhD have been made possible by a significant number of people in my life.

First and foremost, my husband Matt DeLyser. Your ongoing partnership in our marriage has made all that I do possible and I could not be the successful woman and mom that I am without your continued support, love, encouragement, and friendship, thank you.

My family has taught me the value of hard work, especially my father David Jervis, who never did anything without care, thoughtfulness, and a work ethic that I can never match, thank you.

To CMUs Computer Science Department, thank you for taking a risk on me and being willing to recognize that you are not my target research population. I am honored to have worked along side you for years.

Thank you to my Pittsburgh people, the Doherty crew who welcomed me into grad school, and Kami Vaniea, Ciera Jaspan, and Christy McGuire who helped me overcome my imposter syndrome and realize I had expertise to offer.

Thank you especially to my advisors Mark Stehlik and Sharon Carver. Neither of you regularly take graduate students but without the individual expertise of both of you this thesis would not have been possible. Sharon, thank you for holding me to high standards, you have changed the way I write and the lens I use to look at research opportunities. Mark, your belief in me, continued support, ongoing challenges, questioning, and late night writing sessions have made you a PhD advisor anyone would be proud to have, and I am proud and thankful to be your “one and only ever” PhD student.

Contents

List of Figures

List of Tables

Chapter 1

Translating Algorithms to Formal Language

In 1967 Herb Simon defined computer science as the study of computers and all the phenomena that surround them. As the discipline of computer science matured over the next half-century, subdomains including theoretical computer science, artificial intelligence, robotics, human computer interaction, software engineering, and machine learning have developed and were recognized by the Computer Science community. Although the field itself has diversified, a common thread among subdomains, and an important part of many introductory classes, is the communication of a series of steps, or algorithm, to a computational device.

A key skill in the development of fluency in constructing and communicating algorithms is the ability to use specific, often formalized, language to describe the steps of the algorithm to the computational device. Although formalism adds a layer of complexity to the process, the use of a formalization can help novices think precisely about the algorithmic components that are often implicit in natural language. As an example, a favorite exercise in many introductory Computer Science classes is to enact student written directions for common tasks, such as making a peanut butter and jelly sandwich. Students often rely on implicit directions when writing the algorithm, and the teacher will take a literal interpretation of the students' commands to demonstrate the need for absolute clarity and specificity. For example, the direction "Put the peanut butter on the bread" results in the peanut butter jar being stacked on the loaf of bread.

The formalized language used by a large majority of introductory courses to construct and communicate algorithms is a programming language. These languages can range from industry standards such as Java, to pedagogical languages such as Scratch or Alice. Regardless of the language used, the communication of algorithms is done through combinations of commands expressed through the rules of the formalized system, often called code.

1.1 Code Production: Expressing Algorithms in Formal Language

An algorithm can be defined as a "series of instructions for completing a task". Humans express algorithms every day in their communications with each other. Giving directions, sharing a recipe, or laying out tasks for a work product all require that a plan of action for solving the problem be developed and then be decomposed/partitioned into the subtasks required for completion, and then effectively communicated to another individual.

Difficulty arises when communicating with a person who speaks another language, such as Spanish or French, who perhaps comes from a different culture or backround, where implicit references and idioms are not the same as your own. Communicating with a computational device or computer has many of the same challenges as communicating with someone who does not speak your language. Nuance and implicit understanding are not conveyed and a level of formality and precision is needed in order to express meaning without misunderstanding. There are many formal languages that can be used to communicate algorithms. Flow charts, logic or mathematical languages, programming languages, and even formalized pseudo code can all be used to communicate the steps of an algorithm. Often in our introductory computer science courses, we use one or more of these formalizations; however in a vast majority of courses, students are asked to implement their solutions in a particular programming language.

The formalization of steps or directions is an important component of Computational Thinking as defined by Wing [91]. Wing states that Computational Thinking is the "thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be carried out by an information processing agent" [90]. In order to practice representation and formalization, we often ask students to write programs that implement basic algorithms to gain the repetition and practice needed to identify the algorithmic patterns that will become abstractions. The abstractions are formed by the identification of patterns within the specific formalized implementations students practice, and then recognizing and re-using those patterns in the solutions to future problems. Students often do this with little success, especially as novices [53].

The requirements of formality ensure that we use precise descriptions and constrain the discourse to a specific set of instructions that have been defined for the formalism. Additionally, especially for programming languages, there is a syntax requirement for the formalization in order to craft even a basic implementation of the problem solution. The syntactic requirement means that in addition to the correctness of the sequence of instructions, you must also include the correct punctuation, potentially the correct spacing, and often the correct grouping of commands to ensure they are understood.

In order to adopt a common educational practice of leveraging prior knowledge, a number of studies have been conducted to determine the prior knowledge introductory students have of some complex algorithms. These studies asked the students to describe a task in natural language, and then analyzed the responses for appropriate algorithmic components and structures. With varying degrees of success, students were able to express algorithms for counting the number of letters in a series of mailboxes [80], selling tickets to a movie(queuing problem) [75], debugging a program [74], and efficiently searching a space [29].

All algorithms are composed of a series of atomic commands understood by the computational agent. These commands, or algorithmic components, are rigorously defined by the formalism and are able to be enacted without further explanation or decomposition. For example, a small list of such commands for a computer may include assignment (a=b), addition (a+b), decision (if-else), and repetition (while, for). Even this simple list masks a range of abstract layers, from the lowest level (machine code) to the highest (e.g., repetition), however most programmers, including novices, work with a programming language or formalism that contains these abstractions as atomic commands.

The complexity of *describing the task* faced by novices when producing code that implements algorithms is almost equal to the complexity that the novices themselves face in their attempts to translate algorithms to code.

1.2 Difficulties Novices Face with Translating Algorithms to Code

Despite being able to express some algorithmic solutions to problems in natural language, novices often face difficulty expressing the algorithms in the formal language required in an introductory programming course [53]. As with many academic endeavors, the whole is greater than, or more difficult than, the sum of its parts. In order to produce code that implements an algorithm to solve a specified problem, students must first understand the problem, then associate it with an appropriate abstraction, for example associating the counting of the votes in an election across precincts with a sum abstraction. After the abstraction is chosen, the student must then decompose the algorithm into its relative components, creating a variable to hold the sum, a loop to repeat the process over each precinct, and a statement to add the precinct's total to the overall total. The choice of components can be dependent on language features [1]; however, within an introductory course, students often use only a single language and therefore do not need to make the choice between languages or paradigms.

Once the components are identified, the novice must then translate the mental model of the component sequence into the formalism required by the programming or practice environment. For example, the recognition that a variable needs to be initialized to store the sum would turn into the following code snippet

```
int sum =0;
```

in the programming language Java. The formalism in the Java language requires the specification of a variable type (integer) and a starting value for the variable before it can be used.

In addition to the components required by the formalism, there are often syntax requirements. For example, assignment statements in Java require that the value for the

[1]For example, in a functional language, the repetition through the list may be implemented by a recursive function call instead of a loop.

variable be placed on the right side of the equal sign. For the expert, this happens without thought; but for the novice, whose experience with variables may come from a mathematical context where order does not matter, the inversion of the statement to 0=sum provides a cryptic message that must be interpreted and corrected before receiving feedback about the larger algorithmic structure. Although computer science education researchers have not yet explored the specific impact of learning syntax on programming language acquisition, the impact of syntax on the acquisition of a second natural language (i.e., an English speaker learning Spanish) has been well studied. A focus on the construction of meaning using appropriate vocabulary over the syntax or structure of novice statements will produce more efficient learning of new languages [22, 86].

When learning a programming language, the combination of the translational step to a program and the unforgiving requirements of the formalism, as often enforced by a compiler, compound the difficulty of implementing an algorithm to solve a problem.

1.3 Existing Feedback Mechanisms Are Not Designed For The Novice

In the process of becoming a proficient producer of algorithms, novices write code in order to practice the translation to formalism, find the correct sequence and structure of commands, as well as learn to think about edge cases and test code for correctness. Feedback during practice is an important component in the development of fluency or expertise [38]. Unfortunately, many of the systems used by novices learning to write code do not offer feedback that is directed at or useful to the novice programmer.

Computer scientists, both novice and expert, employ various tools to provide feedback during code production. The most common tool for both the novice and professional is the IDE, or Integrated Development Environment. IDEs provide an editor, an environment for writing the textual code required, a compiler to provide feedback about syntax, and often runtime and debugging functionality allowing for testing and debugging of code. Although IDEs can have advanced features, such as in-code drop down menus for completion of statements, they are primarily designed to be generalizable and therefore cannot offer built in problem feedback for the programmer. Problem specific feedback is done through the creation of test cases that evaluate the output of the code based upon a series of provided inputs. Often experts create tests that check the behavior of runnable[2] code by entering a series of input values they construct themselves and checking to see if the output matches expectations or requirements. The creation of test cases itself requires a level of expertise to know what appropriate values are, and what input values are most likely to cause an error. This feedback is often very helpful to the expert and very unhelpful to the novice. Novices often lack sophisticated understanding of the algorithm they are trying to produce so they have trouble determining which of the many components could be responsible for a change to the output.

[2]Runnable code indicates no compiler errors.

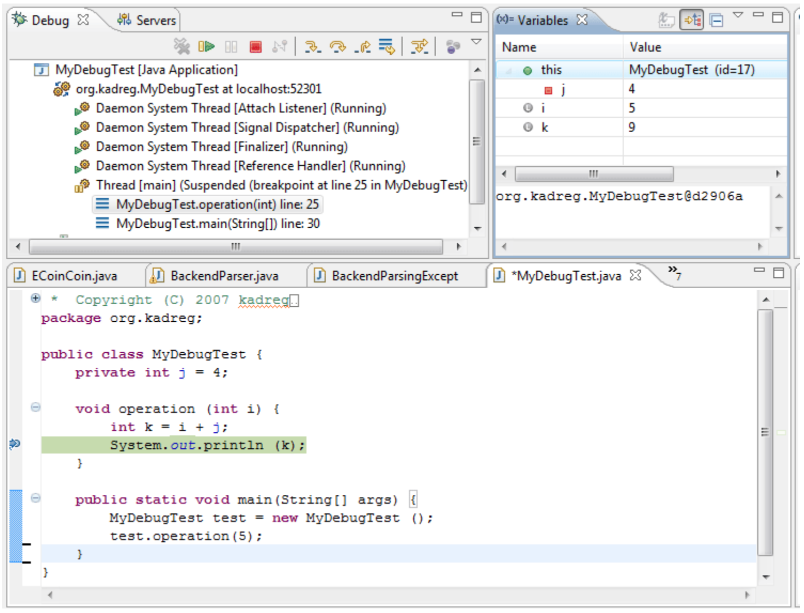
1.3.1 Support for Code Production: An Artifact of History

An inspection of early expert processes involved in code production can shed light upon the difficulties of today's novices. Although the first computers were built to perform calculations, the execution of those computations by modern day standards was painfully slow. Early computer programmers could not compile their programs in an instant, so they followed a process that was heavily influenced by the cost of compilation. Algorithms were written with extreme care before they were even communicated to the computer through a keyboard (or other device). Writing a computer program involved careful thought about a solution. With a compilation and execution time of days or hours, the algorithmic components were often verified or hand traced before the code was submitted to the compiler. The programmer would return to the computer lab a day later to pick up the result of the execution - either compiler errors or a list of outputs based upon the predetermined inputs provided by the programmer. Again, with the cost of computation, it was up to the programmer to unpack this feedback to determine whether the program worked correctly or not.

With the early model of programming, despite having access to computational tools, the cost of computation was prohibitive and self reflecting strategies were key to being successful. Over time, the computational cost of compilation and execution reduced, and professors of computer science tried to prompt the planning and self reflection in students through instruction [5, 49]. The tools in a majority of introductory courses, however, remained the same. Feedback is centered on compilation and output based execution, based on the idea that what the humans do well is the big picture, the algorithmic abstractions, and what the computer does well is process the details. While this may be true for the professional, it is not true for the average novice.

The average novice goes through a very different process in order to plan, write, enter (type), and execute algorithms in code. First, the novice must unpack the problem statement. In this step, the student must take the high level description of the problem and decide what algorithms or structures should be applied in the code. As a novice, it is likely that the student is mostly recalling the worked examples seen in class or a textbook and is trying to match the problem to one of them. The structures or algorithms must then be decomposed into the relevant individual components required for the task. These components must be ordered correctly, and even more so must have references to each other at appropriate parts. In a modern computer science class, a novice may enter whatever he or she remembers and press compile as a way to check the solution.

Notice the difference. The expert or early programmer will read (or even trace with values) the code that has been written in order to check the correctness of the algorithm implemented. Novices instead put together what they remember and attempt to use the compiler or other development tool as a feedback mechanism that can help them correct errors. The types of feedback both users need to efficiently arrive at a solution is very different, just as the process both users are going through is different.



```
MyDebugTest [Java Application]
  org.kadreg.MyDebugTest at localhost:52301
    Daemon System Thread [Attach Listener] (Running)
    Daemon System Thread [Signal Dispatcher] (Running)
    Daemon System Thread [Finalizer] (Running)
    Daemon System Thread [Reference Handler] (Running)
    Thread [main] (Suspended (breakpoint at line 25 in MyDebugTest)
      MyDebugTest.operation(int) line: 25
      MyDebugTest.main(String[]) line: 30
```

Name	Value
this	MyDebugTest (id=17)
j	4
i	5
k	9

```
org.kadreg.MyDebugTest@d2906a
```

```java
 *  Copyright (C) 2007 kadreg
package org.kadreg;

public class MyDebugTest {
    private int j = 4;

    void operation (int i) {
        int k = i + j;
        System.out.println (k);
    }

    public static void main(String[] args) {
        MyDebugTest test = new MyDebugTest ();
        test.operation(5);
    }
}
```

Figure 1.1: An Example Debugger: Variable Values are Displayed in Upper Right

1.3.2 Feedback in Integrated Development Environments

An Integrated Development Environment (IDE) is a piece of software that combines a text editor with a compiler and other useful tools for writing programs. The feedback offered by IDEs varies between the environment and the specific purpose for which it was designed. Some IDEs offer feedback in the form of visualizations built upon the code produced by the programmer. The visualizations take many forms and can focus on different aspects of the code structure or operations. Most IDEs provide a view of the structural components of a program or larger project. In some cases, the structure is represented by a file structure, possibly including the names of methods (or subprograms) found within the files. The structure could also be used to display an object hierarchy in languages where appropriate. The visualizations of file or object structures can help with locating particular methods or understanding code structure, but they will not often help debug errors arising from incorrect logic in code.

Imagine trying to determine why a car will not start. You have a diagram of the car systems. The diagram can light up the systems in the car that are being manipulated in the car starting sequence. For example, you turn the key and the key lights up, followed by some lights along the connections from the starter to the battery. Imagine the lights flash in some sequence and you are left to determine how the flashed sequence is different from an optimal sequence. Without expert knowledge of how cars operate, the diagram may help you locate trouble areas, but cannot tell you what to change in order to correct the problem.

Although structural representations of the program are important, additional representations are useful when trying to debug unexpected program behavior. A second type of visualization focuses on the data being stored and manipulated by the program. In its

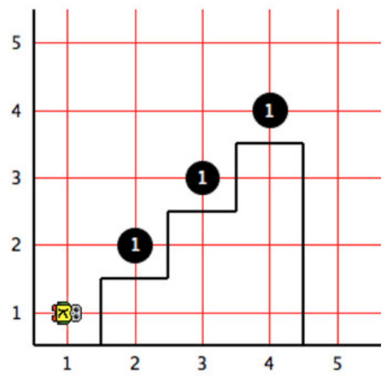


Figure 1.2: Karel J. Robot Context

simplest form, the feedback is the display of values held by individual variables available through most debuggers. Figure 1.1 shows an example from a popular IDE. The upper right panel displays the values held by variables i, j, and k. The upper left panel displays the stack trace, or series of methods that have been called to arrive at the current code location. The larger lower window displays the code being debugged, with the highlighted line the last line to be executed. Each "step" or single line execution is triggered by pressing a key or button on the screen. The programmer will step through the program execution, one command or block of code at a time, and watch the variable values change. This approach is similar to output based feedback, however it allows the programmer to observe intermediate states within the code and infer correctness based upon those values. In more sophisticated IDEs, the visualization will extend to an image of the data structure or even an animation of the construction and changing values within the structure that is produced during execution of the program. For the novice, although this information could help troubleshoot a problem, the complexity of the screen, combined with the need to observe state change over time, is often too difficult to parse and understand.

1.3.3 Feedback in Contextualized Environments

A third type of visual feedback centers on the task or context provided to novice programmers, either by the problem assignment from the teacher or embedded in the environment itself. In these task-based environments, programming fundamentals are taught and practiced through the contextualization of the concept within problems in a world or environment. One example of such a context is displayed in Figure 1.2 and makes use of the Karel J. Robot library and curriculum. In Karel, the primary actor in most programs is a robot named Karel that completes tasks navigating on a 2D grid. Environments such as Karel have been shown to be successful at engaging students in an introductory course [9], learning the basics of objects and fundamental control structures [70], and assisting in the understanding of data handling in objects [25].

The key to the contextualized environments is the ability to describe the desired result of the program in natural language, and have a visual display (or other cues for the novice) that creates a pictorial representation that is easily matched to the natural language description. For example, in Figure 1.2 the robot will need to walk up the stairs and pick up each beeper (labeled with a 1). The novice should understand the idea behind walking up the steps (a move in the positive y direction, then positive x direction) and what would change visually if each beeper is successfully picked up. If Karel does not walk the appropriate path, or does not pick up the beepers, the novice will not only see the incorrect final state of the environment, but also the series of events (each step the robot makes) that yielded the incorrect solution. The direct relationship to natural language removes a barrier of translation from an abstraction or idea to a concrete representation for the novice and allows for a mental model of the output to be directly tied to the individual actions of the program. Novices, however, are still left to make the connection between the visual representation (output) and the code that they have written in order to debug any algorithmic mistakes. Some environments provide code highlighting during execution to assist novices in locating where their algorithm deviated from their intended solution [71], but this feedback can only help the novice determine a potential code location and not the appropriate algorithmic component or command to use.

1.3.4 Strengths and Weaknesses of Current Feedback Mechanisms

Each of these feedback approaches has its own individual strengths and weaknesses. The trade-off between generalizability and feedback to support novices has long been a characteristic of the division between IDEs and tutoring systems. From an IDE perspective, generalizability is important so programmers can use the tool to create software unimagined by the IDE designers. For the novice programmer, however, the generalizability is less important as many introductory courses focus on similar topics and assign practice targeting similar learning objectives. Pedagogically focused tools do provide more feedback, often through visualizations to the student, but still are restricted by the need for generalizability across an almost unlimited set of problems. Although they visually represent the code produced, these tools or contexts still rely on the student to infer the mistake in the code from the feedback. Additionally, *novices must parse and correct compiler errors before the contexts or visualizations will provide the intended scaffolding.*

As described in Chapter 2, the practice programs or exercises assigned to novices often involve more lines of code or elements than a novice can chunk or hold in working memory. The generalizable feedback mechanisms offer symptoms of the problems in the student's code, and they rely on the student to jump between levels of abstraction in determining (1) the actual cause of the error (which may or may not directly relate to the error message) (2) the way to correct the symptom observed by the error message presented, and (3) the overarching effect of the correction on other aspects of the program. For example, an error resulting from a lack of variable initialization could be a syntax error, a null pointer exception during runtime, or an unexpected output value during output-based testing. The difference between the type of error given for the same underlying cause depends upon the programming language and the type of variable that was uninitialized. Even if the error is the most specific, a compiler error indicating the variable was not initialized, the error message often displays the line number where the variable is used and not necessarily where the variable was created or needed to be initialized. The novice is left to perform the complex diagnosis, identifying which variable is at fault, or the reason for the incorrect value, relying on her own mental model of what the algorithm should be (which may be faulty) and then attempting to match the code produced to that model.

Depending upon the environment producing the feedback, the task the programmer is engaged in, and the desired outcome of the activity, there are a number of steps the programmer employs to make productive edits after receiving feedback from a system. An expert reads the feedback and then often uses a locator such as a line number to determine the general area in the code producing the error, reads the code, using cues from the feedback message to identify the source of the error, and finally conceptualizes the appropriate modification to fix the error. An important distinction to make is although feedback is produced in response to a particular error in the code, the feedback in most cases will not be exact in identifying what needs to be changed in order to make the code correct. For example, a misspelled variable will not produce feedback indicating a "spelling" error, as it would in a word processor. Instead, the feedback for a misspelled variable would most likely indicate that there is a variable that has not been defined. Due to the need for generalizability, feedback mechanisms in IDEs are often imperfect and have steep learning curves for novices, since the feedback only exposes a symptom and not the actual cause of the error.

1.3.5 Evidence of Student Difficulties in Understanding Feedback

Feedback can be a powerful tool during practice to promote the shift from novice to expert. To be most effective, however, the feedback must be comprehensible by the student and the student must believe that reading the feedback will assist in problem solving efforts. As previously discussed, many of the IDEs used by students for general practice when writing code are designed by and for experts. The feedback produced is often only indicative of a symptom of a problem within the code and not directly related to the algorithm being produced. This misalignment between novice knowledge and feedback leads to difficulties that are too challenging to be desirable.

The challenges faced by students with error messages, the most common form of feedback during practice, have led to multiple proposed alternatives to the traditional programming editor. Drag and drop environments such as Alice [20] and Scratch [51] avoid errors by only allowing students to "drop" programming blocks in appropriate places. The inability to use a particular block in a place is in itself a subtle form of feedback, however the systems do not provide a message as to why the block is incorrect. The implicit feedback relies on the novices to spontaneously reflect on the attempted incorrect action and determine for themselves the source of the error and how to correct it. With regards to algorithmic components, these systems are similar to other visual contextualized environments, as the result of the program is most often displayed as an animation on the screen. Watching the animation, users must make the inference themselves between the actions of the objects or sprites and the program code.

Other work has focused on evaluating or modifying the messages themselves. One example of simplified error messages can be seen in the BlueJ environment [40]. The error messages are displayed one at a time and have been edited to remove some of the message complexity. For example, when you receive an error in the BlueJ environment, you can click on the error to take you to the line of code where the error message was triggered (not always the line containing the error). In a standard IDE, the line number is included in the error message and then the novice must find the line of code corresponding to the line number.

Other work has taken a data driven approach, looking at the source of errors in the Dr. Racket IDE [52]. Authors found common underlying mistakes causing program errors in novice code, and propose modifying specific error messages based on these data. Again these efforts are designed to correct student difficulties with compiler messages, and studies show that even in environments such as BlueJ, where the messages have been modified, students often struggle and engage in cycles of unproductive edits [34].

Although the analysis of student responses to error messages and the proposed modification of the messages is a useful usability endeavor, it is not aligned with a direct learning goal outcome. When analyzing the ability of students to correctly apply compiler error messages to code edits, Marceau et al. in [52] observed "It is tempting to interpret these graphs as indicating students' *conceptual* difficulties in the course. This interpretation is invalid because the error *message* the student sees is not a direct indicator of the underlying *error*. For instance, Lab 6 had numerous 'unbounded-id' errors, but a careful manual analysis revealed that many of them were from students improperly using structures, not merely being bad typists. To further confound matters, the precise error that is shown is a function of parsing strategies, because many invalid expressions can be flagged in multiple ways. Therefore, an additional manual analysis is necessary to understand what actual errors students were making." As the parsed errors themselves are symptomatic of underlying issues with the student code, and vary by the assignment the student is completing, a rework of the error messages for a generalizable IDE can at best offer students a menu of options representing potential sources for the error.

In the quote, Marceau et al. refer to the misconception, hidden in an expert blind spot, that compiler errors often result from students being "bad typists". This claim would be similar to asserting that a novice second language learner forms incorrect sentences not because of a misunderstanding of the new language, but because he made typos that went uncorrected. Clearly novices will make semantic and syntactic mistakes in a new language based upon imperfect knowledge more often than mistakes based on typing errors. Yet in computer science most of the tools students use for practice require that the syntax be correct before any feedback relating to the meaning of the code can be produced.

The approach of analyzing student attempts to determine appropriate modification of error messages is an approach to the difficulties by looking only at the observable.

By treating the issues as an HCI problem, we can understand the user base and make changes that impact the easily observable behavior of program edits. The primary goal of novice practice, however, is not just to reduce the number of unproductive edits. Instead instructors assign practice problems in order to provide opportunities for a student to apply the program components (variables, control structures, or data structures) to a variety of problems and develop abstractions surrounding these components so they may be used again in future problem solving scenarios. As an instructional designer, I propose focusing feedback on the desired learning outcomes and not the observable symptomatic difficulties students have.

1.4 Research Questions: Improving Feedback by Aligning AbstractTutor with Algorithm Production Goals

The difficulties that students have translating algorithms into code, combined with ineffective feedback mechanisms, have inspired the research questions addressed by this thesis. Rather than studying the efficacy and modifications of feedback messages produced at compile time, this work will analyze the addition of a feedback cycle before compiler error messages when possible. In this cycle, the system uses knowledge of the problem the student is attempting in order to preliminarily evaluate the potential "meaning" of the code.

The work presented in this thesis makes contributions to both Computer Science and Cognitive Science through the construction of the AbstractTutor system, and then the use of that system to study novice responses to feedback during practice. Similar to an English teacher checking the outline of an essay before reading the text, AbstractTutor verifies the existence of specific algorithmic components before checking for syntax or using test cases to provide output based feedback. The verification and feedback regarding potential meaning prior to the display of compiler errors has the potential to improve novice learning, and the alignment of feedback messages regarding algorithmic components with instructional goals should only strengthen the potential learning gains.

The two main research questions addressed in this work are: **(1) Can a pre-compilation feedback mechanism be constructed that operates with reasonable accuracy (85% of student generated submissions)? and (2) Will pre-compilation feedback regarding algorithmic components produce better (a) within-problem performance and (b) across-problem learning.**

Answering these questions required a design exercise and a series of qualitative and quantitative studies with secondary research questions. Chapter 2 provides the motivation behind the choice of algorithmic components as a path towards fluency and expertise in algorithmic construction of code. Chapter 3 presents a brief overview of the inspiration for the feedback model and its alignment with introductory instructional goals. In Chapter 4, I present the results of a think aloud study, used to validate the feedback model. Chapter 5 details the technical implementation of the feedback model, as well as contributions to the learning sciences in the analysis of within problem progress. Chapter 6 presents methods of assessing student code submissions as a contribution to the Educational Data Mining community. Chapter 7 presents the results of a multi-institutional study where subjects used the online AbstractTutor system during practice for an introductory computer science course, where students who saw algorithmic feedback were more likely to make productive edits. In Chapter 8, I present my conclusions and directions for future work.

Chapter 2

Using Models of Expertise to Guide Feedback Development

Despite the variety of interventions discussed in the last chapter, novices still struggle with code production. Novice computer science students are often able to express rules for appropriate syntax, translate individual lines of code to English at a very basic level, and follow pre-written code using tracing strategies. Although the mastery of these individual skills is highly correlated with increased skill in code production, the combination of mastery in individual tasks does not signal expertise in code production, especially when longer programs or more complex algorithms are involved. Prior research has demonstrated a disconnect between knowledge of individual program components, or the ability to understand lines of code, and the ability to produce complex code structures or algorithms. The lack of a strong connection indicates that additional support is needed during learning and practice to help students make explicit connections between the code components they understand and the algorithms they are attempting to author. In this chapter I will discuss a variety of research highlighting the differences between low level code knowledge and the skill of algorithm production. These studies are framed with two models, the SOLO Taxonomy and Block Model, derived from experts and novices, showing that experts understand code at a different level of abstraction than novices. These results are used to motivate a framework for generating within-practice feedback and measuring student progress, detailed in Chapter 6. The feedback uses language that connects mistakes in the students' code to the algorithmic abstractions that I seek to foster through the AbstractTutor system.

2.1 Expertise in Algorithm Abstraction and Production

Expert performance in a domain is often characterized by fluency/accuracy, proceduralization of common tasks and complex conceptual structures. Additionally, these complex conceptual structures are organized based upon context and experience as much as topic within the domain. The complex structures of chess masters, for example, have been shown to not only provide for problem solving assistance, but also for chunking or encoding board states [14]. The same studies showed an inability to chunk impossible board states, indicating that it was not a pictorial representation borne out of familiarity, but one deeply rooted in the problem solving embedded in the game of chess. Experts are also more likely to be able to identify problems of similar types, even when the context of the problem solving situation has changed [16].

These theories of expertise have important implications for the learning of programming, especially when practice is comprised of complex tasks with interrelated structures. Just as experts in chess are able to chunk combinations of pieces in representative board states, so too do computer scientists chunk series of commands into complex interrelated structures, most often representing algorithms, in working memory. Evidence of this can be seen not only in experts' ability to recreate code in a memory test, but also in studies where students are asked to debug or describe code [73]. In questions that contain algorithms with a slight error or small inconsistency, weaker students are much more likely to locate the error or identify the changed behavior of the code [2]. Although this seems to be contrary to theories of expertise, it actually strengthens the argument for chunking by the expert. The higher accuracy on the part of the novice could indicate a closer scrutiny on the part of the weaker student in order to form an understanding of the code, while the expert will make a classification based on the larger structure rather than the individual details. Similarly, in studies of reading, an expert reader can easily parse a sentence where the internal characters of words are scrambled and, depending on the severity, will not even notice, while the novice will struggle with understanding [67].

The size of the conceptual chunks may play a role in students' ability to not only set and carry out goals during practice exercises, but also in their ability to learn the complex abstractions that rely on multiple interacting components of algorithms. If a novice is working at the token level of abstraction, seeing each variable name, syntactical structure, and operator as a separate entity, he may not be able to hold the entire task in working memory. This constraint will make it difficult for the novice to see the whole plan or algorithm at one time and remember the connective structures from practice when later assessed. Without directed metacognition after the practice exercise is completed, many students simply are relieved to have the "correct answer" and do not often reflect on the components they have assembled in order to arrive at that answer. Instead, students often express relief or pride at the completion and seek the next goal, homework assignment, practice attempt, or concept without reflecting on the difficulties they encountered and the appropriate solutions.

Due to the expression of difficulty and a significant failure or dropout rate from early courses, educators often debate what can be changed about introductory courses in order to alleviate the difficulties that students are experiencing, produce a more positive learning experience, and increase student learning during practice. With self reflection, many faculty arrive at the conclusion that the difference from their learning experience to the current introductory curriculum is the programming language, not the concepts, and so debates over which language to choose are ongoing. Many of the arguments over an appropriate first programming language center on paradigm (object oriented vs. procedural vs. functional) and syntactical difficulties are often specified as a reason to choose one language over another [39]. Spohrer and Soloway in 1986 took a different view, stating

> Our empirical study leads us to argue that (1) yes, a few bug types account for a large percentage of program bugs, and (2) no, misconceptions about language constructs do not seem to be as widespread or as troublesome as is generally believed. Rather, many bugs arise as a result of plan composition problems - difficulties in putting the pieces of a program together - and not as a result of construct-based problems which are misconceptions about language constructs. [77].

Some pedagogical approaches have targeted the formation of plans, goals, or abstractions in the form of patterns for the student to follow [6]. This work was inspired in part by the building architecture literature [4] and has been shown to have impact on student learning [56]. The relationship between expertise and findings regarding student mistakes and subsequent debugging attempts have prompted the focus on feedback regarding algorithmic components in this thesis, as opposed to a study of error messages and their efficacy[1].

2.2 Algorithmic Abstraction in Code Comprehension

The attempts to measure novice programmers' ability to understand and produce complex programs has resulted in several correlational studies and two models to explain student reasoning. Similar to the relationship between reading comprehension and writing in natural language, a positive correlational relationship has been established between the ability to comprehend code and produce code [50]. In addition, this correlational relationship is not predictive, indicating there is another skill set required from the identification and translation of code to the production of lines of code to produce an algorithm [88]. Many interventions have asked students to explain code and then evaluated the answers for complexity and indications of algorithmic chunking or abstraction [48, 89]. No prior interventions have looked at the result of feedback containing messages about algorithmic abstraction on students while writing code.

Although the prior literature has focused on measuring the relationship between proficiency at code writing and the ability to 'translate' code, little work has connected these correlational studies with fluency and expertise. The differentiation between fluency and expertise in problem solving domains is a fine line. Fluency is representative of one's ability to translate and converse in the language of the domain, while expertise involves the ability to reason and transfer the fluent knowledge to new problem solving situations. In computer programming, novices at the early stages of fluency can accurately describe individual lines of code, or trace a series of statements, correctly producing an output value when given particular inputs. These novices, however, cannot be considered experts and often have difficulty rearranging the lines they have translated into a coherent whole, or describing the code in perceptual chunks instead of individual lines [21].

A favored assessment item for determining the expertise of students is the "Explain in Plain English" code comprehension question [57]. In these items, students are presented with a piece of code, often involving several lines to complete an algorithm, and asked to

[1]Additional work is being conducted on the efficacy of error messages and the implications for student problem solving during practice [28]

explain what the function of the code is, or the resulting changes to variable states. The students are often not provided with initial assignment values for the problems, as the goal of the question is not to produce a numeric output, but instead require the student to use natural language to describe the parts of the algorithm presented. In a previous study [83], I recorded open ended responses to similar types of questions. In the question, students were provided with the code shown below that counted the number of items in a list whose value was less than the parameter val.

```
int countLessThan(int myList[], int val){
  int c=0;
  for(int i=0; i<myList.length; i++)
    if(myList[i] < val)
      c++;
  return c;
}
```

The students were asked to describe the role of the variable i in the algorithm presented. Table 2.1 shows several student responses to the question. Student 1 answered the question with understanding of the code, but almost a direct translation of the meaning; this answer shows little descriptive understanding of the purpose of the variable, only its state changes throughout the looping structure. Students 2 and 3 are slightly more sophisticated in their answers, connecting the variable i to the idea of an index and even to the use of it to obtain values from the array myList. Student 4 answers with a much more complete answer, indicating that while the variable is representative of the indexes of the values stored, its primary purpose is to access and possibly alter the values stored within the list. Results of this study also indicated that with feedback connecting the code to abstractions, students were able to produce more sophisticated open-ended responses over time [82].

Student 1	"Counts from 0 to length minus 1"
Student 2	"i acts as a temporary index in the for loop"
Student 3	"i runs through each index of myList"
Student 4	"i is used as temp values from 0 to len(mylist) such that each element in the array mylist can be accessed and altered."

Table 2.1: Student Responses to an Explain in Plain English Question

The diversity of student responses, and the need for descriptive categories of explaining these responses has lead to the use of two main schema for classifying open ended student code descriptions. The Structure of Observed Learning Outcomes, or SOLO, Taxonomy [11] has been used on several occasions to map students' responses to a hierarchy of complexities [89]. In the SOLO Taxonomy, students move from a Pre-Structural answer (oversimplified), to an Extended Abstract response. Table 2.2 describes the different levels of the SOLO Taxonomy. From the examples provided above, Student 1 would be in the Uni-Structural category, Students 2 and 3 would be Multi-Structural, and Student 4 would be Relational. The answer from Student 1 is less abstract and represents a direct translation of the code seen in the problem. Student 4 offers a more abstract explanation, giving not only the values that i will assume through the loop, but also the purpose that the variable serves within the algorithm.

Extended Abstract	The previous integrated whole may be conceptualized at a higher level of abstraction and generalized to a new topic or area.
Relational	The different aspects have become integrated into a coherent whole. This level is what is normally meant by an adequate understanding of some topic.
Multi-Structural	The student's response focuses on several relevant aspects but they are treated independently and additively. Assessment of this level is primarily quantitative.
Uni-Structural	The student's response only focuses on one relevant aspect
Pre-Structural	The task is not attacked appropriately; the student has not really understood the point and uses too simple a way of going about it

Table 2.2: The Solo Taxonomy

The SOLO Taxonomy has limitations because it only employs a single dimension of complexity in analyzing the student responses to code. A second model for evaluating student code that is used frequently in the literature is the Block Model [72], outlined in Table 2.2. The Block Model differs from the SOLO Taxonomy as it includes a second dimension regarding not only the size of the perceptual chunk used by the student, but also the interrelational aspects of the code description. The spectrum from text and atoms, to the macro-structure of a program represents a student's ability to reason about the blocks and how they interact with each other (relations). The other axis of the Block model deals primarily with Duality in terms of Structure and Function. These terms are used differently than Structure-Behavior-Function Theory [30][2] in that Function here refers to the purpose or goals of the program. The structure of the program refers to the relationship between the sequence or ordering of the lines and the intended outcome of the program.

Returning to our previous examples of student code descriptions in Table 2.1, Student 1 would be operating at the atom level in the Structure domain. Students 2 and 3 would still be at the atom level, but focused more in the Function domain. Student 4 would be at the Relation level in the Structure domain. Student 4 could move to the Function domain if he/she included statements about the purpose of the overall program; however the question asked in the example does not prompt the student to describe the larger problem.

In both of the models, the more abstract the student's reasoning about the code, the higher on the chart the response will score. The concept of abstraction becomes even more

[2]Structure-Behavior-Function Theory is a well known theory in cognitive science and is cited here to illustrate the difference in the language used.

Macro Structure	(Understanding the) overall structure of the program text	Understanding the "algorithm" of the program	Understanding the goal/the purpose of the program (in its context)
Relations	References between blocks, e.g.: method calls, object creation, accessing data...	sequence of method calls "Object sequence diagrams"	Understanding how subgoals are related to goals, how function is achieved by subfunctions
Blocks	Regions of Interest (ROI) that syntactically or semantically build a unit	Operation of a block, a method or a ROI (as a sequence of statements)	Function of a block, maybe seen as sub-goal
Atoms	Language elements	Operation of a statement	Function of a statement. Goal only understandable in context
	Text Surface	Program Execution (data flow and control flow)	Functions (as means or as purpose), goals of the program
	Duality	Structure	Function

Table 2.3: The Block Model for Student Code Explanations

important in the development of fluency as we consider numerous studies where significant correlations were found between students' ability to produce statements that score highly on these models, and their ability to produce correct algorithms in code writing tasks [48, 49, 50, 61]. Despite these correlations, much of the feedback that students receive while engaged in the practice of code production is focused on the individual details of the code, or the atoms referenced in the block model. A standard compiler, and most Integrated Development Environments (IDEs) will only provide feedback about syntactical errors, such as punctuation out of place, as a student is working. The feedback produced either by IDEs or through run-time testing creating output based feedback[3] favors the expert. The low-level syntax corrections are meant to signal an error that may or may not be a direct indication of the problem, but instead a symptom of the real issue. An expert not only understands this relationship but also can use the symptom presented in the error to diagnose the problem.

Once all the compiler errors are corrected, the student can receive feedback at the macro-structure level, but she is often left on her own to reason about the relations between blocks of code. This feedback is once again only an indication or a symptom of something that may be incorrect and the novice is left to reason about the underlying cause of the mistake. This process often leads to struggling, and sometimes guessing on the part of the student. Unless a student engages in unprompted metacognition at the completion of the problem, there is often a large disconnect between the debugging effort based upon feedback and the conceptualization of the algorithmic components the student produced at the start of the problem solving process.

Through the use of this example, previous work, and the descriptions of the models used to evaluate the complexity of student reasoning, I have illustrated the importance of addressing multiple levels of abstraction in the student learning process. In the next section, I present a model of code production that highlights where the difficulty with low level reasoning and program comprehension can interfere with production, thereby inhibiting the benefits of practice exercises on student learning. The model of code and algorithmic production seeks to identify code blocks and potential relations that can be used to produce feedback for students during problem solving, thereby scaffolding the jump from atom level (compiler feedback) and macro-structure (output based feedback) from novice to expert.

2.3 Models of Code and Algorithm Production

Students are often assigned code-writing tasks as practice in introductory computer science courses. These code writing tasks vary from large, complex programs that build on an established code base [47], to smaller snippets of code designed explicitly to practice specific skills or implement common algorithms [41, 59]. There have also been several tutoring systems focused on the construction of a single line of code or expression and focused on building student skills over time [27]. The following model, shown in Figure 2.1, is offered

[3]In run-time testing, a series of values are used to provide initial values for input to the code and then resulting output is tested against expected output or values.

