In presenting this thesis in partial fulfillment of the requirements for an advanced degree at Idaho State University, I agree that the Library shall make it freely available for inspection. I further state that permission for extensive copying of my thesis for scholarly purposes may be granted by the Dean of the Graduate School, Dean of my academic division, or by the University Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Signature _____

Date _____

Extending KeystrokeExplorer for Visualization of Diverse Student Coding Strategies via Temporal Abstract Syntax Trees

by

Delaney Moore

A thesis
submitted in partial fulfillment
of the requirements for the degree of
Master of Science in the Department of Computer Science
Idaho State University

To the Graduate Faculty:

The members of the committee appointed to examine the thesis of Delaney Moore find it satisfactory and recommend that it be accepted.

Paul Bodily,
Major Advisor

Farjana Eishita,
Committee Member

Tracy Payne,
Graduate Faculty Representative

Table of Contents

Extending KeystrokeExplorer for Visualization of Diverse Student Coding Strategies via Temporal Abstract Syntax Trees

Thesis Abstract – Idaho State University (2023)

For many beginning CS students, being able to abstractly conceptualize a programming solution and implement that solution in an organized and effective manner can be difficult. We present a method to measure and compare student coding processes in introductory courses. We describe the *temporal abstract syntax tree* (TAST), a model that enables visualization of the student coding process and from which can be derived metrics for characterizing a student's programming strategy. We define a TAST and provide the method of deriving a TAST from keystroke data and an *abstract syntax tree* (AST). We define visualizations and metrics derived from the TAST model that summarize and highlight a student's programming process. We illustrate the application of these visualizations and metrics on a dataset of real-world student code submissions and two different controlled validation datasets in order to demonstrate the range of applications and insights that can be gathered from our methods. Our visualizations and metrics have been implemented into a web tool called KeystrokeExplorer [21] and give the user the ability to investigate a student's abstract thinking process. Our added extension, labeled as the *TAST extension* (TASTE), allows for the exploration of basic and custom metrics to give insight into the coding strategy of each student. It also allows for comparative analysis across a group of students, assisting in the identification of unique and varied coding strategies exhibited among a set of student programming submissions for a particular assignment. We demonstrate and discuss different ways in which this tool can be applied to student data, and we suggest ways in which this work might be expanded upon in order to further refine efforts to identify, characterize, and study diverse coding practices.

Keywords: abstract syntax trees, abstract computational thinking, student coding strategy, assessment tools, visualization

Chapter 1

Introduction

In recent years, computer science (CS) pedagogical research has identified a need for a deeper understanding of different coding practices and principles due to a lack of success overall specifically in beginning programming courses [5]. It has been noted that student performance in beginning programming courses commonly results in bimodal grade distributions [3]. This result has naturally contributed to a common binary classification of students as either "those who get it" or "those who don't". Despite the temptation to group "those who get it" into a single common class, it is apparent that even among successful beginning programmers there is exhibited a variety of successful programming strategies, and CS education researchers have noted the need for more careful and nuanced characterization of these varied strategies. Educators in the field of computer science commonly attribute skills such as abstract and computational thinking as being essential for any computer scientist, and it has also been shown that poor program planning and poor problem-solving are among the main reasons for failure in introductory computer science classes [1, 18]. Students who demonstrate strong computational and abstract thinking skills have shown greater success in introductory CS courses compared to students who have weaker computational and abstract thinking skills [18].

In CS education, a common way to measure student success is by assessing a final product submitted by a student whether it be a project, an assignment, or an assessment [17]. Only in a handful of cases do we start to assess the *process* that a student demonstrated over the period of time it took to formulate the final product. Piech et al.[17] state that students in introductory courses can still find successful solutions to assignments even though they are not fully comprehending necessary conceptually abstract concepts that they will need for future success in the field (e.g., by trial and error, excessive reliance on teaching assistants or other students, etc.). Introductory courses are key building blocks in a student's education

and being able to identify and address these deficiencies in a student's programming ability early is important. Assessing how a solution was developed is a meaningful task that can provide more detailed feedback for students. However, manually sorting through student data to analyze the student thought process is a daunting and time-consuming task for instructors. There is a need for the development of methods that allow the analysis of the student coding process in a way that is automated and is easy to gather information at a glance.

A common way that the literature groups different types of programmers are into *tinkerers* or *planners* [5]. There exists a tendency to assume that one group, the planners, are the superior programmers. This binary way of grouping programming strategies leaves little room to characterize the intricacies of coding a solution. This way of grouping also leads us to assume that there are only two types of programmers: those who get it and those who don't. Blikstein et al.[5] reveal through their research that "... success in computer programming is a tale of many possible pathways." Their research has highlighted that an effective way to determine the factors that lead to student success in introductory courses is by categorizing student coding patterns and strategies. There is a need for new ways to reference a greater number of programming strategies and ways to measure these strategies. Identifying and measuring these coding processes can reveal the strengths and weaknesses of novice programmers and can lead to student success.