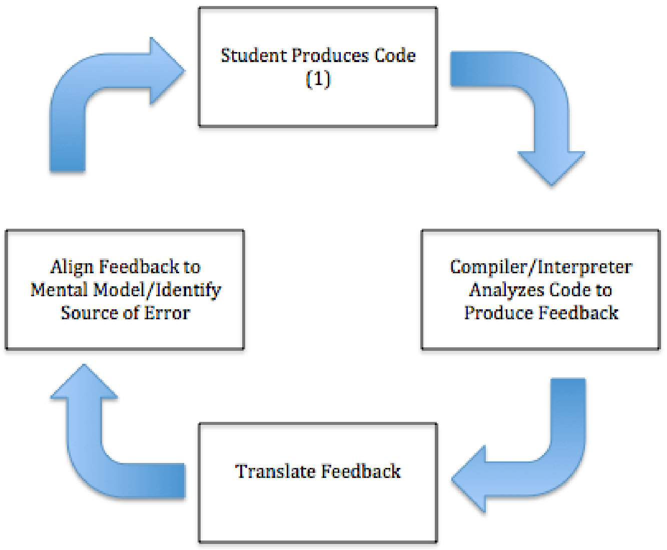

Student Produces Code (1)
Compiler/Interpreter Analyzes Code to Produce Feedback
Translate Feedback
Align Feedback to Mental Model/Identify Source of Error

Figure 2.1: A Model of Student Code Production Practice

for student code production activities, except for those cases where the system focuses on a single line of production at a time. This flow diagram is an elaboration of the model presented by Jadud in [34] and includes descriptions of student processes while problem solving.

In the model, a student will make an initial attempt to construct a solution (1), and submit the code to a compiler, interpreter, or other construct of the system in order to check the syntax, and sometimes the output of the program. After the code is submitted, the system will respond with feedback. The feedback produced can be in response to many factors, such as a compilation error or a failure of an output-based test, and it can take many forms, such as text on the screen or a visualization of the program execution. Once again, this feedback is focused on the line-level or atomic details of the code. It is at this time that the student must perform the first step in translating the feedback from the system and aligning it with the mental model of the program he has constructed [44]. The student then engages in a code comprehension step to determine the appropriate location in the program where an edit would correct the error conveyed by the feedback. Upon identifying a location, the edit is made and the cycle begins again.

Regarding the identification/translation of the feedback received by the system, there is evidence that novices often do not read the error messages with which they are presented, and often have difficulty understanding the messages they do read [28]. Additionally, in his work on the compile-edit cycle, Jadud found many different types of student problem solving patterns, and the difficulties encountered with system messages that often do not reflect the actual errors in the code [34]. This lack of attention to the feedback presented by the system often results in an attempt to solve the problem through unfocused, often unproductive, edits, sometimes referred to as "shotgun debugging".

Regardless of whether the student has fully translated and parsed the feedback, or simply understood that there is some error (not what the error is), the next stage in the cycle is one of code comprehension. The code comprehension activity is performed in order to identify the location within the code for the next edit, in an attempt to correct the error producing the feedback. As many of the programs that students are working with, especially in this study, are larger than their working memory [55] this code comprehension step is not merely a recall of the code produced, but an actual prompt for self explanation[4]. If the student only understands the code at the atom level, his self-explanations will be limited to the low level translational aspect of his understanding. These students will often struggle with the appropriate modifications to correct their code and may use tracing strategies[5] in order to identify the source of the problem. Additionally, the students with atom-level understanding may be more likely to resort to shotgun debugging, especially if they lack the self efficacy with regards to their ability to fix the problem to expend the mental effort to find the exact source of the error.

Once the student has decided where the error in the code exists, he will engage in an editing process that requires him to decide how to fix the error. This edit is observed in the compile submissions and will be used to infer, combined with other metrics such as time between submissions, the effort expended by the student and the intent of the edit. Once again, an edit can be severely hampered by an atom-level representation of the code if the student is fixing more than a compiler error. A structural or logical error in the algorithm may not be easily represented by individual atoms and once again the student is left with the decision to guess what the appropriate edit may be.

In this model of code production, there are multiple steps that could be influenced with feedback. I could clarify the messages and highlight the lines of code responsible for errors as suggested [52] however this approach will only reinforce the line or atom level evaluation of the code in question. Instead, I offer a strategy where feedback is presented that focuses on the relational structure of the code, and prior to working with the compilation issues provides students with information regarding the algorithmic components for the problems they are solving. The feedback focuses on blocks of code, such as a for loop or if statement, and the relationships of particular variables in order to bridge the connection between the atoms and the macro-structure they produce. In the next section, I describe this feedback, the intended impact on the students' problem solving process, and a bottom-out feedback message for students who are still unable to make a productive edit after receiving the algorithmic component message.

[4]The Think Aloud protocols collected and analyzed in Chapter 4 provide additional evidence of self explanation at this step.

[5]Tracing strategy: substituting sample values for the inputs and tracing the code to determine output values.

2.4 Feedback Representing a Model of Algorithmic Construction

The work presented in this thesis aims to help students who are mostly fluent in the coding structures they are producing and are in the early transition stages to expertise [48]. These students have completed most of a first course in programming and are working with course assignments that require the construction of multiple lines of code, making use of variables and structures that play important parts in the construction of the overall algorithm. Similar to the goal/plan analysis work of Spohrer and Soloway [76, 77], the feedback will be generated based upon the usage of algorithmic elements required to complete the task presented to students. The algorithmic elements and their descriptions will be abstract in nature, as described in the SOLO Taxonomy and Block models.

The pedagogical IDE (PIDE) developed in this work relies on assessing individual constraints in order to provide feedback about the algorithmic components that may be lacking in student code. These constraints rely on parsing the code, and often rely on knowledge of multiple aspects of the program in order to ascertain if the constraint has been satisfied. The feedback messages were designed to include language about how the components either worked together, or about the purpose of the missing constraint in the larger algorithm. Recognizing that some weaker students may still be unable to parse the more advanced feedback, the system will provide secondary feedback if the error is still not corrected in the next testing cycle.

In this chapter I have described two models of algorithmic abstraction from the literature on expertise. Additionally, I highlighted the disconnect between the current feedback mechanisms and the practice of the novice. In this thesis I seek to align feedback with the algorithmic components and algorithmic abstraction as described in the SOLO Taxonomy and Block Models, in order to help students move from novice to expert. In the next chapter, I describe a model of algorithmic components appropriate for the problems solved in this thesis, and then detail a think aloud study in Chapter 4 to validate the model.

Chapter 3

Developing Model of Algorithmic Components for Feedback

3.1 Algorithmic Components Assessed for Generation of Feedback

Teachers have been evaluating student code, by hand and automatically, as long as programming has been an academic subject. The evaluation has served many purposes, including the generation of feedback to help novices correct errors and learn from their mistakes. In order to ensure consistency, award partial credit, or evaluate specific subparts of a larger problem, teachers often employ a rubric or other codified metric to evaluate code. The creation of such rubrics often requires pedagogical content knowledge as the rubric should represent the instructional goals of the assignment, as well as the common mistakes that students make during the problem solving process.

Although instructors can weigh, through the use of rubric points, the importance of program subgoals as aligned with instructional focus, many automated assessment systems treat all errors as equal and focus only on achieving full program correctness. While full program correctness is usually the overall goal, systems that take a broad approach often cannot provide feedback for the learner with the appropriate level of abstraction necessary to promote the most efficient learning. For example, a novice who uses a variable without declaring it in the Java language would receive the same feedback as an expert who had a typo, misspelling the name of the variable. These two errors have very different underlying causes, omission of a key part of the process and a small typo, yet receive the same feedback. The sacrifice of feedback specificity for the generalizability of the tool often leaves novices confused and without the support they need to have productive practice.

The balance between generalizability and the construction of tools appropriate for the novice can be seen in a comparison of automated grading systems. Automated grading systems and feedback mechanisms employ a number of strategies for the design and implementation of systems that evaluate student code. In this chapter, I focus on a model used to decide what feedback messages are appropriate for a particular student code submission. The components used in the model are aligned with appropriate instructional goals for a novice in a first computer science course in Java. Chapter 5 discusses the implementation of the system that uses the model and the technical details of the feedback generation. In this chapter, I present a brief overview of the inspiration for the feedback model and its alignment with introductory instructional goals.

3.2 Assessment of Code for Evaluation of Student Knowledge

Instructors in introductory computer science courses give assessment questions requiring students to read, explain, and write code. The largest standardized assessment of introductory computer programming in the United States is the Advanced Placement Computer Science A (APCS) Exam. The APCS exam is taken by approximately 39,500 students in the United States and abroad in 2014 [1]. The exam has two parts, a multiple choice section and a free response section. The free response section is comprised largely of problems where the student must write code to solve a problem presented. Both the problems and the assessment methodology in my AbstractTutor system were largely inspired by the free response part of the APCS exam and its grading methodology.

As an instructor, I participated with the APCS program on a variety of levels. I taught the course in high schools for 9 years. I am also a certified CollegeBoard consultant, training other teachers in the best practices of teaching APCS. Additionally, for 7 years I worked as an APCS exam reader and question leader. As an exam reader, I was responsible for applying a rubric to hand written student code produced during the exam. During an average exam reading session of one week I would grade approximately 2,000 exams, and over 7 years I have developed an expertise in evaluating student code by hand. For three of the 7 years, I served as a question leader where I was responsible for creating the rubric used to assess the student code.

The CollegeBoard rigorously evaluates the grading methodology for consistency and validates the rubrics and problems against college level students using their course grades and scores on assessment items. Carnegie Mellon is a frequent participant in the piloting of assessment items for the APCS exam. Based on the rigorous evaluation of the methodology and my extensive experience with the APCS assessment, I feel confident in applying similar techniques to the evaluation of student code for the production of feedback related to algorithmic components. As a national curriculum based on a survey of US colleges and universities, the common rubric points that are aligned with algorithmic components should also align with instructional goals in most introductory computer science courses in the United States.

Table 3.1 shows a solution to a problem from the 2010 APCS exam and the accompanying rubric used in the exam scoring. The problem asked students to access a List storing CookieOrder objects to "compute and return the sum of the number of boxes of all cookie orders." [1] The canonical solution is shown on the left and the rubric on the right.

An important feature of the rubrics used in the scoring of student code is the decomposition of the algorithm into the relevant components that when combined in the right structure make a working solution. The rubric must also be generalizable to a variety of

Canonical Solution	Rubric (5.5 Points Total)
public int getTotalBoxes(){	+1 Consider all CookieOrder objects in orders
int sum = 0;	+1/2 access any element of orders
for(CookieOrder co : orders){	+1/2 access all elements of orders
sum += co.getNumBoxes();	
}	+1 1/2 Computes total number of boxes
return sum;	+1/2 Creates an accumulator (declare and initialize)
}	+1/2 Invokes getNumBoxes on object of type CookieOrder
	+1/2 Correctly accumulates total number of boxes
	+1/2 returns computed total

Table 3.1: APCS Example and Grading Rubric

different solutions, as the APCS course is taught by a variety of instructors and although a testable language subset is given, students may use any valid Java code in order to solve the problem. The scoring rubrics for the APCS exam are used to award partial credit for code that demonstrates knowledge by evaluating the inclusion of algorithmic components in student hand written code. Although in an automated system focused on feedback, the use of partial credit is not a goal of the system, the structure of the rubrics makes them ideal for evaluating student knowledge for the application of feedback.

Within each problem is embedded many smaller problems, solvable by individual code atoms, or chunks. These code atoms or chunks represent a single concept, from declaring a variable to accessing the length of an array, required for a student to produce a working solution. During initial practice for a novice, exercises that have a singular, specific answer are used to develop fluency in individual topics. Within a few weeks, however, novices must engage in programming practice requiring the construction of more complex programs involving algorithms with multiple lines of code (and containing multiple atoms) arranged in a particular structure.

As the complexity increases, so too does the answer space of possible correct answers, and the number of atomic units, or code chunks, necessary to construct a correct solution. Although it is time efficient and much easier technologically to only categorize the entire solution as correct or incorrect, the feedback provided by such evaluation lacks the granularity to be useful for the novice, and the data collected by such a system would be too course grained to be of use to the educator or education researcher. Instead, I propose to identify the relevant indicators of mastery of the individual components (code atoms or chunks) and evaluate each of those pieces individually for correctness and appropriate placement in the code structure.

Consider the following two incorrect answers to a similar problem to find the sum of items in a list of numbers, shown in Table 3.2 In both of these examples, the output-based feedback would see the same value for every possible test case. The example on the left most likely is a typo, or a small misunderstanding on the part of the student, but the example on the right represents a much larger misconception about what the question is asking. If the designers of the feedback mechanism knew that code that produced the same value as the last number in the list was a common wrong answer, they could prepare a feedback message related to the error. In this case, however, two very different student submissions would have generated the same output, and the feedback would be about the *output value* as opposed to the code produced by the student.

1	public int findSum(int []myList){	public int findSum(int []myList){

2
3
4
5
6
7
	int sum = 0;
	for(int i=0; i<myList.length; i++){
	sum = myList[i];
	}
	return sum;
	}
		int end = myList.length;
		return myList[end];
		}

	Line number 4 above contains a mistake: sum = myList[i]; should be sum += myList[i];	There is no one line with an error here, instead the student shows a large misconception in not accessing every element with a loop before returning a value.
	Output Based Feedback: “When provided the input [1,2,3,4] your code returned 4 and should have returned 10.”	Output Based Feedback: “When provided the input [1,2,3,4] your code returned 4 and should have returned 10.”
	Desired Feedback: “When provided the input [1,2,3,4] your code returned 4 and should have returned 10.”	Desired Feedback: “The task you are trying to achieve requires a loop to access each element in the array.”

Table 3.2: Two Incorrect Sums

The solution presented on the left has many of the necessary algorithmic components in order to produce a correct solution - the student only needs to change one line of code so that each item in myList (myList[i]) is added to the sum on line 4. The code on the right however only accesses a single variable, and therefore needs more algorithmic components - the student should be told that each element of the array needs to be accessed in order to find the sum of all the elements. The equality of output of these two very different solutions emphasizes the need for a more fine-grained feedback mechanism than what is currently employed in many educational tools.

3.3 Assisting Students with Code Writing

There have been many approaches over the last 40 years to assist students with the production of code as a pedagogical task. I used two dimensions of assistance in order to

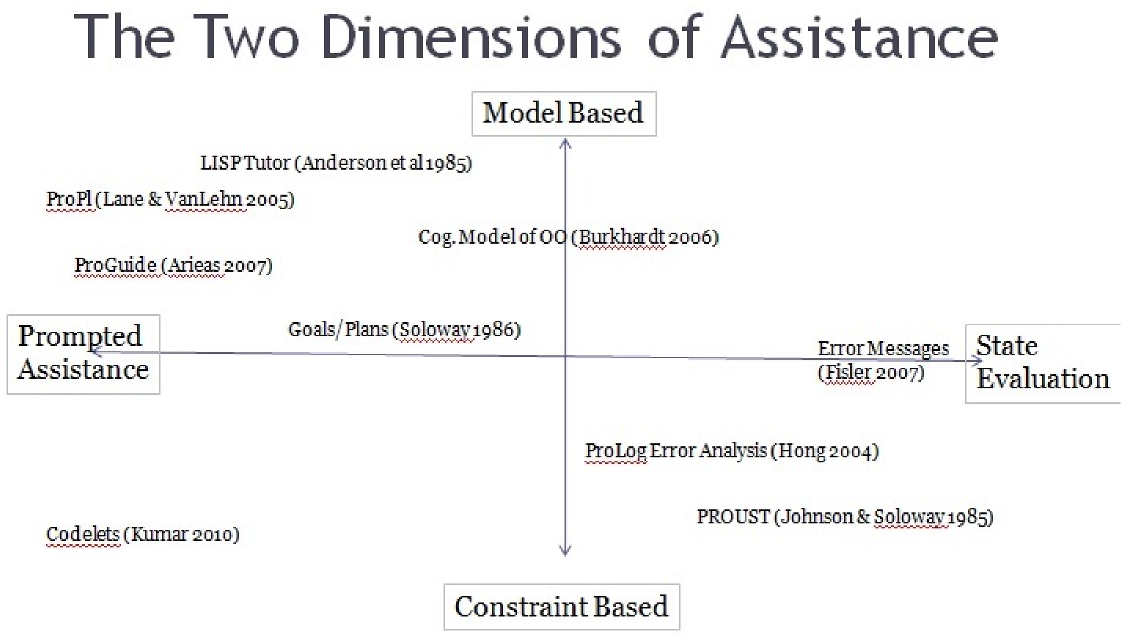

The Two Dimensions of Assistance
Model Based
LISP Tutor (Anderson et al 1985)
ProPl (Lane & VanLehn 2005)
Cog. Model of OO (Burkhardt 2006)
ProGuide (Arieas 2007)
Goals/Plans (Soloway 1986)
Prompted Assistance
Error Messages (Fisler 2007)
State Evaluation
ProLog Error Analysis (Hong 2004)
PROUST (Johnson & Soloway 1985)
Codelets (Kumar 2010)
Constraint Based

Figure 3.1: Two Dimensions of Assistance

provide a framework for discussing some of the systems used in the past. Figure 3.1 shows a sample of tutoring systems plotted on a two dimensional axis to highlight the relative features of each system.

The horizontal dimension of the Assistance Axis deals with the generation of support or feedback messages to the learner. On the left side of this axis are systems that prompt the learner for the next step before the learner has entered any code. The prompting is an excellent strategy for the novice who is unsure of where to start, or for a system that is used for both practice and instructional purposes. The weakness of a prompting system is that the learner will eventually need to learn the problem decomposition required to write algorithms on his own, and the prompting can offer subtle cues about sequence and structure based on the prompts.

On the right side of the support/feedback axis is the opposite of prompted support, state evaluation and feedback. In a state evaluation system, the learner is allowed to make an attempt at writing code to solve the problem, and the system then determines what feedback is appropriate. AbstractTutor deals specifically with state evaluation, and is not trying to become a prompted assistance tutor. Instead, in this thesis I focus on student practice that mirrors professional practice of constructing an algorithm (or solution to a problem) and then testing it in a system.

The vertical dimension of the Assistance Axis deals with the method by which the next prompt or the next feedback message is decided. At the top of the axis is a model based system. Model based systems use some type of model, cognitive or otherwise, to either infer intent or misconception and provide an appropriate message to move the learner closer to a correct solution [13, 54]. At the bottom of the axis are constraint based systems. A constraint based system uses a series of rules related to observable correct or incorrect features of the solution or code in order to select the next feedback message [45]. One major difference between a constraint and model based approach is the evaluation of the learner. In a model based approach, the model is an inference about the learner - either in terms of their intent or knowledge. In a constraint based system there is no evaluation of the learner, instead only the observable solution is used to determine his intent or knowledge at any given point.

As with many implementations of theory, there are many systems that are a hybrid of the two dimensions of assistance. A selection of these systems are plotted and will be discussed as they relate to the design decisions made for AbstractTutor.

There are a number of systems that offer prompted assistance to students as they learn to code. ProPL [43], ProGuide [5], Soloway's Goal/Plan system [76] and Codelets [41] are just a few. In these systems, a student is prompted for the next step in completing an algorithm and, based on the answer compared to a model (ProPL, ProGuide) or a series of constraints (Codelets), feedback is given to help the student arrive at a correct solution. There are also a few systems that offer the evaluation of student code with no initial prompting. Systems such as Dr. Java [79], Proust [36], and CodingBat [59] allow the user to read the problem and make an attempt to construct a complete solution on his own. After constructing the solution, the system evaluates the code submitted by the student and either generates feedback based upon a model of desired student behavior or a series of constraints meant to identify a correct solution.

As discussed in the previous section, the use of constraints focused on algorithmic components is inspired by the rubric based grading that expert human raters use when evaluating student code. AbstractTutor's feedback mechanism uses algorithmic components, allowing the feedback generator to distinguish between very different solutions that produce the same output. Additionally, AbstractTutor is currently a practice based environment where students are only interacting with a few problems, therefore the development of a rich model would be difficult.

The AbstractTutor system uses inspiration from the rubric based grading methodology of an assessment focused on the knowledge demonstrated by students.The grading methodology translates well into a constraint model.The choice of constraints is appropriate as computer programming is an ill defined domain with a large search space of potential solutions, especially once students begin to write multi-line algorithms to solve problems. Within the system, I focused on a set of algorithmic components to be used as constraints. The algorithmic components allow for the generation of feedback that is explicitly tied to the algorithmic components necessary for the algorithm. The explicit feedback is an important feature of a novice support system according to Sack and Soloway."To be effective, an automatic program debugger must have methods allowing it to both: identify errors in computer programs, and explain to the student the errors it has found." [69] Many of the systems in use today only focus on identifying the errors for students, and not explaining the errors. In the rest of this chapter, I detail the algorithmic components used by the system in order to generate feedback messages, and the feedback produced by incorrect code relating to the component.

3.4 Algorithmic Components for Introductory Array Problems

3.4.1 Instructional Setting for the Model

In order to discuss the model of algorithmic components developed for this thesis, I need to first discuss the expected preparation of the learner, the desired outcomes from practice, and the problems provided during practice. The problems in AbstractTutor focus on simple array algorithms that commonly occur in computer science, specifically calculating a sum, finding the maximum value in a list, counting the number of values in a specified range, and returning the index of a specified value. As described in Section 2.4, the problem space for AbstractTutor was selected to target students toward the end of their first course in computer science. Although abstractions can be used to describe the atoms or code chunks used in early program assignments, prior work by the author has demonstrated the a ceiling effect with novices on easier problems [80]. The simple array algorithms chosen here are representative of the worked examples and programming practice students engage in when first encountering algorithms of this complexity, so they would be appropriate for students in any introductory course with a chapter on arrays and array algorithms [85].

Expected Preparation of the Learner

The problems and feedback evaluated by AbstractTutor are designed to provide assistance to novice computer programmers in their first course. The topic of simple array algorithms is not the first topic presented in such a course; in fact, it often appears near the end of the sequence of topics, as it combines many key skills - variable declaration, looping, using data types (arrays), and decision structures (if statements).

Students are expected to have attempted, if not completed, practice exercises containing variable declarations and decision structures prior to encountering the AbstractTutor topics. Students should also have been exposed to, through reading or lecture, worked examples containing a loop structure (often a for loop) and how to access all the elements of an array using a loop. The worked example most students encounter is to print all the values in the array.

As students in their first course, they may not have mastered any of the above topics. In fact, many students will still be struggling with a variety of the subtopics involved in the AbstractTutor system. Although ideal instruction would provide individualized practice at the particular level of the student, the AbstractTutor system is not currently adaptive and is meant to provide appropriate feedback and not do problem selection.

The Desired Outcomes from Practice

The desired outcomes from practice in AbstractTutor are twofold. First, the practice should reinforce the individual concepts required to complete the algorithms specified (variable declaration, looping structures, decision structures, returning variables). Secondly, the practice should help students understand the composition of the algorithmic components into a more sophisticated algorithm to process a data set.

With these two outcomes in mind, I designed a model to evaluate student code in order to produce feedback. The individual algorithmic components were selected to represent the individual concepts that students would need in order to complete a correct algorithm in this selected set of problems. In addition to providing feedback about the components themselves, the feedback messages were designed in order to highlight the way in which the components can be combined to produce the effective algorithm.

The Problems

The four problems implemented in the system were chosen to have common algorithmic components for focused practice. The four algorithms participants were asked to implement were a sum, max search, counting the number of elements in a particular range, and finding the index of a particular value. Each problem was presented in either a numerical context (find the sum of all the numbers in the list) or a story context (find the total price of all items on a receipt). The two contexts were presented as a part of an early research question attempting to measure whether the story context would be easier or more difficult for students. In all data collected for this thesis, there was no difference between the mathematical context and story context, and all results are collapsed from this point forward. The tables below show the two versions of each problem description, one with a numerical context and the other with a story context, as well as a potential solution for each problem.

Problem	Numerical Context	Story Context
Sum Description	Write a method to find and return the sum of all the values stored in the array `public int findSum(int []myList){`	The array items contains the price of items on a sales receipt. Write a method to find and return the total (sum) of all the items on the receipt. `public int findSum(int []items){`

Solution	`int sum = 0;`
	`  for(int i=0; i<myList.length; i++){`
	`    sum += myList[i];`
	`  }`
	`  return sum;`
	`  }`
		`int sum = 0;`
		`  for(int i=0; i<items.length; i++){`
		`    sum += items[i];`
		`  }`
		`  return sum;`
		`  }`

Max
 	
Description
	Please complete the method findMaximum below. The method takes an array of numbers called myList as a parameter and returns the maximum values stored in the array myList.
	
	`public int findMaximum(int []myList){`
		Please complete the method findMaximum below. The method takes an array which contains the MPG of several vehicles. Find the largest MPG stored in the list.
		
		`public int findMaximum(int`
		`                  []vehicleMPG) {`

Solution	`int max = myList[0];`
	`for(int i=0; i< myList.length; i++){`
	`    if(max < myList[i])`
	`      max = myList[i];`
	`    }`
	` return max; }}`
		`int max=vehicleMPG[0];`
		`for(int i=0;i<vehicleMPG.length;i++){`
		`  if(max<vehicleMPG[i]){`
		`    max = vehicleMPG[i];`
		`  }`
		`  return max;`
		`}`

Table 3.3: Sum and Max Descriptions and Solutions

Count
Description
	Please complete the method countLessThan below. The method takes an array of numbers called myList as a parameter and returns the number of non-negative items(including 0) less than the parameter value in the array myList.
	
	`public int countLessThan(int []myList,`
	`int value){`
		Please complete the method countLessThan below. The method takes an array of grades from a midterm exam. Find the number of positive scores (greater than or equal to 0) less than the passing grade given.
		
		`public int countLessThan(int []scores,`
		`int passingGrade){`

Solution	`int count = 0;`
	`for(int i=0; i< myList.length; i++){`
	`  if(myList[i] < = 0)`
	`    count++;`
	` }`
	` return count;`
	` }`
		`int count=0;`
		`for(int i=0; i<scores.length; i++){`
		` if(scores[i] >=0 && scores[i] < value)`
		`     count++`
		` }`
		` return count;`
		` }`

Index
Description
	Please complete the method indexOf below. The method takes an array of numbers called myList as a parameter and returns the first index of the parameter value if it is contained in the array. If it is not contained the method should return -1.
	
	`public int indexOf(int []myList,`
	`int value){`
		Please complete the method indexOf below. The method takes an array of office numbers as a parameter and returns the index of the first Office Number that matches the value idno if it is contained in the array. If it is not contained the method should return -1.
		
		`public int indexOf(int []roomNumbers,`
		`int idno){`

Solution	`for(int i=0; i<myList.length; i++){`
	`  if(myList[i] == value)`
	`    return i;`
	`  }`
	`  return -1;`
	`  }`
		`for(int i=0;i<roomNumbers.length;i++){`
		`  if(roomNumbers[i] == value)`
		`    return i;`
		`  }`
		`  return -1;`
		`  }`

Table 3.4: Count and Index Descriptions and Solutions

3.4.2 The Model

The following model specifies the algorithmic components necessary to solve the problems currently used by the AbstractTutor system. AbstractTutor uses constraints in order to assess the partial correctness of a student solution. The constraints for the problems are the detailed description of the algorithmic components necessary to generate appropriate feedback. The constraints described are represented in a hierarchy based upon the associated algorithmic components. Each of the three categories displayed with dependencies from left to right (for example you cannot check if the code uses the length of the list to terminate a loop without first having a repetition (loop) structure). The identifying appropriate elements category also relies on the implementation of some looping structure.

The following sections detail the constraints, as well as the part of the problem solving process where each constraint is checked. Most constraints are checked upon submission, before the code is executed, however the checks for correctness are confirmed through the use of output-based testing. Each submission is an attempt by the student to either confirm correctness, or seek feedback. Even if the student does not believe that the submission is completely accurate, it is an attempt to move forward with the problem. It is the combination of the checking at submission with the output based testing that contributes to the accuracy of the system, as described in Chapter 5.

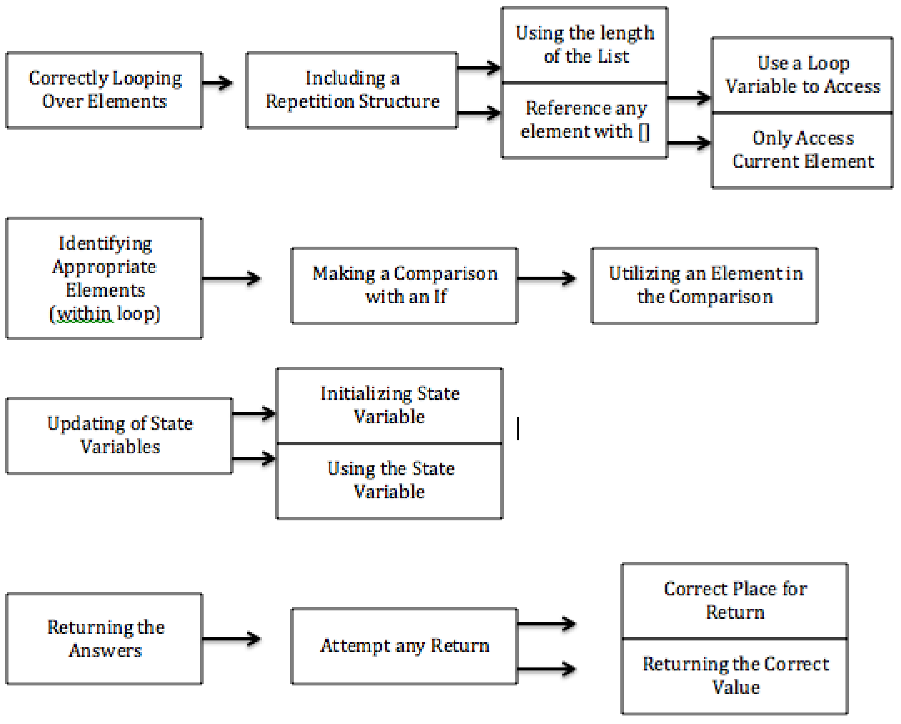

Correctly Looping Over Elements
Including a Repetition Structure
Using the length of the List
Reference any element with []
Use a Loop Variable to Access
Only Access Current Element
Identifying Appropriate Elements (within loop)
Making a Comparison with an If
Utilizing an Element in the Comparison
Updating of State Variables
Initializing State Variable
Using the State Variable
Returning the Answers
Attempt any Return
Correct Place for Return
Returning the Correct Value

Figure 3.2: A Model of Algorithmic Components

Correctly Looping Over Elements

Five of the algorithmic components are focused on looping over, or iterating through, the entire list of elements. In order to assess proficiency in this skill, I define the following components observable from the code. The components are:

- Including a Repetition Structure - Within each problem students are presented with an array of undefined size. In order to solve each problem correctly, students will need a repetition structure in their code.[1]
- Using the length of the List - The number of items in the array is not specified in the problem statement, therefore a correct solution should use the length property of the array to correctly access all the elements.
- Referencing Any Element From the List with [ ] - A looping structure's primary purpose in the problems chosen is to assist in the access of each element in the array. This model state checks for at least one element being accessed within the code.
- Using the Loop Variable to Access - Once we determine that students have attempted to access any element (component 3) we then need to make sure they can potentially access every element in the array. The use of the loop variable inside the [ ] ensures the potential to access every element.
- Only Accessing the Current Element - This is the only component where evidence results in a fail state. All of the algorithms can be implemented while only considering one element at a time. This state checks for an attempt to access multiple array values in one loop step.

The model states for looping over the array are common to all problems for this work, as well as generalizable to a larger set of problems common in introductory computer science.

Identifying Appropriate Elements

In three of the required algorithms (max search, count, and indexOf), the correct code requires a decision structure. The structure most often used for this type of decision is an if statement. For example, in the counting algorithm, participants need to use a decision structure in order to determine if the current array element is greater than or equal to 0 and less than the provided parameter value. There are three algorithmic components corresponding to this part of the algorithm.

- Making A Comparison with an If - Code receives credit for this component when it contains a decision structure inside the looping structure.
- Utilizing an Element in the Comparison - Code receives credit for this component if an element of the array is used for comparison inside the decision structure.
- Correctly Identifying with an If - Code receives credit for this component if it correctly identifies the appropriate elements as tested. This model state is confirmed with output-based testing.

[1]A recursive structure could also be used to solve the problems, however a recursive approach is not supported by the current system.

Updating of State Variables

Three of the required algorithms (sum, max search, and count) require the use of a state variable in order to implement the algorithm correctly. The state variable maintains a value, for example the sum of all the elements encountered so far, while the array is being processed. For the algorithms in this study, it also holds the return value when the loop has completed. There are three algorithmic components for the correct use of state variables.

- Initializing State Variable - Before the variable can be used, it needs to be declared and initialized before the looping structure in the algorithm.
- Using the State Variable - The state variable needs to be updated within the looping structure in order to maintain an appropriate state.
- Correctly Updating for Algorithm - Depending upon the algorithm, the state needs to be assigned a value (max search) or added to (sum, count). This model state is tested using output based testing.

Returning the Answers

The structure of the code writing problems presented to the participants require that they implement the solution as a method and return the resulting value produced by the algorithm. For example, when calculating the sum of all of the numbers in the array, the return value is the computed sum. There are three algorithmic components that correspond to returning the correct answer.

- Attempting Any Return - Code receives credit for this algorithmic component if there is any return statement within the implementation.
- Correctly Placing the Return - There must be a return statement after the execution of the loop for both algorithmic, and compilation reasons. Credit is received for this component if a return statement exists after the looping structure in the code.
- Returning the Correct Value - Credit is received for this component only if the code passed all of the output based testing specified for the problem.

3.4.3 Mapping to Code

Table 3.5 shows a student created solution, and the accompanying model components marked as present or absent for the code above.

3.4.4 Evaluating the Components

In the next chapter, I will detail a study to evaluate the model of algorithmic components proposed here for array algorithms. The model will be evaluated for appropriateness and completeness (i.e., does it cover the space of student answers?) and correctness (i.e., can it be used to accurately represent the types of errors that students make?). The study presented will focus on (1) the analysis of code in AbstractTutor using the described model, and (2) the use of abstraction by novices in the construction of solutions to array algorithm

```
public int findSum(int []myList)}
  int total = 0;
  for(int index = 1; index < 10; index++)
    total += 5;
  return total;
}
```

Including a Repetition Structure	Present
Using the Length of the List	Absent
Referencing Any Element From the List with [ ]	Absent
Using the Loop Variable to Access	Absent
Only Accessing the Current Element	Absent
Making a Comparison With An If	Absent
Utilizing an Element in the Comparison	Absent
Initializing State Variable	Present
Using the State Variable	Absent
Attempting Any Return	Present
Correct Place for Return	Present
Correct Return Value	Absent

Table 3.5: Code to Model Mapping

problems. In the study I demonstrate the appropriateness of the model, as it relates to common errors made by students. The validation of the described model is important for the computer science contribution of this thesis. Additionally, I demonstrate differences between high and low proficiency students and the increased frequency of transitions between statements of high and low abstraction during problem solving. The use of abstraction by novices is useful in supporting the learning sciences contribution, and supporting the use of abstractions in feedback messages to help improve student performance.

Chapter 4

Using Think Aloud Protocols to Understand Student Code Production

In Chapter 3, I specified the algorithmic components that are used as a model for analyzing student code and producing feedback for practice problems wherein students are asked to write simple array algorithms. Although the components were inspired by the Advanced Placement Computer Science rubrics, no similar models have been tested by having students interact with a computer system to determine their completeness and appropriateness in categorizing student coding attempts. In this chapter, I present the design and results of a proof of concept study focused on (1) the use of the proposed model to analyze student code in a computer system, and (2) the potential for abstract feedback to impact novice performance in AbstractTutor. The two focuses of this study tie into the main research questions of the thesis, whether (1) Can a pre-compilation feedback mechanism be constructed that operates with reasonable accuracy? and (2) Will pre-compilation feedback regarding algorithmic components produce better (a) within-problem performance and (b) across problem learning?

The proposed model was used to generate feedback messages based upon the presence or absence of algorithmic components in student code. In order for the feedback to be useful for the students, the model had to generate the messages at the appropriate time and represent common errors that students make in the problem solving process. In this chapter, I evaluate the model with novice students in order to answer 5 specific sub-research questions. The first four questions focus on the use of the model to analyze student code, and relate to the validity of the model for constructing accurate pre-compilation feedback. Two of these questions are focused on the appropriateness of the algorithmic components in the model as a basis for providing feedback to students. The other two research questions analyze student submissions using the model as a measure of proficiency and learning. The final question focuses on the abstractions students use in their verbalizations, correlating abstractions with expertise, and is offered as evidence that algorithmic feedback can impact student learning.

1. Do the algorithmic components each represent errors that a student is likely to make

while problem solving?

2. Do the algorithmic components together represent a model that covers the space of common algorithmic errors students make while problem solving?
3. Are the proposed algorithmic components useful for evaluating student progress within a single problem?
4. Are the proposed algorithmic components useful for evaluating learning across multiple problems?
5. Does the verbal expression of abstraction correlate with student proficiency?

In order to evaluate the research questions, a series of think alouds were conducted with students from Pittsburgh area universities. The following sections detail the subjects, the data collected, and the evidence for answers to the research questions.

Research Question	SubQuestion	Data	Section
Analyze Code	Representing Errors	Transcriptions	4.2
Analyze Code	Covering Common Errors	Code Submits	4.2
Analyze Code	Evaluating Progress	Code Submits	4.2
Impact Performance	Evaluating Learning	Code Submits	4.3
Impact Performance	Expressing Abstraction	Transcriptions	4.3

Table 4.1: Research Questions Mapped to Sections

4.1 Characteristics of Novice Students Thinking Aloud

In this section, I describe the characteristics of the participants, the structure of the think alouds, and the questions presented to the subjects, as well as the data collected by the system.

4.1.1 Participants

This study was conducted with 24 students who had either recently completed an introduction to programming course, or were in the final month of such a course. The four problems presented in the online system focus on simple array algorithms, a topic usually presented at the end of introductory programming classes. Although the system was online, students only interacted with a beta version of the system in the researcher's office as a part of this study.

Participants were recruited from Carnegie Mellon University and the University of Pittsburgh. The participants were solicited through an email forwarded from the instructor of the introductory course they had completed or in which they were enrolled. Participants did not receive any course credit for the study, but were compensated for their time with $20. Participants varied in the amount of time to solve the four problems. The average time of the think aloud including the directions was approximately 39 minutes (median time 36 minutes). Six participants completed the task in under 20 minutes, with the shortest time being approximately 8 minutes. Five participants took over an hour to complete the task, and the longest time was approximately 1 hour and 40 minutes.

The respondents were both graduate and undergraduate students and a mix of genders (M=18, F=6). One additional respondent was disqualified after completing the entrance survey wherein he indicated that he had completed several programming classes, so he is not counted in the above numbers. Six (6) of the participants took the introductory course using Processing - a special subset of the Java language, focused on art and visual design as well as computer programming fundamentals.

4.1.2 Self-Efficacy Characteristics

Upon entering the session, participants completed a survey with questions about demographics, prior coursework, expected (or received) grade in the introductory programming class and a self efficacy scale [66]. Of the 24 participants, 15 indicated they received or anticipated receiving a grade of an A or A- for the appropriate programming class. Nine (9) participants indicated they would receive a grade less than A, or were unsure of the grade since they were currently enrolled in the class. No participants indicated that they would or had received a failing grade for the class.

A self efficacy scale was used to collect information regarding students' self confidence in programming. The scale has been previously validated [66] to strongly correlate with student outcomes in an introductory computer science course. The self efficacy scale had 33 questions regarding a wide range of topics usually covered in an introductory course. It required students to select a number from 1 (absolutely able to complete this task) to 7 (unable to complete this task).[1]

Table 4.2 shows the results of the self efficacy scale for questions that explicitly reference skills relevant to the problems students solved as a part of this study. Students' responses to individual self efficacy questions were correlated with the total number of submits necessary to solve all problems, a rough measure of proficiency. The correlations were calculated using the number of submits the participant required to answer the questions correctly, which correlated with their self efficacy rating for the question stated in Table 4.2. A positive correlation indicates that the confidence of the student matched with the proficiency, as measured by the number of submits, during the think aloud. It is unsurprising that the first four statements correlate with the number of submissions. These statements directly address the tasks required to solve the problems presented in the think aloud.[2]

It is interesting, however, that there is essentially no significant correlation between the statements regarding correcting all errors, completing an assignment with built-in help, and overcoming problems. These are general skills employed by the participants when solving programming problems and may be too broad to analyze well in self reflection.

[1]For purposes of correlation analysis, the original responses were rescaled so 1 would indicate most confidence and a 7 would indicate least confidence.

[2]Although participants are not asked to complete a program to find an average, they are asked to compute a sum which is a part of the average algorithm.

Statement	Mean	Median	Correlation With Total Number of Submits (p)
I could write syntactically correct Java statements.	4.8	5	0.43 (.04)
I could write a Java program that computes the average of three numbers.	5.77	7	0.50 (.02)
I could write a Java program that computes the average of any given number of numbers.	5.32	6	0.58 (.004)
I could write a small Java program given a small problem that is familiar to me.	5.18	5.5	0.40 (.06)
I could debug (correct all the errors) a long and complex program that I had written and make it work.	4.14	4.5	-0.10(.6)
I could complete a programming project if I had just the built-in help facility for assistance.	3.68	4	-0.2 (.36)
While working on a programming project, if I got stuck at a point I could find ways of overcoming the problem.	4.68	5	0.006 (.97)

Table 4.2: Entrance Survey Responses (1= highest confidence/ 7 = low confidence)

The number of submits is a direct indication of the participants' ability to correct errors with the built in help. And yet, there is no significant correlation between efficacy statements regarding this skill and the actual number of submits the students used to answer the question. This finding may be a further indication that the feedback messages previously provided to novices are not appropriate to help novices either correctly finish a single problem, or provide the metacognitive prompts to help students learn across problems. Students had less self efficacy about their ability to debug, to use the built-in help in their development environments, and to find ways to overcome a problem when stuck. The lower ratings here could indicate that students understand that better pedagogical tools are needed in order to focus on learning goals and make appropriate progress during practice.

4.1.3 Procedures

After completing the survey, participants were introduced to the study and given a description of the task. They were asked to "think out loud and describe any code they were writing or choices they were considering". Participants interacted with a web-based environment on a desktop computer at the researcher's desk.

The environment shown in Figure 4.1 contains an editable text window, a submit button, and a feedback area. When starting a problem, the text window, or code area, contains a stub, or partial implementation of a program. In the stub, there is a line of code for the class and a line for the method header[3], providing participants with the name of any parameters [4] needed to solve the problem presented. The code stub also contained a comment[5] that described the problem to the student. The submit button was used to compile and test the code entered in the code window. The feedback area presented the appropriate feedback for the code the student submitted and would show below the Compile and Test message and button.

During the think aloud, the researcher sat in a chair slightly behind the participant. The participants' verbalizations were recorded on a digital recorder, and the online system recorded the code when the participant pressed submit.

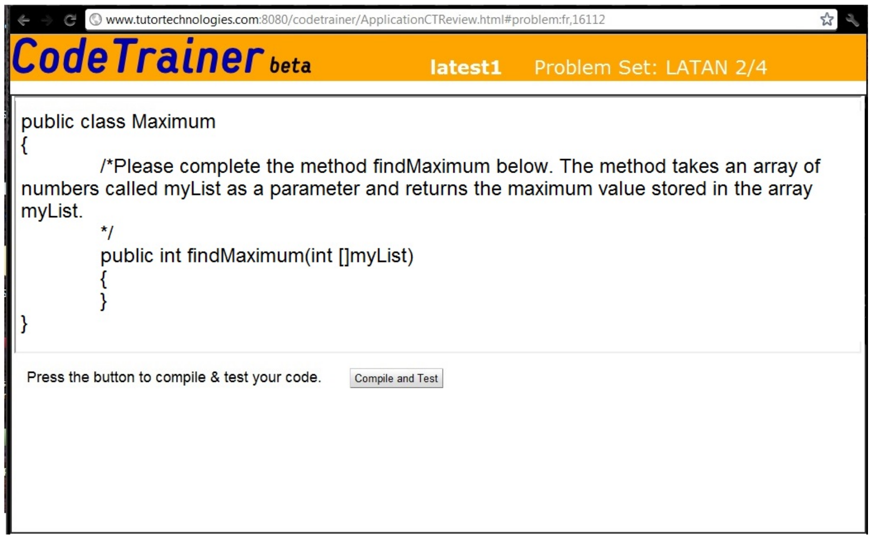
4.1.4 Problems

The four problems presented by the system were chosen to assess participants' ability to write code in the Java programming language that implemented a common array algorithm. Although half of the participants saw a contextualized problem description, all participants were required to complete the same four algorithms in the same order. The four algorithms participants were asked to implement were: sum, max search, count, and

[3]The class and method header provide initial lines of code setting up the space for the student to implement the algorithm.

[4]A parameter is a variable whose values are defined in another part of the program. The values are passed to the method, and code within the method can act upon those values through the parameter name.

[5]A comment is text contained within the code that the system does not try to compile or execute.


www.tutortechnologies.com:8080/codetrainer/ApplicationCTReview.html#problem:fr,16112

CodeTrainer beta   latest1   Problem Set: LATAN 2/4

```
public class Maximum
{
	/*Please complete the method findMaximum below. The method takes an array of
numbers called myList as a parameter and returns the maximum value stored in the array
myList.
	*/
	public int findMaximum(int []myList)
	{
	}
}
```

Press the button to compile & test your code.

Figure 4.1: The Tutoring Screen Used By Participants

index of. The exact problem stubs as well as contextual descriptions are available with solutions in Appendix A.

4.1.5 Research Conditions

Because the study was conducted while the system was still in development with the purpose of assessing the appropriateness of the algorithmic model for evaluating any student answer, rather than testing the between condition effects, participants were not assigned to conditions randomly. Participants were assigned to one of four research conditions based on the order they responded to the study advertisement in order to assess the parts of the system as they were being constructed. The four research conditions represented a combination of two factors of the student's experience in the study. Half of the participants received only output-based feedback, and half of the participants received feedback regarding the algorithmic components necessary for completing the assignment, based upon the model described in Chapter 3, in addition to the output based feedback. The output-based feedback messages used were similar to messages produced by other practice systems such as CodingBat or myProgrammingLab. The feedback was produced by executing the student code with particular input and comparing the output to a correct answer. An example of output based feedback for this study would be "When your code was executed with the values [1,2,3,4,5] it returned a value of 10 when 15 was expected."

The second factor impacting the student experience was the use of a contextualized problem description. Students without a contextualized problem description had the parameter described as an "array of numbers" and were asked to write a specific algorithm such as "find the sum of all the numbers in the array." Students with a contextualized problem description had the array described in terms of a context for the numeric values. For example, instead of asking students to find the sum of all the numbers in the array, the contextual description told students the array held the price of items on a sales receipt and asked the students to find the total bill.

The first 6 participants completed the problems without contextual problem descriptions and with output-based feedback only. The next 7 participants completed the problems without contextual problem descriptions and with algorithmic component feedback[6]. The algorithmic component feedback was read/shown to the participants after they pressed the submit button if the code submitted should have generated the feedback[7]. Feedback was read/shown to the participants as data collected from the submits in the think aloud was used to refine the automatic feedback mechanism that was used in subsequent studies.

Participants 14-19 completed the problems with a contextual problem description, and with output-based feedback only. The final 5 participants completed the problems with a contextual condition and received algorithmic component feedback presented to them on small slips of paper with only one message at a time. After completing the four problems, participants were thanked for their time and compensated.

4.1.6 Data

Data for this study were collected in two ways. First, the online system recorded the participants' code every time they pressed submit. The code was hand scored based on the constraints the AbstractTutor system will use for automatic analysis.[8] During the think alouds, participants entered 482 submissions across the four problems. Table 4.3 details the number of submissions by problem for all participants (ALL), as well as the total number of submissions made by participants in each condition (MO - Mathematical Problems, Output Based Feedback, MA - Mathematical Problems, Algorithmic Feedback, CO - Contextual Problems, Output Based Feedback, CA - Contextual Problems, Algorithmic Feedback).

Problem	ALL (N=24)	MO (N=6)	MA (N=7)	CO (N=6)	CA (N=5)
1	117	22	21	39	35
2	175	91	20	26	38
3	106	50	15	28	13
4	84	35	18	16	15
Total	482	198	74	109	101

Table 4.3: Code Submissions by Problem

Second, while the participants were completing the problems, they were asked to think out loud, describing the code they were writing and the choices they were making. The

[6]An extra participant was included in the non-contextual/algorithmic feedback condition as the contextual problem statements were being coded into the system.

[7]Feedback was produced based on the constraints listed in chapter 3.

[8]Although the automatic scoring of model states is a part of the larger thesis work, hand scoring was used at this time for purposes of accuracy. Data from the think alouds were used to refine the automatic scoring mechanism. At the time of this study, students would not have received appropriate error messages with high frequency. Chapter 7 - discusses the automated scoring results, but all results presented in this chapter rely on the hand scoring.

audio from the session was recorded, and then later transcribed for coding purposes. The transcriptions were then separated into utterances (segments of speech) and coded for abstractions as detailed in Section 4.1.8. Additionally, exemplar statements were identified for use in evaluating the algorithmic components in section 4.2.

4.1.7 Initial Student Attempts

Although no formal pretest was administered to participants, the accuracy of the first submit to the system, before any feedback was received, can be considered a measure of students' ability at the start of the problem. The first problem that students encountered was to find the sum of the numbers in an array. When done correctly, a solution to the problem would contain 11 of the algorithmic components described in Chapter 3 in the code[9]. Table 4.4 shows the number of subjects per condition, the average number of algorithmic components that were correct on first submit (FS Average), and the percent of algorithmic components that were correct on first submit. An ANOVA showed no significant difference in the number of algorithmic elements in the first submits of subjects when compared by Feedback condition (p=0.76). The ANOVA did show a significant (p<0.01) factor of Contextual condition; however because subjects were not randomized to condition, we cannot determine if this difference was from the contextual problem description or participant differences. The further exploration of this difference is recommended for future work.

Condition	Number of Subjects	FS Average	FS Percent
MO	6	8.3	75.8
MA	7	9.9	89.6
CO	6	7.2	65.2
CA	5	4.6	41.8
Total	24	7.7	70.1

Table 4.4: First Submits by Condition

4.1.8 Abstraction Coding

The think aloud statements students made while completing the programming tasks were recorded and transcribed. Due to the lack of a formal "answer", the statements made by the students often do not make complete sentences or thoughts, and they are not easy to separate. In order to do data analysis, however, the text needed to be divided into individual statements for coding.

Student statements were divided into individual data points by looking for either a pause in the speech, a clear step to a new idea or topic, or a change in the problem or sub task the student was coding. Overall, 2,950 individual statements were identified from

[9]Calculating the sum of the numbers does not require the selection of individual elements because you use all elements in the sum.

Category	Label	Description	Example
No Abstraction	NA	These statements refer specifically to the code. They do not include statements that generalize.	“int count =0”, “We want it to be greater than or equal to 0”, “That is myList dot length”
Block Labeling	BL	These statements refer to a block of code with a label as opposed to the character being typed.	“I need to define an int parameter named sum”, “So if both of these are true, then we add one to the counter”
Macro Structure or Relations	MR	These statements connect elements or relate an element to its purpose in the algorithm.	“So after we are doing with the for loop we want to return the counter”, “I think I may need to use bubble sort to sort the..”
Feedback Response	FR	These statements refer to the feedback messages or the specific debugging elements in the problem solving process.	“Um, I don’t know what that means”, “So it was looking for a 12 but returned..”, “Ok, forgot a semicolon”
Not Code Specific	NC	These statements are focused on the student’s feelings or progress, not on the code.	“Ok, I want to try it”, “I think that’s right”

Table 4.5: Categorization of Student Statements

the transcriptions. Table 4.6 shows the number of utterances made by participants in each condition per problem.

Problem	ALL (N=24)	MO (N=6)	MA (N=7)	CO (N=6)	CA (N=5)
1	861	295	120	212	234
2	969	479	124	101	265
3	626	279	108	126	113
4	494	245	57	101	91
Total	2950	1298	409	540	703

Table 4.6: Utterances by Condition and Problem

Each statement was classified based on the categorization shown in Table 4.5. The classification scheme is a modification of the SOLO taxonomy designed specifically for this data set.

The purpose of the classification was to determine the relative number of abstract statements made by students, not the degree of abstraction in the entire problem and therefore the original SOLO Taxonomy was modified to the classification scheme presented in Table 4.5 and described below. The classification is helpful in indicating if the student was thinking only about the individual code elements and syntax, or if attempting to connect the larger program goals to the code being written. Also, previous work in self explanation [63] indicates that connections to the abstract ideas or larger goals of the problem are important, and the coding scheme is designed to specifically highlight the percentage of time students spend connecting the code statements they are writing to these abstract ideas.

The first category, NA (No Abstraction), was used to classify statements that involved no abstraction beyond an individual line of code. This aligned with the SOLO categories Unistructural and Multistructural. The NA Category also aligns with the first column of the Block Model. The difference between the two categories in SOLO represents the student understanding of the problem being asked or expressing some misconceptions. The statements made in this study are difficult to judge for accuracy, and the accuracy is less important than the level of abstraction for the analysis presented here. Due to those factors, the two SOLO categories are collapsed into one.

The second category, BL (Block Labeling), was used to classify statements that move beyond the simple reiteration of the code presented. Students who engage in Block Labeling often connected pieces of code in their statements as shown in the examples in Table 4.5. The BL classification aligns closely with a part of the Relational classification for SOLO. The BL classification also aligns with the second column of the Block Model, where the structure of the program is explicit.

The MR (Macro Relational) category was used for statements that demonstrated a connection to a macro relationship between elements of the code and the abstract nature of the problem. Statements about the problem itself, without any reference to code were also classified as MR; this type often emerged at the beginning of each section as the student read the problem aloud. The MR classification encompasses the Extended Abstract SOLO response, however some statements could be classified as relational under SOLO if there is no larger connection to another problem. The MR classification aligns with the third column of the Block Model, where students call out not only the code, and its relationship to the code structure, but also the function that it serves in the algorithm.

The category labeled FR is used to classify statements made by the participant directly referencing the feedback the student was shown. The statements are separate from the MR, BL, and NA category in that the student may have simply read the feedback message on the screen or made a statement about the code that was directly prompted by the feedback statement.

The final category, NC, is used to classify statements that are not related to the problem, or its solution. Although SOLO has a Prestructural category that is not represented in the coding used here, the NC category does not require the expression of a misconception, only a statement not directly related to the code of the problem.

Although SOLO is helpful to connect this work to prior work of the Computer Science Education community, the granularity of analysis is not the same between this work and prior uses of SOLO. The SOLO taxonomy applied to novice programmers as described in [49] was used to categorize an entire answer, or description of an entire problem by a student. In this work, we are categorizing individual statements during a problem solving task. The individual statements are of a much finer granularity, and therefore I used a coding scheme based on SOLO without replicating it exactly. For that reason, I consider the modification of the coding scheme and the evaluation of student statements at a different granularity a contribution of this thesis.

In order to verify the classification accuracy of the primary author, an interrater reliability analysis was performed to assess the assignment of categories to student statements. A second expert rater was trained in the classification scheme and used the categories to rate 40 random statements from the data set. Kappa was computed for agreement [17]. The ratings produced by the primary author and second expert generated a Cohen's kappa of 0.601, a good agreement [37]. After evaluation, the raters discussed the differences and all differences but 1 were mutually agreed to realign with the primary author's answers.

4.2 Analyzing Student Code: Appropriateness of Algorithmic Components

With this qualitative analysis, I seek to inform the primary research question "Can a pre-compilation feedback mechanism be constructed that operates with reasonable accuracy?" Specifically, in this section I focus on the alignment of the statements students make and the code they submit with the model of algorithmic components described in Chapter 3. This section focuses on the sub-questions of: (1) Do the algorithmic components each represent a model that can be used to evaluate errors a student is likely to make while problem solving? and; (2) Do the algorithmic components together represent a model that covers the space of common algorithmic errors students make while problem solving?; and

(3) Are the proposed algorithmic components useful for evaluating learning within a single problem?

In order to answer that question, the audio recordings of the programming think alouds were transcribed and coded for the verbalization of each algorithmic component. In this section, I present the number of subjects that expressed an algorithmic component, either prior to their first submission to the system, or in response to feedback regarding an incorrect answer. Additionally, I present examples of each algorithmic component from the audio recorded. Finally, I discuss aggregate data evaluating student progress within problems.

4.2.1 Correctly Looping Over Elements

Including a Repetition Structure: All but one participant (23) recognized immediately the need for a looping structure in order to access all of the elements of the array. Participants verbalized the need for a for loop as a part of their problem solving process. Some examples of participant statements include "I think I have to use a for loop to add elements of array mylist" (A3) and "So I would want to use a for loop that goes through the equal number of times as there are entries in the array."(A7). The one student who did not use a for loop in her code expressed that she needed an "if loop" in order to accomplish her task. Anecdotally, in the experience of the author, this is a common misconception among novice students, who mistake a decision structure with a looping structure.

Using The length Property Of The List: The language used to express the need to use the length of the list as a termination condition for the looping structure varied based upon the level of abstraction used by the participant in describing the loop bounds. Participants who expressed a high level of abstraction chunked the details of the for loop into a single statement while typing the correct code. An example of this is found in the statement: "I know we need to loop over all the elements in the array so I will make the for loop."(A4). Other participants were more concrete with their verbalizations and used language that explicitly listed the components of the loop they were typing, for example "for i=0, oh variable i, int i, for i = 0 i <mylist, capital L, mylist length and i++"(A6). In the second example, the student included each part of the loop, verbalizing even the capital letter 'L' from the parameter myList. Often the weaker participants, as measured by accuracy of the first program submission (prior to feedback), would express the code in more detail. This finding is consistent with prior work regarding code comprehension activities [73] where students who were able to "chunk" code into the more abstract algorithms when reading code, performed better on code production tasks.

Referencing Any Element From The List With [ ]: Similar to the use of length to describe the bounds of the loop, we see varying levels of abstraction as participants expressed the need to reference an element from the list with the characters [ ]. Participants used the word bracket, or the computer science term "sub", to indicate the access of an element from the list. For example, subject A11 used the language "so.. the sum will equal each of the items sub I" to indicate the access of each element, and this particular subject also used a similarly low level of abstraction when talking about the for loop, explicitly stating the components of the for loop. Other subjects were much more abstract in their language, including the elements of the array as a singular concept such as "and for each iteration of the for loop, I would just want to add the current value in the array to some sort of int sum, start at 0."(A7). In the example "the current value in the array" does not even use the name of the variable to be accessed, instead abstracting the statement to a more general statement about the array.

Using The Loop Variable To Access: The use of a loop variable to access the array in student statements followed the same pattern as the previous algorithmic component. If the participant used an abstract statement to reference the "current value in the array", that language included both the variable and the array access in a single statement. Some participants did struggle at first indicating that they needed to use some value to access an individual element (thereby satisfying the prior algorithmic component) but not using the loop control variable. Here is an example of a student in the output based feedback condition with problem context, working through a mistake where they used a 1 in the [ ] instead of their loop control variable i.

" *so we'll go if vehicleMPG 1 is greater than max um*
then max is vehiclempg i
then we will return that
submission: compile error
semicolons
hm..
line 9
uh oh
submit it again
Submission: compile error
less than
let's see
Submission: Incorrect Output
oh. so we have to do i, just a trip up on vehicleMPG"(A9)

At the beginning of the quote, the participant's construction of the first attempt, as well as the reaction to the feedback provided by the system. The participant used a 1 in the first line of the passage (inserting the 1 in the if statement), however used i in the assignment statement on line 2. Because of the paradigm of output-based feedback, this participant needed to fix two compiler errors in order to get feedback, in the form of incorrect output, that helped him fix the main error with his code. In the algorithmic feedback condition, the participant would have seen immediate feedback surrounding the use of the 1 instead of i.

Only Accessing The Current Element: The algorithmic component of only accessing the current element is different from all other components in the entire model in that a participant who produces the correct solution will not use any language or explanation referring to NOT accessing neighboring elements. The "current" or "element at i" will be used, to compare to the state variable, the "max so far". The following example is a participant who struggled with this concept.

" *"when this one is bigger, is less than, next one then the next one is ..*
I don't know whether there is a maximum function to compare the two variables so I just write like this
so I think if I take .. if max is equals to the bigger one..
and if bigger is equal then I will take the bigger one.
So let me see
first I count 0, when 0 is less than -1 then I take -1 or I take my mylist 0 and when the next come -1 hm..
so I need to take this n, I need to make a comparison with n.
so this is n
this is n
and if n .. if n less than mylist i+1 then it goes to mylist.." (A2)

The participant used the phrase "mylist i + 1" in order to compare with "the next" element. Participants were most likely to make this mistake when completing the second problem, requiring them to find the largest number in a list. The second problem was the only problem involving a comparison between multiple elements of the list. Five participants (21%) made at least one submission (mean = 17, median = 14) containing code with a reference to an element that was not the current element[10]. Students making this error often verbalized the algorithm as checking an item against the "next" item or an item's "neighbor".

4.2.2 Selecting Appropriate Elements

Making A Comparison With An If: In three of the problems students were asked to solve, they were required to use a decision structure or if statement in order to identify appropriate array elements for use in the algorithm. A variety of terms were used by participants in both conditions to describe the selection code including "use a conditional", "compare each one", "if statement", and "check that it's less than the passing grade". Some participants were abstract in their language and used an English "if" instead of stating their code. One example from a participant solving problem 2 (max search) is "In this case we have to use a conditional to test if the element we are looking at in the current moment in the for loop is greater than the minimum."(A4)

Utilizing An Element In The Comparison: The purpose of the decision structure within the algorithm is to identify the elements of the array that help answer the problem posed to the student. For example, problem 3 asked the student to count the number of values between 0 and a parameter. The comparison within the decision structure is used to check to see if the current element is between the two values. Examples can be found

[10]Note: The use of a constant, such as in the example for Using the Loop Variable to Access would not be counted as referencing a non-current element. The non-current element is only triggered by the use of + or - in a mathematical expression to calculate another element's location.

of participants who expressed the algorithmic component abstractly, as in the previous paragraph, as well as more explicit examples about the code they were writing "If myList of x is less than value and myList of x is greater than .. lets say negative 1"(A5).

Correctly Identifying With An If: Identifying the correct element is also a part of the same statements used to express the need for an if statement, as well as using an element in the comparison. In the example at the end of the last paragraph, the participant correctly identified an element greater than or equal to 0 and less than the appropriate parameter value. Participants sometimes missed edge cases, such as not including the 0. The output based testing helped participants understand if they had made an error in their logic, just as it will be used to assess the program code in the system.

4.2.3 Updating of State Variables

Initializing State Variable: An algorithmic component specifically assessing the initialization of the state variable was added to the model in response to a number of participants who either submitted code without the component, or who had to go back during the think aloud to initialize such a variable. The observation leading to the addition of the component was made during the first set of think alouds when participants were receiving output-based feedback only, therefore the addition should not impact the outcome of the analysis as its inclusion would not have impacted the think-alouds before its addition. Overall, 15 participants (63%) had at least one submission where the code was missing the initialization of the state variable.

Over all the participants, six (6) participants had two or fewer submissions where the initialization was missing, the other 9 participants had as few as 3 submissions and as many as 52 submissions (mean = 12.1, median = 6) across all 4 problems. For example, the following is an excerpt from a participant who had to go back to define the state variable after he had already typed his for loop. During that problem, the subject declared the loop first, and had to go back, before the loop code, in order to declare an "int to store the sum".

"Ok, so going to make a for loop having i equal the length of the array minus 1, because that would be how many items are on the sales receipt. I'm going to have that go while it is less than the length so that it is only that number, and increment it by one every time, and I'm going to need an int to store the sum. So.. the sum will equal each of the items sub i."(A11)

Using the State Variable: The problems in this study that need the use of state variables (sum, max search, and count) require the state variable to be updated within the looping structure. An example of a participant vocalizing exactly what he was typing is "sum plus equals myList access the element"(A5). An example of a participant refering to the state variable not by the name he gave it in the code, but instead by its role in the algorithm expressing an abstraction is, "So if both those are true, then we add one to the counter"(A3).

Correctly Updating For Algorithm: Many participants had the correct update for the algorithm if they attempted to update the state variable at all in their code. One participant had an incorrect update that was caught by the output based feedback from the problem. The following quote occurs right before the participant submitted the program to the system, and his reaction to the feedback from the output based testing. “So, if it satisfies the two things, total plus equals that element, and then after it’s done we will just return whatever the total is.” At this point the participant engages in a debugging session. First he checks his code and find nothing visibly wrong, and then he rereads the question and respond ”I see it now, I thought it meant return the sum of all those numbers, it just wanted the total number of those numbers. I see the difference”(A8).

4.2.4 Returning the Answers

Attempting Any Return: The methods the students are asked to write in these activities require the code to return a value that is the result of their algorithms. The return value is designated in the code with an explicit statement, and participants used the word return to indicate this action in their think aloud sessions. There are a multitude of examples of this, such as, “it’s going to return the total”,“just return sum”, “return -1 outside of it”, and “return max value”.

Correct Place For Return: The return statement will end the method, causing any other code to not be executed. It is important to put the statement in the correct place for the algorithm being implemented. In three of the algorithms the participants are asked to implement, the return statement belongs at the end of the code, while in the last algorithm a return statement can be placed inside of the loop once the correct element is identified. Fourteen participants who included the return statement always put it in the right place. One participant verbalized his decision making process by saying

> *“If room numbers in the index position is equal to .. and this is the double equal sign because you are checking equality .. to idno which is the identification variable which is passed into the method. If that is equal to .. then we can go ahead, go ahead and return, um, negative.. can I do that? No, that’s wrong, you want to return i because that’s the index position, I was about to return negative 1 which is what I do if I don’t find it. So outside the for loop I can return negative 1.”(A10)*

Of the other 10 participants, when a return statement was included there was, on average, 3.2 submissions that did not have the return statement in the correct place (median=3).

Correct Return Value: Similar to the correct update for the algorithm, the correct return value is also checked with the output based testing. For example one participant had an incorrect loop bound and received feedback that his maximum search algorithm was incorrect. “So what I did there, while I was adjusting the parameters in my for loop, I made length minus 1 which I didn’t need to do.”(A2) No participant solved every problem correctly on the first try, so every participant had at least one submission without a recorded correct return value.

4.2.5 Algorithmic Components are Expressed

As demonstrated by these data and examples, the algorithmic components proposed in the model presented in Chapter 3 can be found in student statements while they are thinking aloud. During the analysis of the student statements, it became clear that students used various levels of abstraction within a given problem and across the algorithmic components. A student may have been very abstract, not using a literal reading of the code, with the initialization and update of the state variable, as initialization and mathematical calculations with variables are often taught quite early in the course allowing for significant practice before the end, while being more literal in his verbalizations of the loop or array access. Looping and arrays often fall later in the course, and therefore students have had less time to master those statements.

Overall, there is evidence from the student statements that the algorithmic components are a part of the thought processes used by novices during problem solving. As a part of student thought processes, feedback regarding these components should be beneficial to the novice. As a next step, I validate the completeness and correctness of the set of components to ensure the model has the components necessary to address student errors and misconceptions.

4.2.6 Model Appropriateness and Correctness

In order for the model of algorithmic components to be considered appropriate and correct for mapping student solutions of the problem, there should be evidence that the students (1) can submit code lacking each of the components and (2) demonstrate progress through the various components through successive code submissions.

Table 4.7 shows the number of times each component appeared correctly in a student submission for each problem. The table includes the data for an entire condition, as well as the values per problem. Overall there were 482 submissions for the 24 users while solving 4 problems. The data from the table are used to generate the graphs and in the discussion of each algorithmic component in the following sections.

Correctly Looping Over Elements

All of the algorithmic components for correctly looping over the elements of an array meet the two criteria for evidence above. Figure 4.2 shows the percent of submissions where an algorithmic component was missing from the code submitted. The darker bar indicates the percentage across all submissions while the lighter bar indicates the percentage across first attempts (FA Percent) or the first submission made by each participant for each problem. The state for the inclusion of a looping structure (loop-for) is the component most likely to appear in student's code, with only one(1) participant excluding a loop from the submission.

	Looping Sec. 4.2.1					Selecting Sec. 4.2.2			State Var Sec. 4.2.3			Return Sec. 4.2.4		
	Loop For	Loop Length	Loop List	Loop Ref.	Loop Cur.	Identify Comp.	Iden. Use	Iden. Correct	State Init	State Use	State Correct	Rtn Any	Rtn Place	Rtn Val.
All(N=482)	480	452	450	438	395	353	342	238	281	367	328	422	390	101
1	115	105	92	84	114	0	0	0	91	101	89	82	78	25
2	175	163	169	168	103	168	160	81	85	160	136	162	158	24
3	106	100	106	106	101	102	102	78	105	106	103	98	93	25
4	84	84	83	80	77	83	80	79	0	0	0	80	61	27
MO(N=198)	198	196	196	196	113	170	170	97	106	161	140	186	167	29
1	22	21	22	22	20	0	0	0	17	22	21	16	15	6
2	91	91	89	89	20	89	89	21	39	89	69	86	85	6
3	50	49	50	50	45	46	46	42	50	50	50	49	46	8
4	35	35	35	35	28	35	35	34	0	0	0	35	21	9
MA(N=74)	74	74	74	70	74	52	52	52	44	56	55	73	71	30
1	21	21	21	17	21	0	0	0	16	21	21	21	21	8
2	20	20	20	20	20	19	19	19	13	20	20	20	20	7
3	15	15	15	15	15	15	15	15	15	15	14	14	12	7
4	18	18	18	18	18	18	18	18	0	0	0	18	18	8
CO(N=109)	108	99	88	85	108	65	58	37	77	80	67	82	79	24
1	38	36	21	21	38	0	0	0	35	35	25	26	26	6
2	26	19	24	24	26	22	18	17	15	17	16	19	18	6
3	28	28	28	28	28	28	28	8	27	28	26	24	24	6
4	16	16	15	12	16	15	12	12	0	0	0	13	11	6
CA(N=101)	100	83	92	87	100	66	62	52	54	70	66	81	73	18
1	34	27	28	24	35	0	0	0	23	23	22	19	16	5
2	38	33	36	35	37	38	34	24	18	34	31	37	35	5
3	13	8	13	13	13	13	13	13	13	13	13	11	11	4
4	15	15	15	15	15	15	15	15	0	0	0	14	11	4

Table 4.7: Algorithmic Components by Condition (N indicates total submissions)

The algorithmic components for only accessing the current element (loop-current) illustrates an interesting phenomenon. The graph shows that there is a higher percentage of observations where the participant incorrectly attempted to access multiple elements of the array in all observations (18%) compared to first attempts (6%). There are two factors that contribute to this phenomenon for the loop-current component. First, the loop-current component is a fail state, meaning that it is assumed to be correct unless there is evidence, in the form of incorrect code, present in the submission. This means that a participant who did not use [] to access an element of the array could not have an incorrect loop-current component assigned.

Secondly, some participants introduced the access of multiple elements as they were attempting to correct other problems with their code. This indicates that students were actually more accurate on their first attempt and then introduced errors in response to feedback provided during the problem solving process. This phenomenon happened for 3 students in the output based feedback condition, and one student in the algorithmic feedback condition. The output based feedback tells the student that the code is not correct, but no further information about why the code is not correct. The student will then often resort to complicated reasoning or shotgun debugging (guess and change things) in order to try to get the feedback to indicate the code is correct. Unfortunately, because most output based feedback mechanisms are unable to measure whether a change makes you closer to a correct solution or not, the student is just left with the idea that it is "wrong". The one student in the algorithmic feedback condition revised the mistake after one submit (with appropriate algorithmic feedback) while the three students in the output based feedback condition took 46, 6, and 13 submits each to correct the problem.

An extreme case of this would be a subject in the output based/mathematical context condition in the think aloud study. She was originally correct in her access, using myList[i] in her code; but after receiving the output based feedback, she modified it to myList[i+1] in order to compare neighboring items to find the max (a common misconception and the reason for the accessing the current element model state).[11] She has three correct submissions (including her first) of this model state, and 40 incorrect submissions.

Selecting Appropriate Elements

The percentage of missing algorithmic components for selecting the appropriate elements from the array were among the most observed components in the data. Figure 4.3 shows the states as identify-compare, identify-use, and identify-correct. The identify-correct component is judged by output based testing and therefore requires the code to compile before it can be marked as correct. Also, similar to the loop-current component, there are more instances of an incorrect across all submissions (34%) than in first attempts (20%). In addition to participants modifying code to render it incorrect with regards to the model, there are also a number of incorrect submissions resulting from compiler errors introduced as a part of the debugging process.

[11]Her method was storing the index and comparing the index against the array values, instead of using the max index to access the appropriate element in the array.

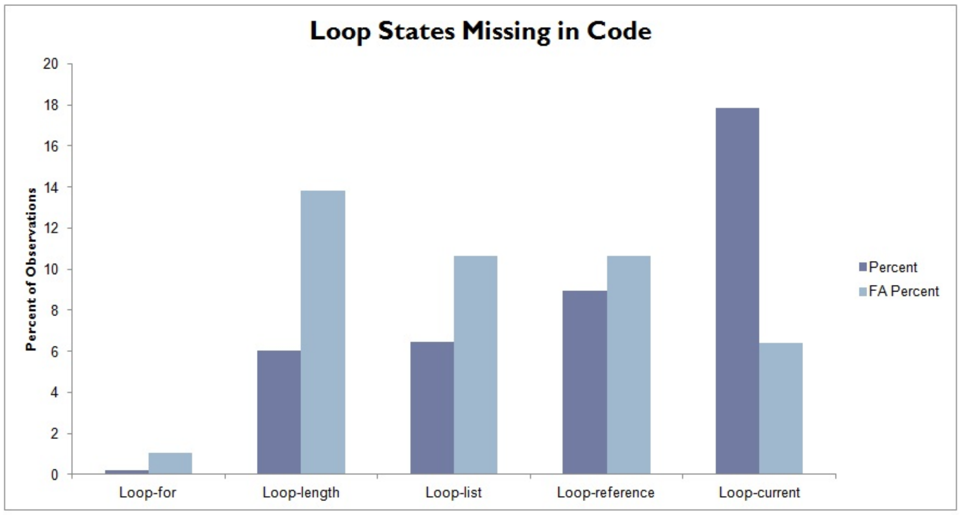


Figure 4.2: Looping Over An Array, Percentage of Missing Observations

Using the State Variable

The percentages of missing algorithmic components for the declaration and use of state variables are also shown in Figure 4.3. The large number of incorrect declaration and initialization of the state variable (state-init) prompted the addition of an algorithmic component particularly focused on this concept. The component was added to the model after the observations were made during the first 13 participants, who received output based feedback only. In the think aloud sessions where participants received algorithmic feedback, the state-init component was used as a part of the model, and appropriate algorithmic feedback was presented when appropriate. The figure shows the missing percentage of submissions over all think aloud participants.

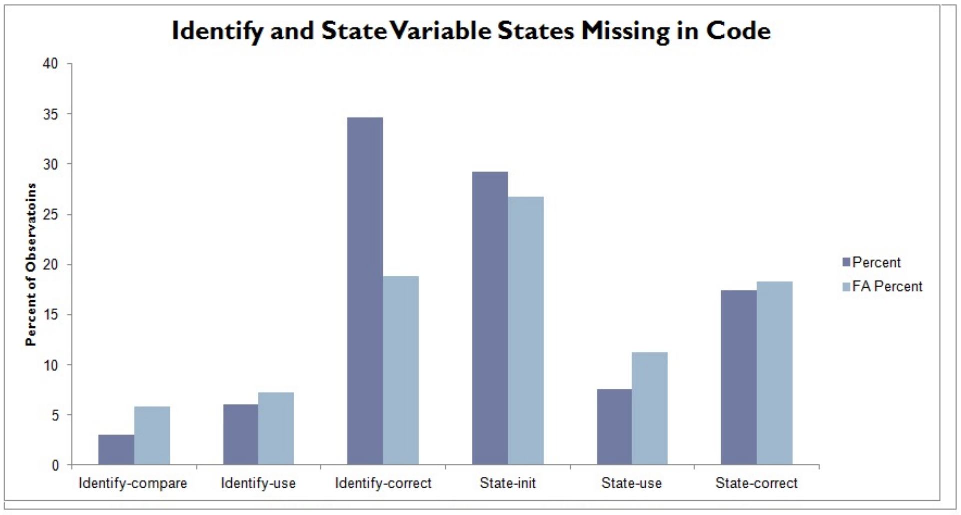


Figure 4.3: Identify and State Variable Counts, Percentage of Missing Observations

Returning the Answers

The percentages of missing algorithmic components for returning the appropriate answer, as well as the percentage of code that did not compile, are in Figure 4.4. Similar to the identify-correct and state-correct components, the components for returning the correct value (return-value) are also checked with output based testing and therefore can only be checked with code that compiles. Once again the data show that participants can both (1) submit code that does not contain each of the indicated components for returning the answers, and (2) make changes to their code that includes those components over time.

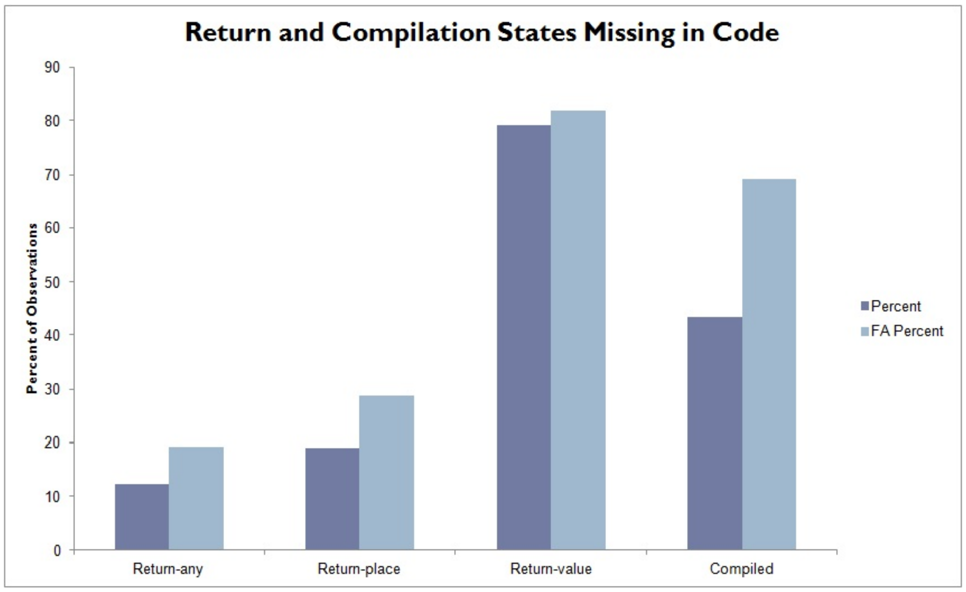


Figure 4.4: Return and Compilation States Counts, Percentage of Missing Observations

Model Evaluation

In section 4.2 I showed the percentage of missing observations of model components in student code in order to answer the question: "Do the algorithmic components each represent a model that can be used to evaluate errors a student is likely to make while problem solving?". For each model state, at least one student submitted at least two attempts missing each of the model states. The student submissions demonstrate that the model can be used to represent student programs and evaluate the errors a student is likely to make. Additionally, when taken together, the components represent a model that covers the space of the common algorithmic errors students make while problem solving.

4.3 Impacting Novice Performance: Evaluating Students

In this Section I discuss evidence that AbstractTutor has the potential to produce student learning, and can measure the learning using the algorithmic components model. This section focuses on the two sub-questions of (1) Are the proposed algorithmic components useful for evaluating learning across multiple problems, and (2) Does the verbal expression of abstraction correlate with student proficiency?

4.3.1 Using Algorithmic Components to Evaluate Student Progress Within Problems

The construction of a correct algorithm in computer code requires a student to combine the required skills of writing correct syntax, as well as apply the knowledge of algorithmic components in order to produce compilable and correct code. Often the difficulties involved in constructing accurate program code result in shotgun debugging, a process by which random changes are made in the hopes of producing desired output. This behavior is similar to guessing strategies shown in other technological-supported systems where inputs are guessed in order to get to help or feedback that will allow the student to progress through the system [7]. Within this study, participants also made attempts to submit code that did not result in progression towards a correct solution. Overall, I observed 387 edits that were submitted to the system. Over half of the submissions made by students did not result in a significant change or improvement to the program the students were working on. Of the 387 edits, only 183 (47%) resulted in a change to the algorithmic components of the model used as a rubric for scoring attempts. Of those 183 edits, 13 (7%) resulted in a submission with fewer observed algorithmic components, i.e., a less correct solution attempt.