In order to be able to investigate the student coding process in an efficient way, we have designed and implemented an extension, which we have named the *TAST extension* (TASTE), in a web-based tool called KeystrokeExplorer [21]. In addition to allowing visualization of the correlations between basic measurable attributes of programmatic solutions (e.g., grades, number of comments, and number of keystrokes), this extension also facilitates visualization of *custom* metrics (i.e., metrics that a user defines that can be derived from keystroke and other code data in an attempt to represent the measurement of a subjective, abstract, qualitative aspect of the coding process). As an example of such a custom measure, we attempt to define metrics representing *attentiveness* (i.e., the act of programming a programmatic element to

completion) and *foresight* (i.e., the ability to see what key programmatic elements the final submission will need to contain in order to solve the problem at hand). These custom metrics rely on measurements of the code represented via a novel data structure—a *temporal abstract syntax tree* (TAST)—designed to represent the temporal evolution of a student's abstract thinking process. From TAST we derive a number of custom metrics that aim to summarize aspects of the student's abstract thinking process (e.g., student foresight, attentiveness, etc.) which can subsequently be examined for correlation with other basic or custom metrics. TASTE has potential uses in a pedagogical setting, which includes student and instructor use to understand student coding processes in real time to improve student understanding, and in a pedagogical research setting, which includes the use of the tool to develop new metrics that can measure programming skills or strategies in a single value. The ability to compare different student coding metrics alongside a code playback tool can give professors, students, and researchers the capability to identify important information about the ability of novice student programmers and insight into different coding strategies. The goal of this paper is to present TASTE and our visualization methods along with possible uses for interpreting and applying our extension.

In Chapter 2 (published in the *Proceedings of the 2022 Intermountain Engineering, Technology, and Computing (i-ETC) Conference* [15]), we prove the existence of TAST for any compilable project along with the method for deriving a TAST from a Code Process Chart (CPC). In Chapter 3, we explain the development of the TASTE tool and its incorporation into KeystrokeExplorer. We revisit the TAST model and define a method for deriving a TAST from keystroke data. Additionally, we define other visualizations and custom metrics derived from TAST. A set of examples is included to explore the different visualizations generated from TASTE to illustrate the possible applications of our extension. We conclude by offering our ideas for future work and reflections on the work presented.

Chapter 2

Temporal Abstract Syntax Trees for Understanding Student Coding Thought Process

Paper submitted and accepted to Intermountain Engineering, Technology, and Computing (i-ETC) Conference 2022

Abstract

In computer science (CS) academia it can be a challenge to help beginning students develop the thought process to be successful software engineers. Although code is implemented in a linear manner, the mental construction and problem-solving process is commonly nonlinear, requiring a high-level vision of class structures, control flow, etc., before any code is physically written. This concept can be difficult for beginning CS students to comprehend and use in their own coding projects. We provide a visualization that aims to help students more easily understand the coding thought process. This is accomplished by collecting keystroke data and incorporating it into an *abstract syntax tree* (AST) which creates a *temporal abstract syntax tree* (TAST). We provide the necessary information to prove that this visualization exists for any student's project where keystroke data is collected. We also refer to another type of keystroke visualization known as a Code Process Chart (CPC) that provided inspiration for TAST. The goal is to eventually use these TASTs alongside their corresponding CPCs to help students understand and improve their own coding thought processes.

2.1 Introduction

It is commonly observed in computer science (CS) academia that grades in introductory CS classes suffer from a reverse bell curve, or a bimodal grade distribution [3]. Instead of most of the class passing with few students on the extreme ends of the spectrum (A and F), in bimodal distribution students either pass with flying colors or fail, with few students in the middle. Bridging this gap can be challenging. We hypothesize that one of the possible causes for this divide in introductory classes, and even between beginning programmers and experienced programmers, is the thought process a student uses when coding a project.

When a student begins coding, most project tasks are simple (e.g., print "Hello World", create a for loop, etc) and only require students to implement a few lines of code. As projects become more complex, it can be difficult for students to comprehend the whole project at once. By contrast, when given a project of similar scope, an experienced coder is often able to immediately identify a plan of action. This includes identifying classes to be implemented, control flow structures needed to solve a particular part of the problem, etc. They can see the project as a whole and not just one section at a time. This skill is typically heavily underdeveloped for a beginning programmer. Without the vision to see the project as a whole, the student will write the code line by line, with little vision of future sections of the project. We hypothesize that the sooner a student can learn to see past the next line of code, the more success the student will find. We identify the experienced coder's thought process as "abstract."

Explaining this concept to a beginning CS student can be difficult. To help alleviate the difficulty of teaching this concept, we have developed a visualization to show students these different types of thought processes. With the end goal of investigating these and other questions, we have designed and implemented a novel visualization called a *temporal abstract syntax tree* (TAST). We combine keystroke data, in the format of a Code Process Chart (CPC), with *abstract syntax trees* (ASTs) to create a new type of visualization we refer to as a TAST. An AST is a tree representation of compilable code. Each node represents an

element in the code. Bigger, more broad objects such as functions and for loops will appear higher in the tree while variables and constants will appear closer to the end of the tree. The nodes of the tree are connected to the other nodes associated with it. For example, the leaf nodes branching off a function node will represent different loops and variables located within that function.

The TAST model conveys information about the student coding process that can be used to teach students about the abstract thinking process. We provide an explanation of how a TAST is derived by also proving its existence.

2.2 Related Works

In recent works, ASTs have been used to assist in software evolution analysis [11, 16], to visualize project contribution [8], and to detect plagiarism [12]. ASTs provide a meaningful way to visualize code in that they highlight high-level aspects of code while abstracting away from detailed syntax. We can isolate what is really going on. For example, we can compare an AST with one version of code to another to see what nodes were added and therefore what functionality was added. This makes ASTs great for software evolution analysis and analyzing contributions to a group project. ASTs are often unique, and two pieces of code that solve the same problem do not often have the same tree. That is why ASTs can be used to detect plagiarism.

Research involving keystroke data has been applied in a multitude of ways. One major application has been to detect cheating, especially in online classes [6, 13]. Being able to track pasting into projects is an obvious way to use keystroke data to identify plagiarism, but research has even identified ways to identify student patterns through keystroke data, so if another person is coding for a student, it can be detected. Keystroke data has also been collected in hopes of being able to predict which students will pass and which students will fail their introductory coding class in order to identify which students will need help [4, 7]. This type of information can help professors give extra help to students in need before it is

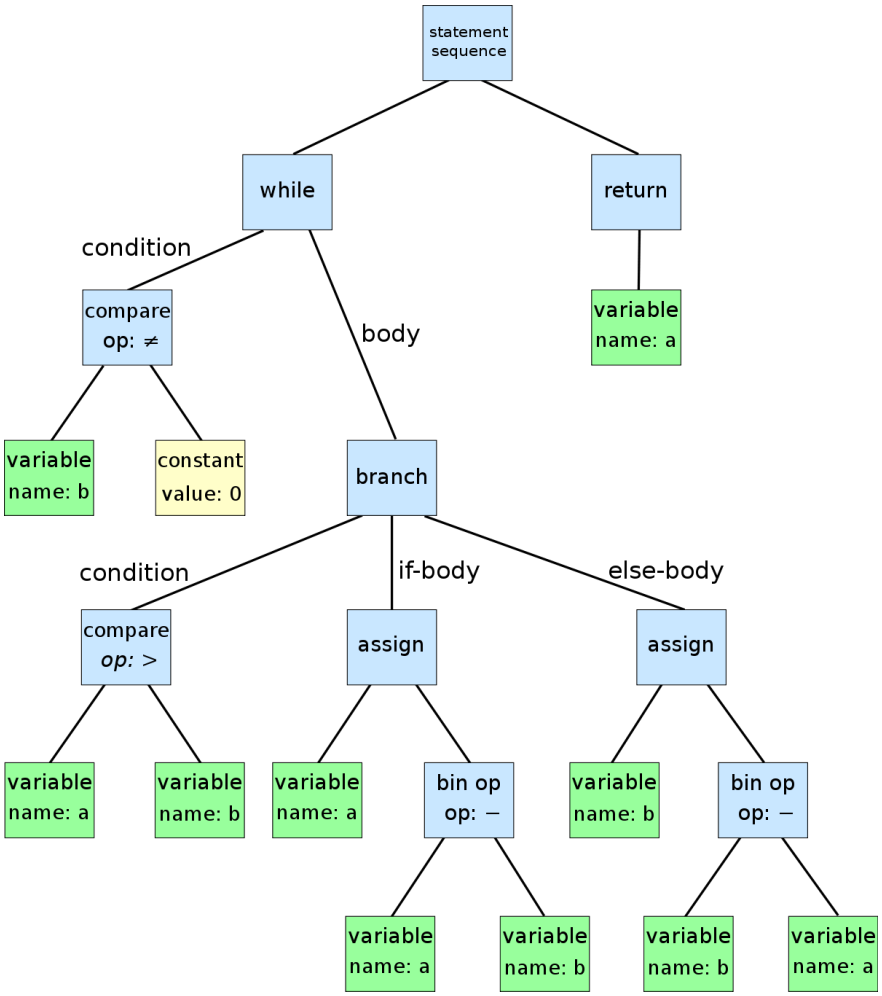


Figure 2.1: **Abstract Syntax Tree (AST) Example.** Shown is an arbitrary example of an AST. Because of the nonlinear nature of control flow structures, proficiency in designing programmatic solutions necessitates thinking of solutions in nonlinear ways. The AST provides a visual abstraction of the nonlinear structure of a programmatic solution and the nonlinear thought process behind a problem solution. Higher layers in the AST are associated with more abstract, non-linear constructs whereas lower layers in the AST are associated with more linear constructs. Image courtesy of Wikimedia Commons.

too late in the semester. Our approach is similar because of our intention to help introductory CS students. We are more interested in the order in which the keys were pressed for a given project. We hypothesize that the temporal keystroke data combined with an AST for the same project can give us insight into the thought process of a student and that the resulting TAST model can help students understand how to improve their own coding thought process.