These data imply that a simple count of the number of submissions would grossly exaggerate either the fixing of compiler errors or the use of guessing as progress in completing the program. The use of targeted feedback and its evaluation therefore requires a finer grained measurement of student performance. The algorithmic components offer a way to determine if a submission is a productive edit - moving the student closer to a solution, or an unproductive edit[12]. Unproductive edits may still be thoughtful, or upon closer examination reveal a knowledgeable attempt on the part of the student. Although human evaluation of program code is the most accurate, it is a lengthy and costly process. The use of the algorithmic component model in AbstractTutor is an intermediate step between the use of inaccurate and overly generalized metrics of student proficiency, such as number of submissions, and human evaluation. As described in the following sections, the model is useful for disaggregating the individual concepts with which students struggle and, as de-

[12]An unproductive edit here is defined as an edit that does not correct an incorrect algorithmic component of the code. I recognize that fixing compiler errors is also productive, but if the remainder of the code is correct, the fixing of compiler errors will result in the output-based algorithmic elements to be marked as correct.

scribed in Chapter 6, is useful for tracking students across multiple submissions to evaluate within problem progress.

4.3.2 Evaluating Across Problem Gains

The first attempt at a problem by a participant can be seen as an indication of the knowledge the participant had at the time of problem solving. By evaluating the algorithmic components marked as correct at the first submissions for each problem, we can observe an estimate[13] of the algorithmic elements the participants included in their solutions. When we compare the first attempts across all four problems, we can see evidence of learning gains. For a simple initial measure, I counted the number of missing algorithmic elements for each participant and calculated the average for each problem. Problems 1 and 2 had relatively the same number of missing elements with averages of 3.85 out of 10 elements (38.5%) and 4 out of 13 elements (30.8%), respectively. Problem 3 showed a significant decrease with an average of 2.22 out of 13 elements (16.9%), and problem 4 also was improved with an average of 1.96 out of 11 elements (17.8%).($p<.01$)

4.3.3 Correlating Abstraction with Proficiency

This section seeks to address the research question "Does the verbal expression of abstraction correlate with student proficiency?", specifically looking at the relationship between the abstract statements student make and their ability to solve the simple array problems presented in the study. Measures of student abstraction are often correlated with student proficiency at writing algorithms in code. There are many ways to indicate student proficiency at writing algorithms in code. I considered two possibilities for this work, a scoring of each first submission made in attempt of the problem (analogous to performance on an assessment in the absence of feedback [48]), and a count of the number of submissions required by the student in order to solve the problem.

The first submission score is a snapshot of the student's performance, taken only a part of the way through the problem solving process. The statements collected and analyzed are from the entire problem solving session. Ultimately all students completed every problem within the system as a part of the study, and the measure of their proficiency should match the time frame for which the statements are collected.

There is a weaknesses in using the number of submissions, as students sometimes struggled with compiler errors that slightly increased the number of submits, but had no bearing on the student's understanding of the abstractions involved in the algorithm. Despite the weakness, the number of submits is the best measure of student proficiency throughout the problem solving process available at this time so I will use it as the measure of proficiency throughout the analysis in this section.

[13]Some students may use an incremental approach to the problem, submitting small programs that are not complete, but can be checked for syntax and basic structure, however none of the think aloud subjects verbalized this strategy during the study.

Problem Condition	NA	BL	MR	FR	NC	Total
1	305	89	194	61	212	861
MO	133	45	50	12	55	295
MA	40	16	32	10	22	120
CO	60	13	63	22	54	212
CA	72	15	49	17	81	234
2	310	216	239	40	164	969
MO	168	121	86	22	82	479
MA	26	32	54	0	12	124
CO	33	7	40	7	14	101
CA	83	56	59	11	56	265
3	243	72	186	35	90	626
MO	114	31	76	19	39	279
MA	27	15	48	2	16	108
CO	48	13	42	9	14	126
CA	54	13	20	5	21	113
4	166	67	170	26	65	494
MO	85	40	66	19	35	245
MA	18	12	23	1	3	57
CO	20	9	57	4	11	101
CA	43	6	24	2	16	91
Total	1024	444	789	162	531	2950

Table 4.8: Statement Codes by Problem
Table Key: NA - No Abstraction, BL - Block Labeling, MR - Macro Structure, FR - Feedback Response, NC - Not Code Specific (as detailed in Section 4.1.7), MO - Mathematical Problems, Output Based Feedback, MA - Mathematical Problems, Algorithmic Feedback, CO - Contextual Problems, Output Based Feedback, CA - Contextual Problems, Algorithmic Feedback (as detailed in Section 4.1.6)

4.3.4 Abstraction Labeling of Statements

The data contain 2950 statements made by 24 subjects across the four problems they were solving. Table 4.8 shows the aggregate count of each code broken down by problem the students were solving at the time the statement was made, as well as aggregate totals[14]. The most frequently used code was NA, making up 35% of the statements made by students. Surprisingly, 20% of the statements were NC or Not Code Specific (i.e., "I wish this window was larger"), not focusing on the code the student was writing for the problem. The particular problem students were working on was not a significant factor (F(1,22)=57.4, $p>0.05$) in the breakdown or total number of statements made, and therefore the problems are collapsed for analysis after Table 4.8 (as shown in Table 4.9).

[14]

Condition	NA	BL	MR	FR	NC	Total
MO	500 (38.5%)	237 (18.3%)	278 (21.4%)	72 (5.5%)	211 (16.3%)	1298
MA	111 (27.1%)	75 (18.3%)	157 (38.4%)	13 (3.2%)	53 (13.0%)	409
CO	161 (29.8%)	42 (7.8%)	202 (37.4%)	42 (7.8%)	93 (17.2%)	540
CA	252 (35.8%)	90 (12.8%)	152 (21.6%)	35 (4.8%)	174 (24.8%)	703

Table 4.9: Statement Codes Percentage by Condition

During the session, students were engaged in the task of writing code which involved the typing of the actual syntax, so it is unsurprising that many of their statements were made about the code itself and were labeled NA. Additionally, students also needed to parse problem descriptions in order to determine the appropriate code to write, and so the high percentage of abstract (MR) statements is also to be expected [19]. The next section details the relationship between the statements of each type (No Abstraction, Block Level, and Macro Relation) and a rough estimate of the proficiency of the student as measured by the number of submissions it took to solve the problems.

4.3.5 Abstractions Correlate with Proficiency

The initial expectation based upon prior work with explain in plain English (EiPE) questions [49, 50, 57, 82], was that students with a higher proficiency would use more abstract statements and fewer non-abstract statements when thinking aloud while programming.

There was a correlation between the number of submits and the raw number of statements made in each category (NA, BL, MR), however this is not an accurate metric for determining the difference between students. A student who took more submits to solve the problem often spent more time solving the problem and therefore talked for a longer period of time, making more statements in general. The raw data were transformed to represent the percentage of statements made by the student in each category, and the transformation is used for the remainder of the analysis in this chapter. An aggregate table of submissions per research condition is shown in Table 4.9.

An Analysis of Variance (ANOVA) showed no significant relationship (p=.38) between the raw proficiency of the student (as measured by number of submits to solve each of the 4 problems) and the percentage of statements made at each level. Although the number of statements made in total was high, the number of students in the study was relatively low and a lack of power may account for the lack of significance. As a discipline, we often categorize students into high and low performers, and a similar categorization proved useful for analysis here.

The median number of submits across all four problems for all students was 15.5. Students were classified as high proficiency (fewer submits than the median) or low proficiency (more submits than the median). The median score of the high proficiency students was 8.5, while the median of the low proficiency students was 25.

As shown in Figure 4.5, there was no significant difference (p>.1) between the percentage of NA or BL statements made by students of high or low proficiency. There was, however, a significant difference between the mean percentage of MR statements made by students of high (Mean=.35) and low (Mean=.27) proficiency students (p <.05, SD=.12, .14). This finding would indicate that higher proficiency students are more likely to use abstract statements during the self explanation process.

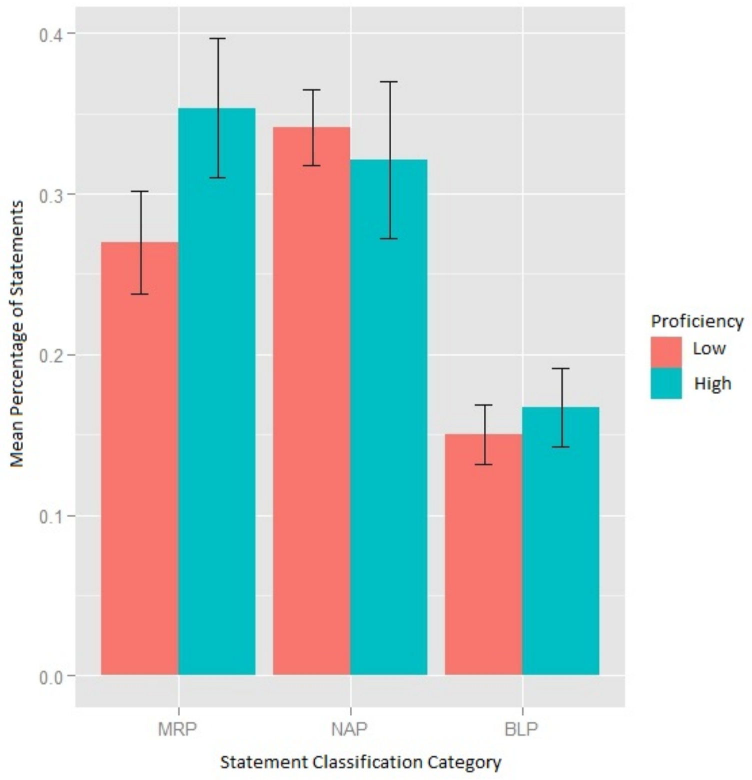


Figure 4.5: StatementClassifications by Student Proficiency

4.3.6 Transitions Correlate with Proficiency

There is an additional dimension to the data beyond the percentages of statements students make at various levels of abstraction. The statements were also coded for transitions between classification categories. A transition is defined as a statement that was coded differently from immediately prior statements. For example, if a student said *"So after we are done with the for loop, we want to return the counter"*, it was coded as MR. In the next statement, the student may have verbalized typing the return statement with *"return c, semicolon"* which would be coded as NA and marked as a transition statement.

Statements coded as NC or FR were not considered transitions. It was common for a student to express a feeling or read a piece of feedback as a part of the problem solving process without changing the level of abstraction in the surrounding statements. For example, a student could be typing a for loop and make the statement *"that is myList dot length"* (NA), and then say *"I remember this from the last class"* (NC), and return to a statement *"i plus plus parenthesis"* (NA). The student inserted a contextual cue, or statement focused on something outside the problem scope, but still continued to act at the same level of abstraction as prior to the statement. A sequence of codes such as NA, NC, MR would have the last MR marked as a transition as it represented a change in category from the NA.

Similar to the original classification of statements, the raw number of transitions is skewed towards students who took longer to solve the problems. To normalize the number of transitions, the percent of statements which were transitions between classification categories in all of the following analysis were used.

Figure 3 shows a comparison box plot between high and low proficiency students and the percentage of statements they made which transitioned between classification categories. There is a significant (p<.01) difference between the percentage of transitions in low and high proficiency students. High proficiency students transitioned between categories on average in 38.3% of statements, while low proficiency students transitioned between categories, on average, only 30.2% of the time.

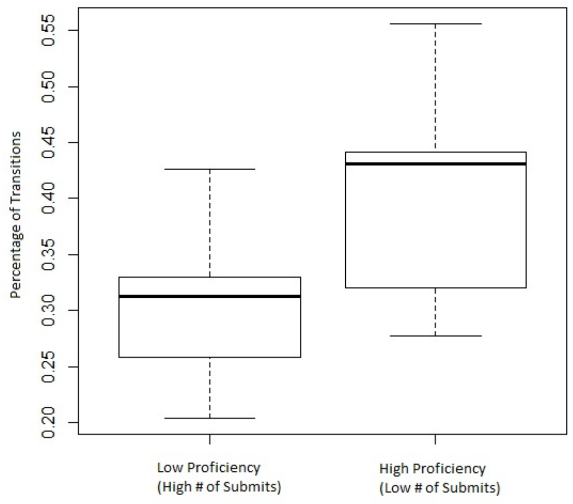


Figure 4.6: Transitions by Proficiency

The increase in the percentage of transitions indicates that students of higher proficiency move between the concrete code they are writing and the more abstract problem or relationship between an individual line of code and the entire algorithm more often. Examples from the think alouds indicate that students were engaging in self explanation, connecting the specific code to the larger goals of the problem.

For example, a high proficiency student engaged in the following dialog where he was able to transition between the specific code he was writing, to the relationship to the larger problem as he progressed. The dialog from the student, along with category classifications is shown in Table 4.10. Dialog from a low proficiency student at a similar point in the same problem is shown in Table 4.11.

Statement	Category
for int i, so check out the for loop at first	NA
just so I am understanding, this argument passed in, this is the number that each element in the array has to be less than? ok	MR
Ok, I'm pretty sure the loop is right on	NC
myList i is greater than or equal to 0	NA
and less than value	NA
I see it now	NC
I thought it meant to return the sum of all those numbers, it just wanted the total number of all those numbers. I see the difference	MR

Table 4.10: A High Proficiency Student Solving The Count Problem

Statement	Category
0 less than scores.length i plus plus	NA
so if, I'm just checking the condition	NA
if scores dot of I hm, ok.	NA
is less than 0	NA
so that I won't get confused if I use places	NC
and scores of I is greater	NA
and 0 also comes under that	NA
so 0 to greater than or equal to 0	NA
and less than passing grade	NA

Table 4.11: A Low Proficiency Student Solving the Count Problem

In the same part of the problem, the low proficiency student did not relate the code he was writing verbally back to the larger problem, or the abstract nature of the algorithm he was attempting to write.

When looking at the difference in the percent of transitions, 8% may seem small. However, when the relative amount of time spent on each problem is considered, combined with the higher use of MR statements by the high proficiency students, low proficiency students will spend a significantly longer amount of time discussing the individual code elements and syntax while engaging in practice.

4.4 Proof of Concept: Students Use Algorithmic Components in Code Production

This study addressed 5 research questions in the validation of the algorithmic components using student problem attempts and think aloud transcripts. This chapter sought to provide evidence to answer five research questions with data from a think aloud study conducted with novice programmers. The analysis included both qualitative and quantitative work using the student statements during the think aloud, as well as the code that was written and submitted to an online system.

Although arguments can be made for the addition (or removal) of components based upon different languages or different levels of student ability, for the population evaluated in this study, data indicate that the components are appropriate and cover a majority of student errors with regards to algorithm construction. Each of the components were observed to be incorrect in at least one participant's submission, demonstrating the ability for students to make such an error and supporting the need for each of the listed algorithmic components in a feedback model.

In addition to observing the components initially proposed, research questions 2 and 3 also focused on evaluating the completeness of the set of algorithmic components. As a result of evaluating the think aloud transcripts and student submissions, two additional components were added to the feedback model after the initial 6 participants[15]. Initially, I hypothesized that declaring and initializing a variable to keep track of the state of the algorithm (i.e., the sum of the array elements encountered so far within the for loop) was a skill that students would have mastered by the last month of the course. A number of submissions made by students were missing the declaration or correct initialization of a state variable, and so a component was added to the model to check for this omission. I also observed a misconception in the use of +1 to access the "next" element as a part of the ongoing evaluation of array elements as discussed in section 5.5.2. The loop-current algorithmic component is the only component focused on a misconception specifically, however it was observed frequently enough to indicate the appropriateness of the component's inclusion in the feedback model.

While evaluating research question 1, in addition to observations about the model to

[15]The initial 6 participants received output based feedback, so the addition of additional algorithmic components would not have changed any of their interactions during the think aloud.

be used to generate algorithmic component feedback (as detailed in Chapter 5), analysis of the students' use of abstraction provides additional evidence supporting the evaluation of a hypothesis of this thesis: *Feedback regarding algorithmic components presented to students pre-compilation will produce better (a) within-problem performance and (b) across-problem learning.* The transition to abstraction when describing the algorithmic components in code was more prevalent in students of higher proficiency.

As reported in Section 4.3, the decrease of errors in first attempts at the problems indicates that student learning occurred during the interaction with the system. Indications of learning could mean two things: (1) that the practice problems were of appropriate difficulty - causing mistakes to be made but not so difficult as to prevent learning over time and (2) that the model can be used to show progress towards expertise in student practice. The demonstration of student learning from the practice problems also answers the question of whether learning can be observed over a short practice session. Although the problem order was fixed, the required algorithmic components of each problem are mostly the same, and the appearance of appropriate components at first submit indicates an increase in student understanding of when the models must be applied.

Not only can learning be observed, the algorithmic components also appear to be a viable way to evaluate the difference in student submissions across problems. Because many of the problems share similar algorithmic components, the consistency provides for a measure of student progress despite the difference in algorithms the students are asked to produce. A more sophisticated discussion of measures of student progress and the use of the algorithmic components for additional measures can be found in Chapter 6.

In this and previous chapters for this thesis, I have laid out the theoretical foundation for the development of a pre-compilation feedback mechanism using algorithmic components and abstractions (Chapters 1-2), and described (Chapter 3) and tested (Chapter 4) a model of algorithmic components useful for practice problems using simple array algorithms. In the next chapters I move forward with an analysis of the mechanism that is used to generate feedback in AbstractTutor (Chapter 5), a discussion of metrics for measuring student performance in AbstractTutor (Chapter 6) and the results of online studies evaluating the effectiveness of the feedback (Chapter 7).

Chapter 5

Implementing Algorithmic Feedback in AbstractTutor

I have presented evidence from literature and data supporting the case for the construction of a pedagogical integrated development environment (IDE) to support novice programmers. Drawing from the literature surrounding expertise and models of abstraction (Chapter 2) and building upon personal expertise and task analysis (Chapter 3), I designed a model of algorithmic components to be used to produce feedback for novices while they practice writing code to construct simple array algorithms. Having validated the algorithmic components in the model and demonstrated the potential usefulness for tracking student progress in a pilot study with a prototype of AbstractTutor (Chapter 4), I now detail the technical aspects of the feedback mechanism implementation for a fully implemented online AbstractTutor.

5.1 Automated Testing of Student Code

There are numerous approaches to the automatic generation of feedback messages for programmers. The evolution of professional tools such as compilers and IDEs is well documented. Educational tools offer a more diverse set of feedback categories, and therefore employ a wider range of methods to produce the feedback. In Chapter 1, I described several types of feedback in development environments, as well as some structural or visual attempts to supplement the feedback provided by the compiler. In this chapter, I focus on the mechanism behind the feedback production in AbstractTutor. The analysis presented here, as well as the results of testing against student code, speak directly to the research question: *Can a pre-compilation feedback mechanism be constructed that operates with reasonable accuracy (85% of student generated submissions)?* To answer the question, I present the design of the feedback mechanism of AbstractTutor and the results of data analysis used to refine the feedback mechanism with respect to simple array problems.

5.1.1 Desirable Components of Feedback Mechanisms

At the core, feedback mechanisms communicate structure and state to the user. Structure is important to the programmer for many reasons. Structure can be used to describe a macro relationship between the files, objects, or methods used in code, and the dependencies that exist. Structure can also be used to describe a more micro view of the code, specifically looking at the scope and relationship of individual lines of code, or even variables and expressions, within a particular algorithm. Both types of structure can be very useful to the expert who has a mental model of the interworking parts of the solution they are trying to express in code. State refers to the stored result of computation performed in code. The state associated with programs or code segments can be useful in comparing expected values with actual outputs. Both structure and state can be difficult to determine if code produced by a novice does not conform to minimum standards necessary for the environment to parse.

Many current environments focused on languages entered by typing, [1] which have been used in introductory computer science courses, rely on feedback mechanisms designed for and by the professionals the language was built to support. For an expert, the most important feedback comes when she tests her code against input and output values, often with carefully thought out examples of "edge cases" - or unique situations that could produce unexpected behavior. Compiler errors are most often produced by a typo, or a misreading of documentation, and are easily understood and corrected. This testing behavior is very different from the behavior of the novice. A novice often requires feedback just to produce minimally executable code, and is looking not to test edge cases, but the overall performance of the program or algorithm. The errors a novice makes that result in a compiler error occur just as often from a misunderstanding of semantics or appropriate program structure as from a simple typo [52].

In addition to the gap in knowledge and skills between a novice and expert, there is also a significant difference in the primary purpose of writing code for a novice and an expert. In most cases, when an expert writes code, he is attempting to build and test a solution to a problem. Often that problem will not have an acceptable previously implemented solution; otherwise the expert would simply use the existing solution. And although the expert may not have a mental model of an absolutely correct solution to the problem he is addressing, he often will have a correct implementation of an approximate solution. For example, if an expert programmer is trying to create an application to display a user's twitter feed with the most important messages first, he may not know how best to choose the "most important" messages, but for the criteria he has decided upon, he will know a way to implement a program that displays a messages from a users' twitter feed or how to find packages and documentation providing worked examples the expert can understand and use to solve the problem. The challenge of the problem lies not in implementing the code once decisions are made, but instead modifying and testing that solution in a process of iterative refinement.

The testing a potential solution to a coding problem has a different purpose for the novice. In contrast to the expert, the novice writes code to implement a working solution

[1]as opposed to drag and drop languages such as Alice or Scratch

to a well defined problem. A significant part of the struggle for the novice is not only the implementation or code production, but also the formulation of a mental model of the algorithm itself. The code writing exercise itself is an assigned practice in the implementation of specific concepts, and the novice is expected to learn from the experience, implying that she is not proficient before the exercise. The novice uses the environment and its feedback as a learning mechanism, therefore expecting assistance with, not just correction of, the code being written. The difference between the needs of the novice and expert would indicate that novices would benefit from a feedback mechanism focused on the content of the problem in addition to the language.

5.1.2 Limits on Implementation

There are two main barriers to designing a perfect feedback mechanism for novices. The barriers are generalizability of the tool to different problems and difficulty in proving correctness of a student solution. Although better feedback can be obtained through the use of problem knowledge, and matching against a correct solution, the structure needed to have this knowledge limits the generalizability of the tool to the problems defined within its structure. Additionally, by comparing to a correct solution, the students are constrained to a particular solution space, although Computer Science is a domain that is celebrated for its creative thinking and problem solving features. Although these arguments make sense when compared against the needs of the expert, for the novice the goal is not flexibility, but practice and learning.

The second barrier to designing a perfect feedback mechanism is the difficulty for a computer to prove the correctness of an arbitrary solution. With the difficulty of proving correctness, many methods have been developed to test the correctness of computer code. Educators have adopted methods such as unit testing [24, 60] and verification [62] to automate the grading of student programs. In addition to reducing the time it takes to verify and score student work, automated testing can also shorten the feedback loop and provide novices with a more rigorous validation of the code they have produced [58].

Automated testing treats the student's code as a black box and determines accuracy based on comparing expected outputs against output generated by a student's code. Imagine testing a function that finds the sum of two numbers. You could put in a 3 and 5 and receive an 8 for an answer. Although at the surface, this appears to validate the performance of the code, perhaps the code always returns 8, or has a more difficult bug to catch such as always returning a positive number. The ability to construct output-based tests to test all possible cases is itself a very difficult problem, especially for novices [58].

Verification strategies have been used to provide a more rigorous evaluation of student code while problem solving [62]. The verification mechanism, however, suffers from some of the same issues as a generalized compiler. First, students need to write assertions in a language the computer can understand, in order for the generalizable mechanism to provide appropriate feedback for the particular solution the student is trying to write. This addition of a level of complexity ensures that the student understands the problem, but does little for the novice who is having algorithmic difficulties and cannot produce correct assertions.

In addition to the difficulty of testing all possible cases, such testing also focuses only on the output. Although this type of testing is useful in industry, where experts care mostly about expected behavior and not the details of implementation, output based testing cannot be used efficiently to generate feedback regarding the individual components of student code. Additionally, the construction of unit tests focuses the feedback on particular problems and limits the generalizability of the practice environment. The algorithmic component approach used by AbstractTutor will allow for future development of similar problems about array algorithms without needing to develop new feedback mechanisms or statements.

Another possible way to test student code is to treat the code as text and use techniques such as string matching or regular expressions to check for predetermined commands within student generated code [78]. These methods of checking for a predetermined set of commands impose a rigidity on student solutions that is more applicable to domains of recall such as spelling, than to subjects involving complex problem solving.

Alone, neither output based feedback, nor a rigorous inspection of the novice's code is a viable strategy for producing feedback about the algorithmic components in student code. Despite the prior work in using unit testing and other mechanisms to evaluate student code, the application of these mechanisms to the algorithmic components of code, specifically for the purposes of feedback during code production, is new and a contribution of this thesis.

5.2 Evaluating Code Pre-Compilation

AbstractTutor operates with the knowledge of the problem the user is attempting to solve in order to provide the most detailed feedback about appropriate components. Although there is a solution space for correct answers, novices will create an unreasonably large space of incorrect answer attempts for feedback generation mechanisms to handle efficiently or accurately [77]. In this section, I discuss several approaches to the analysis of student code for either grading or more detailed feedback purposes, and the strengths and weaknesses of each approach as it applies to the goals of the AbstractTutor system.

As noted in Chapter 3, there have been many prior attempts to create feedback mechanisms for student-produced code. Almost immediately, researchers agreed that enumerating all of the possible correct and buggy states for student produced code was an impossible task.

> "Clearly, one can't possibly enumerate beforehand the space of program interpretations: there are just too many ways to construct correct and buggy programs. Rather, starting with the problem specification and a database of correct and buggy plans, transform rules, and bug-misconception rules, PROUST constructs and evaluates interpretations for the program under consideration. In effect, the goal decomposition and the plan analysis of the program evolve simultaneously." [36]

5.2.1 Determining the Program Components

Computer science programming problems pose more difficulty for determining the correct aspects of a partially correct solution than many other domains, except perhaps essay writing. While not trivial, it is usually within the range of the average computer science instructor's ability to write output based test cases to determine correctness for the programs they assign at the introductory level, so this practice has become a standard in many CS classrooms [23, 61]. At the introductory level, treating the program as a black box, these forms of output-based testing require that the student, or an automated system, provide input to the student's code and then based upon the output of the program determine if it is operating correctly. Sometimes when a program gives incorrect output, a probable cause of the bug can be inferred from the output. This approach, however, only works for a small subset of the possible solutions attempted by students.

A detailed analysis of the actual code authored by the student is needed in order to more accurately determine the source of the incorrect result. The code produced by a student is essentially text, and prior work has shown the text-based evaluation of code through string matching to be effective in a constrained environment [78, 80] where the solution space is limited due to the nature of the questions. This approach of matching the exact letters that a student uses will not work except for problems where students are writing small code segments. In larger problems, students have the ability to name variables as they choose, which introduces variability that makes string matching difficult. For example, a variable to keep track of the sum of all the numbers in a list could be named sum, s, sumOfNumbers, or even myCatFluffy. All of the examples are valid variable names and represent just a small fraction of the almost infinite space from which students choose.

Additionally, it is not only important that a particular line of code appears in a solution, it is also important where it appears, and how that line fits within the larger structure of the code. Consider the two examples in Table 5.1. I have added appropriate spacing make the code easier to read; however, the spacing has no effect on the program execution. In the examples, the code appears to be very similar, and in fact the only difference is the addition of { and } in the left hand example. By adding the braces, the return statement will be considered part of the loop and will return the value stored in s after only the first item has been added to it. A string-matching evaluation of the code could ascertain this detail, but it would be very difficult to determine all of the possible combinations of algorithmic structures with this methodology.

The most useful way to map the code would include both a representation of variables that did not depend on the variable name, and a structural representation that would be useful in determining the relative placement of algorithmic components. Abstract Syntax Trees (AST) meet both of these specifications and have been used by others to evaluate student code in computer science environments [41, 68]. AbstractTutor uses abstract syntax trees in order to evaluate not only the algorithmic components of student code, but also their relationship to each other in the program structure in order to provide the appropriate feedback to students.

The contributions of AbstractTutor lie in the application of abstract syntax trees to infer *algorithmic structure* of student programs and provide feedback that not only identifies

Table 5.1: Two Code Examples Difficult to Distinguish by String Matching

```
int s=0;
for(int i=0; i< myList.length; i++)
{
     s = s + myList[i];
     return s;
}
```

```
int s=0;
for(int i=0; i< myList.length; i++)
    s = s + myList[i];
return s;
```

the presence of appropriate code, but its existence within the structure of the student-produced algorithm. Additionally, AbstractTutor then applies the research on abstraction to the development of the feedback messages, encouraging students to consider the way the algorithmic components must work together in order to achieve the desired result. No other feedback systems in the literature review explicitly use feedback attempting to have the students think abstractly about the components of their code.

5.2.2 Focusing on Pre-Compilation Feedback

Abstract Syntax Trees are a form of static analysis [32] to evaluate code prior to execution. Static analysis techniques were developed in professional environments to combat some of the same issues facing our development of a system to provide feedback to novices [92]. Pradel and Jaspan created a system for professionals learning new software frameworks using static analysis techniques to validate code against framework constraints [65]. The system maintains generalizability through the use of javadoc comments provided by the programmer to cue the validation of code.

Although Jaspan's work focused on feedback for professionals who are experts at producing algorithms and need assistance identifying constraints that are required by a library, the work is a precedent for the application of static analysis to feedback mechanisms in computer science.

In AbstractTutor, the static analysis techniques allows for pre-compilation feedback regarding algorithmic structure. As discussed in Section 1.2, the focus on syntax early in the problem solving process is contrary to research on efficient second language learning [22, 86]. Additionally, as shown in the case studies at the end of this chapter and in Chapter 7, compiler errors are often generated by incorrect algorithm implementations, or student misconceptions. Focusing on fixing the errors through compiler messages without addressing the underlying cause, when they are produced by a deficiency in student knowledge, does little to aid the student in making productive edits, and eventually arriving at a well thought out solution.

The production and display of feedback regarding the algorithmic components prior to compiler errors gives the opportunity for students to focus on the macro and ensure the structure is correct, and stays that way, before worrying about the atom level, which should reduce the number of symptomatic corrections that do not relate to the larger algorithmic problem.

The following sections detail the code evaluation process used to generate the pre-compilation feedback in AbstractTutor.

5.2.3 Evaluating Code in AbstractTutor

In the AbstractTutor system, student code is parsed to produce an Abstract Syntax Tree (AST) and the tree is checked for evidence of appropriately located algorithmic components. The system generates a series of 1's and 0's representing the appearance (1) or absence (0) of each algorithmic component in student code. I describe the set of 1's and 0's produced by the evaluation as an alignment vector. The alignment vector is produced with a consistent ordering of the algorithmic components and therefore alignment vectors can be compared across submissions. Tables 5.2 and 5.3 show two code snippets and the corresponding alignment vector relative to the feedback model described in Chapter 3.

The first code snippet, show in Table 5.2, has a repetition structure (for) containing a statement using the length of the list (myList.length) as a bound. Additionally, inside the structure there is a reference to an element from the list (myList[index]) and it uses index, the loop variable, in the reference. The code only accesses the current element (there is no [index + or - 1]). The algorithm is implementing a sum, which does not select any particular elements, therefore there are 3 0's for the components related to selecting appropriate elements. The code does not initialize a state variable (the fourth 0), and therefore although the student wrote total += myList[index] we cannot observe the attempt to use, or correct update of the state variable. Finally, there is a return statement, and it is in the appropriate place for the algorithm (the next two 1's in the vector), but the output cannot be observed to be correct because this code will not compile without declaring the total variable.

Three of the alignment vector bits are checked by the system at runtime: *Correctly identifying with an if, Correctly Updating for Algorithm*, and *Returning the Correct Value*. These three components are checked when the system compares return values from the method with both predetermined edge case values and randomly generated values and answers. In the current version of the system, all three components are checked at the same time and are marked as 1 or 0 simultaneously. There are plans for future work to disaggregate test cases that would indicate a specific error with either the if statement or the update to the state variable. The *Returning the Correct Value* bit is used as a final check to ensure that, even if a student has all of the right components, there is not additional code that would yield an incorrect algorithm.

Although the student did presumably attempt to use a state variable without declaring or initializing it, the constraint for using the state variable is tied to the declaration. The feedback the student receives in this situation is about the initialization; and once the line of code "int total = 0;" is added to the beginning of the algorithm, the use of the state variable can be observed and assessed. Feedback is presented in the order of the constraints, so the first constraint that the student fails will trigger that feedback message. The constraints in the alignment vector are ordered so that declarations, initializations, or

```
public int findSum(int []myList){
 for (int index = 0; index<myList.length;
                 index++)
    total += myList[index];
 return total;
}
```

Including a Repetition Structure	1
Using the Length of the List	1
Referencing Any Element From the List with [ ]	1
Using the Loop Variable to Access	1
Only Accessing the Current Element	1
Making a Comparison With An If	0
Utilizing an Element in the Comparison	0
Initializing State Variable	0
Using the State Variable	0
Attempting Any Return	1
Correct Place for Return	1
Correct Return Value	0

Table 5.2: Code to Alignment Vector Mapping 1

```
public int findSum(int []myList)}
  int total = 0;
  for(int index = 1; index < 10; index++)
    total += 5;
  return total;
}
```

Including a Repetition Structure	1
Using the Length of the List	0
Referencing Any Element From the List with [ ]	0
Using the Loop Variable to Access	0
Only Accessing the Current Element	0
Making a Comparison With An If	0
Utilizing an Element in the Comparison	0
Initializing State Variable	1
Using the State Variable	0
Attempting Any Return	1
Correct Place for Return	1
Correct Return Value	0

Table 5.3: Code to Alignment Vector Mapping 2

other dependencies will receive feedback messages first, before feedback related to the use of a particular item. For example, students cannot receive feedback indicating that they should use a loop variable to access an item in the list without first having constructed a repetition structure with an appropriate looping variable.

In the second code snippet, shown in Table 5.3, the student did include a repetition structure (for), however did not use the length of the list to appropriately bound the loop. The code does not contain a reference to the list in the loop body, and therefore cannot use the loop variable in that access. In not accessing any element, the student has not accessed more than the current element, so the constraint focused on only accessing the current element is upheld. The constraint for the current element is the only "fail" state in the model - requiring the observation of incorrect code in order to be incorrect. An empty submission will still uphold this constraint. As in the last example, there is no need to implement a selection of particular elements, so there are three 0's for those constraints. The state variable is initialized (1) and updated in some fashion during the loop execution (1) but as the result of the code when checked against output based testing is not correct, this code does not get a correct update for the variable (0). There is a return statement (1), it is in the appropriate place (1), however it will not produce the correct value when checked against output based testing (0).

In the alignment vector, the correct update and the correct return are checked with output based testing; in the AbstractTutor system that is done by comparing expected outcomes against random and specific test cases. A more detailed description of the output based testing is found in the next subsection.

5.2.4 Automated Assessment Mechanism

The AbstractTutor system was implemented to operate in a web browser, and uses the Janino framework for program compilation, Abstract Syntax Tree (AST) creation, and program execution against test cases. As discussed in Section 2.1, the variability of individual student code solutions requires an intermediate step to evaluate student code against the algorithmic components used to generate feedback in the system. In AbstractTutor, the intermediate representation of student code is an Abstract Syntax Tree (AST).

An AST is a structural representation of student code that allows for the removal of identifiers whose name is not salient to the solution. Figure 5.1 shows an AST representation of a generic problem in AbstractTutor. The generic representation was used to help design the AST walker - a java program that took the completed AST and looked for the appropriate algorithmic components in expected places in the student code submission.

Extraneous code that does not affect the structure of the code is ignored by the AST evaluation[2]. If the extraneous code impacts the correctness of the algorithm, the elements of the alignment vector concerned with the correctness of the output based feedback will be affected.

[2]The AST will still contain the code, the evauluator will just not use it when constructing the alignment vector

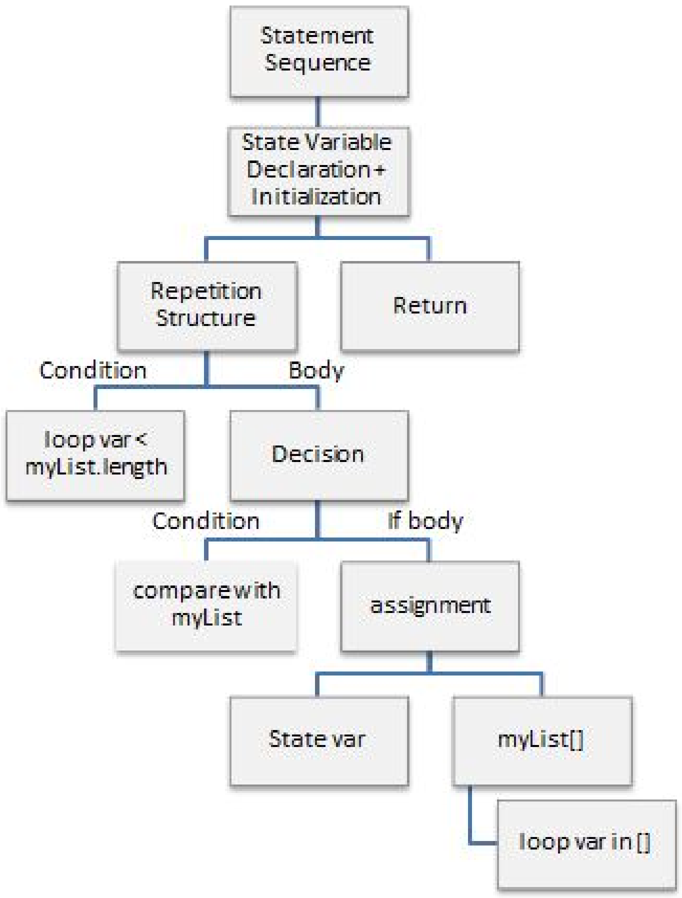


Figure 5.1: A Generic AST

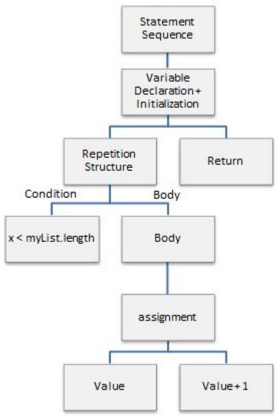
5.3 Confirming Code Evaluation Accuracy

An important part of feedback for the novice is the accuracy of the feedback messages. In this section, I present the results of a study with goals of (1) to evaluate the efficacy of an AST approach to code evaluation and (2) to provide opportunities to fine tune the feedback mechanism to provide the most accurate feedback possible. These two goals contribute to the overall design of the tutor and an evaluation of its accuracy for the main thesis research question: *Can a pre-compilation feedback mechanism be constructed that operates with reasonable accuracy (85% of student generated submissions)?*

5.3.1 Analyzing Existing Think Aloud Data

Novice programmers can produce erroneous code, and while some common misconceptions and errors exist, the full extent of student mistakes is difficult to produce artificially. It is therefore essential that we evaluate and fine tune the feedback mechanism on code written by the novice. In this section I explain a detailed assessment of the AbstractTutor code evaluation mechanism based on hand evaluated code, and an analysis of a large data set

Code	`public int countLessThan(`
	`        int []myList, int value){`
	`     int x;`
	`     for (x = 0;`
	`          x < myList.length; x++){`
	`        value += 1;`
	`     }`
	`     return(myList.length - value);`
	`}`

AST	 Statement Sequence Variable Declaration+ Initialization Repetition Structure Return Condition Body x < myList.length Body assignment Value Value+1
Alignment Vector	11001-00-00-110

of automatically evaluated code.

To determine the accuracy of the automated assessment mechanism, I used the submissions gathered by the system during the think aloud study described in Chapter 4. These data were produced by 24 subjects solving 4 problems for a total of 482 submissions. Table 5.4 shows a breakdown of the data by feedback condition. Students in the think aloud were placed in one of two conditions, as described in Chapter 4. One group relied only on the feedback automatically produced by the system equivalent to a standard IDE (compiler and output based testing), and the other group had algorithmic feedback read to them at appropriate times. Each row of Table 5.4 gives a particular statistic (like average submissions) for a particular group of students (either by feedback condition or for all participants). For the analysis of the think aloud data I was assisted by Alexandra Johnson, an undergraduate research assistant.

For a larger and more diverse dataset we use data collected in the Spring of 2011 from a web-based study. The full details of the dataset are described in Chapter 7 with regards to participant recruitment and engagement. For this assessment of AbstractTutor, we were only concerned with the performance of the parser and therefore used the student submissions from the study to evaluate the system accuracy. During the study, 160 students

	Algorithmic Feedback	System Feedback	All Participants
Average Submissions	14.5	25.6	20
Max Submissions	26	72	72
Min Submissions	6	7	5
Unique Alignment Vectors Observed	54	37	75

Table 5.4: Think Aloud Student Submission Data for all Four Problems

made 4730 submissions to the AbstractTutor system while solving 4 problems. Table 5.5 shows the details of the dataset.