2.3 Methods

Before going into depth, we define some key terms. A *Code Process Chart* (CPC) is a 2D graph that compares the final version of the code against a *snapshot* of the code at each keystroke [21]. The x-axis represents every character in the final code from beginning to end and the y-axis is used to denote how the current snapshot differs from the final version. An example of a CPC can be seen in Fig. 2.2. A TAST is an AST that incorporates keystroke data. Each node will be labeled based on the snapshot number in which that node was created. Each node can be altered in size or color based on this snapshot number.

The remainder of this section provides proof that there exists a TAST for a given CPC as long as the final snapshot of the CPC is a compilable piece of code.

2.3.1 Proof

We will now prove the existence of a TAST given a compilable piece of code and a CPC with proof by construction.

Assume we have a CPC where the final piece of code is compilable. The compilable piece of code, with n characters and m words, along with annotated keystroke data can be represented as two lists. The first list $C = [c_1, c_2, ..., c_n]$ is a list of each character, or letter, in the code, where c_i represents the ith character in the final code. The second list $L = [l_1, l_2, ..., l_n]$ represents a list of snapshot values associated with each character in the code where l_i is the number of the snapshot where the ith character first appears. In order to find L, we need access to the CPC. The CPC can be represented as a 2D matrix $D = [d_1, d_2, ..., d_y]$

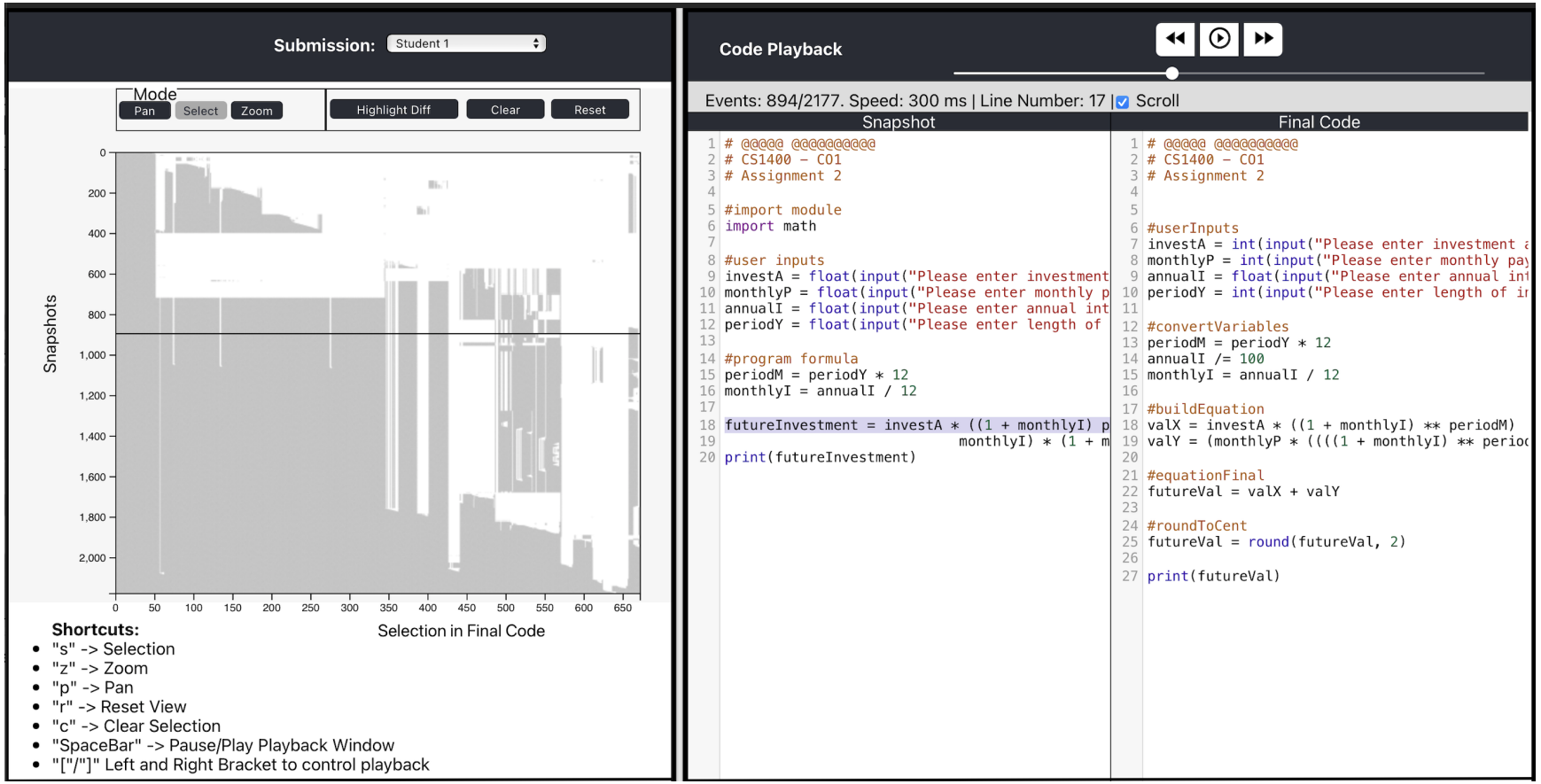


Figure 2.2: **Example of a Code Process Chart (CPC).** The right-hand side of the code playback panel represents the final code that the CPC on the left represents. The left side of the code playback panel gives the state of code that the horizontal line in the CPC represents. This line can be moved up and down to scan through the keystroke data using the slide bar at the top of the code playback panel. This allows users to break down the process of the student while also learning about the visualization. A temporal abstract syntax tree for the code in the far right panel is shown in Fig. 2.3

where d_r represents the rth row in the CPC and the CPC consists of y rows. Each row d_r is represented as a list of 0s and 1s so that $d_r = [b_1, b_2, ..., b_n]$ where b_i represents whether or not c_i exists on the rth snapshot of CPC, where 0 means c_i does not exist and 1 means c_i exists. With reference to D and C, the list L is created using Algorithm 1.

Algorithm 1 Character Snapshot

```
Initialize a snapshots list L, pass in a string list C, and a 2D matrix D
for char in range(len(C)) do
    creation_index = 0
    reverse_counter = len(D) − 1
    for row in range(len(D)) do
        current_row = D[reverse_counter]
        if current_row[char] == 0 then
            break
        else
            creation_index = reverse_counter
            reverse_counter− = 1
    L.append(creation_index)
```

By tokenizing *code* into words, we get a list $W = [w_1, w_2, ..., w_m]$ where there are m words in the final code and w_j represents the jth word in the code. We then need a new snapshot list $S = [s_1, s_2, ..., s_m]$ where s_j represents the snapshot value where the jth word was created. We use L and C to match each word in the code to its creation snapshot value. This can be done with Algorithm 2.

Once each word in the code has an associated snapshot value, the next step is to build an AST for the code. We know that such an AST exists based on our assumption that the code is compilable. Using the Python library "ast", we pass the library object the code string, and it creates an AST object of the code. We then walk through the tree, storing each node and edge in associated lists called *nodes* and *edges*. When visiting each node, we use "ast.get_code_segment(code, node)" that, given a node in the AST and the code that created the tree object, will return the code responsible for that node. This is possible because the tree object stores information about the start and end locations of the node in reference to the code. This function returns a string *seg* which is the segment of the

Algorithm 2 Character to Word

Initialize a snapshots list S, pass in a list L and a string list C
$code\_index = 0$
for *word in W* **do**
    $length = len(word)$
    $cont = True$
    while *cont* **do**
        if $C[code\_index] == word[0]$ **then**
            $S.append(labels[code_index])$
            $code\_index+ = length$
            $cont = False$
        else
            $code\_index+ = 1$
return S

Algorithm 3 Node to Snapshot

Pass in a node g_a, a string list C, a list W, a list S, and a number num_rows
$seg = ast.get\_code\_segment(C, g_a)$
$P = seg.split()$
$q = len(P)$
$start\_index = -1$
$k = 0$
$j = 0$
$cont = True$
while *cont = True* **do**
    $word = W[j]$
    $p\_k = P[k]$
    if $word == p\_k$ **then**
        if $k == 0$ **then**
            $start\_index = j$
        if $k == q - 1$ **then**
            $cont = False$
        $k+ = 1$
    else
        $start\_index = -1$
        $k = 0$
    if $j == len(W) - 1$ **then**
        $cont = False$
$v_a = S[start\_index]$
return v_a

code that relates to the node. When tokenized, we get $P = [p_1, p_2, ..., p_q]$ where seg has q words and p_k represents the kth word in seg. Next, we need to create a sequence of ordered pairs $G = ((g_1, v_1), ..., (g_a, v_a))$ where there are a nodes, g_z represents the zth node, and v_z represents the snapshot number where the zth node was created. We can connect each node with its snapshot value using Algorithm 3.

We can then use the *nodes* and *edges* list to draw an AST and can use G to label each node g_a with the associated snapshot value v_a to create a TAST. This value can also be used to change the size or color of the node. Therefore, given a CPC where the last snapshot is a compilable piece of code, there exists a TAST for that code.

2.4 Results and Discussion

Within a TAST, the lighter a node is in color, the earlier the node was created. Nodes closer to the root of the tree represent more abstract constructs of programming, like loops and class structures. Leaf nodes represent variables and variable values. We hypothesize that a tree representing a skilled and experienced programmer, or someone who understands the abstract thought process of coding, would have the most yellow nodes near the root and the most orange nodes in the leaves. On the other hand, a novice programmer or someone who codes more linearly would have the opposite effect, with yellow nodes at the leaves and orange nodes near the root, or might have a random disbursement of color.