	Per Problem	Per Four Problems
Average Submissions	8.7	28.1
Median Submissions	3	14.5
Maximum Submissions	248	315
Min Submissions	6	4
Unique Alignment Vectors Observed	N/A	121

Table 5.5: Online Study Student Submission Data

Table 5.6 shows some of the most frequent Alignment Vectors (more than 80 observations) with a description of the errors students make to produce the Alignment Vector, and, where appropriate, the problems most likely to cause the state to be observed. The largest majority of vectors are those that have all the appropriate algorithmic components for the algorithm (25%) but do not pass the output based testing. In these cases, the student will often be struggling with a compiler error preventing the code from compiling, or a detail of the implementation focused on an edge case of the algorithm (not including 0 in the count of numbers for problem 3). The feedback for students in this case would either be the compiler errors generated by their code, or the output based testing result (expected and resultant values with a given input). A more complete description of the model states and the transitions between them appears in Chapters 6 and 7 when I focus on student learning outcomes.

The goals of this analysis are three fold. First, I evaluated the ability of the AbstractTutor parsing mechanism to accurately categorize student code solutions with respect to the Algorithmic Component Model proposed in Chapter 3 using the Think Aloud data from Chapter 4. Second, I compared the accuracy of the AbstractTutor parsing mechanism to unit testing, as a proxy for output based feedback, with an expert human rater using a subset of the Think Aloud Data. Finally, I used the online submissions to evaluate the accuracy of AbstractTutor over a large data set, including the conditions where AbstractTutor failed to accurately parse the student code submissions.

Alignment Vector	Count	Description
11111-11-11-110	935	This vector represents almost correct code. The submission has all of the appropriate algorithmic components but does not pass the output based testing. There may be a problem with an edge case, a loop bound, or other detail in the algorithm that causes it to fail the output based testing.
11111-00-11-110	515	The vector shown on the left is a solution that does not contain an if statement, and does not pass the output based testing. Since problem 1 did not require an if statement (sum of all numbers) all but 15 of the observations of the vector (from 3 participants) were from problem 1.
11111-11-00-100	181	The vector shown on the left is a solution without an update for a state variable, and where the code is missing a final return, and has not yet passed output based testing. The majority of these submissions (165) came from problem 4 (return the index of a number in the list) which can be solved without a state variable. Eight users, representing 16 submissions, obtained this vector on problems other than 4.
11111-11-11-000	115	This vector is produced by code with no return statement, a common error among students. This error was distributed across problems 2 and 3, as problem 1 without a return produced the vector 11111-00-11-000 (observed 51 times) and problem 4 without a return produced the vector 11111-11-00-000 (observed 14 times). The converse vector (only a return - 00000-00-00-100 or 00000-00-00-110) was observed 34 times.
11111-11-00-110	83	This vector is produced by code without a state variable correctly declared or updated, and that has not passed output based testing. A majority of the submissions resulting in this vector came from problem 4 (62) since it does not require a state variable. The remaining submissions (21) were from 13 users and a review of the data showed most of the submissions did have a state variable, but did not declare it correctly before the loop.

Table 5.6: Online Study Student Submission Data

Accurate Categorization

I, as an expert human rater, coded each submission from problem 3 in the think aloud data set for alignment with the model of program components. Problem 3 was chosen because it uses all of the states from the algorithmic component model described in Chapter 3.

In the coding phase, the program submission was evaluated for the presence or absence of a component from the perspective of a human rater evaluating against the rubric the alignment vector was based upon, with each component was marked with a 1 or 0 based on this standard and no partial credit was possible per component. The 1/0 scoring produces a human generated alignment vector. This form of rubric based grading is used by the Educational Testing Service (ETS) to consistently evaluate thousands of exams each year. All judgements of accuracy for the AbstractTutor system are based on a comparison against the standard set by the evaluation of the AP Computer Science exam, as the largest human rating effort for computer programs available.

An automated script was used to evaluate the student produced code and generate the alignment vectors for each individual student submission. The automated script used the same framework as the web system in order to generate an AST for each student submission, and then evaluate the AST for the necessary algorithmic components to produce the alignment vector.

In preparation for the data analysis for this thesis, I inspected approximately 1/3 of the student code submissions visually looking for specific reasons for the code not to parse, examples of model states, and case studies for the current chapter and Chapter 7. In early versions of the online tutor, the parser and alignment vector generator was not tuned to recognize while loops and made incorrect alignment vectors for 25 submissions (creating vectors of all 0, looking like a failure to parse). Later versions of AbstractTutor included appropriate parsing for while structures.

Anecdotally, no other instances have yet been found where the code parsed, generating an alignment vector, but parsed incorrectly, creating an alignment vector that was not equivalent to the ideal alignment vector produced by the author. I recognize that instances of incorrect vectors may exist, but they appear to be very rare.

Partial Credit Assessment Compared to Unit Testing

Although the accuracy of the abstract syntax tree parser is very good, perhaps perfect, the metric alone does not provide an accurate comparison with either a human rater or the current standard of unit testing. In order to determine accuracy and do a comparison, unit tests were designed to test as many parts of the algorithm for correctness as possible. Forty nine responses from the think aloud were selected to compare against a human rater, an AST, and unit testing. The forty nine responses represent all submissions for problem 3 of the think aloud described in chapter 4, from the first 18 participants in the study. The first 18 participants were chosen because those were the participants who had completed the think alouds before the study was done to validate the AST mechanism. Although this is a partial data set, the results are useful to emphasize the difference between the

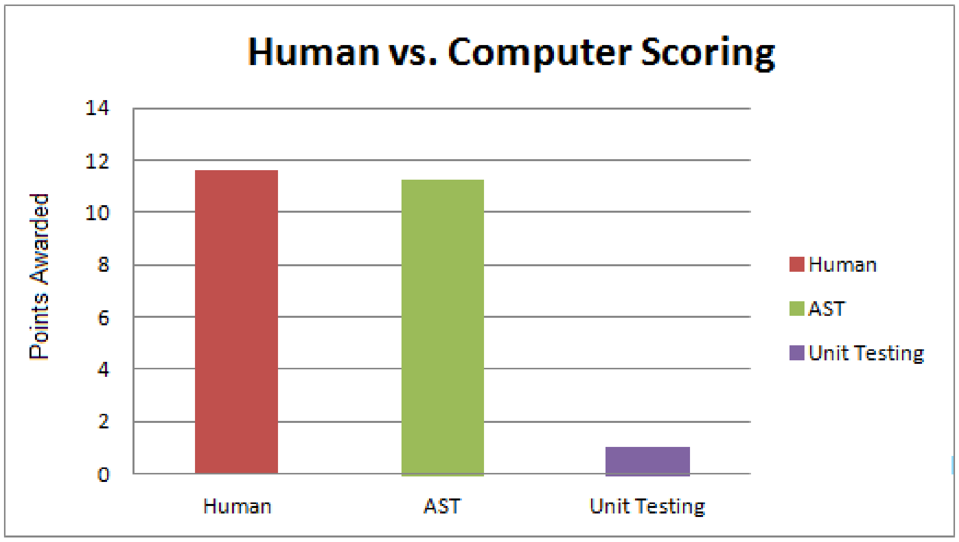


Figure 5.2: Human vs. Computer Scoring of Student Code Submissions

accuracy of output based unit testing[3] and ASTs.

Figure 5.2 shows the average results of the human rating, the AST rating and the unit testing for the 49 responses. In all but one of the cases the human and AST awarded similar partial credit score for the code submission. One of the cases was a failure to parse, and the AST awarded a 0. Unit testing was unable to approach the accuracy of the AST or human rater, often awarding 0 points as the code failed to compile and therefore could not be tested.

In the think aloud data for all problems (355 submits over 4 problems by 18 users) at the time, only 222 of the submissions compiled (63%). Of the 222 compiled solutions, 77 of them were judged to be correct solutions to the problem. This means that the output based feedback was only able to provide information about the structure of the students' code, specific to the problem they were trying to solve, 145 times or 40.8% of student submissions. In the other 133 non-compilable submissions (37%), the student needed to work with generic feedback before correcting any problem specific errors. To clarify further, in only 32 (out of 132, 24%) cases was the output based feedback sufficient, causing the student to submit a correct solution on the next attempt.

Overall, these data show that while output based testing can be useful in a percentage of student submissions, it is unable to provide reliable feedback in a majority of submissions. Additionally, the lack of granularity as demonstrated by the comparison of the AST, human rater, and output based feedback will make it difficult for a tutoring system to accurately determine which subskills the student has mastered while solving complex problems.

[3]Unit testing refers to the testing of individual methods using input data and checking the method's return value against known answers.

Accuracy over a large dataset

Novice students are especially good at producing unexpected attempted solutions to problems they are solving. In order to check the accuracy of the feedback mechanism in AbstractTutor, the large internet dataset was used. From this dataset I observed 4730 submissions from users across four problems. Although not all students in the online study were in the algorithmic feedback condition, the system still recorded the alignment vector for each submission. For all submissions, there were 869 total submissions that recorded an all-zero alignment vector when code was present (18%). I call this state a "failure to parse" as the code contained an error that was severe enough to prevent the formation of an abstract syntax tree to be translated into an alignment vector. Case Study 4 at the end of the chapter provides an example of code that failed to parse.

Reason	Number of Submits	Number of Users
For Statement Errors	128	22
Errors with { and }, method not closed properly	127	27
No statement inside loop	106	18
Used Enhanced for loop	98	12
Errors inside loop caused no parsable statements	86	12
Undetermined Errors	64	13
Attempt to access array incorrectly ([]myList or others)	64	10
No ; before for statement	56	16
Errors with if statement parenthesis	36	7
Use of a while loop	25	8
Code outside of the method	25	5

Table 5.7: Online Study Student Submission Data

Table 5.7 shows all reasons for a failure to parse with 25 or more submits. Other reasons for failure to parse included renaming the method, incorrect use of relational operators (<, >, etc.) and other errors seen fewer than 10 times. The errors that occurred fewer than 10 times were each committed by a single (but different) user and were recovered within a few submissions.

The online study where the 4730 submissions were recorded was not the final study, and its results were used to fine tune the AST generator and the feedback mechanism. An error encountered by approximately 200 users was the appearance of an illegal character in the code. I hypothesize that the character was added by certain browsers as an End of Line (EOL) or End of File (EOF) marker. In future iterations of AbstractTutor, student submissions are first checked for any characters outside the language subset, removing any characters that may cause a failure to parse, and then generating ASTs and alignment vectors.

Additionally, code was added to correctly identify and parse code containing a while structure. The enhanced for loop, unfortunately, could not be handled by the compiler version used in AbstractTutor, however an error message was added informing students that enhanced for loops were not supported by the system.

5.4 Case Studies: Code Evaluation

This section presents several case studies of actual student submissions and output produced by AbstractTutor. In the case studies, a description of the student's code, the Alignment Vector (AV), and actual output are shown. Additionally, I discuss the ideal Alignment Vector as well as feedback. Cases are presented where AbstractTutor correctly parsed the student code, as well as cases where AbstractTutor could not have provided correct algorithmic feedback to the student[4]. For comparison purposes, Appendix A contains reference solutions to each problem in AbstractTutor.

These case studies are meant to specifically highlight the code evaluation mechanism for the purposes of producing the algorithmic feedback as a computer science contribution of this thesis. Therefore, I focus specifically on the features of the code related to the generation of feedback, and not the knowledge or learning of the student.

5.4.1 Case Study 1: Failure to Access Array

A clear example of when output based feedback and algorithmic component feedback would evaluate student code differently is the case where the student does not access the array inside the loop structure as a part of the code submission. In the countLessThan problem, students are asked to count the number of positive items in the array that are less than a certain value. The student whose code is shown in Table 5.8 is not only missing an access of the array, but also a conditional statement to evaluate if each element is less than the value.

Although this participant's code did not compile, the parser was able to determine the appropriate alignment vector and provided the student with appropriate algorithmic component feedback, where the output based feedback would have been the compiler message shown.

5.4.2 Case Study 2: Incorrect Loop Bounds

Novice students often produce code that is incorrect in both expected and unexpected ways. The example shown in Table 5.9 demonstrates three errors in student reasoning, two common misconceptions, and a third unexpected error. The two common misconceptions are the use of = instead of == in the if statement, and the ability to return inside the loop without a default return on the outside. The unexpected error is the use of
`myList[i].getLength`

[4]There are case studies presented where they student saw the correct feedback for their condition (output based) despite an incorrect parse by AbstractTutor

Attempt	This was the third (3rd) problem completed by the user, and the code displayed was the first submit.
Start-End Time	21:58 - 28:43
Condition	Algorithmic Feedback

Code	`public int countLessThan(int []myList,`
	`                          int value){`
	`    int x;`
	`    for (x = 0; x < myList.length; x++){`
	`        value += 1;`
	`    }`
	`    return(myList.length - value);`
	`}`

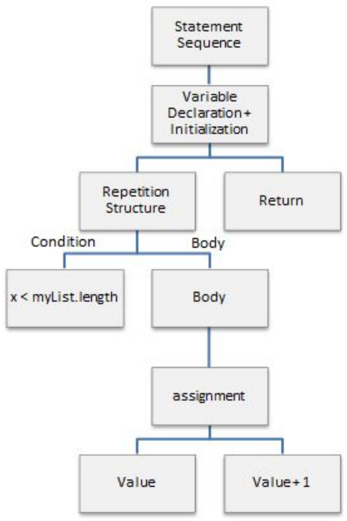
AV (Actual)	11001-00-00-110 Compile bit: 0
AV (Desired)	11001-00-00-110 Compile bit: 0
AST	 Statement Sequence Variable Declaration+ Initialization Repetition Structure Return Condition Body x < myList.length Body assignment Value Value+1
Output-Based Feedback	Undeclared Identifier value
Algorithmic Feedback (Given)	The loop you wrote supports accessing each element in your array (myList), yet you do not access the array myList within the loop. These components must work together for the problem to be solved.

Table 5.8: Case Study 1: Failure to Access Array

Attempt	This was the 4th problem completed by the participant, and their first submit for the problem.
Start-End Time	30:35 - 33:31
Condition	Output-Based Feedback

Code	`public int indexOf(int []myList,`
	`                  int value){`
	`    for(int i=0; myList[i].getLength; i++)`
	`        if(myList[i] = value){`
	`          return value;`
	`        }`
	`}`

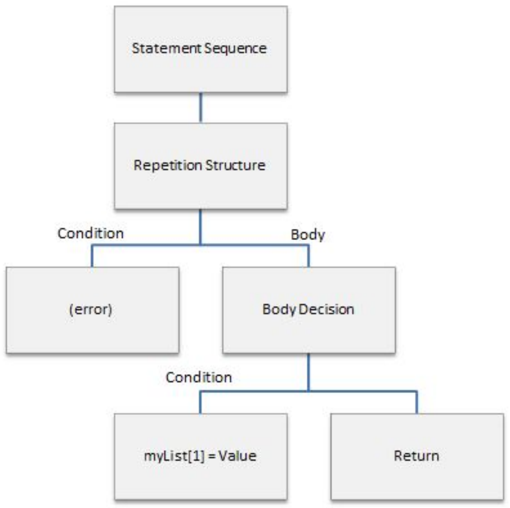
AV (Actual)	10101-10-00-100 Compile bit: 0
AV (Desired)	10101-10-00-100 Compile bit: 0
AST	 Statement Sequence Repetition Structure Condition Body (error) Body Decision Condition myList[1] = Value Return .
Output-Based Feedback (Given)	Your code did not compile and generated the following errors: Line 8: int cannot be dereferenced, 9: incompatible types found :int required: boolean
Algorithmic Feedback	The variable myList is a parameter. This means another part or parts of the program decide what is stored in it, and it may change every time code is run. Use a variable in the loop combined with a property of myList in order to determine the number of steps for the loop to take.

Table 5.9: Case Study 2: Incorrect Loop Bounds

as a loop bound without a comparison.

Rather then speculate about the student thought processes for each answer, I focus here on the parse and generation of feedback by AbstractTutor. The system correctly identifies the errors as they relate to the algorithmic components. In the algorithmic feedback condition, the student would have been prompted to look at the loop bound. Each feedback message has a secondary message if the student does not correct the problem, and the secondary message would have directed the student to "Use myList.length to determine the number of steps for the loop to take.", giving the student the example of how to access the length of the list. This particular student may have continued to struggle, however, as an error message was not prepared to focus on the lack of a comparison as a part of the loop bound.

5.4.3 Case Study 3: Mistaken Array Name

In the third case study shown in Table 5.10, the participant incorrectly used the name of the method (indexOf) instead of the name of the array (roomNumbers). AbstractTutor correctly parsed the student's code to indicate that it did contain a loop structure (for), but the code does not use the array parameter (roomNumbers) in order to determine the bounds for the loop. Although the code submitted by this participant generates the same feedback message as Case Study 2, the reason behind the message is quite different. The compiler error points the student at the first incorrect usage of "indexOf", the name of the method, instead of "roomNumbers" the actual name of the array. The student does not actually want to declare a variable for indexOf, which is the normal correction for a "cannot find symbol" error message. The ability for the system to generate messages directing the student to the correct location of the code to be modified despite very different mistakes, is a strength of the constraint-based approach to analyzing the student code.

5.4.4 Case Study 4: Failure to Parse

AbstractTutor did have a special case that caused the parser to fail with reasonably well formed code submissions, as seen in Table 5.11. Although the abstract syntax tree parser is robust to many compile errors, making it possible to generate pre-compilation feedback, it still relies on the code submitted to contain cues as to the structures for the syntax tree.

In the Table 5.11 example, the code does not contain any code that compiles after the for statement. Without any code inside of the loop, the parser could not correctly determine what was a part of the loop and what was not. AbstractTutor then generated a message beginning with "Your code could not parse for better feedback. The compiler says: " and included any compiler messages the system returned. Although this is a failure to generate the appropriate algorithmic feedback, the student still received the same feedback he/she would have gotten outside of AbstractTutor; and often by correcting the syntax errors mentioned in the compiler feedback, the student will also be fixing the code to the point where a correct parse could happen.

Additionally, the example in Table 5.11 also exposes a weakness in the AbstractTutor evaluation. Although the student does use myList.length to access the length of the array,

Attempt	This was the 4th problem seen by the participant, and the code represents the second submit for the problem.
Start - End Time	33:23 - 35:53
Condition	Ouput-Based Feedback

Code

```
public int indexOf(int []roomNumbers,
                int idno){
    for(int i=0; i<indexOf.length; i++){
         if(idno == indexOf[i])
            return idno;
          else
             return -1;
     }
 }
```

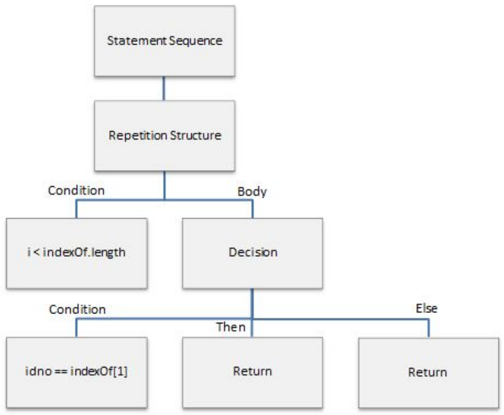
AV (Actual)	10001-10-00-100 Compile bit: 0
AV (Desired)	Compile bit: 0
AST	 Statement Sequence Repetition Structure Condition Body i < indexOf.length Decision Condition Then Else idno == indexOf[1] Return Return
Output-Based Feedback (Given)	Your code did not compile and generated the following errors: Line 9 cannot find symbol, symbol: variable indexOf
Algorithmic Feedback	The variable myList is a parameter. This means another part or parts of the program decide what is stored in it, and it may change every time code is run. Use a variable in the loop combined with a property of myList in order to determine the number of steps for the loop to take.

Table 5.10: Case Study 3: Mistaken Array Name

Attempt	This code is the participants' first attempt at the first problem.
Start - End Time	2:00 - 6:57
Condition	Algorithmic Feedback

Code	`public int findSum(int []myList){`
	`        for(int index=0;`
	`            index > myList.length; index++)`
	`        int total`
	`        total+=myList[index]`
	`    }`

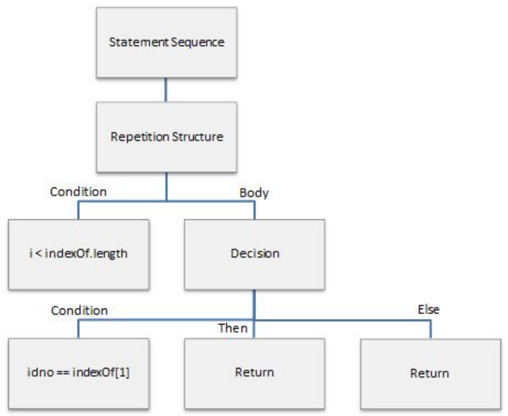
AV (Actual)	00000-00-00-000 Compile bit: 0
AV (Desired)	11111-00-00-000 Compile bit: 0
AST	 Statement Sequence Repetition Structure Condition Body i < indexOf.length Decision Condition Then Else idno == indexOf[1] Return Return
Output-Based Feedback	9: '.class' expected, 9: not a statement, 10: ';' expected
Algorithmic Feedback (Given)	Your code could not parse for better feedback. The compiler says: 9: '.class' expected, 9: not a statement, 10: ';' expected

Table 5.11: Case Study 4: Failure to Parse

the comparison with index uses the wrong relational operator (>instead of <). AbstractTutor does not currently check for these logical errors in the recompilation testing, and the student would need to rely on feedback from the output based testing to determine the source of error in their code.

5.4.5 Case Study 5: No Declaration of State Variable

The student code submission shown in Table 5.12 illustrates a dependency in the Alignment Vector and AST that is useful to present for clarity. In the code, the student did not declare the type of the variable maximum; it should be preceded with the type int. As a result, the AST parser for AbstractTutor did not give the student credit for correctly declaring and initializing the state variable (the second 0 in the AV). Although the student did use maximum inside the if statement in an appropriate way (to maintain state through the algorithm), because it was not correctly declared, AbstractTutor will not give credit in the AV for the update (the third 0).

Although this decision can be applauded for awarding, or noting the absence of, the state update part of the AV, the dependance upon the proper initialization has greater benefit with fewer adverse effects. First, the student was in the algorithmic feedback condition and received a message about the declaration or assignment of maximum, which was similar to message a student in the output based condition would have seen from the compiler error focused on that problem. In the system, the student corrected the error on the next submission, and the AV was updated to include both the initialization as well as the update components, thereby producing no possible error in the feedback messages communicated to the student from AbstractTutor. The only detriment of this approach is the sequential evaluation of the student submits as it appears he corrected two AV components on the single submit instead of just one.

5.4.6 Case Study 6: Bad Variable Update and Return

The code and feedback presented in Table 5.13 is intended to highlight the different ways that a compiler and AbstractTutor handle undefined variables. In the code submission, there are no definitions for any of the local variables used (i or count). Since a general compiler is reactive in nature, it reacts to an error in the code once encountered so it can only say that count is undefined the first time it encounters the variable (on line 10)[5].

Because AbstractTutor has knowledge of the problem the student is trying to solve, it reacts before seeing count in the loop. Instead, it identifies that the code should have a variable defined before the for statement and will respond with feedback stating that a state variable was not declared or initialized properly. Had the participant not corrected the error before the next submission, AbstractTutor would say: "Create and initialize a variable before the loop." providing a clear and explicit direction for the correction of the code.

[5]The code in the computer window included the comments about the problem and therefore had more lines than appears in the case study tables.

Attempt	This code is the participants' fourteenth attempt at the second problem.
Start - End Time	48:03 - 54:02
Condition	Output Based Feedback

Code	`public int findMaximum(int []vehicleMPG){`
	`  maximum = vehicleMPG[0];`
	`  for(int i=0; i<vehicleMPG.length; i++){`
	`    if(vehicleMPG[i] > maximum){`
	`      maximum = vehicleMPG[i];`
	`    }`
	`    return maximum;`
	`  }`
	`}`

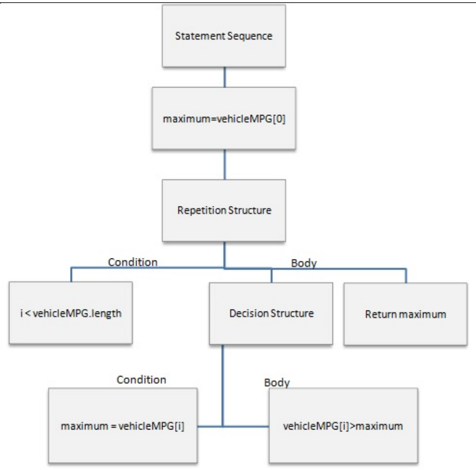
AV (Actual)	11111-11-00-100 Compile bit: 0
AV (Desired)	11111-11-00-100 Compile bit: 0
AST	 Statement Sequence maximum=vehicleMPG[0] Repetition Structure Condition Body i < vehicleMPG.length Decision Structure Return maximum Condition Body maximum = vehicleMPG[i] vehicleMPG[i]>maximum
Output-Based Feedback	9: Your code did not compile and generated the following errors: Line 7: Cannot find symbol, Symbol: variable maximum
Algorithmic Feedback (Given)	In order to solve your problem you need to maintain a particular state in a variable. This state will give you the answer you are seeking. You have not correctly created or assigned a starting value to the state variable.

Table 5.12: Case Study 5: No Declaration of State Variable

Attempt	The code presented here is the participant's fourth attempt at the second problem.
Start - End Time	4:29 - 4:55
Condition	Output-Based Feedback

Code	
	```
	public int countLessThan(int []myList,
	        int value){
	        for(i=0; i<myList.length; i++){
	          if(myList[i]<value)
	            count++;
	            else return 0;
	        }
	    }
	```

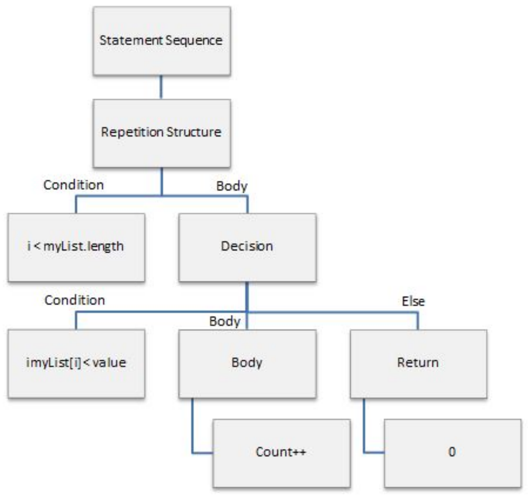
AV (Actual)	11111-11-00-100 Compile bit: 0
AV (Desired)	11111-11-00-100 Compile bit: 0
AST	 Statement Sequence Repetition Structure Condition Body i < myList.length Decision Condition Else Body imyList[i]<value Body Return Count++ 0
Output-Based Feedback (Given)	Your code did not compile and generated the following errors: Line 8: cannot find symbol, symbol: variable i, 10: cannot find symbol, symbol: variable count
Algorithmic Feedback	In order to solve your problem you need to maintain a particular state in a variable, this state will give you the answer you are seeking. You have not correctly created or assigned a starting value to the state variable.

Table 5.13: Case Study 6: Bad Variable Update and Return

5.4.7 Case Study 7: Alternative Looping

Although the reference solutions to the problems presented in Appendix A use a for structure to loop over the array, it is possible to correctly solve the problems using a while structure as seen in the example in Table 5.14. This participant was in the output based feedback condition, and although the abstract syntax tree did not recognize the looping structure, the student received feedback that she was correct because the code passed all of the output-based testing. In recognition that students will create solutions that are unique, the AbstractTutor system will allow a student to receive credit for, and advance past a problem that passes all of the output-based testing, even if it is missing a model state. The system was modified after this example was collected to recognize while structures in addition to for structures as a looping mechanism.

5.4.8 Case Study 8: Loop Variable Undeclared

The example shown in Table 5.15 is interesting for two reasons. First, although there are errors within the for statement (not declaring i), the loop and subsequent code parsed correctly. The student in the output based condition would be focused on declaring i to be a variable, at which point the code would execute and an infinite loop would ensue.

In the algorithmic feedback condition, the student would be prompted to use the array (myList) and its length in order to control the number of executions of the loop. Even if the student did produce an infinite loop, AbstractTutor has a time out feature. If the code requires more than 10 seconds to run any individual test case (with a maximum of 20 numbers), the student receives a message indicating that an infinite loop may be present. The feedback regarding infinite loops is provided to students in both the algorithmic and output based feedback conditions as a usability issue. If a web-based system just does not return any feedback, it is difficult to know if the system or the network has a problem. The infinite loop feedback is provided to students so that they can fix the error and continue to work with code that will provide output based feedback where appropriate.

5.4.9 Case Study 9: No Semicolon Before Loop

Case Study 9 shown in Table 5.16 highlights another special case of an error that causes the parser to fail. The student's code is reasonably well formed, however it is missing a semicolon on the line before the for statement. A semicolon is an indicator of a statement to the java compiler, and as such the parser assumes the for statement is a part of the assignment made on the line before. Ideally, the above code would generate an algorithmic feedback message asking the student to use the length of the array as a bound for the for loop (the student is currently only using vehicleMPG instead of vehicleMPG.length).

Although the failure to parse in this instance could be an issue, the system is designed for fast recovery. The student receives the compiler message that a semicolon is missing (the same feedback as the output based condition) and if the error is fixed, the code will parse correctly and algorithmic feedback will resume. When the parser fails, the student is no worse off than the current standard of output based feedback; and as each submission

Attempt	The code presented here is the participant's second attempt at the first problem. The first attempt had ( ) instead of [ ] for the array access.
Start - End Time	27:31 - 31:25
Condition	Output-Based Feedback

Code

```java
public int findSum(int []myList)
{
     int x = 0;
     int sum = 0;
     while(x < myList.length) {
          sum += myList[x];
          x++;
     }
     return sum;
}
```

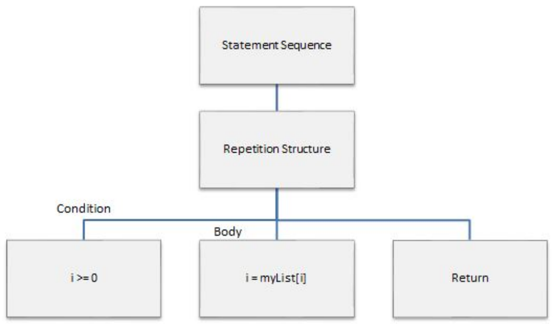
AV (Actual)	01111-11-00-111 Compile bit: 1
AV (Desired)	11111-11-00-111 Compile bit: 1
AST	 Statement Sequence Repetition Structure Condition Body i >= 0 i = myList[i] Return
Output-Based Feedback (Given)	Correct
Algorithmic Feedback	Correct

Table 5.14: Case Study 7: Alternative Looping

Attempt	The code presented here is the participant's first attempt at the third problem.
Start - End Time	52:57 - 63:19
Condition	Algorithmic Feedback

Code	
	```
	public int countLessThan(int []myList,
	                int value)
	{
	      for(i=0;i>=0;i++){
	           i=myList[i];
	      }
	      return i;
	}
	```

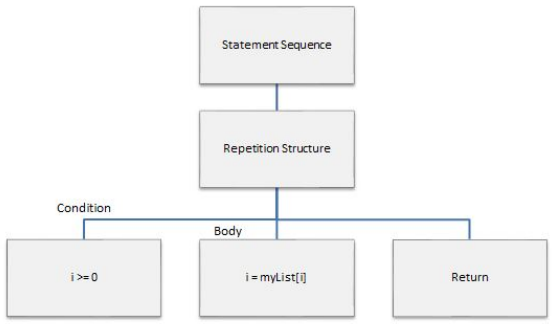
AV (Actual)	10111-00-00-110 Compile bit: 0
AV (Desired)	10111-00-00-110 Compile bit: 0
AST	 Statement Sequence Repetition Structure Condition Body i >= 0 i = myList[i] Return
Output-Based Feedback	Your code generated the following syntax error: Line 6, Column 30: Expression "i" is not an rvalue
Algorithmic Feedback (Given)	The variable myList is a parameter. This means another part of parts of the program decide what is stored in it, and it may change every time code is run. Use a variable in the loop combined with a property of myList in order to determine the number of steps for the loop to take.

Table 5.15: Case Study 8: Loop Variable Undeclared

Attempt	The code presented here is the participant's third attempt at the second problem.
Start - End Time	35:02 - 35:18
Condition	Algorithmic Feedback

Code	`public int findMaximum(int []vehicleMPG){`
	`     int temp = 0`
	`     for(int i = 0; i < vehicleMPG; i++) {`
	`          if(vehicleMPG[i] > temp){`
	`               temp = vehicleMPG[i];`
	`           }`
	`      }`
	`      return temp;`
	`}`

AV (Actual)	00000-00-00-000 Compile bit: 0
AV (Desired)	10111-11-11-110 Compile bit: 0
AST	The code did not parse so no tree was created.
Output-Based Feedback	Your code generated the following syntax error: Line 7, Column 44: Operator ";" expected
Algorithmic Feedback (Given)	It appears you have a poorly structured for loop or a syntax error. Unfortunately in this version of the system we are unable to support for-each loops. Please reformat your loop and resubmit. Your code generated the following syntax error: Line 7, Column 44: Operator ";" expected

Table 5.16: Case Study 9: No Semicolon Before Loop

is evaluated independently, a failure only affects the current submission. The student in the case study took three edits to correct the missing semicolon; and once the semicolon was added, the parser evaluated his code correctly.[6]

5.4.10 Case Study 10: Compile but Algorithmically Wrong

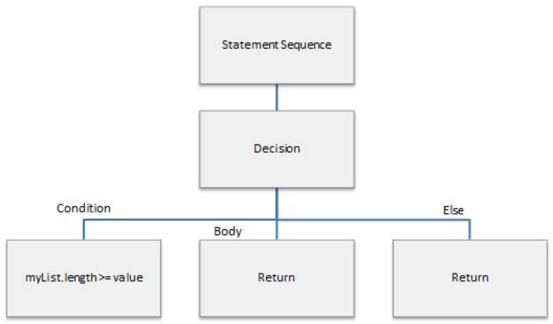
The submission shown in Table 5.17 demonstrates an example of a student who has produced compiled code that illustrates either a significant misunderstanding of what algorithm the question is asking for, or a weakness with Java semantics. The student completed questions 1-3 in two submits each, fixing only a minor syntax error before moving on, implying a level of expertise with the content. For question 4, however, the student did not even attempt to loop over the elements in the array. The student submitted similar code (only an if statement checking various properties of the list against the parameter value) however gave up after 10 submissions - moving to the next problem without correctly solving the indexOf problem. Seven (7) of the submissions included output based feedback with numerical values.

This case study not only highlights a successful evaluation by the parser, but also an instance where the output based feedback did not provide adequate information to even a relatively sophisticated novice about the code he produced.

5.4.11 Case Study 11: Compile but No Array Access

Case Study 11 shown in Table 5.18 again shows a student solution that compiles but is not correct. The student did not access the array, named items, inside of the for loop, therefore creating a situation where he could not possibly calculate the correct answer. This case is another example of very different feedback from the output based and algorithmic feedback conditions. Although the student fixed the error on the next edit, she submitted code in the next two problems with errors around the access of a single element from the array inside the loop. The AST generated here shows how an evaluation of the individual states can be useful in identifying common student problems and providing appropriate information to the system designers to refine feedback.

[6]A recommendation for future versions of AbstractTutor include line numbers displayed to the side of the code editor so students can more easily identify syntax errors. Although the student received the missing semicolon error, he made two edits in other locations before adding the semicolon in the correct place. He had initially left a semicolon off the line temp=vehicleMPG[i] as well.

Attempt	The code presented here is the participant's 10th attempt at the fourth problem.
Start - End Time	35:02 - 35:18
Condition	Algorithmic Feedback
Code	`public int indexOf(int []myList, int value){ if(myList.length >= value){ return value; }else{ return -1; } }`
AV (Actual)	00000-00-00-100 Compile bit: 1
AV (Desired)	00000-00-00-100 Compile bit: 1
AST	
Output-Based Feedback (Given)	Your indexOf method returned the wrong value when I passed it [23, 51, 46, 74, 57, 22, 45, 79, 25, 61, 31, 13, 21, 52, 93, 36, 22, 17, 97, 50]and a value of 36. Your method should have returned 15 but it returned -1
Algorithmic Feedback	In order to accomplish the task assigned, you need to access every element in the array. Try to implement (write code for) a control structure that allows you to repeat a task over many times. You will use this with other variables in order to complete your task. You also need to write at least one line of code inside the structure.

```
public int indexOf(int []myList,
                int value){
     if(myList.length >= value){
          return value;
      }else{
          return -1;
     }
}
```

Table 5.17: Case Study 10: Compile but Algorithmically Wrong

Attempt	The code presented here is the participant's third attempt at the first problem.
Start - End Time	35:02 - 35:18
Condition	Algorithmic Feedback

Code

```
public int findSum(int []items){
     int sum = 0;
     for (int i = 0; i < items.length;
                                 i++){
          sum += sum;
     }
     return sum;
}
```

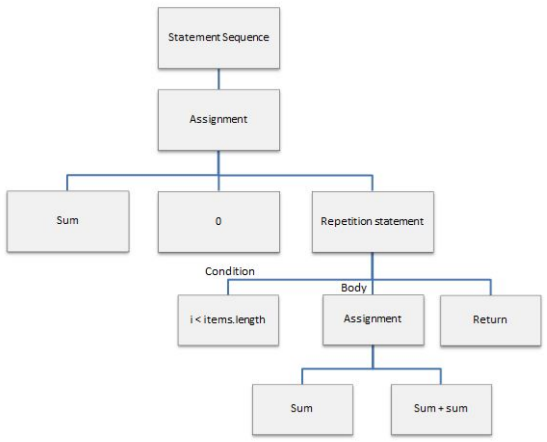
AV (Actual)	11001-00-11-110 Compile bit: 1
AV (Desired)	11001-00-11-110 Compile bit: 1
AST	 Statement Sequence Assignment Sum 0 Repetition statement Condition Body i < items.length Assignment Return Sum Sum + sum
Output-Based Feedback (Given)	Your findSum method returned the wrong value when I passed it [2, 92, 4, 28, 51, 84, 22, 87, 41, 36, 68, 42, 66, 21, 85, 68, 23, 34, 18, 14]: Your method should have returned 886 but it returned 0
Algorithmic Feedback	The loop you wrote supports accessing each element in your array (items), yet you do not access the array items within the loop. These components must work together for the problem to be solved.

Table 5.18: Case Study 11: Compile but No Array Access

5.5 Conclusions

To be effective, a feedback mechanism must reliably provide students with feedback specific to the submission they have entered. For this thesis, I postulate that an 85% accuracy rate of algorithmic feedback for the AbstractTutor system will be a contribution. The 85% represents over double the possible accuracy of the current standard of output based testing as measured in the think aloud data. Additionally, in the instances where AbstractTutor is unable to provide the appropriate algorithmic based feedback, it defaults to displaying the compiler message for the code submitted. This feature makes AbstractTutor equivalent to the output based feedback in the instances where the specialized algorithmic feedback mechanism fails.

In order to determine the overall accuracy of the algorithmic feedback mechanism, the large online study data was used. Of the 4730 submissions made to the system, 869 were classified originally as a failure to parse, therefore preventing AbstractTutor from displaying the appropriate algorithmic feedback. From the 869 failures, 98 of the failures were due to the use of an enhanced for loop, a feature not supported at the time by the Janino framework, but a feature that has since been added. Additionally, 25 of the failures were due to the use of a while loop, which has been corrected in the implementation of the feedback mechanism and tested in the final study detailed in Chapter 7. This leaves us with 749 (15.8 % of all submissions) instances where the system failed to parse the code submitted.

Although not quite passing the 85% threshold, many of the submissions made to the system were sequential duplicates - the student pressed the submit button multiple times without making a change to the code. In the think aloud, this phenomenon was often ascribed to two particular cases. First, the output based feedback (unit tests) used a random set of generated numbers, therefore repeat submissions will provide the user with additional data regarding the code. Also, students may press the submit button multiple times if they believe their code is correct but that there is a flaw in the system - especially if there is a lag in the web browser. One particular student in the large online study made 50 sequential submissions within a short period of time. Removing just 49 of those 50 identical submissions gives us 700 instances where the system failed to parse the code out of 4681 submissions a 14.6% accuracy rate. Although the 14.6% accuracy rate is not exact as there are other instances of duplicate code submission, I offer this result as indication that the 85% threshold is approached, if not crossed in a large diverse dataset.

In this chapter, I have shown the underlying mechanism for the production of algorithmic feedback and used multiple data sets to estimate the accuracy of the feedback on actual student code submissions. The unique combination of a representation of the algorithmic components in an Abstract Syntax Tree, combined with output based testing has yielded an accuracy rate providing the possibility for algorithmic component feedback during novice practice. Overall, the tutor feedback mechanism performs significantly better than is possible by output based feedback, and it results in not only the opportunity for more specific feedback for the novice, but also for the evaluation of the individual parts of the student solution for each submission. In the next chapter, I detail how the use of the analysis of each submission for individual algorithmic components can provide a more robust measure of student progress across submissions, within problems, and even across problems in the AbstractTutor system.

Chapter 6

Quantitatively Understanding Student Coding Attempts

6.1 Problem Solving in Complex Spaces

In Chapter 3, I detailed a model of algorithmic components to be used to evaluate student code for the purpose of providing feedback. Chapter 4 detailed the results of a think aloud study to validate those model components and highlight their usefulness in evaluating student code. The work presented in Chapter 5 analyzed the implementation and accuracy of the feedback mechanism, and the importance of the algorithmic model to that accuracy. In this chapter, I offer an analysis of methods for evaluating student coding attempts to determine student proficiency. I postulate that "correctness on first attempt", or "number of submits for a problem", are too course grained to use for analyzing student problem solving in complex multi-attempt spaces. Instead, I offer two metrics that are used in this thesis to evaluate student problem solving in a programming environment. A portion of this work was published in [81] with Thomas K. Harris and Kelly Rivers.

6.1.1 Granularity of Current Methods

Current methods of evaluating student progress in tutoring or practice systems focus on student *attempts* as a means of progressing towards *correctness*. In AbstractTutor, students are focused on the production of multipart algorithms and often require feedback about various parts before arriving at a correct solution. Additionally, as discussed in Chapter 2, current feedback mechanisms require the novice to extrapolate the connection between an error and the source of the error in code. Student attempts therefore have been shown to represent a cycle of compilation (feedback seeking) and edits [34]. Although the word edit implies a thoughtful change, it has also been shown that students engage in "shotgun debugging"[1] and in this Chapter I demonstrate that students also engage in multiple

[1]Shotgun debugging is similar to Gaming the System [7] where students attempt to achieve correctness through random edits. Each edit represents a guess as opposed to a thoughtful change. For example, a student may just add } characters in random locations in the code hoping for a successful compilation.

duplicate submissions.

The combination of multiple expected edits and duplicate submissions (41% in a large online study) offer an initial warning that each submit may not represent an attempt at a thoughtful change and therefore a raw accounting of attempts may not represent an appropriate granularity for measuring within problem process or proficiency.

In this section, I offer data from a large online study as supporting evidence of insufficient granularity of "attempt only" or "correctness" metrics in understanding student proficiency. The large online study has been referenced in Chapter 5, and learning outcomes and recruitment details will be presented in Chapter 7. For the next sections, we will use only the users who completed all four problems in the system. There were 73 users, out of 160, who correctly answered all four problems in the system.

6.1.2 Attempt Only Metrics

In the cognitive tutoring literature, it has been demonstrated that students sometimes attempt to "game the system" by guessing or "hint-mining" [7]. This behavior has been shown to result from a dislike for the subject, a lack of educational self-drive, and frustration [8]. In many cases, the set of appropriate tokens for solutions to academic tutoring software limit the potential search space and students are able to arrive at a correct solution through a guess and check process.

In complex, multipart problems, like the production of code to implement a computer algorithm, the solution space is not easily constrained, and therefore a guess and check strategy is often unproductive without minimum proficiency in the domain. In the large online study, the minimum number of submits to solve all four problems was four (one submit per problem). No user was able to solve all four problems in four submits, however one user managed 5 submits. The maximum number of submits to solve all four problems was 315. The difference between the mean (34.6) and the median (17) number of submits indicate a number of outliers on the high end. Most students (40) managed to complete the four problems in under 20 submits. The students who employed guess and check strategies are the likely outliers, as demonstrated in the sections below.

6.1.3 Raw Correctness on First Attempt

In many tutoring systems, each submission is an opportunity to fully answer a problem, or take a step in a larger multi-step problem being solved. These opportunities are often of sufficient granularity that a user who has mastered the concept can construct a correct answer on his first attempt. If a correct answer is not submitted, in most cases there exists feedback (or hints) that are designed to help the user reach a correct solution within one submit, as opposed to guiding the student through repeated edits.

The construction of a multi-part algorithm is often a multi-step process, even for an expert. Because of the formalisms required in writing computer code, as described in Chapter 1, even experts use tools such as IDEs or editors to check the code they write for errors. Due to the complexity of constructing algorithms, novices also use tools to provide feedback during the algorithm construction process. It would be unrealistic to assume that the first attempt of a novice is able to be scored as completely correct or incorrect and provide appropriate information about learning over multiple problems. Even among proficient novices, the probability of a completely correct answer is very low on first submission [53]. Even within this thesis, I have replicated those results in Table 4.5. Table 4.5 shows a breakdown of participants in the Think Aloud study and the percentage of correct components in their first submits. Overall, participants had a first submit average of 70%.

6.1.4 Two Metrics for Assessing Student Submissions

For the purposes of this thesis, I am especially interested in the appropriate algorithmic components necessary to produce algorithms to solve the array problems presented to students. Although the correction of a missing semicolon demonstrates an understanding of the required syntax of the programming language, it does not represent a modification to improve an incorrect algorithm. I therefore offer two metrics to focus specifically upon edits that modify the alignment vector corresponding to student submitted code, both of which better reflect the state of the algorithm.

First, I explore the probability of an algorithmically productive edit. This metric is meant to be a singular snapshot, similar to the metric used in many intelligent tutoring systems of "correct on next submit" [18]. Although the algorithmically productive edit does not require a full, complete correct answer, it can be thought of as completing a part of a multi-step problem. I will detail the metric and provide some initial data analysis with submits from the think aloud study presented in Chapter 4.

Second, I look beyond the single snapshot with Probabilistic Distance to Solution (PDS). Unlike the single attempt analysis of probability of an algorithmically productive edit, PDS is a metric designed to encapsulate the entirety of the student's problem solving over the whole problem. Since publication in 2012, the paper in which this first appears has been cited 6 times in relevant literature reviews.

6.2 Probability of Algorithmically Productive Edit

Intelligent tutoring systems have used probabilistic measures to assess student learning and within-problem progress for numerous years. Multistep problems have been a feature of many complex tutoring systems, however practice involving tutoring in the construction of algorithms with the writing of large segments of code offers a new challenge. Although the construction of an algorithm can be seen as a multi-step problem, novices rarely take a linear approach to the writing of a program or the acquisition of feedback regarding the solution they are writing.

Although the non-linearity may be an artifact of the feedback mechanism with which the students are familiar, it nonetheless exists as a phenomenon. Students will often produce what they believe to be a complete (if not correct) solution before engaging with the feedback mechanism. And, in the production of algorithms where multiple interacting parts make stepwise refinement difficult, this approach is not dissimilar from the expert.

Finding the Sum
```
public int findSum(int []myList){
      int sum = 0;
      for(int i=0; i < myList.length; i++){
         sum = sum + myList[i];
      }
      return sum;
  }
```

```
public int findSum(int []myList){
      int sum = 0;
      for(int i=0; i < myList.length; i++){
         sum = sum + myList[i];
      }
      return sum;
  }
```

Table 6.1: Finding the Sum of the Numbers

Consider the algorithm to find the sum of the numbers in a list shown in Table 6.1. As an expert, if I was forcing myself to do stepwise refinement, I may declare a sum variable, initialize it to 0, and then return that variable to ensure accuracy of my code. For step two, it would be difficult to do anything other than produce the for loop to iterate over the list and update the sum variable as appropriate. The alignment vector for this problem checks for 11 distinct algorithmic components as defined in Chapter 3, and as an expert I still would have difficulty breaking the problem solving process into more than two submits. This fact implies that each individual submit requires consideration of multiple algorithmic components.

Also, since students often construct a full attempt at the problem solution before pressing submit, it means that they will make edits on multiple parts of the problem in order to attempt to correct a faulty solution. Measuring edits and effectively using that measurement to infer knowledge or competency cannot be as straightforward as looking for the "next correct step" in a single edit.

6.2.1 Code Classifications

In order to look at the effect of a single edit, I propose a classification of edits to allow for focus on algorithmic components before syntax, and the ability to discount multiple submissions. Because of the desirability of automation, as these metrics may eventually be applied for real-time evaluation of student knowledge, it is also important that the classification be able to be assigned with accuracy by a computation and not by a human rater. I therefore propose a classification of five distinct types of submits from a user: First Attempt, Duplicate Submission, Algorithmically Counterproductive Edit, Algorithmically Neutral Edit, and Algorithmically Productive Edit. *First Attempt* is categorized as its own label because we are unable to judge a modification until we have received an original submission.

I label a code submission that does not produce any change to the code as a *Duplicate Submission*. A segment is labeled Duplicate Submission (DS) when the code submitted is the same (ignoring white space - extra returns, etc.) as the previous submission. A submission will NOT be marked DS if the code matches *any* prior submission, only if it

Num	Code Submission	Label

1	`for(int i=0; i<scores.length; i++){`
	`if(scores[i]>=0){`
	`if else{`
	`return sum;`
		Not DS

2	`for(int i=0; i<scores.length; i++){`
	`if(scores[i]>=0){`
	`if else{`
	`return sum;`
		DS

3	`for(int i=0; i<scores.length; i++){`
	`if(scores[i]>=0){`
	else{
	`return sum;`
		Not DS

4	`for(int i=0; i<scores.length; i++){`
	`if(scores[i]>=0){`
	else if{
	`return sum;`
		Not DS

5	`for(int i=0; i<scores.length; i++){`
	`if(scores[i]>=0){`
	else{
	`return sum;`
		Not DS

Table 6.2: Duplicate Submission Examples

matches the submission immediately preceding the submission being labeled. Although an argument could be made for including any matching submission, students may arrive at a prior submission after an unproductive edit without realizing the two submissions are the same. Table 6.2 shows a sequences of code from a participant and marks each submission as DS or not DS. The user was solving the countLessThan problem and for space constraints the initial code stub is left out of the submission data.[2] Although submission number 5 is the same as 3, it is not marked as a Duplicate Submission because there is an edit in between the two submissions.

The second category of submission is an *Algorithmically Counterproductive Edit.* An Algorithmically Counterproductive Edit (ACE) is an edit that results in a submission having an alignment vector with fewer correct algorithmic components than the previous edit. A counterproductive edit is indicative of an error being introduced as the student is attempting to process and address feedback. Table 6.3 has examples of ACE and other edits from a participant. Table 6.3 shows and example of an Algorithmically Counterproductive Edit in a series of participant submissions. In the table, code submission number 4 demonstrates the ACE. The student is struggling with the appropriate way to access an individual item in the array. He mistakenly believes that .length is required (possibly because of the items.length used in the for statement) and tries different combinations of using the [i] index to get to a singular item.

The *Algorithmically Neutral Edit* (ANE) does not match the previous submission (DS), but does not result in a change to the alignment vector from the previous submission. Although students may make a productive edit by correcting a compiler error, this would not be considered a change to the algorithm except in a few instances where the compiler error clarified the structure of the algorithm. The exceptions occur when the code is initially unparsed, and the edit modifies the code structure so that an Abstract Syntax Tree (AST) can be constructed, or if the code does not compile, and the edit allows for compilation and passage of output based testing. In both exception cases, the edit is not focused on a specific algorithmic component, however would result in a change to the alignment vector.

The fifth category of submission is *Algorithmically Productive Edit.* An Algorithmically Productive Edit (APE) results in a change to the alignment vector from the previous submission, indicating that the student made an edit that added algorithmic correctness to the submission. The APE is the most desired of all edits. Table 6.4 shows a series of student submissions with alignment vectors and appropriate labels. The submissions were for the findMaximum method and the common code stub and method header are removed for space purposes.[3]

[2]The method header was public int countLessThan(int []scores, int passingGrade)

[3]The method header was public int findMaximum(int []vehicleMPG).

Num	Code Submission	AV	Label

1	`{int sum=0;`
	`for(int i=0; i<items.length; i++){`
	`sum += items.length;}`
	`return sum;`
	`}`
		11001-00-11-110	ANE

2	`{int sum=0;`
	`for(int i=0; i<items.length; i++){`
	`sum += items.length;}`
	`return sum;`
		11001-00-11-110 All code is the same.	DS

3	`{int sum=0;`
	`for(int i=0; i<items.length; i++){`
	`sum += items[i].length;}`
	`return sum;`
		11111-00-11-110 Participant has referenced any element from the list with [] and used the loop variable (i) in that reference.	APE

4	`{int sum=0;`
	`for(int i=0; i<items.length; i++){`
	`sum[i] += items.length;}`
	`return sum;`
		11001-00-11-110 Participant has removed the [i] from code.	ACE

Table 6.3: Algorithmic Counterproductive and Neutral Edit Examples

Num	Code Submission	AV	Label
1	`{int max = 0;` `for(int i=0;i<vehicleMPG.length;i++){` `if(vehicleMPG[i]>max){` `max=vehicleMPG[i];` `}else{max=max;}`	00000-00-00-000 Code does not parse, missing .	FA
2	`{int max = 0;` `for(int i=0;i<vehicleMPG.length;i++){` `if(vehicleMPG[i]>max){` `max=vehicleMPG[i];` `}else{max=max;}}`	11111-11-11-000 Code will now parse, and AV is correct. Only missing return.	APE
3	`{int max = 0;` `for(int i=0;i<vehicleMPG.length;i++){` `if(vehicleMPG[i]>max){` `max=vehicleMPG[i];` `}else{max=max;}}`	11111-11-11-000 Still no return.	ANE
4	`{int max = 0;` `for(int i=0;i<vehicleMPG.length;i++){` `if(vehicleMPG[i]>max){` `max=vehicleMPG[i];` `}else{max=max;}}` `return max;`	11111-11-11-110 Student adds return, but code does not perform as expected with negative numbers.	APE
5	`{int max = -1000000;` `for(int i=0;i<vehicleMPG.length;i++){` `if(vehicleMPG[i]>max){` `max=vehicleMPG[i];` `}else{max=max;}}` `return max;`	11111-11-11-111 Solution is now correct.	APE

Table 6.4: Algorithmically Productive Edit Examples

6.3 Probabilistic Distance to Solution

In section 6.2, I focused on a single submission granularity, explaining a categorization scheme used on each individual code submission made by the participant. In this section, I focus on a metric, Probable Distance to Solution, that is used to look at the entire problem solving process of a student, from initial starting code stub to final solution.

6.3.1 Complex Problem Solving Spaces

Complex problem spaces are problems with multiple components where there is not an exact ordering or prescribed process for completing each individual step. In complex problem spaces like the problems in my studies, novices will often attempt several unique edits and approaches in order to create a finished correct solution. The model of algorithmic components is used by AbstractTutor to generate feedback for student attempts based on desired components in a solution. The codification of student submissions using the algorithmic components can be seen as an opportunity to apply data mining or classification techniques.

Modern data mining and classification techniques allow for increasingly complex solution spaces to be automatically modeled and assessed. For example, automated essay grading [87], mathematical proofs [35], and even complex computer programs [33] can be analyzed for completeness and scored. Although the models, feedback strategies and mechanisms used by each domain vary, it is still important for researchers to assess students' progress and make comparisons between research conditions in order to refine and improve such pedagogical systems.

In complex problem solving spaces, such as natural language production or computer programming, students may make edits or submit attempts that are not directly related to the specific learning outcomes of the tutoring task [34]. For example, in computer programming, a student may struggle with a compilation error, such as having a parenthesis out of place, which is not reflective of their understanding of the desired learning goal. Students may also produce submissions that progress through multiple skills, creating a complex path to solution, with many possible states [77]. In this thesis, the focus of the feedback and analysis is on the algorithmic components necessary to construct simple array algorithms. The placement of semicolons and parenthesis, although possible learning objectives for somewhere in the course, represent skills that are not the central learning outcomes. Therefore, counting additional submissions where students are correcting typos will artificially introduce variability in the measurement of student progress.

6.3.2 Other Within-Problem Set Performance Metrics

Measures of performance focused within the problem solving activity are sometimes used by tutoring systems instead of running pretests and posttests outside of the tutor, when the creator of the tutor wants more immediate learning feedback. Some within-tutor metrics have already been created and used effectively; for example, number of submissions and amount of time taken to get to a correct state were used in a system focused on improving math scores [26]. These metrics are not as effective in complex problem solving domains where the solution contains multiple parts produced simultaneously, due to the variety of strategies used to solve problems and the difficulty of merging them [43].

Other Intelligent Tutoring Systems use constraint-based modeling to determine how well a program matches the expectations of the problem; for example, Mitrovic built an ITS for SQL that used over six hundred constraints to provide accurate and useful hints to students [54]. Le and Menzel also describe techniques for building constraint-based tutors in 'ill-defined domains', similar to the complex domains described [46]. However, both of these approaches require that the author of the ITS generate the constraints by hand, which becomes very time-consuming when applied to a broad domain (such as programming). The metric we propose aims to improve on these models by examining more fine-grained aspects of the problem states.

In this section, I detail a new metric, Probabilistic Distance to Solution (PDS) and describe its implementation in assessing student progress in the AbstractTutor problems. We then apply this metric to the think aloud dataset described in Chapter 4 and highlight cases where PDS offers additional insight into misconceptions and problem solving paths.

For this thesis, the metric is applied after the full data set is collected in Chapter 7, as a way of analyzing the difference between students in different research conditions. For potential future work, the metric can be automated and used in real time to provide additional feedback to students about probable progress through the problems.

Traditional Measures of Performance

Before applying PDS to the think aloud data, a description of the traditional measures of student performance in the pedagogical system is offered to provide clarity and contrast.

In Table 6.5, I tested the similarity of the problems given during the think aloud study described in Chapter 4, by analyzing student performance per question. We found that Problem #2, finding the maximum, required the most submissions to complete on average due to outliers, but that the other three problems had very similar submission patterns. The outliers on problem #2 required over 45 submissions each, while every other participant took less than 20 submissions to complete the task. Students tended to reduce overall time taken to solve problems as they moved through the set of problems (see Table 6.5, where SD is the standard deviation of total submissions for the indicated problem). Students were performing think-aloud protocols while completing the problems, making the time to submit slightly exaggerated due to verbalization. Qualitative data (from think-aloud observations) suggest that the speed-up was due in part to difficulty with the interface rather than differences between the problems. A linear regression model on the number of submissions did not indicate significance for problem number, but indicated marginal significance between students. This regression indicates that the four problems were approximately the same difficulty, and applicable to group for this work.[4]

The ability level of individual participants varied greatly, with some participants submitting final solutions with minimal modifications from their first attempt, while other

[4]We acknowledge the low power involved in this study, and will continue to evaluate cross-problem difficulty as more data is gathered.

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

Table 6.5: Traditional Statistics for Think Aloud Data

participants struggled and progressed through multiple incorrect model states before arriving at a correct solution. Individual participants were consistent in their performance across problems, either doing well or struggling with all of them. The traditional measurement metrics can be used to separate participants into two groups: high performers (students who were able to quickly solve the problems), and low performers (students who needed more time and several attempts to get a problem right).

Of the seven students requiring more than six submissions to solve at least one problem, only two averaged fewer than six submissions per problem. These seven students also tended to take more than 500 seconds (8.33 minutes) overall to solve their problems, apart from the two mentioned above, who have individual outliers above that line. This disjoint grouping suggests that I can subdivide the low performing group into students who performed uniformly badly (5 participants) and students who struggled only with a specific problem (2 participants).

The eleven high performers all averaged four or fewer submissions to reach a correct answer, and all clustered under an average of 400 seconds (6.66 minutes), with the exception of one student who took nearly eighteen minutes to finish the first problem, but only needed to submit once. The increased time could be a factor of the think aloud protocol, as the student spent time describing the code he had produced as a part of the think aloud process.

Although it becomes evident that there are differences among individuals, The use of number of submits and time do not offer sufficient granularity or analysis power to discuss the actual difficulty students had with the problems. Using the metrics above, there is no inherent way to evaluate the multiple learning objectives present in the complex problem or the difference between a student struggling with a compiler error, simply being cautious about correct code (as with the high performing outlier who took 18 minutes), or struggling with the algorithmic concepts of the algorithm.

Two similar solutions

In multiple places throughout this thesis, I have postulated that students take different paths to constructing a correct solution, yet I have not modeled those pathways. An important distinction between the paths that students take and the submissions they make, is that two students may use the same number of submissions but take demonstrably different routes to the correct solution. The analysis of the solution route can provide additional detailed information to the researcher regarding the specific difficulties the student is having.

Figure 6.1 illustrates a very simple example of two participants' paths from an empty start state to a correct finished state. The figure shows each code submission as a circle with a letter for reference. The letters are only for reference, with S indicating an empty starting state, and F representing the finishing state. The ordering of the states is based upon the order of student submissions and position is not relative to the accuracy of the code submission.

For the PDS metric, an extra bit was used to distinguish between code that compiled and code that did not compile with the same alignment vectors. Two code submissions with the same alignment vector, one that compiled and one that did not, would have separate states in the diagram. State A represents code that has all of the correct algorithmic components, is compilable, but does not return the appropriate value. The Alignment Vector for State A is 1111-11-11-110. State B represents code that has all of the correct algorithmic components, but does not compile, so it cannot check the final return state. The Alignment Vector for B is the same as A 11111-11-11-110.

Participant 5 (the solid line and upper pathway) corrected a sign error (step 2) which introduced a compiler error, then corrected the compiler error (step 3), and finally another sign error (step 4), which resulted in a correct solution.

Participant 12, on the other hand, initially submitted code that did not contain a return statement, indicating a misunderstanding about how information is communicated back from the function. The Alignment Vector for State C is 11111-11-11-000, and the code did not compile[5]. The participant then made a small change resulting in the same model state for the code (step 2), then added a return statement based on a compile message (step 3), and finally fixed another compile error and submitted a correct solution (step 4). Although these two participants have the same number of submissions, the reasons for, and the nature of, the submissions are very different and a more detailed analysis of the steps expose a misconception about return statements by Participant 12.

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

Through these data and example, I have demonstrated the lack of granularity in performance metrics used in traditional learning environments. It could be suggested to look at each alignment vector component individually, recording number of submissions and time to complete each piece, to obtain a more detailed picture of student problem solving. Unfortunately, decoupling the alignment vector components will strip away their relationship to each other and make it difficult to look at each problem as a whole.

[5]The missing return caused a compile error.

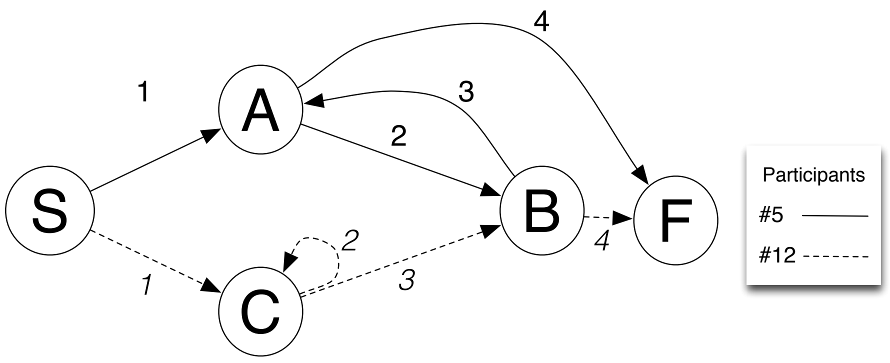


Figure 6.1: Comparing Two Student Paths

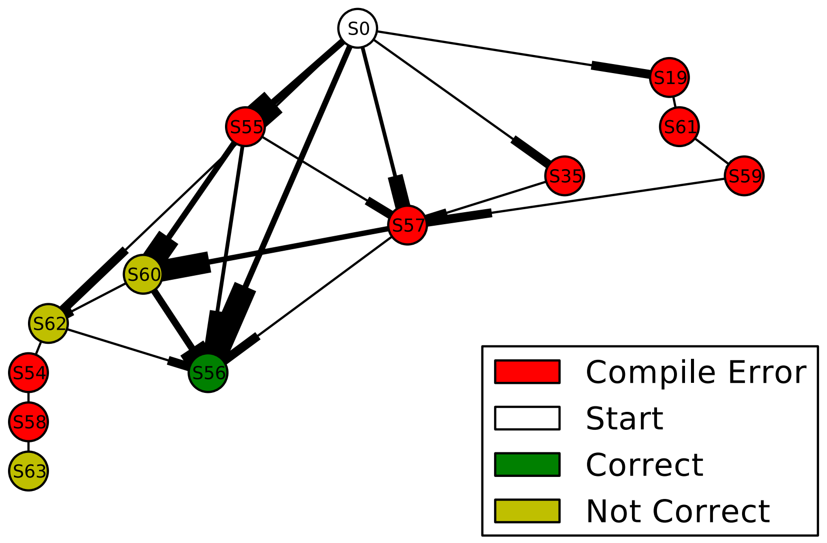
6.3.3 Probabilistic Distance to Solution

In order to visualize students in the programming and problem solving process, I use each student's submission as a stepping stone of the longer pathway each student follows. The submissions represent a student-dictated attempt at a correct solution or request for feedback. The alignment vector used to produce feedback is a representation of the state of the program, and since the alignment vector was designed to align with the instructional goals of the student's practice, the representation and subsequent visualization have meaning for discussing student knowledge and learning.

To draw generalizations about how program states correspond to student performance and other latent factors such as learning, I aggregated all student submission paths for each problem into a network (see Figure 6.2 for an example using Problem 4). The network nodes $S_1 \cdots S_{n-1}$ are possible program states with a finish state node F, and the edges are the observed transitions between states. This is similar to what is shown in Figure 6.1, and, in visualizations of the network, student code submissions are represented by circles (nodes) and connected by lines (edges) based upon the series of submissions made by any one student. Each alignment vector representation produces a different node in the network. If two students submit code with the same alignment vector, that would be represented by the same node in the network. Figure 6.2 shows the network constructed from student submissions to Problem 4 in the think aloud study described in Chapter 4. Each node represents an observed alignment vector, where the F-states are the alignment vector components. A solid green box indicates the component was correctly present in student code (1), and a white box indicates the component was not correctly present in student code (0). Notice that S0 has no correct alignment vector components - this is the starting state and represents the empty starter code provided to students, and S56 has all green boxes representing a correct final state[6].

From any given node representing a student submission, theoretically a student could

[6]The S-states in Figure 6.2 are not completely sequential as all four problems were computed together and state numbers were assigned as states were encountered in the dataset. The numbers of each state (S0..Sn) are assigned as the algorithm encounters a particular alignment vector and are provided only for reference.



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

Figure 6.2: Student Program States – Problem 4
Student program states for problem 4. Table columns Fx are alignment vector features. Table rows Sx are observed alignment vectors in program submissions. Only observed states (nodes) and transitions (edges) are shown. Node self-transitions also exist in the model, but they are not shown here. The thicknesses of the edges are proportional to the log of observed transitions in the data. The lengths of the edges are arbitrary and do not relate to the model.

produce any other alignment vector on the next submission. Many of the potential alignment vectors, however, are not likely to be observed as a next step in the student problem solving process. Instead I see common pathways emerging between fixed alignment vector representations (nodes) which can be seen by the thicker lines in the network in Figure 6.2.

For example, five students in the think aloud completed problem 4 in only one submission. These data are represented in the thick line in Figure 6.2 between S0, the starting state, and S56, the final correct state. A second example visible in the diagram is the S0, S55, S60, and S56(Final) path. We know the path is not trivial (one student) because of the thickness of the lines. By looking at the alignment vector, I can see that from the starting state, students then submitted code missing component F8, a correct return statement, that did not compile. Some students transitioned to other states (S57, S62, and S56), however a number of them fixed the compiler error, but still did not achieve a correct return value, and arrived at state S60. From S60 it was a common transition to S56, the final state.

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

$$\hat{p}(S_y(t)) = P(S_y(t)|S_x(t-1)) = \frac{T_{x,y}}{\sum\limits_i T_{x,i}} \tag{6.1}$$

This is equivalent to a Markov chain estimate with a 1-state history.[7]

6.3.4 Computing Predicted Number of Submissions to Solution

As a complex, non-linear, problem solving space, the number of submissions between any given point in the students' process and a final completed project is not computable based only on a measurement of an individual submission. Although the alignment vector gives us a representation of the completeness of the students' code, there is not a one-to-one correspondence between the alignment vector value and the number of submissions needed to produce the value, or the number of submissions from that value to a finished program. Instead, a probabilistic approach can be applied using student data.

A classic problem in computer science is the random walk. Given some network of nodes and edges, if an entity randomly chooses any edge to walk on from any node, how long will it take the algorithm to reach a designated final state? Although student submissions may *appear* random, there are some common, and mostly logical, pathways students take during the process, especially for the more sophisticated novice. Each outgoing edge from a given node can be given a weighted probability, so that the chance of walking on that node is more likely based on observed data. I have applied that method to the networks described to calculate a *Probable Distance To Solution*, referring to the number of likely

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

edits it will take a student from any given node (alignment vector state) to a final state (complete solution).

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

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

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

$$\begin{aligned}
d_f(S_1) &= \left[\sum_{s=1}^{n-1} P_{1,s}(D_{1,s} + d_f(S_s))\right] + P_{1,f}D_{1,f} && (6.2)\\
d_f(S_2) &= \left[\sum_{s=1}^{n-1} P_{2,s}(D_{2,s} + d_f(S_s))\right] + P_{2,f}D_{2,f} && (6.3)\\
&\vdots && (6.4)\\
d_f(S_{n-1}) &= \left[\sum_{s=1}^{n-1} P_{n-1,s}(D_{n-1,s} + d_f(S_s))\right] \\
&\quad + P_{n-1,f}D_{n-1,f} && (6.5)\\
d_f(F) &= 0 && (6.6)
\end{aligned}$$

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

$$\begin{aligned}
d_f(S_1) &= \vec{P}_1 \bullet d_f(\vec{S}) + 1 && (6.7)\\
d_f(S_2) &= \vec{P}_2 \bullet d_f(\vec{S}) + 1 && (6.8)\\
&\vdots && (6.9)\\
d_f(S_{n-1}) &= \vec{P}_{n-1} \bullet d_f(\vec{S}) + 1 && (6.10)\\
d_f(F) &= 0 && (6.11)
\end{aligned}$$

6.3.5 Applying PDS

The PDS metrics accompanied by the transition graph are rich sources of information about the paths that participants pursued in order to arrive at a correct solution. Figure 6.2 shows a Program States Graph for problem #4 from the data collected during the think aloud study described in Chapter 4. Each circle (node) represents a unique Alignment Vector (AV) observed in the data. Each line is a transition between states, observed when students in a state (the thin side of the line) made a single edit which produced the AV for the second state (at the thick side of the line). The thickness of the line indicates the number of times the transition was observed over all students. For example the transition between S0 (the starting state) and S4 has a relatively thick line, indicating it was observed more frequently than the transition between S0 and S32. The thickness of the line underscores the common problem solving pathways, and it can be used to help identify intermediate states in the process. All lines are single directional - indicating a transition from the thin side of the line to the thick side.

Figure 6.2 includes a table illustrating the observed model states in the binary vector, as well as the PDS for each state. In problem 4, S56 is the solution state and S0 the initial starting state. Before even evaluating the student paths, we can observe that an additional state, S63, was also a terminal state for a participant. This participant located a bug in the evaluation system that has since been corrected.[8]

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

With the possibility of including terms in the algorithm for syntactic but not model changes (i.e. two states that are identical except for a compilation error would not be counted as a full step), PDS can be adapted to focus on model state transitions that indicate misconceptions of not only the students but also of the researchers as well. For example, instructors may believe that novices all take similar pathways to solve the problems given, using techniques demonstrated in worked examples from lecture. The diversity of the PDS graphs however, indicate that students take very different paths, and although there are a finite set of paths, students do not use a logical stepwise refinement process to produce a solution.

6.3.6 Usefulness of PDS

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

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

Although tested against data from a computer programming dataset, I believe that the PDS metric could be valuable across many domains with complex solutions demonstrating multiple skills, as illustrated in references to this work [81] papers related to computer

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

programming [42], a logic tutor, survey administration [31], and a simulation-based assessment task [10]. In the next chapter, I will use the PDS values and network graphs to evaluate students in different tutoring conditions to determine if algorithmic based feedback produced different pathways in the student problem solving process.

6.4 Conclusion

In this chapter I have presented the framework for two analyses of student data. First, I disaggregated student submits based upon a classification of the type of edit made from the previous submission, using the model of algorithmic components not only as a mechanism for generating feedback, but also as a method of assessing student code. In the following chapter I will use these metrics to assess differences within and across problems for students participating in online studies.

Chapter 7

Impacting Novice Code Production with Feedback

The studies in this chapter are the culmination of the work presented in this thesis. My theoretical foundation supports explicit feedback regarding algorithmic components for the novice programmer (Chapters 1 and 2). The model of algorithmic components that I presented and validated (Chapters 3 and 4) is used to generate feedback for the participants in an online system, whose implementation was described in Chapter 5, and then used to analyze the progress participants make through problems in the tutor (Chapter 6). In this chapter, I seek to answer the research question **Will pre-compilation feedback regarding algorithmic components produce better (a) within-problem performance and (b) across-problem learning?**

To answer the research question, I use data from two online experiments with novice programmers across the country. In Study 1, I use the Algorithmically Productive Edit (APE) metric, and for Study 2, I use both APE and Probabalistic Distance to Solution (PDS) discussed in the last chapter. Analysis will be both quantitative, working with aggregate information from student submits, as well as qualitative, showing case studies of a series of submits, to describe observed phenomena.

7.1 Study 1: Likelihood of Productive Edits

In the Spring of 2012, I conducted a large online study to test the effects of algorithmic feedback on novices learning to program. This section details the methodology used in the study, the participants who completed activities, and an analysis of the data collected. All participants logged into the AbstractTutor system from the internet and feedback was generated automatically and in real time while they were solving problems involving simple array algorithms.

School	State	Number Registered	Problems Completed
Bismark State University	ND	3	6 (50%)
Johns Hopkins University	MD	14	27 (48%)
Loyola University	MD	16	13 (20%)
Mt. Saint Mary's University	MD	22	60 (68%)
North Carolina State A&T University	NC	14	3 (5%)
Otterbein University	OH	1	0 (0%)
Southern Polytechnic State University	GA	4	7 (44%)
William Penn University	IA	58	136 (59%)
Wooster College	OH	28	72 (64%)

Table 7.1: Study 1: Schools with Participating Students

7.1.1 Study Design

In the Spring of 2012, I recruited participants from colleges and universities with faculty on the ACM Special Interest Group in Computer Science Education list serv. I sent a message to the list asking faculty who teach an introductory computer science course to respond if they would be willing to ask students to complete four problems in an online java programming practice environment. Nine (9) faculty responded and were provided with an access code so students could establish online accounts in the AbstractTutor system. Each faculty member presented the activity in a different way, some requiring participation for a homework grade while others offered it as extra review before the final exam.

From the 9 universities that received an access code, 160 students created online accounts and attempted at least one problem. Table 7.1 shows the universities and the number of registered users from each university. Not every student who created an account completed the activity, and Table 7.1 also shows the number of students who successfully completed at least 1 problem, and percentage of possible problems (4 per student) that were completed. Faculty members who requested could receive an email list of usernames for students who registered and the number of exercises completed in the system for grading purposes in their courses.

When participants created a user account with the given access code, they were randomized to one of four conditions. Half of the participants saw problems in a mathematical context, while half of the participants saw problems in a story based context as described in Chapter 4. Table 7.2 shows examples of a mathematical context and story context problem; the full set of problems with suggested solutions is available in Appendix A. Half of the participants received only compiler error messages and output based feedback providing input and output values from the method. The other half of the participants were placed in the algorithmic feedback condition, with the potential for pre-compilation algorithmic feedback when appropriate.

Each participant saw the sum problem first (Table 7.2), with wording appropriate to the contextual condition (Math or Story) of the participant. Problem 1 (sum) was shown to all participants first because it was a relatively simple algorithm, and one most likely

Mathematical Context	Story Context
Write a method to find and return the sum of all the values stored in the array. public int findSum(int []myList){	The array items contains the price of items on a sales receipt. Write a method to find and return the total (sum) of all the items on the receipt. public int findSum(int []items){

Table 7.2: Problem Context Examples

to have been seen by students in the worked examples covered in their course. Problem 1 was also the only problem that did not require an if statement to select specific elements from the array. Each student then saw a random ordering of problems 2-4 (max, count, index). After completing the four questions with feedback appropriate for their condition, the participants then saw a fifth question with directions that they were only going to be given one chance to answer the question and no feedback (either algorithmic or compiler). The fifth question was used as a post test for the session and asked students to find the sum of the even numbers contained within the array. All students received the post-test question worded for the math context.

Although students were distributed evenly across conditions, the assignment took place when the user first created a login to the system. Unfortunately, not every student who created a login observably attempted a problem. I am defining an observable attempt as a participant who created a login, consented to participation in the study, and then made at least one submission for the first problem. Table 7.3 displays the number of users, by condition, who made an observable attempt at each problem in the system. The first section of the table is divided by the problem the participants saw and the conditions they experienced (Mathematical vs. Story context and Algorithmic vs. Output Based feedback). The second section of the table shows the number of participants per problem they attempted. For example, in the first column, 32 participants in the Math Context with Output Based Feedback condition made an observable submission for the first problem they saw in the system (Sum). In the same research condition, 24 students made an observable submission for the second problem they encountered, which was randomly chosen from Max, Count, and Index.

Overall, 21 participants only attempted problem 1 (left without attempting problem 2). Fourteen (14) participants attempted only two problems, and 14 participants attempted only three problems. One hundred and eleven (111) participants attempted all four problems in the system. Overall, most of the participants completed all four problems, and an inspection of the incomplete attempts shows students who were engaged with the task (not submitting blank code or irrelevant text). Not all of the professors required the students to complete problems (as evidenced by requests for registered users), and so many factors may contribute to the attrition rate.

Problem	Math Context Output-Based Feedback	Math Context Algorithmic Feedback	Story Context Output Based Feedback	Story Context Algorithmic Feedback	Total
Sum	32	43	51	34	160
Max	24	35	41	31	134
Count	24	31	40	27	122
Index	25	31	37	29	122
Order					
1	32	43	51	34	160
2	27	35	46	31	139
3	24	34	38	29	125
4	22	28	34	27	111

Table 7.3: Study 1: Number of Participants per Condition

Condition	Num Students	Ave Problems Attempted	Ave Problems Completed	Ave Time To Complete
Math Context	75	3.28	1.96	19:55
Story Context	85	3.42	2.10	25:09
Algo Feedback	77	3.40	1.75	20:50
Output Feedback	83	3.31	2.30	23:42

Table 7.4: Study 1: Problem Attempt and Completion Rate by Condition

7.1.2 Submissions

The 160 participants entered a total of 4835 submissions into the system, including 115 submissions for the post test question[1]. Each non post-test submission (4720) represents an attempt to either submit a potential correct solution, or receive additional feedback about a partial solution. When the participants submitted their code, they received a feedback message from the system that contained information about the mistakes in the program, or whether the code was correct. The content of the feedback message varied based upon the assigned feedback condition. Participants also had a next problem button they could use to move to the next problem, even if they had not correctly solved the problem they were viewing.

Unfortunately, during the study, a bug in the code prevented AbstractTutor from providing algorithmic feedback in instances where students had a compiler error. The bug triggered a default message normally given when the code does not parse; however, due to the error, the message was shown any time the student had a compiler error. Although the error message started "Your code could not parse for better feedback," it then provided the appropriate compiler messages. An ANOVA comparing subjects in the faulty algorithmic feedback condition and output based condition showed no significant impact of the type of feedback [F(1,158)=1.4, p=.24] on the number of submissions made by any user, implying that the addition of the message "Your code could not parse" did not impact the users' ability to read and use the compiler errors.