We have provided an example of a TAST, using color to represent the time dimension, created by using the methods described in the above proof (see Fig. 2.3). At first glance, this example shows signs of a software engineer who visualizes the code at a higher level. The nodes at the top of the tree near the root are yellow and light orange, indicating that these nodes correspond with the code written in the earliest stages of the coding process for this particular project. The bottom left of the tree is a dark orange, which means these nodes were created near the end of the coding process. The bottom right of the tree is intermixed

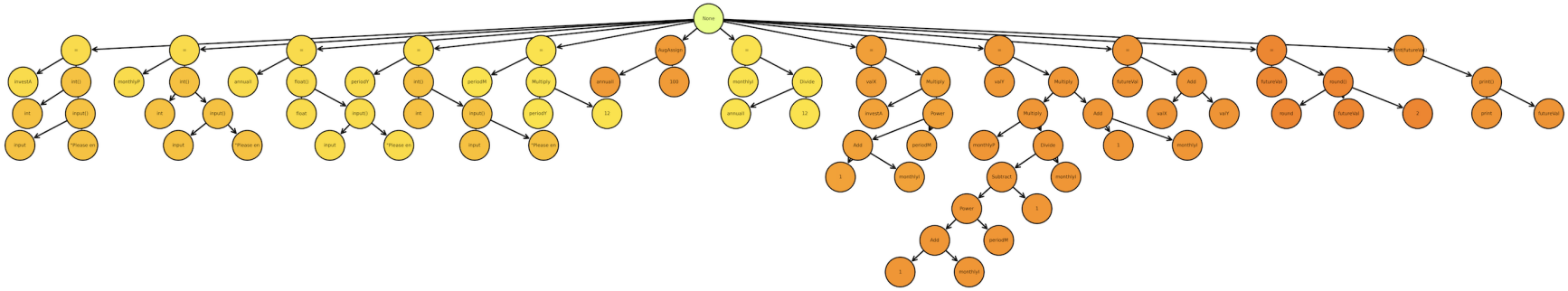


Figure 2.3: **Temporal Abstract Syntax Tree (TAST) Example.** Shown is the TAST generated from the code and keystroke data represented by the CPC in Fig. 2.2. The lighter the node, the earlier the keystrokes were for the associated code. In this TAST, we see that generally speaking, each branch is a similar color, meaning that the student coded branch by branch. (We discuss quantitative ways of measuring the distribution of color within a TAST in Chapter 3.)

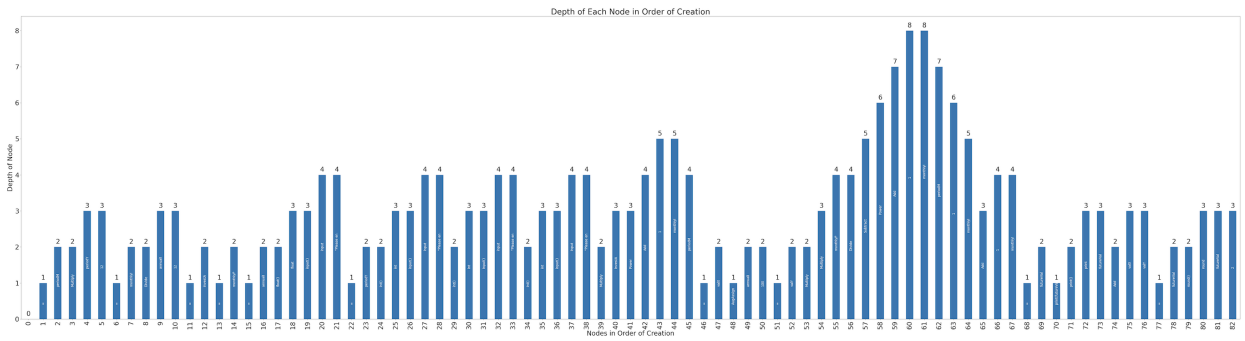


Figure 2.4: **Depth Chart Example.** Shown is an example of a tree depth chart for a TAST. For a given time step t, the chart shows the average height of all TAST nodes completed before t. Our hypothesis is that this type of visualization can be used to highlight differences in the level of abstraction at which programmers of varying levels of expertise begin thinking about and encoding solutions to programming problems.

with yellow and orange. This indicates that the engineer might have created these nodes initially and then came back and revised them at a later time.

This particular example represents a fairly linear piece of code and it lacks the nonlinear elements we previously discussed (loops, class structures, etc.). Moving forward we expect to find more examples of student code that are nonlinear in nature. In this way, we can start to test our hypotheses. We also intend on studying pieces of code created by higher-level students, in order to compare novice programmers to experienced ones.

Moreover, we anticipate improving this visualization tool by implementing the slide bar feature that the CPC tool already has implemented. This way, we can see the code and the visuals side by side, but we can also see the TAST being built in real-time. This gives us a chance to isolate nodes and see the code at the same time. An example of another type of visualization can be seen in Fig. 2.4. This visualization represents the average height of a TAST over time. For example, if the first node created had a tree depth of 7, then the first bar would have a value of 7. Then if the next node created had a tree depth of 5, then the second bar would be 6, and so on and so forth. We intend into using this visualization and possibly develop others to find what best helps students understand the abstract coding process.