Although the error in the feedback mechanism prevents us from aggregate comparisons across conditions, as much of the feedback that participants saw was the same, the data can still be useful. Within the Algorithmic Feedback condition, 300 submissions generated appropriate algorithmic feedback, so although the treatment was not as intensive as desired, the data are still potentially useful. Each single instance of algorithmic feedback can be seen as an intervention, and the performance of students in (1) their immediate next action, and (2) over the remainder of the problem can tell us about the impact the feedback had on the participants' problem solving process.

When conducting an analysis of the data, the number of submissions is only a rough estimate of the difficulty a participant has with the conceptual understandings of a problem. Table 7.5 shows the total number, mean, and median number of submissions for users from each research condition and by problem. The obvious difference in Table 7.5 of the Story Context problems with output based feedback is mostly the product of a single user who needed 152 submits on the sum problem, 44 submits on the max problem, 102 submits on the count problem, and 17 submits on the Index problem. Numbers in Table 7.5 with an asterisk (*) indicate a single user contributed over 100 submits to the problem count. The mean and median number of submissions per user are also reported as more stable measures of comparison.

Recall that the Sum problem was offered first, and not randomly as a part of the series, because it was hypothesized to be the easiest problem (not requiring an if statement) to solve, however a one-way within subjects ANOVA was conducted to compare the number of submissions per problem number (based on the order in which the problems were seen)

[1]Two students skipped the fourth problem but did enter an attempt at the post test question.

Problem	Math Context Output-Based Feedback	Math Context Algorithmic Feedback	Story Context Output Based Feedback	Story Context Algorithmic Feedback	Total
Sum (Mean/Median)	333 10.4 / 3	319 7.4 / 4	856* 16.8 / 4	237 7.0 / 3	1745 10.9 / 4
Max (Mean/Median)	130 5.4 / 4	183 5.2 / 3	309 7.5 / 3	222 7.2 / 3	844 6.3 / 3
Count (Mean/Median)	169 7.0 / 4	178 5.7 / 3	394 9.9 / 3	265* 9.8 / 3	1006 8.2 / 3
Index (Mean/Median)	249 10.0 / 8	276 8.9 / 5	383* 10.3 / 6	225 7.8 / 4	1133 9.3 / 5.5
Order					
1 (Mean/Median)	333 10.4 / 3	319 7.4 / 4	856* 16.8 / 4	237 7.0 / 3	1745 10.9 / 4
2 (Mean/Median)	206 7.6 / 5	187 5.3 / 3	432 9.4 / 4	232 7.5 / 3	1057 7.6 / 4
3 (Mean/Median)	216 9.0 / 4	189 5.6 / 3	491* 12.9 / 4	162 5.6 / 3	1058 8.5 / 3
4 (Median)	126 5.7 / 4	261* 9.3 / 2	165 4.6 / 4	318* 11.7 / 3	870 7.8 / 3
* indicates a large number of submissions from one user					

Table 7.5: Study 1: Number, Mean, and Median of Submissions per Condition, Number of Participants available in Table 7.3

and no significant effect of problem number was found [F(1,3)=.996, p=.39]. These data, however, include submissions from students who attempted a problem but did not complete it, or who completed some problems, but not all.

Table 7.6 shows the number and median submissions per user for participants who completed all four problems within the system. In total, 72 participants completed all four problems successfully. They were distributed across conditions as follows: Math/OBF: 17, Math/AF: 14, Story/OBF: 23, Story/AF: 18. Again, a one-way within subjects ANOVA found no difference for Problem Number [F(1,70)=0.402, p=.53], Problem Name [F(3,70)=1.80, p=0.16], Feedback Condition [F(1,70)=.32, p=.57] or Context [F(1,70)=.063, p=.80]. The results indicate the number of submissions did not change significantly among users as they progressed through the four problems. Although the lack of significance in the Feedback Condition does not aid in positively answering the primary research question of the thesis, it would indicate for these data that the feedback mechanism did not significantly negatively impact the students in the abstract feedback condition.

Using the raw number of submissions does imply that practicing the four problems had no measured effect on student proficiency at solving the problems in AbstractTutor. A closer evaluation of the actual submissions, however, tells a different story.

7.1.3 Individual Submission Analysis

Although an analysis of the number of submits did not yield a significant difference for problem or condition, in Chapter 6 I presented evidence that the raw number of submits is not an appropriate measure of student proficiency. The raw correctness of an attempt is also a poor measure, as most (86%) of submissions are incorrect on first attempt. For this analysis I use the full dataset of any student who made at least two submissions to any problem, giving them the chance to receive feedback and perform an edit.

The submits for the problems students attempted to solve in the AbstractTutor system can be disaggregated using the model of algorithmic components proposed in Chapter 3. Each algorithmic component can be seen to represent an individual piece of a correct solution, and as discussed in Chapter 5, the combination of the components into an Alignment Vector can provide a detailed aggregate of each submission. Table 7.7 shows the average percentage of correct algorithmic components in alignment vectors for each submission.

Although the disaggregation of submits by alignment vectors offer more granularity than raw submits or correctness on first attempts, it does not give us a picture of student progress over time. In Chapter 6, I discussed a categorization of student attempts into Duplicate Submissions (DS), Algorithmically Productive Edits (APE), Algorithmically Neutral Edits (ANE), and Algorithmically Counterproductive Edits (ACE). Categorizing the type of edits made by participants can also give us a more detailed view of the participants' struggles while solving the problem. Table 7.8 shows the distribution of all submits by the problems based on condition and problem number.

A within subjects ANOVA of submissions that were not First Attempts (n= 4165) yielded significant results for a number of factors both across and within subjects. Prior submissions were coded into one of three possible categories: Algorithmic Feedback (244), Output Based Numeric Feedback (555), and Other Feedback (3366). Algorithmic feedback

Problem	Math Context Output-Based Feedback N=17	Math Context Algorithmic Feedback N=14	Story Context Output Based Feedback N=23	Story Context Algorithmic Feedback N=18	Total N=72
Sum (Mean/Median)	183 10.8 / 3	107 7.6 / 5	318* 13.8 / 4	115 6.4 / 3	723 10.0 / 3
Max (Median)	95 5.6 / 4	95 6.8 / 3	161 7.0 / 3	122 6.8 / 3	473 6.6 / 3
Count (Median)	144 8.5 / 4	97 6.9 / 4	167 7.2 / 3	120 6.7 / 3.5	528 7.3 / 4
Index (Median)	158 9.3 / 6	198* 14.1 / 3	288* 12.5 / 4	157 8.7 / 3.5	801 11.1 / 4
Order					
1 (Mean/Median)	183 10.8 / 3	107 7.6 / 5	318* 13.8 / 4	115 6.4 / 3	723 10.0 / 3
2 (Mean/Median)	156 9.2 / 5	114 8.1 / 5	259 11.3 / 4	125 6.9 / 3.5	654 9.1 / 4
3 (Median)	151 8.9 / 4	86 6.1 / 3	268* 11.7 / 3	88 4.9 / 2.5	593 8.2 / 3
4 (Median)	90 5.3 / 3	190* 13.6 / 2	89 3.9 / 3	186 10.2 / 3.5	555 7.7 / 3
* indicates a large number of submissions from one user)					

Table 7.6: Study 1: Number and Mean of Submissions per Condition for Participants Completing 4 Problems

Problem	Math Context Output-Based Feedback	Math Context Algorithmic Feedback	Story Context Output Based Feedback	Story Context Algorithmic Feedback	All
Sum	69.3	52.0	34.7	57.4	47.6
Max	62.7	69.3	54.4	50.3	57.8
Count	67.0	72.1	60.0	41.6	58.5
Index	48.8	61.5	66.8	89.1	66.0
Order					
1	69.3	52.0	34.7	57.4	47.6
2	60.6	68.1	58.8	52.8	59.5
3	50.8	72.3	66.4	75.5	65.6
4	64.7	61.6	49.4	55.8	57.6

Table 7.7: Study 1: Alignment Vector Averages for All Submissions

refers to instances where students received the feedback resulting from the model presented in Chapter 3 and detailed in Appendix A. Output Based Numeric Feedback refers to submissions where students code compiled and they received feedback regarding a series of numbers as input to the method, and the resulting value from the code, as well as the expected correct resulting value. The Other Feedback encompasses all other feedback provided to the students, this included compilation errors, infinite loops, and exceptions[2].

Tables 7.9 and 7.10 show the results of a within-subjects repeated measures ANOVA. The effects of being in the mathematical context condition, algorithmic feedback condition, the current sum of alignment vector components, the type of feedback received for the previous submission, the name of the problem, and the number of the problem[3] were calculated for the likelihood of having produced an Algorithmically Productive Edit. Although there was no large effect for being in the algorithmic feedback condition, this is unsurprising since the feedback mechanism failed and the treatment was significantly diluted (i.e., there were many instances where participants should have seen algorithmic feedback, but where instead they received a compiler error message). There was however a significant effect of the type of feedback received in the prior submission on the likelihood of an Algorithmically Productive Edit.

Table 7.11 shows the coefficients of the model for the types of feedback received immediately prior to the current submission. The coefficients, combined with the significance shown in Tables 7.10 and 7.11, can be used to evaluate the additive impact of algorithmic feedback. The coefficient of 0 for algorithmic feedback is assigned as a reference number (since alphabetically "algorithmic" is the first name of the type of feedback). All of the other coefficients represent a change from the reference point of Algorithmic Feedback.

[2]An exception in Java is generated at run-time when the code behaves in an unexpected way based upon the state of any variables. Common exceptions from introductory programming include attempts to divide by zero, or trying to access an index past the length of an array.

[3]First, Second, Third, or Fourth problem seen by the user.

Condition	Duplicate Submission	Algorithmically Counter Edit	Algorithmically Neutral Edit	Algorithmically Productive Edit	First Attempt
Math/OBF (N=32)	335 35.8%	50 5.35%	274 29.3%	141 15.8%	135 14.4%
Math/AF (N=43)	357 35.4%	66 6.6%	264 26.2%	150 14.9%	169 16.8%
Story/OBF (N=51)	809 40.3%	107 5.33%	660 32.85%	224 11.15%	209 10.4%
Story/AF (N=34)	441 43.8%	48 4.8%	251 25.0%	113 11.3%	150 15.0%
Problem Order					
1 (N=160)	691 39.7%	109 6.3%	549 31.5%	223 12.8%	169 9.7%
2 (N=139)	365 34.6%	51 4.8%	326 30.9%	167 15.8%	147 13.9%
3 (N=125)	479 45.4%	53 5.0%	268 25.4%	126 11.9%	130 12.3%
4 (N=111)	407 46.8%	58 6.7%	181 20.8%	112 12.9%	112 12.9%
Total (N=160)	1942 41.1%	271 5.7%	1324 28.0%	628 13.3%	558 12.9%

Table 7.8: Study 1: Labeling of Code Edits with Percentage of Submissions Per Row

ANOVA	Df	F Value	P Value
Mathematical Context	1	9.2	.002
Algo Feedback	1	0.01	.94
AV Sum	1	13.6	<.001
Previous Feedback	5	5.2	<.001
Problem Name	3	.67	.57
Problem Number	4	.85	.50

Table 7.9: Across Subject Results for Algorithmically Productive Edits

ANOVA	Df	F Value	P Value
AV Sum	1	393.7	<.001
Previous Feedback	5	102.8	<.001
Problem Name	3	3.65	0.01
Problem Number	3	2.98	0.03

Table 7.10: Within Subject Results for Algorithmically Productive Edits
A positive coefficient indicates a contributing factor for making a productive edit

Across Users Type of Feedback	Coefficient
Algorithmic	0
Compiler	-0.017
Correct	-0.29
First Attempt	0.10
Output Based Feedback	.01
Failed to Parse	.28
Within Users	
Algorithmic	0
Compiler	-0.15
Correct	-0.35
First Attempt	-0.32
Output Based Feedback	-0.35
Failed to Parse	.08

Table 7.11: Within Subject Results for Algorithmically Productive Edits

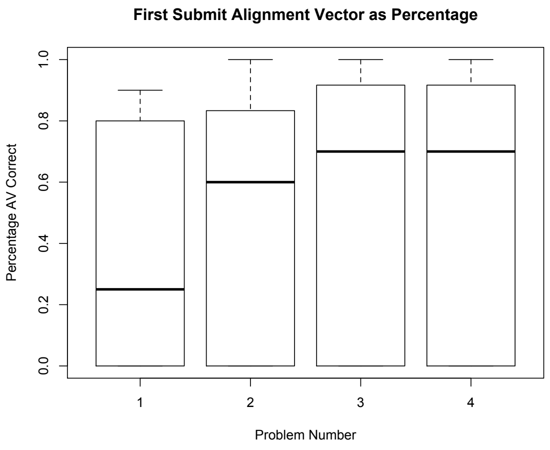


Figure 7.1: Study 1: First Attempt Alignment Vectors (As Percentage)

As argued in previous chapters of this thesis, the general compiler-based feedback received by students does not necessarily translate to productive edits of code. Compiler messages, messages regarding infinite loops, and other code-based feedback for participants, were the least likely to generate Algorithmically Productive Edits, having negative coefficients for both within and across user effects in the model. Additionally, the alignment vector percentage was a significant positive effect for both within and across users. The alignment vector is an approximate measure of algorithmic completeness of the algorithm, and therefore it is unsurprising that a participant who is closer to a correct solution will be more likely to make productive edits.

These results offer supportive evidence that exposure to algorithmic feedback improves students' within problem performance.

7.1.4 Learning Gains from Practice

As shown in Section 7.1.3, the raw number of submits is not an ideal measure of within problem student performance, and the use of number of submits or raw correctness on first attempt would also not be a good measure of student learning across problems. Figure 7.1 shows a box plot of the alignment vector at first submit for each problem students saw. The Alignment Vector is transformed from a raw number to a percentage to compensate for the fact that Sum and Index only have 10 necessary components in their relative alignment vectors. An ANOVA indicates a significant positive effect of problem number on the alignment vector of participants' first submission. [F(1,1)=8.6, p<.01] The increase in alignment vector totals at first submission indicate that algorithmic learning is taking place across problems, despite our inability to detect a difference using only the raw number of submissions. These results do not offer an answer to the research question regarding the effect of algorithmic feedback on across problem learning; however due to the failure of the feedback mechanism, the treatment was significantly diluted.

7.1.5 Study 1: Conclusions

Although the feedback mechanism did not produce algorithmic feedback in all of the instances where it is desired, an analysis focused on the individual submission level shows the algorithmic feedback is not only beneficial, but has a greater impact on the likelihood that students will make a productive edit. The algorithmic feedback provided through AbstractTutor in the large online study has a positive impact on student performance, both within an individual subject and across the data set. In the next section, I describe a study wherein the algorithmic feedback was produced at expected levels and I evaluate not only submission level student outcomes, but differences across condition as well.

7.2 Study 2: Full Problem Evaluation

After the malfunction of the feedback mechanism in the previous study, I designed and executed a second study in the fall semester of 2012. Students again participated in the problems previously described and the feedback mechanism operated appropriately for student submissions.

7.2.1 Study Design

In the Fall of 2012, I recruited participants from colleges and universities with faculty on the ACM Special Interest Group in Computer Science Education list serv. I sent a message to the list asking faculty who teach an introductory computer science course to respond if they would be willing to ask students to complete four problems in an online practice environment. 12 faculty responded and were provided with an access code to give to students to set up online accounts in the AbstractTutor system. Each faculty member presented the activity in a different way, some requiring participation for a homework grade while others offered it as extra review before the final exam.

From the 12 universities that received an access code, 61 students created online accounts. Table 7.12 shows the universities and the number of registered users from each university. Not every student who created an account completed the activity, and Table 7.12 also shows the number of students who successfully completed at least 1 problem, and percentage of possible problems (4 per student) completed. Faculty members who requested could receive an email list of usernames for students who registered and the number of exercises completed in the system for grading purposes in their courses.

When participants created a user account with the given access code, they were randomized to one of two conditions. As no previous effects of math or story based problem description were found, all participants saw a mathematical context for the problems they completed.

7.2.2 The Data

Overall 36 students from 4 schools visibly attempted at least one problem in the online tutor. Students were randomly assigned by the tutor to a feedback condition upon creation

School	State	Number Registered	Problems Completed
Olivet Nazarene University	IL	14	53 (95%)
Illinois Valley Community College	IL	5	9 (45%)
Johns Hopkins University	MD	15	41 (68%)
Portland Community College	OR	2	1 (13%)

Table 7.12: Study 2: Schools with Participating Students

Problem	Math Context Output-Based Feedback (N=21)	Math Context Algorithmic Feedback (N=15)	Total (N=36)
Sum (Mean/Median)	93 6.2/4	85 4/3	178 4.9/4
Max (Median)	54 5/4	67 4/3	121 4.3/3
Count (Median)	112 10/6	53 4/2	165 6.3/2
Index (Median)	113 11/3	57 4/1	170 7.4/3

Table 7.13: Study 2: Number and Mean of Submissions per Condition

of a username with the AbstractTutor system, however not all students who created a username consented to the study or observably attempted a problem. The 36 participants were divided amongst conditions as follows: Ouput Based Feedback 21 students, Algorithmic Feedback 15 students. Although a more balanced student distribution across conditions is desirable, it is impossible to know when students will persist or leave the system.

The 36 participants generated 634 submissions in the online system. Table 7.13 shows the distribution of submissions by problem and condition. All students saw all problems in the same order, Sum, Max Search, Count in Range, and IndexOf. In the larger Study 1 from this chapter, there were no effects of algorithm the participants were trying to implement, similarly, there were no effects of problem on student performance in the smaller study.

Again the student submissions were categorized based upon Algorithmically Productive, Neutral, and Counterproductive edits as described in Chapter 6. The distribution of edits is seen in Table 7.14. Although it appears that students in the Algorithmic Feedback condition make fewer Algorithmically Productive Edits to their code, students will also make fewer Algorithmically Counterproductive Edits or Neutral edits in the algorithmic feedback condition. The lower percentages were balanced by an increased number of first attempts and duplicate submissions in the algorithmic feedback category. It is possible that students in the algorithmic feedback condition either did not understand the algorithmic feedback, or were attempting to get the second level hints and therefore submitted the same code multiple times in a row.

Condition	First Attempt	Duplicate Submission Edit	Algorithmically Counter Edit	Algorithmically Neutral Edit	Algorithmically Productive Edit
Math/OBF (n=21)	20 18.3%	23 21.1%	8 7.3%	32 29.4%	26 23.9%
Math/AF (n=15)	20 27.0 %	18 24.3%	1 1.4%	20 27.0%	15 20.3%

Table 7.14: Study 2: Labeling of Code Edits with Percentage of Submissions Per Row

7.2.3 Individual Submission Analysis

In this section, I focus on analyzing the within-problem solving differences, attempting to answer the research question **Will pre-compilation feedback regarding algorithmic components produce better within-problem performance?** Similar to the analysis performed in Section 7.1, I use an individual submission granularity to assess the impact of algorithmic feedback on students' immediate actions after receiving feedback.

In order to estimate the effect of each type of feedback in a single attempt, each submission was labeled with the type of feedback the user had seen on the prior attempt. Consider the following sequence from a participant in the output based feedback condition, a participant submits a first attempt to a problem (1) and then receives a compiler error message. The participant makes an edit to the program and resubmits (2). The resubmit has no compiler errors, but is not correct and the user sees a message describing the numbers in the array, the expected answer, and the resulting value calculated by the participant's code. The participant then makes a third edit and resubmits the code (3) which is judged to be correct. Attempt (1) is labeled a First Attempt as the participant saw no feedback that prompted the code he wrote. Attempt (2) is is labeled Compiler, as the participant saw a compiler error message before editing and resubmitting the code. Attempt (3) is labeled OBF for Output Based Feedback as the user saw a message about the output produced by his function.

The type of feedback seen by the participant had a significant relationship with the likelihood that the participant would make a change classified as an algorithmically productive edit. A mixed effects ANOVA shows a significant effect of the previous feedback on the expectation that students will make a productive edit, or APE [F(4,489)=11.8, p<.001].

Although the coefficients of the model can inform how each type of feedback impacted the likelihood, a more fine grained analysis is possible if we look at the magnitude of the edits with relationship to the feedback seen. A linear mixed effects model, including an error term for each subject, was used to estimate the effects of each type of feedback on the alignment vector components.

Each submission was evaluated by the AbstractTutor system for the appearance of each algorithmic component in the model as described by Chapter 3 and 5. The sum of the components was then used similar to a rubric score to create an alignment vector score for each submission. For example, a blank submission would receive a score of 0, while a completely correct solution for a Sum problem would receive a score of 10. A

Problem	Min Score	Max Score	Mean Score	Median Score
1	0%	100%	46%	60%
2	0%	100%	68%	83%
3	0%	100%	72%	90%
4	0%	100%	76%	90%
All	0%	100%	65%	83%

Table 7.15: Study 2: Alignment Vector Sums as Percentage of Points Possible

correct solution for a Max problem would receive a score of 12, as there are two more algorithmic components required for a correct implementation of Max. Table 7.15 shows the distribution of alignment vector scores as a percentage of points possible for the problems solved in the tutor.

Each subsequent submission has the chance of providing a productive edit. The magnitude of that edit can be seen by the overall increase in the alignment vector sum. Not all submissions, however, have the same potential for productive edits, as some submissions are very close to correct, missing only one alignment vector component (perhaps having a score of 11 out of 12), while other submissions have many areas for improvement (perhaps having a score of 0 or 1 out of 12). To normalize the value of each edit made, the change to alignment vector resulting from an edit was calculated as a percentage of the possible change (the difference between the previously submitted value and the possible correct value). The actual number of alignment vector points is slightly skewed as some of the problems (sum, index) could be correct with only 10 alignment vector components, while the other two problems (max and count) required all 12 alignment vector components.

Figure 7.2 shows the result of different types of feedback (x axis) on the percentage improvement to submitted code after an edit. Each blue line represents the *comparative* effect of a particular type of feedback, using algorithmic feedback as the baseline. The grey boxes show the 95% confidence intervals for each category. The means and confidence intervals were determined based on the coefficients of a Linear Mixed Effects model fit by REML in the statistical package R (shown in Table 7.16). Algorithmic feedback has no confidence interval (grey box) as all of the other lines are produced as a difference from the baseline. A feedback category would therefore be significantly different from Algorithmic feedback if the red line (added for reference) lies outside the grey confidence interval for any particular type of feedback.

To relate the graph to a particular student submission, let us consider a student who has just submitted a piece of code for problem one, finding the sum. The last feedback message the student saw was an Output Based Feedback message, such as "Your countLessThan method returned the wrong value when I passed it [71, 41, 29, 10, 76, 63, 15, 31, 56, 32, 0, 35, 4, 3, 68, 67, 51, 62, 31, 97] and a value of 89: Your method should have returned 19 but returned 18". In the example, the student did not count the 0 (used a strictly greater than operator instead of greater than or equal to when comparing to 0). According to the model, the student is not likely to make a productive edit, and on average will make an edit that makes the program, on average, 1% further from correct than the previous submission.

Type of Feedback	Graph Label	Coefficient Value	p-value
Algorithmic Feedback	AF	.121	N/A
Correct	C	-0.281	0.026
Compiler Error	Compile	0.053	0.12
Output Based Feedback	OBF	-0.135	<.001
Failure to Parse	Parse	-0.140	0.024

Table 7.16: Study 2: Estimated Change to Alignment Vector As a Result of Feedback

The same student submits a different piece of code for either the same or a different problem (problem was not a significant factor), and sees a piece of algorithmic feedback. That student is now likely to make a productive edit, NS the edit will on average improve the AV by 12% (.121). Table 7.16 shows the coefficients and significance for each of the categories of feedback presented to participants. There is no p-value for Algorithmic Feedback because it is the baseline value.

Across students, there was still no significant effects of condition (feedback or context) or problem number. A model selection process was used to refine the model and minimize the AIC value as a measure of fit. The model selection was done by first including all of the potential variables (feedback condition, problem number, problem name, and story or math context) and then removing the variable with the highest p-value at each iteration. The A Information Criteria (AIC) value was used as a measure of model fit [3] and at each iteration the removal of the variable produced model with better explanatory power. The resulting model was used to generate the graph shown in Figure 7.2 and Table 7.16

Supportive of the claims of this thesis, students who saw feedback regarding the algorithmic components in their code were likely to produce a positive edit, and that positive edit is significantly greater than edits produced by output based feedback messages.

7.2.4 Conditional Differences

One of the hypotheses of the thesis was that students in the algorithmic condition would show across problem learning, increasing the sophistication of their answers at first attempt, or completing the problems in less time. There were no significant effects of being in the feedback condition on either the number of submits it takes to solve a problem (p=.25), or the sum of components in the alignment vector of the first attempt(p=.68) at a problem.

Figure 7.3 shows a boxplot of the sum of alignment vector components, as a percentage of the AV points available for max, count, and index of, on the first attempts of participants. The plot was constructed using all the first attempts from all users on problems 2, 3, and 4. The plot labeled FALSE is students in the output based feedback condition, while the plot labeled TRUE represented students in the algorithmic feedback condition. The boxplot illustrates the lack of significance between the two conditions, although there may be a ceiling effect happening as the novices become more proficient over time. Table 7.17 shows the mean first attempts by algorithmic feedback condition for each problem.

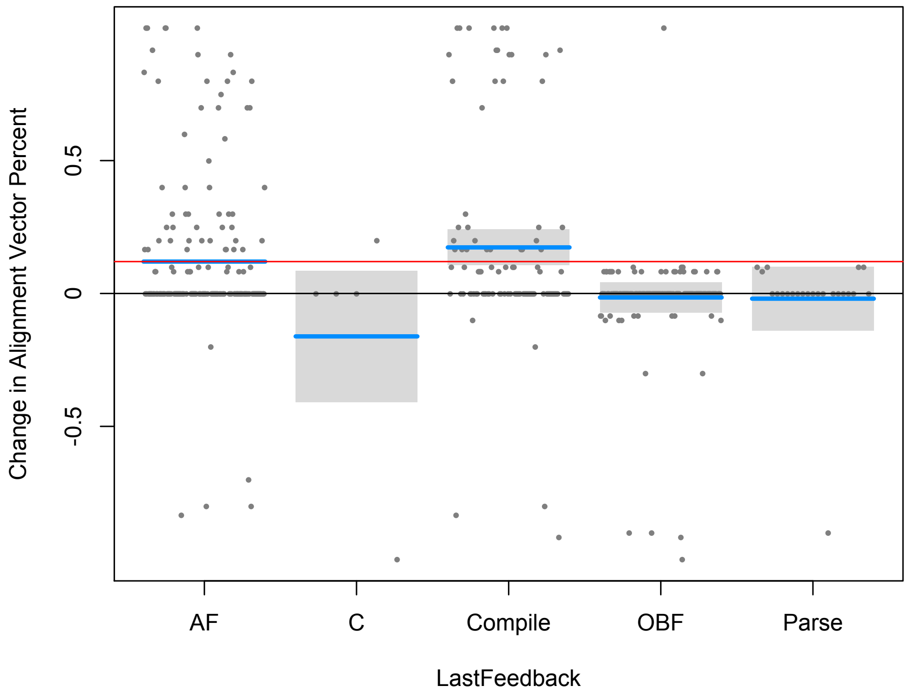


Figure 7.2: Study 2: Changes in Alignment Vector based on Feedback Received
Key: AF = Algorithmic Feedback, C = Correct, OBF = Output Based Feedback,
Parse = Failure to Parse Message

	Algorithmic Feedback		Output Based Feedback	
	Mean	SD	Mean	SD
Max	0.65	0.38	0.69	0.32
Count	0.76	0.39	0.77	0.33
Index	0.72	0.36	0.75	0.40

Table 7.17: Study 2: Mean Alignment Vector Percentage on First Attempt

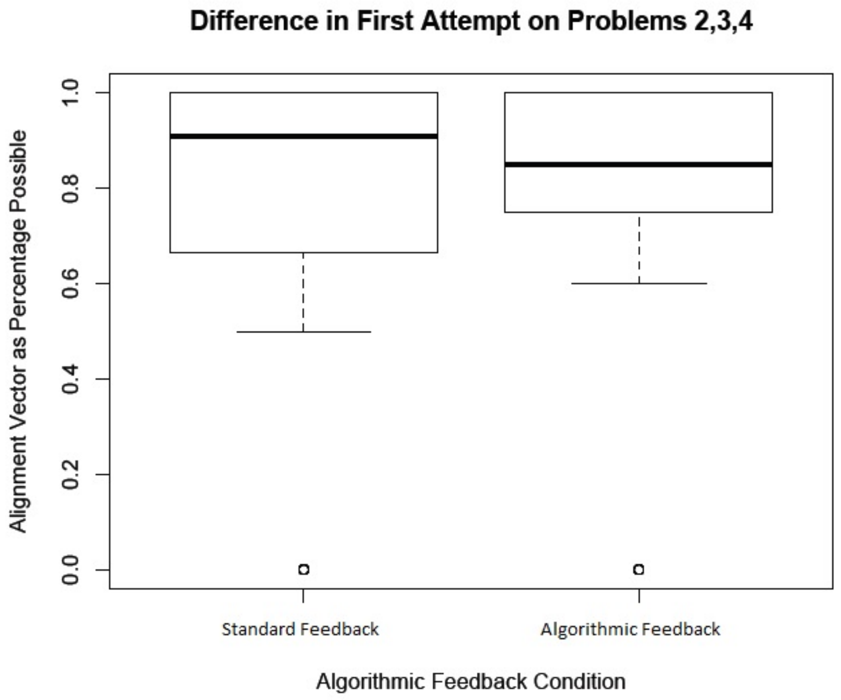


Figure 7.3: Study 2: First Attempts at Problems 2, 3, and 4

7.2.5 Algorithmic Feedback Results in No Compiler Errors

In Chapters 1 and 2, I discussed work implying that novice programmers encounter compiler errors as an artifact of larger structural issues, as opposed to mistakes in typing. The participants in this study, while novices in the larger discipline of copmuter programming, have still had almost an entire semester of practice in the syntactical structures of the Java programming language. The data below from the online study confirm the claim that student difficulties arise not out of syntactical struggles, but algorithmic ones.

In the study, 23 of the 372 submissions from participants in the algorithmic feedback condition were unable to parse due to a syntax error (11 submissions from 1 user). Aside from the failure to parse messages, *no compiler messages needed to be shown*. Students in the algorithmic feedback condition not only corrected any algorithmic errors based on the algorithmic feedback, but produced syntactically correct code as they did it. In comparison, participants in the output based feedback condition saw compiler messages 102 times out of 262 submissions (40%).

These data support the claim that the widely reported and studied struggles of novices with compilers is not an result of the struggles novices face with syntax, but instead with semantics. *This finding could help refocus the nature of pedagogical tool design research in the CS Education community*, as much of the current research [12, 28, 52] is on categorizing the type of compiler errors students receive during practice and attempting to modify the feedback produced as a result of compiler errors to be more useful for students. These works

	Algorithmic States	Feedback PDS from S0	Output Based States	Feedback PDS from S0
Sum	15	6.62	14	4.05
Max	16	4.77	15	4.29
Count	13	9.64	10	3.60
Index	20	11.80	13	4.14

Table 7.18: Study 2: Number of Observed AV States and PDS Distance by Problem

focus not on the actual types of errors the students make, but instead a categorization of what students *see* as a result of current feedback mechanisms (mostly compilers).

7.2.6 Probabilistic Distance to Solution

The Probabilistic Distance to Solution (PDS) algorithm described in Chapter 6 was applied to the data from Study 2. Table 7.18 shows the number of distinct states observed, and the PDS from start (provided code header) to a complete solution. Surprisingly, students in the output based feedback condition had fewer states, and a shorter PDS for every problem.

In Section 7.2.3, I show evidence that students in the Algorithmic Feedback condition(AFC) are more likely to make Algorithmically Productive Edits (APEs) than students in the Output Based Feedback condition (OFC). The analysis from the start state PDS and the number of observed alignment vector states would indicate participants in the AFC made edits that resulted in additional intermediate states before arriving at a final answer. This is evidenced especially in the final two problems when the PDS is more than twice the size for Count and Index in the AFC, compared to OFC.

Although the PDS results seem to counter the hypothesis that algorithmic feedback will produce better problem solving, consider that there was no significant difference between the number of submits for users in either condition or the alignment vector sum for the first attempts at problems. What appears to be occurring is that the algorithmic feedback focuses students on one particular problem with the code, which they fix relatively quickly, and then address the next incremental change. In the output based feedback condition, the student makes syntax changes resulting from compiler errors and then is forced to deal with the feedback regarding the resulting values of the algorithm, which may be related to several algorithmic mistakes. Students then struggle with the output based feedback, often engaging in various forms of shotgun debugging, trying lots of code mutations that have little foundation in actual syntax or algorithmic structure, until they find a mutation that satisfies the code requirements.

With two very different models of practice, I hypothesize that different learning is taking place. In the next section, I offer case studies that present a clear picture of one OFC participant engaging in shotgun debugging, and a second and third instance where algorithmic feedback targeted the specific error a student was having and produced an immediate fix to the problem. The case studies highlight the focus provided by the algorithmic feedback and help to clarify the iterative process all three students take.

7.3 Case Studies: Impact of Feedback on Individual Students

Throughout this thesis I have made the argument that novices have difficulty connecting the output based feedback to the required algorithmic components necessary to solve common practice problems. In this section, I present examples of students with both Algorithmic and Output-Based feedback. In the examples, the feedback received by the student is shown, and the student misconceptions or difficulties are discussed to demonstrate the problem solving process taken by each student.

In each case study I will detail what problem the student is working on, how many submissions occurred prior to the given submissions, and how many submissions after the given submissions the participant needed to solve the problem. I have created names for the participants in each case study to make it easier to refer to them in the qualitative analysis. These names are not the real names of the student and selected by the author, the gender of the name has no meaning.

7.3.1 Case Study 1: Accessing a Single Element

The goal of the first case study is to demonstrate the impact of the mismatch between compiler messages and output based feedback with a common student misconception. The first case study focuses on Sarah, a student who is solving the first problem, finding a sum, and who is in the output based feedback condition and seeing story context problems. Sarah is struggling with accessing a single element of her array in order to accumulate a sum of all the items. The correct code for accessing a single element is items[i]. It has taken Sarah 145 submissions to get to this point, 42 of those submissions were duplicate from the immediately previous submission. (The duplicates were distributed across the 145 submissions with the longest consecutive set of duplicates being 7 submissions.) During those submissions, Sarah struggled with placement of } and { characters to appropriately mark the bounds of her code, and constructing the for loop. It took her 14 tries to get an initial submission that compiled (just a declaration and return of a sum variable) and then she took another 116 tries to get another compilable solution (struggling with placement of } and { as well as how to access the length of the array for a for loop).

In Table 7.19, I show the series of submissions where Sarah struggles specifically with the correct way to access an element from the list. Notice that she is using items.length as the value to add to her sum in submission 1. From subsequent submissions it appears that she believes the .length is necessary for any use of items. The two submissions made (1 and 2) were 46 seconds apart, indicating that it was not just a quick button press causing a duplicate. Notice that the output based feedback produced a different set of numbers for the two duplicate submissions (1 and 2). Each submission was checked against 100 randomly generated data sets, as well as additional data sets that were specified when creating the problems to check for edge cases.

Sarah recognized that her method returned the same value each time (400) although the numbers in the array had changed. This lead her to make a change on submit 3 and include the [i] as a part of the sum statement. Unfortunately, she still left the .length and produced a compiler error about dereferencing an integer value. The error is because you cannot find the .length of an integer (the number stored in items[i], however Sarah moves the [i] to the sum instead of removing the .length. It took Sarah 3 more submits to solve the problem, first resubmitting code exactly the same as submission 1. Sarah's other two submissions first removed the .length from items, and then adding the [i] for a correct solution.

Sarah's attempts highlight the mismatch of the feedback regarding syntax and the mistakes commonly made by novices. In the first two examples, Sarah had code that compiled, however did not correctly access each individual element to add to the sum. Because the compiler is meant to be generic, it cannot assume that Sarah wanted to access each element, and so all it could provide was a comparison between an expected output and actual output. If Sarah was in the Algorithmic Feedback condition she would have received a message indicating that she needed to use the brackets ([ ]) along with the array and the loop variable in order to access each element. The algorithmic feedback presented could have helped Sarah identity the problems she was having and produce a more efficient problem solving process.

7.3.2 Case Study 2: Return Placement

The second case study of Noel, is intended to demonstrate a common student misconception, and the benefit of algorithmic feedback. Noel was in the algorithmic feedback condition and incorrectly placed the return statement for the algorithm in problem 4 (find and return the index of a particular value). Noel's code is shown in Table 7.20. In the large online study, there were 254 submissions where students either omitted the return statement completely, or placed it in an inappropriate place for the algorithm. Of the 254 submissions, 159 were similar to Noel's in that all of the other alignment vector components prior to the placement of the return are correct, distinguishing those students who have achieved a reasonable correctness and only need to move an already present statement.

Of the 159 submissions, 92 were made by subjects in the output based feedback condition. The 92 submissions were made by 16 different users, 5 of whom made mistakes on at least two problems. The average number of submissions to recover from the error after seeing an error message from the compiler was 4.4, but 48 submissions were made by one user. With the outlier removed the average number of submissions is 2.2.

In the smaller study described in section 7.2, there were 33 submissions with a matching alignment vector, made by 10 users. Three of the users were in the Output Based Feedback condition and saw compiler error messages, and three of the compiler error messages were focused on the return statement[4] Of the three users who saw compiler error messages, one user replicated the error on multiple problems (Count and then Index). Of the seven users who saw the Algorithmic feedback, no user replicated the error on multiple submissions. On his next edit, Noel moved the return statement outside of the loop, correcting the

[4]The compiler processes error messages in a different order then the algorithmic feedback. The other messages seen included incompatible types, and messages regarding mismatched { and }.

	Code Submission	AV	Feedback

1	`{int sum=0;`
	`for(int i=0; i<items.length; i++){`
	`sum += items.length;}`
	`return sum;`
	`}`
		11001-00-11-110	Your findSum method returned the wrong value when I passed it [46, 25, 40, 56, 86, 68, 6, 68, 51, 52, 18, 64, 74, 11, 74, 24, 31, 71, 29, 13]: Your method should have returned 907 but it returned 400

2	`{int sum=0;`
	`for(int i=0; i<items.length; i++){`
	`sum += items.length;}`
	`return sum;`
		11001-00-11-110	Your findSum method returned the wrong value when I passed it [1, 50, 3, 84, 52, 16, 79, 12, 34, 36, 41, 16, 2, 46, 35, 25, 20, 22, 32, 61]: Your method should have returned 667 but it returned 400

3	`{int sum=0;`
	`for(int i=0; i<items.length; i++){`
	`sum += items[i].length;}`
	`return sum;`
		11111-00-11-110	Your code did not compile and generated the following errors: Line: 6: int cannot be dereferenced, 6: operator + cannot be applied to any, int

4	`{int sum=0;`
	`for(int i=0; i<items.length; i++){`
	`sum[i] += items.length;}`
	`return sum;`
		11001-00-11-110	Your code did not compile and generated the following errors: Line: 6: array required, but int found, 6: operator + cannot be applied to any,int

Table 7.19: Case Study 1: Sarah's Struggle with Accessing an Element

Code Submission	
	```
	{int count = 0;
	for(int count = 0; count < myList.length; count++){
	   if(myList[count] == value)
	       return count;
	   else
	       return -1;
	}}
	```

AV	11111-11-11-100
Feedback	Your return statement is not in an appropriate place for the algorithm, or you are missing a return statement at the end. If you place the return too early in the code, any code that comes after it will not be executed. Check your code, perhaps trace it with a few values to determine the correct place.

Table 7.20: Case Study 2: Incorrect Return Placement

algorithmic error, and modified his code in two more submissions with the appropriate statement inside the if statement, and then achieved a correct solution.

Noel's code provides an example of one of two common student mistakes or misconceptions. In Noel's code, he is checking to see if he has found the appropriate value in the array (myList[count]==value). If he does not find the right value in the first location, instead of moving on to check the second location, Noel's code will return -1 right away. Negative 1 is the indicator that the value was not found anywhere in the list. The other common mistake is to return inside the for loop, when the algorithm requires a return at the end of the code.

The algorithmic feedback lets the user know that the return statement is simply in the wrong place. The comparable output-based feedback will say "Missing return statement." Although the code does contain a return statement, it is possible for it not to execute[5], and the compiler reacts as if there is no return statement in the method. This message is often confusing for the novice, who can see the return statement in the code, when it appears that the computer cannot. The placement of a return, and the feedback about that placement, is an excellent example of how professional level tools rely on algorithmic expertise in order for a user to translate the error message into an understanding of incorrect code.