Chapter 3

Measuring Coding Strategies in Novice Programmers via Extension of Temporal Abstract Syntax Trees

Paper written for submission to ACM Global Computing Education (CompEd) Conference 2023

Abstract

Abstract thinking and problem-solving skills are a necessity for enabling students to become proficient in computer science (CS) and programming. Many beginning CS students have difficulty efficiently executing the abstract process of mentally organizing and planning an effective programming solution. We have developed an extension named *TAST extension* (TASTE) that implements *temporal abstract syntax trees* (TAST)—an *abstract syntax tree* (AST) in which tree nodes are labeled with the order in which the code underlying the nodes was created—and several derived metrics in an existing web tool known as KeystrokeExplorer [21]. We define how TAST is derived from keystroke data and an AST. We define several basic and custom metrics to summarize the student coding process in order to compare one student against other students. Additionally, we present the incorporation of these methods into TASTE in order to analyze the different student coding strategies. We illustrate the application of these methods on two handcrafted validation datasets and on a dataset of real-world student code submissions, highlighting how these metrics can be examined for a deeper understanding of the student coding process. We discuss and demonstrate ways in which this tool can help instructors, students, and researchers to identify and examine different student coding strategies.

3.1 Introduction

Whereas grades in many academic disciplines follow a bell curve, introductory computer science (CS) courses are unique in that they tend to suffer from a bimodal grade distribution. [3]. In CS, a high number of students fail, a high number of students excel, and a low number of students receive a grade somewhere in the middle. Research has suggested various strategies to addressing the issue of bimodal grade distributions, including catering to the skill level of students predicted to be at high risk of failure [4] and identifying and instilling habits common among successful students [22]. Among the chief causes of failure in introductory CS courses are poor program planning and poor logic, reasoning, analytical thinking, and problem-solving skills [1]. Abstract thinking is a skill that educators commonly link to CS, and it has been shown that students with stronger abstract thinking skills have greater success in CS courses than those with weak abstract thinking skills [18]. Moreover, novice programmers build and retain these key concepts and programming skills early on in their degree because they are prerequisites to being able to master other CS concepts [9]. Students' ability to think abstractly about programmatic ways of solving problems even before touching a keyboard is one of the foremost fundamental skills taught in any introductory programming course and a skill that remains relevant throughout students' journey as software developers.

We present a new method for investigating the student coding process with the *TAST extension* (TASTE). We use *temporal abstract syntax trees* (TASTs) (see Figure 3.1), along with other derived visualizations and keystroke data, to calculate and gather measurements for a pair of custom metrics—foresight and attentiveness—which we define mathematically as a function of more basic metrics. The TASTE allows for these metrics to be visualized for student assignments. The extension is implemented within an existing web tool known as KeystrokeExplorer [21]. KeystrokeExplorer allows a user to walk through a Python solution with the use of a code playback bar. The KeystrokeExplorer's code playback feature alongside TASTE creates an interactive sandbox for exploring the student coding process. TASTE uses summary statistics that facilitate meta-level comparative analyses of student coding

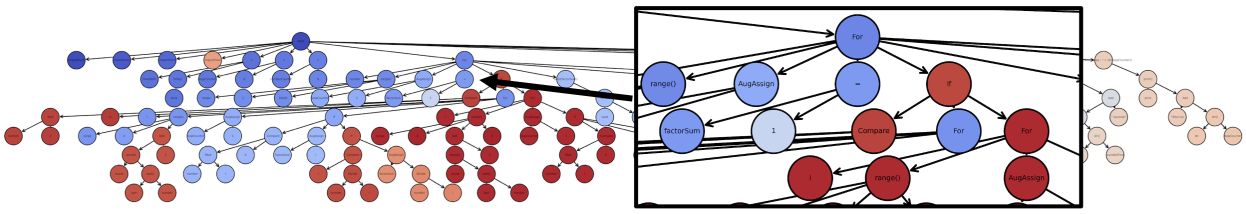


Figure 3.1: **Temporal Abstract Syntax Tree (TAST) Example.** This figure is an example of a *temporal abstract syntax tree* (TAST) for Student 1 and Assignment 6 in the real-world dataset provided by Dr. John Edwards and his research team at Utah State University [21]. The darker blue vertices represent earlier creation values and the darker red vertices represent later creation values. The corresponding depth chart can be seen in Figure 3.5 and the corresponding *Code Process Chart* (CPC) is pictured in Figure 3.3.

processes across students for a given assignment. The code playback feature allows for the in-depth investigation of a particular student assignment.

Our motivation in developing TASTE has been to address the following long-term research questions:

1. How can the student coding process be visualized to identify and understand different problem-solving strategies?
2. What metrics can be used to quantify and characterize different problem-solving strategies in the student coding process?
3. How can visualizations of and metrics describing different problem-solving strategies be leveraged by instructors and students to improve students' abilities to recognize and implement effective problem-solving strategies in the student coding process?