7.3.3 Case Study 3: Unsure Where to Begin

The third case study focuses on Jon, a participant from Study 2 described in this chapter. On the first problem (Sum), Jon struggled with many of the components of the alignment vector during his problem solving process. Table 7.21 shows a progression of Jon's code through the problem solving process and the algorithmic feedback he received.

[5]For example if myList.length == 0 the loop will never execute and the return will never be encountered.

Jon's first submission was just to return the myList array, and was done incorrectly. Instead of giving Jon feedback about the algorithm, the compiler would have told him that he had an incompatible return type, which also would be been fixed by Jon's second submission. The second submission in an Output Based Feedback Condition would have told Jon that is code returned the wrong value, expecting perhaps 1154 but returning 90 with the array [90, 70, 49,...].

On line 4, output based feedback would have told Jon that he made a syntax error in his return statement - Cannot find symbol variable a (on the line number of the return statement). Algorithmic feedback recognized that Jon had initialized the variable in the incorrect place for the algorithm and Jon was able to correct the error and submitted a correct solution on his next attempt.

Jon's case study highlights how a novice could struggle with the mismatch between classic compiler messages and output based feedback and the algorithmic mistakes commonly made by novices.

7.4 Conclusions

The studies presented in this chapter focus on the research question: **Will pre-compilation feedback regarding algorithmic components produce better (a) within-problem performance and (b) across problem learning?** Both online studies confirm that students who get an algorithmic feedback message are likely to make a productive edit and more likely to make a productive edit than those who receive output based feedback. The productive edits are a good measure of within problem performance, as a measure of student progress towards a correct solution. Additionally, students in the algorithmic feedback condition in Study 2 did not need to correct any compiler errors after correcting algorithmic errors.

There are no direct measures which indicate that students had increased across problem learning, as both groups showed non-significant differences in the first attempt score of problems in the sequence. Although this seems to be counter to the second part of the hypothesis, there seems to be the need for additional fine grained assessment of the individual knowledge components contained within each problem. Although the PDS on the smaller study shows an increase in the number of states and a larger overall PDS to obtain a correct solution, this result and the increased likelihood of a productive edit, together with case studies such as Number 2, could be an indication that algorithmic feedback prompts a more incremental problem solving process. This incremental problem solving process allows students to focus on the construction of the algorithm, and potentially could explain the lack of repetition of errors for students in the algorithmic feedback condition. This lack of repetition of errors for students in the algorithmic feedback condition is evidence of an increase in across problem performance by students in the algorithmic feedback condition, however additional studies are warranted to confirm this statement.

In the next chapter I present the conclusions and contributions of the entire thesis and offer ideas for future work.

	Code Submission	AV	Feedback
1	`{return myList[]; }`	0000000 00 000	In order to accomplish the task assigned, you need to access every element in the array. Try to implement (write code for) a control structure that allows you to repeat a task over many times. You will use this with other variables in order to complete your task. You also need to write at least one line of code inside the structure.

2	`{int a=myList[0];`
	`return a;`
	`}`
		00000-00-00-110	The task you are trying to achieve requires a loop to access each element in the array. Write a for loop in your code. If you already have a loop, you need at least one statement - a command followed by a ; in order it to work completely.

3	13 additional submissions where Jon struggles to format a for loop	N/A	Jon saw repeats of the previous message.

4	`{for(int x=0; x<myList.length; x++){`
	`int a = a + myList[x];`
	`}`
	`return a;`
	`}`
		11111-00-00-110	In order to solve your problem you need to maintain a particular state in a variable. This state will give you the answer you are seeking. You have not correctly created or assigned a starting value to the state variable.

Table 7.21: Case Study 3: Unsure Where to Begin

Chapter 8

Conclusions

8.1 Research Focus

The research presented in this thesis focuses on the design, implementation, and testing of an academic intervention to help novice programmers develop expertise in writing simple array algorithms. Although practice is necessary for the development of expertise, appropriate feedback during novice practice can be used to help students see the algorithmic abstractions in the code they are implementing.

The two main research questions addressed in this thesis are: **(1) Can a pre-compilation feedback mechanism be constructed that operates with reasonable accuracy (85% of student generated submissions)? and (2) Will feedback regarding algorithmic components presented to students pre-compilation produce better (a) within-problem performance and (b) across-problem learning?**

To answer the research questions, this thesis presents research with contributions to the Computer Science, Learning Sciences, and Computer Science Education domains. I used qualitative and quantitative analyses to validate a model of problem components and likely student errors (Chapter 4), and I used quantitative analysis with multiple student generated datasets to evaluate and refine the feedback mechanism (Chapter 5). I developed new quantitative methodologies and evaluated them with student data (Chapter 6). Finally, I conducted two online studies to evaluate the efficacy of algorithmic feedback and offer evidence that students who see an algorithmic feedback message make significantly more algorithmically productive edits than students who see output based feedback.

8.1.1 Summary of Contributions

The research presented in this thesis offers contributions, both large and small, to the fields of Computer Science, Learning Sciences, and Computer Science Education. Table 8.1 lists the contributions of the research presented in Chapters 4-7 and indicates the relevant fields for each contribution. Subsequent subsections provide increased detail about each contribution and implications for the relevant literature.

		Learning Sciences	Computer Science	CS Education
	Major Contributions			
1	I demonstrated that students who see feedback regarding algorithmic abstractions are more likely to make a productive edit than students who see traditional output based feedback.			X
2	I applied static analysis techniques for pre-compilation feedback generation.		X	X
3	I designed an implementation of a model of complex, non-linear problem spaces (PDS) and applied the model to compare research conditions.	X	X	X
4	I discovered that students of different proficiency levels both use abstract, and non-abstract statements when describing a programming task, however students with a higher proficiency level transition between types of statements more frequently.	X		X
	Minor Contributions			
5	I discovered that students who correct code based on algorithmic feedback will also correct compiler errors.			X
6	I created and used a model for evaluating abstraction at statement level (instead of whole paragraph or answer).			X
7	I defined and validated a model of algorithmic components for simple array algorithms, and used the model components to measure within problem progression and performance.	X		X
8	I performed an HCI exercise in case studies illustrating the inappropriateness of standard error messages compared to actual novice errors.	X		X
9	I examined and used within problem progress, increasing the granularity of the compile-edit cycle.			X
10	I modified the SOLO taxonomy to use at the utterance level of granularity.	X		X

Table 8.1: Contributions and Domains Offered by this Thesis

8.1.2 Feedback Regarding Algorithmic Abstractions Produces Better Performance

In this thesis, as a contribution to the Learning Sciences, I seek to answer the question **Will feedback regarding algorithmic components presented to students pre-compilation produce better (a) within-problem performance and (b) across problem learning?** Although students are able to express algorithms in natural language [75], prior work has demonstrated that students struggle with code production [34] and the writing of algorithms in formal language [53, 77]. Related work has shown that a correlation exists between students' ability to abstract when describing code [49, 72] and proficiency at code production [48]. Measures of proficiency in the abstraction studies have often relied on the researcher hand scoring the student produced code against a rubric. In this work I create a model inspired by such rubrics and use an automated assessment mechanism to compare student code to the rubric-based model.

The abstractions measured in prior work have been explicitly tied to students' ability to describe algorithmic components and the role of those components in an algorithm. Abstractions about a domain have long been linked to expertise [15], however little work has been done in Computer Science Education, or the Learning Sciences to connect explicit instruction in abstractions with measures of performance. In prior work, I demonstrated that instruction in this specific type of abstraction produced more sophisticated answers, both when students write answers in a text box and in the selection of multiple choice items [82].

Underlying much of the abstraction research is the notion that students will develop these abstractions as a result of course instruction [48]. Yet much of the instruction focuses on the individual components of the algorithm (if statements, loops, arrays) and relies on students to build the expert structures over time. In current methods of practice, students struggle with tools that provide low level feedback and need to infer the algorithmic corrections necessary to construct complete algorithms. In this work I addressed the expert blind spot of many computer scientists, that the algorithmic components can be taught separately without direct instruction on the algorithmic placement and usefulness and still have students apply and use the components together in meaningful ways.

In this thesis, I have taken abstraction and proficiency correlation research further and shown that feedback containing information about algorithmic components can be useful for novices' programming problems, by producing more efficient edits of code (Contributions 1, 7, 9). Additionally, the algorithmic component feedback produced an unexpected result. Participants in the final study, who were in the algorithmic feedback condition, did not see a single error message from the compiler after correcting all of the algorithmic errors in their code[1](Contribution 5).

The approach taken in this thesis offers a variety of contributions to the CS education community. As recent as the summer of 2014, researchers have continued to focus on the

[1]Students did see compiler messages in 23 out of 372 feedback instances for the Algorithmic Feedback Condition when the code was unreadable by the algorithmic parser. Eleven (11) of those messages were to a single user, and 6 of the messages were corrected with one edit. Students went on to correct algorithmic errors after reaching parsable code.

type of compilation errors received by students as an important research question, and look for ways to improve generic compiler messages as a means to improve novice practice [12]. At the same time, the community is also focused on research questions at a very large granularity, attempting to predict course level outcomes with grades on assignments or assessments with multiple learning objectives or parts [64]. The work conducted in this thesis presents an example of identifying a model of multiple components in a single problem type, and a fine-grained analysis of student performance, not based upon aggregate scores but a quantitative metric showing within-problem progression (Contributions 3, 9).

8.1.3 Feedback Mechanisms and Code Evaluation for the Novice

In this thesis, as a contribution to Computer Science, I seek to answer the question **Can a pre-compilation feedback mechanism be constructed that operates with reasonable accuracy (85% of student generated submissions)?** Despite evidence that students struggle with the writing of code in modern computer programming languages [34], much of the research into the creation of pedagogical IDEs has focused on correcting the compiler error messages received by students during practice [28]. Fisler et al. even received a best paper award in 2011 from SIGCSE[2] for measuring the types of error messages encountered by students while programming. The contributions of this thesis are in the development of a pre-compilation feedback mechanism, as well as the postulation that pre-compilation feedback would be useful to the novice and produce more efficient practice.

As a computer science contribution, I applied static analysis techniques to imperfect, student-generated code submissions to generate pre-compilation feedback (Contribution 2). The evaluation of student code against an algorithmic model, not for absolute correctness, but instead for the mere presence of algorithmic components is a unique approach. The model and feedback generation mechanism was created with generalizability in mind, so that it could be adapted to other problems with similar array components, and used with additional output based tests specific to the intent of the code.

While analyzing student code submissions, it became clear that a count of the number of submissions or the ability of a student to produce a correct solution in a future submission were not granular enough measures of student knowledge or problem solving. With generalizability in mind for non-linear problem solving situations, the Probablistic Distance to Solution (PDS) metric described in Chapter 6 and published in [81] was designed and implemented (Contribution 3). The publication of PDS has been cited at least 6 times since publication and used outside of computer science education research.

8.2 Future Work

Because of the diverse nature of this thesis, there are a variety of directions future work could take. In this section, I offer two lines of research and a connection to current tools being built. At a macro level, the contributions to Educational Data Mining and the

[2]SIGCSE has a conference attendance of 1200+ faculty practioners and CS Education researchers.

unique approach of offering pre-compilation feedback on algorithmic components will, I hope, inspire research outside of the direct line of my work.

8.2.1 Larger Treatment for Across Condition Gains

Although algorithmic feedback produced significant improvement in the likelihood a participant would make an algorithmically productive edit, there was no effect of the algorithmic condition on the measured outcomes. In a future study, I would like to increase the dosage, including more problems for students to complete over multiple sessions with the tutor.

An online experiment could be created where students had to solve 1-3 problems a week over multiple weeks. Faculty could assign the work as a part of weekly problem sets, and problems would only become visible at the appropriate time, preventing students from solving all the problems in one sitting.

I hypothesize that continued exposure to the algorithmic feedback would increase the effect on both within and across problem outcomes. Additionally, a second think aloud study could be conducted to determine if students change the frequency of their transitions between No Abstraction statements and Macro Relational statements as described in Chapter 4.

8.2.2 Additional Problems and Feedback Refinement

In the studies presented in this thesis, students were solving the same four problems regarding simple array algorithms. There is a much larger set of array algorithms that students could be asked to write that still use the same set of algorithmic components. Additional problems could be added to the AbstractTutor system and a variety of studies could be run to evaluate student learning. Additionally, isomorphic problems could also be tested. Currently each of the four problems represent a slightly different algorithm; another extension would be to include the same algorithm, but with different variable names or contexts as a part of the series.

First, a small learning gain was shown by looking at the first submission alignment vectors (Figure 7.1). Although there are obvious differences between problems 1, 2 and 3, by problem 4 it appears the curve has leveled off. Additional studies with more problems could be used to determine if there is an optimal number of problems for students to solve for practice in one or multiple sittings. Findings from such a study could be useful, especially for textbook writers who often include 15-20 exercises at the end of each chapter.

Second, although there were no effects of problem name on student performance, it would be interesting to compare problems with exactly the same pattern of alignment vector to each other. For example, the sum problem did not require the use of an if statement, while the other problems did. What would be the effects of eliminating the sum problem from the sequence? What would be the effects of including more problems without if statements? I hypothesize that additional problems with the same pattern of alignment vector will produce little to no difference from the current study. Both studies presented in Chapter 7 show a decreased effect of problem number after the second problem, indicating that after students struggled with the first problem there was little to no gain between subsequent problems. Yet, students were not submitting algorithmically correct code for the final problems and a more detailed study of what recurring errors persist could help create better practice for novices.

Third, in this thesis I classified student errors by the type of feedback they prompted from the AbstractTutor system. Although the classification was useful for the evaluations of the algorithmic feedback, it was not completely descriptive of the type of errors the students were making. Many students struggled with the semantics of the Java language, resulting in either an error from the compiler or an algorithmic feedback message. A more detailed analysis and categorization of the types of errors students make, using multiple submissions and think alouds to help infer student purpose could be useful for refining feedback and practice.

Finally, the feedback messages themselves were never fine tuned in this thesis, except for clarity during the think aloud (some grammar and tenses were changed). Additional studies should be conducted to find the appropriate wording for the algorithmic components and accompanying abstractions to produce the most efficient student practice.

8.2.3 Models of Student Proficiency

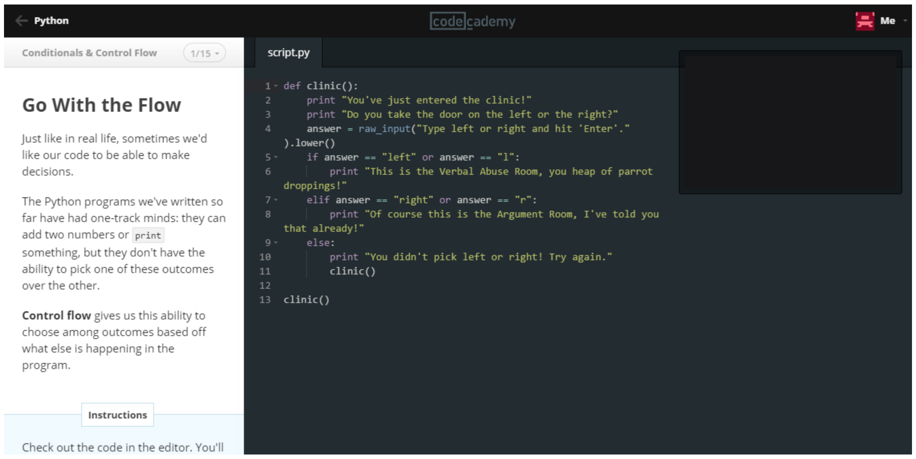
In this thesis I use several models of student proficiency to both categorize and quantitatively measure student performance. In Chapter 4, I use the number of submissions a student makes as a rough measure of proficiency that can help categorize students into high and low performers. Additionally, I use the sum of alignment vector components to measure the "correctness" of a particular submission throughout the problem solving process. The strategy of using the alignment vector sum is similar to the way instructors will grade student work by adding rubric points to provide a score to a particular student answer. Neither of these methodologies provide detailed information about student proficiency, although the alignment vector itself can be used to infer student knowledge.

In Chapter 6, I define two ways to measure changes to the correctness of a solution in the problem solving process. By looking for algorithmically productive edits, and student progression through a problem to a correct solution, I can infer the algorithmic components a student is focused on learning in a particular problem.

In future work it would be interesting to try to order the algorithmic components in a learning progression. To find a learning progression, a model of student proficiency could be developed that takes into account both the static evaluation of student code and the modifications the student makes over time.

8.2.4 Models of Student Problem Solving

The problem solving process exposed by the Probabilistic Distance to Solution metric prompts an interesting question. Students in the algorithmic feedback condition were more likely to make small, productive edits during problem solving, while students in the output based feedback condition made unproductive edits for a time and then fixed a large number of problems in a few edits. This phenomenon of unproductive edits replicates the observations made in [34]. I hypothesize that the second model, a large number of



Go With the Flow

Just like in real life, sometimes we'd like our code to be able to make decisions.

The Python programs we've written so far have had one-track minds: they can add two numbers or `print` something, but they don't have the ability to pick one of these outcomes over the other.

Control flow gives us this ability to choose among outcomes based off what else is happening in the program.

Check out the code in the editor. You'll

```
def clinic():
    print "You've just entered the clinic!"
    print "Do you take the door on the left or the right?"
    answer = raw_input("Type left or right and hit 'Enter'."
).lower()
    if answer == "left" or answer == "l":
        print "This is the Verbal Abuse Room, you heap of parrot
droppings!"
    elif answer == "right" or answer == "r":
        print "Of course this is the Argument Room, I've told you
that already!"
    else:
        print "You didn't pick left or right! Try again."
        clinic()

clinic()
```

Figure 8.1: Codecademy Novice Practice Environment

unproductive edits followed by a large edit, is similar to insight problem solving. The student struggles with the code until she has an insight (or help seek from another source) and then produce a mostly correct answer.

An interesting research question for future work would be the exploration of the implications of the "insight" problem solving on student metacognition and learning. If students need to have an insight as opposed to methodically applying modifications, does that change their ability to reflect on the individual components and impair their learning? Additionally, what effect on student motivation does the constant unproductive edits cause? These questions could help provide information to the ongoing questions in the CS Education community surrounding student attrition from majors and low performance in early courses.

8.2.5 Ongoing Work for New Products

Over the past two years, there has been an explosion of online "learn to code" tools and companies. Almost all of the tools simply embed output based feedback mechanisms from professional level tools into an instructional website. For example, the codecademy website [84] has a coding window placed on the right of an instructional pane. The coding window has an error message provided in the upper right corner as shown in Figure 8.1.

The work presented in this thesis could be used to create feedback more appropriate to the novice in these environments by aligning feedback with instructional goals instead of compiler errors.

8.3 Acknowledgements

This work was supported through the Program for Interdisciplinary Education Research (PIER) at Carnegie Mellon University, funded through Grant R305B040063 to Carnegie Mellon University, from the Institute of Education Sciences, US Department of Education.

The opinions expressed are those of the author and do not represent the views of the Institute or the US Department of Education.

Additionally, I would like to thank the founding team of TutorTechnologies, Thomas Harris, Christy McGuire and Jonathan Steinhart. Without TutorTechnologies the AbstractTutor platform would have been much more simplistic. I would also like to thank the AlphaLab Pittsburgh accelerator for funding TutorTechnologies.

A large thank you goes to the SIGCSE community for offering access to students. Thank you to the teachers who facilitated the online studies by distributing registration codes and asking students to participate. Additionally, a special thank you to Mark Guzdial whose blog post helped advertise the opportunity.

Appendix A

AbstractTutor Problems and Feedback

A.0.1 The Problems and Reference Solutions

A.0.2 Feedback Messages

Each algorithmic component that was evaluated pre-compilation had two feedback messages associated with it. The participant would see the first feedback message after the first time the error was detected on a problem submission as the primary error (Next error to be corrected). If the participant was unable to correct the error, they then saw the secondary message.

Including a Repetition Structure

- Primary Message - In order to accomplish the task assigned, you need to access every element in the array. Try to implement (write code for) a control structure that allows you to repeat a task over many times. You will use this with other variables in order to complete your task. You also need to write at least one line of code inside the structure.
- Secondary Message - The task you are trying to achieve requires a loop to access each element in the array. Write a for loop in your code. If you already have a loop, you need at least one statement - a command followed by a ; in order for it to work correctly.

Using the length of the List

- Primary Message - The variable myList is a parameter. This means another part of parts of the program decide what is stored in it, and it may change every time code is run. Use a variable in the loop combined with a property of myList in order to determine the number of steps for the loop to take.
- Secondary Message - Use myList.length to determine the number of steps for the loop to take.

Referencing Any Element From the List with [ ]

- Primary Message - The loop you wrote supports accessing each element in your array (myList), yet you do not access the array myList within the loop. These components must work together for the problem to be solved.
- Secondary Message - You must access myList[] within the context of the loop by using your variable inside the [] (for example myList[i]).

Using the Loop Variable to Access

- Primary Message - The loop you wrote supports accessing each element in your array (myList), yet you do not use the loop variable when accessing your array. These components must work together for the problem to be solved.

Problem	Numerical Context	Story Context
Sum Description	Write a method to find and return the sum of all the values stored in the array `public int findSum(int []myList){`	The array items contains the price of items on a sales receipt. Write a method to find and return the total (sum) of all the items on the receipt. `public int findSum(int []items){`

Solution	`int sum = 0;`
	 `for(int i=0; i<myList.length; i++){`
	   `sum += myList[i];`
	 `}`
	 `return sum;`
	 `}`
		`int sum = 0;`
		 `for(int i=0; i<items.length; i++){`
		   `sum += items[i];`
		 `}`
		 `return sum;`
		 `}`

Max
 	
Description
	Please complete the method findMaximum below. The method takes an array of numbers called myList as a parameter and returns the maximum values stored in the array myList.
	
	`public int findMaximum(int []myList){`
		Please complete the method findMaximum below. The method takes an array which contains the MPG of several vehicles. Find the largest MPG stored in the list.
		
		`public int findMaximum(int`
		                  `[]vehicleMPG) {`

Solution	`int max = myList[0];`
	`for(int i=0; i< myList.length; i++){`
	    `if(max < myList[i])`
	      `max = myList[i];`
	    `}`
	 `return max; }}`
		`int max=vehicleMPG[0];`
		`for(int i=0;i<vehicleMPG.length;i++){`
		  `if(max<vehicleMPG[i]){`
		    `max = vehicleMPG[i];`
		  `}`
		  `return max;`
		`}`

Table A.1: Sum and Max Descriptions and Solutions

Count
 	
Description
	Please complete the method countLessThan below. The method takes an array of numbers called myList as a parameter and returns the number of non-negative items(including 0) less than the parameter value in the array myList.
	
	`public int countLessThan(int []myList,`
	`int value){`
		Please complete the method countLessThan below. The method takes an array of grades from a midterm exam. Find the number of positive scores (greater than or equal to 0) less than the passing grade given.
		
		`public int countLessThan(int []scores,`
		`int passingGrade){`

Solution	`int count = 0;`
	`for(int i=0; i< myList.length; i++){`
	`  if(myList[i] < = 0)`
	`    count++;`
	` }`
	` return count;`
	` }`
		`int count=0;`
		`for(int i=0; i<scores.length; i++){`
		` if(scores[i] >=0 && scores[i] < value)`
		`     count++`
		` }`
		` return count;`
		` }`

Index
 	
Description
	Please complete the method indexOf below. The method takes an array of numbers called myList as a parameter and returns the first index of the parameter value if it is contained in the array. If it is not contained the method should return -1.
	
	`public int indexOf(int []myList,`
	`int value){`
		Please complete the method indexOf below. The method takes an array of office numbers as a parameter and returns the index of the first Office Number that matches the value idno if it is contained in the array. If it is not contained the method should return -1.
		
		`public int indexOf(int []roomNumbers,`
		`int idno){`

Solution	`for(int i=0; i<myList.length; i++){`
	`  if(myList[i] == value)`
	`    return i;`
	`  }`
	`  return -1;`
	`  }`
		`for(int i=0;i<roomNumbers.length;i++){`
		`  if(roomNumbers[i] == value)`
		`    return i;`
		`  }`
		`  return -1;`
		`  }`

Table A.2: Count and Index Descriptions and Solutions

- Secondary Message - Use your loop variable inside the [] after myList (for example myList[i]).

Only Accessing the Current Element

- Primary Message - You are attempting to access a neighboring array element with a +1. For this algorithm you should be able to correctly and efficiently implement your code while only accessing one element at a time. Think about how you could solve the problem only looking at one number from the list at a time.
- Secondary Message - Do not use a +1 in the []. Instead update your state variable (what is keeping track of the answer through the problem) only for each individual element.

Making A Comparison with an If

- Primary Message - The problem you are trying to solve requires that you only work with SOME of the information in the array, not all if it. In order to accomplish this you will need to select some of the information using a comparison of your code. Think about how to make a comparison with each element to determine which ones are relevant to the problem you are solving.
- Secondary Message - Write an if statement inside the loop to check for relevant items.

Utilizing an Element in the Comparison

- Primary Message - The problem you are trying to solve requires that you only work with SOME of the elements in the array, not all of them. In order to accomplish this you will need to check each individual element to see if it is relevant to the solution. Think about how and where you would need to access each element in order to check to see if it is useful.
- Secondary Message - Access each element inside the if statement by using the name of the array and [] along with the loop control variable.

Initializing State Variable

- Primary Message - In order to solve your problem you need to maintain a particular state in a variable this state will give you the answer you are seeking. You have not correctly created or assigned a starting value to the state variable.
- Secondary Message - Create and initialize a variable before the loop.

Using the State Variable

- Primary Message - In order to solve your problem you need to maintain a particular state in a variable this state will give you the answer you are seeking. With each element in the array you need to update that state to reflect the changes required by the current element.

- Secondary Message - Update your state variable within the loop in order to keep track of required information.

Attempting Any Return

- Primary Message - The method you are writing requires that you return a value as your answer. Look over your code carefully and decide at what point do you have your final answer, and return that value.
- Secondary Message - After the loop has executed, type the word "return" and the name of the variable that has the answer to the problem.

Correctly Placing the Return

- Primary Message - Your return statement is not in an appropriate place for the algorithm, or you are missing a return statement at the end. If you place the return too early in the code, any code that comes after it will not be executed. Check your code, perhaps trace it with a few values to determine the correct place.
- Secondary Message - After the loop has executed, type the word "return" and the name of the variable that has the answer to the problem.

Appendix B

Alignment Vectors Observed and Number of Instances

This appendix contains a table with all of the observed alignment vectors with more than 20 observations and a count of the times they appeared in the large online study (Study 1 of Chapter 7). This is a more complete list of the vectors found in Table 5.6.

Alignment Vector	Count
11111-11-11-110	935
11111-00-11-110	515
11111-11-00-100	181
11111-11-11-111	144
11111-11-11-000	115
11111-11-00-110	83
11111-11-00-111	74
11001-00-11-000	67
11111-11-11-100	60
11111-10-11-110	54
11111-00-11-000	51
11001-10-11-110	50
10111-00-11-110	49
11001-10-11-000	46
11111-00-11-100	46
10001-00-00-000	40
11001-00-00-000	40
01111-00-11-110	39
10001-00-11-110	35
11001-00-00-110	35
11111-00-00-110	34
11111-00-00-100	28
00101-00-11-000	26
10111-11-11-110	23
10111-11-00-100	22
00000-00-00-100	20

Table B.1: Count of Alignment Vector Values from Online Study 1

Bibliography

[1] Ap computer science a exam, free response questions. Accessed from the Web (September 2013), May 2010.

[2] Beth Adelson. When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology:Learning, Memory and Cognition*, 10:483–495, 1984.

[3] Hirotugu Akaike. A new look at the statistical model identification. *IEEE Transactions on Automat*, 19(6):716–723, 1974.

[4] Christopher Alexander, S Ishikawa, and M Silverstein. Pattern languages. *Center for Environmental Structure*, 2, 1977.

[5] C. Areias and A. Mendes. A tool to help students to develop programming skills. *Proceedings of the 2007 International Conference on Computer Systems and Technologies*, 2007.

[6] Owen Astrachan, Garrett Mitchener, Geoffrey Berry, and Landon Cox. Design patterns: an essential component of cs curricula. *SIGCSE Bull.*, 30:153–160, March 1998.

[7] R.S. Baker, A. Corbett, K. Koedinger, and A. Wagner. Off-task behavior in the cognitive tutor classroom: When students "game the system". *In Proceedings of the ACM Conference on Computer-Human Interaction, CHI 2004*, 2004.

[8] R.S. Baker, J. Walonoski, N. Heffernan, I. Roll, A. Corbett, and K Koedinger. Why students engage in "gaming the system" behavior in interactive learning environments. *Journal of Interactive Learning Research*, 19(2):185–224, 2008.

[9] Byron Weber Becker. Teaching cs1 with karel the robot in java. *SIGCSE Bulletin*, 33:50–54, February 2001.

[10] Yoav Bergner, Zhan Shu, and Alina A von Davier. Visualization and confirmatory clustering of sequence data from a simulation-based assessment task. 2013.

[11] J.B. Biggs and K.F. Collis. *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, 1982.

[12] Neil C.C. Brown and Amjad Altadmri. Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 43–50, New York, NY, USA, 2014. ACM.

[13] Jean-Marie Burkhardt, Françoise Détienne, and Susan Wiedenbeck. Mental represen-

tations constructed by experts and novices in object-oriented program comprehension. *CoRR*, abs/cs/0612018, 2006.

[14] N. Charness. Memory for chess positions: Resistance to interference. *Jounral of Experimental Psychology: Human Learning and Memory*, 2:641–653, 1976.

[15] N Charness, R Krampe, and U Mayr. *The role of practice and coaching in entrepreneurial skill domains: An International comparison of life-span chess skill acquisition*, pages 51–80. Erlbaum, 1996.

[16] Michelene Chi, Miriam Bassok, Matthew Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13:145–182, 1989.

[17] JA Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 70(4):213–220, 1960.

[18] Albert T Corbett and John R Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, 4(4):253–278, 1994.

[19] Q. Cutts, S. Esper, M. Fecho, S. Foster, and B. Simon. The abstraction transistion taxonomy: developing desired learning outcomes through the lens of situated cognition. *ICER '12: Proceedings of the ninth annual International conference on Computing Education Research*, pages 63–70, 2012.

[20] Wanda P. Dann, Stephen Cooper, and Randy Pausch. *Learning To Program with Alice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[21] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. Evaluating a new exam question: Parsons problems. In *Proceedings of the Fourth international Workshop on Computing Education Research*, pages 113–124. ACM, 2008.

[22] Heidi C Dulay and Marina K Burt. Natural sequences in child second language acquisition1. *Language learning*, 24(1):37–53, 1974.

[23] Stephen H Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 148–155. ACM, 2003.

[24] Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bull.*, 36(1):26–30, March 2004.

[25] Dave Feinberg. A visual object-oriented programming environment. *SIGCSE Bull.*, 39(1):140–144, March 2007.

[26] Mingyu Feng, Neil T Heffernan, and Kenneth R Koedinger. Predicting state test scores better with intelligent tutoring systems: developing metrics to measure assistance required. In *Intelligent Tutoring Systems*, pages 31–40. Springer, 2006.

[27] Eric Fernandes and Amruth Kumar. A tutor on subprogram implementation. *The Journal of Computing Sciences in Colleges*, 20:36–46, 2005.

[28] Kathi Fisler, Guillaume Marceau, and Shriram Krishnamurthi. Measuring the effec-

tiveness of error messages designed for novice programmers. *In Proceedings of the Special Interest Group on Computer Science Education*, 2011.

[29] Michelle Friend and Robert Cutler. Efficient egg drop contests: How middle school girls think about algorithmic efficiency. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 99–106, New York, NY, USA, 2013. ACM.

[30] Cindy Hmelo-Silver and Merave Pfeffer. Comparing expert and novice understanding of a complex system from the perspective of structures, behaviors, and functions. *Cognitive Science*, 28:127–138, 2004.

[31] I-Han Hsiao, Shuguang Han, Manav Malhotra, Hui Soo Chae, and Gary Natriello. Survey sidekick: Structuring scientifically sound surveys. In *Intelligent Tutoring Systems*, pages 516–522. Springer, 2014.

[32] IEEE, editor. *IEEE Standard Glossary of Software engineering Terminology*, volume 610.12. IEEE Standard, 1990.

[33] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[34] Matt Jadud. A first look at novice compilation behavior using bluej. *Computer Science Education*, 15(1), 2005.

[35] Matthew Johnson and Tiffany Barnes. Visualizing educational data from logic tutors. In *Proceedings of the 10th International Conference on Intelligent Tutoring Systems - Volume Part II*, ITS'10, pages 233–235, Berlin, Heidelberg, 2010. Springer-Verlag.

[36] W. Lewis Johnson and Elliot Soloway. Proust: Knowledge-based program understanding. *IEEE Transactions on Software*, 1985.

[37] GG Koch JR Landis. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[38] Ken Koedinger and Vincent Aleven. Exploring the assistance dilemma in experiments with cognitive tutors. *Educational Psychology Review*, 19(3):239–264, 2007.

[39] M. Kolling. The problem of teaching object-oriented programming, part 1: Languages. *Journal of Object-Oriented Programming*, 11:8–15, 1999.

[40] Michael Kölling and John Rosenberg. An object-oriented program development environment for the first programming course. *SIGCSE Bull.*, 28(1):83–87, March 1996.

[41] Amruth Kumar. Learning programming by solving problems. *Proceedigns of IFIP Working Group, Working Conference on Informatics Curricula*, pages 152–164, 2002.

[42] Rohit Kumar. Cross-domain performance of automatic tutor modeling algorithms. In *Workshop on Graph-Based Educational Data Mining.*

[43] H. Chad Lane and Kurt Vanlehn. Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education*, 15:183–201,

2005.

[44] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 361–370, New York, NY, USA, 2007. ACM.

[45] Nguyen-Thinh Le and Wolfgang Menzel. Constraint-based error diagnosis in logic programming. In *ICCE*, pages 220–227, 2005.

[46] Nguyen-Thinh Le and Wolfgang Menzel. Using constraint-based modelling to describe the solution space of ill-defined problems in logic programming. In *Advances in Web Based Learning–ICWL 2007*, pages 367–379. Springer, 2008.

[47] Marcia C. Linn and Michael J. Clancy. The case for case studies of programming problems. *Commun. ACM*, 35(3):121–132, March 1992.

[48] Raymond Lister. The neglected middle novice programmer: Reading and writing without abstracting. *In Proceedings of the 20th Annual Conference of the National Advisory Committee on Computing Qualifications*, 2007.

[49] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, 36:119–150, June 2004.

[50] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *ICER '08: Proceeding of the Fourth international Workshop on Computing Education Research*, pages 101–112, New York, NY, USA, 2008. ACM.

[51] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10(4):16:1–16:15, November 2010.

[52] G. Marcau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. *SIGCSE'11*, March 2011.

[53] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bull.*, 33:125–180, December 2001.

[54] A. Mitrovic, K. Koedinger, and B. Martin. A comparative analysis of cognitive tutoring and constraint-based modeling. *Lecture Notes in Computer Science*, 2702:313–322, 2003.

[55] Briana B. Morrison. Using cognitive load theory to improve the efficiency of learning to program. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, ICER '13, pages 183–184, New York, NY, USA, 2013. ACM.

[56] Orna Muller. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the first international workshop on Computing education research*, ICER '05, pages 57–67, New York, NY, USA, 2005. ACM.

[57] Laurie Murphy, Renee McCauley, and Sue Fitzgerald. 'explain in plain english' questions: Implications for teaching. *Proceedings of the fourty-third ACM Special Interest Group on Computer Science Education*, 2012.

[58] E. Odekirk-Hash and J. Zachary. Automated feedback on programs means students need less help from teachers. *Proceedings of the thirty-second SIGCSE Technical Symposium on Computer Science Education*, 2001.

[59] Nick Parlante. Codingbat: Practice java and python problems.

[60] Nick Parlante. Codingbat:code practice. Website, 2011.

[61] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. A survey of literature on the teaching of introductory programming. In *ACM SIGCSE Bulletin*, volume 39, pages 204–223. ACM, 2007.

[62] Frank Pfenning. Teaching imperative programming with contracts at the freshmen level. Accessed from http://www.cs.cmu.edu/ fp/papers/pic11.pdf 7/11/2014.

[63] P. Pirolli and M. Recker. Learning strategies and transfer in the domain of programming. *Cognition and Instruction*, 12:235–275, 1994.

[64] Leo Porter, Daniel Zingaro, and Raymond Lister. Predicting student success using fine grain clicker data. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 51–58, New York, NY, USA, 2014. ACM.

[65] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking api protocol conformance with mined multi-object specifications. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 925–935, Piscataway, NJ, USA, 2012. IEEE Press.

[66] Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. Self-efficacy and mental models in learning to program. pages 171–175, 2004.

[67] Graham Rawlinson. The significance of letter position in word recognition. *Ph.D. Thesis, Nottingham University*, 1976.

[68] Kelly Rivers and Kenneth Koedinger. Automatic generation of programming feedback: A data driven approach. *The First Workshop on AI Supported Education for Computer Science (AIEDCS 2013)*, 2013.

[69] W. Sack and E. Soloway. From meno to proust to chiron: Ai design as iterative engineering: Intermediate results are important! *Proceedings of the Invited Workshop on Computer-Based Learning Environments*, 1998.

[70] Dean Sanders and Brian Dorn. Classroom experience with jeroo. *J. Comput. Sci. Coll.*, 18(4):308–316, April 2003.

[71] Dean Sanders and Brian Dorn. Jeroo: a tool for introducing object-oriented programming. *SIGCSE Bull.*, 35:201–204, January 2003.

[72] Carsten Schulte. Block model: an educational model of program comprehension as a tool for a scholarly approach to teaching. *Proceedings of the Fourth Annual Workshop on Computer Science Education*, 2008.

[73] Carsten Schulte, Teresa Busjahn, Tony Clear, James Paterson, and Ahmad Taherkhani. An introduction to program comprehension for computer science educators. In *ITiCSE '10: Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, pages 108–112, New York, NY, USA, 2010. ACM.

[74] Beth Simon, Dennis Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Common sense computing(episode 4): Debugging. *Computer Science Education*, 18:117–133, 2008.

[75] Beth Simon, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Commonsense computing: what students know before we teach (episode 1: sorting). In *ICER '06: Proceedings of the second international workshop on Computing education research*, pages 29–40, New York, NY, USA, 2006. ACM.

[76] Elliot Soloway. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29:850–858, 1986.

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

[78] LA Sudol-DeLyser; S. Carver;M. Stehlik. Learning looping: The relationship between implicit language and code. *In Proceedings of American Education Research Association Annual Conference*, 2013.

[79] Brian Stoler. A framework for building pedagogic java programming environments. Master's thesis, Rice University, 2002.

[80] Leigh Ann Sudol. Teaching students to loop, the effect of worked examples. Edbag Seminar, 2010.

[81] L. A. Sudol-DeLyser, K. Rivers, and T. Harris. Calculating probabilistic distance to solution in a complex problem solving domain. *5th International Conference on Educational Data Mining*, pages 144–147, 2012.

[82] L.A. Sudol-DeLyser, M. Stehlik, and S. Carver. Code comprehension problems as learning events. *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, 2012.

[83] Leigh Ann Sudol-DeLyser and Jonathan Steinhart. Factors impacting novice code comprehension in a tutor for introductory computer science. 4 2011.

[84] Codecademy Learning System. Code academy online instructional system. Webpage, December 2014.

[85] Allison Tew and Mark Guzdial. Devoping a validated assessment of fundamental cs1 concepts. *Proceedings of the 41st ACM Technical Symposium on Computer Science*

Education, 2010.

[86] John Truscott. The case against grammar correction in l2 writing classes. *Language learning*, 46(2):327–369, 1996.

[87] Salvatore Valenti, Francesca Neri, and Alessandro Cucchiarelli. An overview of current research on automated essay grading. *Journal of Information Technology Education: Research*, 2(1):319–330, January 2003.

[88] Anne Venables, Grace Tan, and Raymond Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *ICER '09: Proceedings of the fifth international workshop on Computing education research workshop*, pages 117–128, New York, NY, USA, 2009. ACM.

[89] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *Proceedings of the 8th Austalian conference on Computing education - Volume 52*, ACE '06, pages 243–252, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[90] Jeannette Wing. Computational thinking. *J. Comput. Sci. Coll.*, 24(6):6–7, June 2009.

[91] Jeannette M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.

[92] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4):240–253, 2006.