The goal of our research is to provide a precise description of the TASTE tool and our developed visualizations and metrics along with examples of how to use this model to analyze student coding strategies. Preliminary to defining and demonstrating the application of basic and custom metrics via TASTE, we define the method of generating a TAST model directly from keystroke data (rather than from CPC data) and revisit the purpose of TAST as a model for representing the temporal evolution of programmatic solutions.

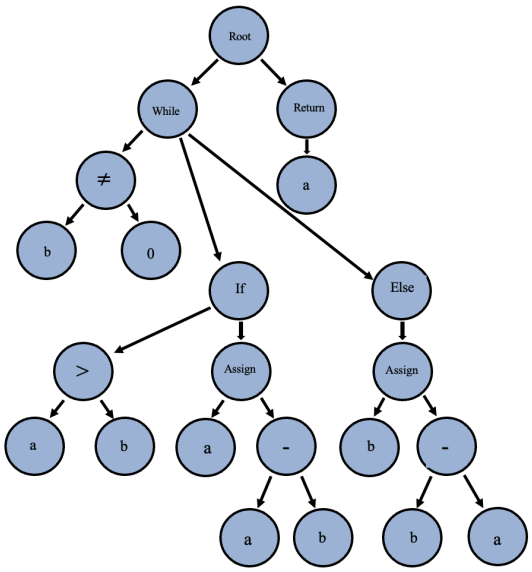
3.2 Related Works

Numerous strategies have been proposed to help address the gap in methods for characterizing abstract problem-solving skills. Lin et al. [14] demonstrate that the implementation of live coding in introductory coding classes helps improve the abstract thinking abilities of students as applied to programming. Students were able to improve their own computational thinking skills (as measured by comparing pre and posttest grades) by watching instructors demonstrate the process in prerecorded videos. However, their approach does not consider the process a student uses to create a solution and only considers their improvement between test grades and represents a return to treating such skills along a linear scale, rather than allowing for and characterizing a variety of successful abstract thinking strategies [5]. Piech et al. [17] argue that the final product of a programming project is not alone sufficient to assess student comprehension in an introductory course, as the process of how a student creates a final programming solution can reveal deeper aspects of a student's understanding. They develop a tool that automatically records coding snapshots as a student codes and generates machine learning (ML) models in order to record feedback representative of the order in which the student chooses to implement programmatic features and syntactical elements over the course of completing the assignment. Their models that used data about a student's process while coding an assignment were more accurately able to predict the performance of students on the midterm than the students' grades for the assignments.

A recent study notes that, although there exists broad agreement that abstract thinking is one of the most important competencies in computer science, there are as yet few sources that provide practical definitions for this competency [24]. This same study also describes a theoretical model of abstraction and abstract thinking in order to enable instructors to codify abstract thinking levels in their students. Although their model does provide some practical applicability, it does rely on subjective qualitative analyses, leaving the bulk of the analysis to manual instructor discretion and preventing the implementation of an automated analytical process.

```
while b != 0:
    if a > b:
        a = a - b
    else:
        b = b - a
return a
```

(a)



(b)

Figure 3.2: **Abstract Syntax Tree (AST) Example.** Figure (a) is a simple segment of code that contains two variables a and b, a while loop, an if statement, and a return statement. The AST that represents the code in Figure (a) is pictured in Figure (b). Although this example is simple, it should be noted that more complex programming structures such as a while loop and an if statement inhabit vertices closest to the root whereas variable assignments and operations occupy vertices closer to the leaves.

One common approach when analyzing the abstract structure of a programmatic solution is an examination of *abstract syntax trees* (ASTs). An AST is a hierarchical syntactical parsing of a program into its fundamental elements. An example of an AST can be seen in Figure 3.2. In an AST, the most basic, concrete operations of the program are represented at the leaf nodes whereas abstract control flow structures appear higher up in the tree. Hovemeyer et al. [10] propose building ASTs that focus only on control flow structures in order to evaluate how the understanding and use of control flow structures in CS students' assignments related to their grades. Another study explains a new tool that combines augmented reality (AR) with ASTs to help CS students learn data structures [2].

One method to help students understand abstract thinking skills is the creation of exercises that blend computational thinking concepts and creative thinking concepts with the intention of improving the retention of computational skills in introductory CS students [20].

A successful way to discover the keys to introductory student success in CS is by detecting and defining types of coding processes and strategies [5]. Just like there are many ways to solve a math problem, there are many ways to code a correct programmatic solution. There is a multitude of strategies that students apply when programming [5, 19]. Turkle and Papert[23] describe that novice programmers are usually put into one of two categories: tinkerers and planners. *Tinkerers* focus on fixing and manipulating smaller subsections of code one at a time while *planners* see the full solution and work on the program as a whole connected piece of work. Conventionally, we might expect that the planners are the superior programmers, but Turkle and Papert originally argued that neither group has superiority when considering programmatic success and both strategies are valid ways of approaching programming. They also suggest that since these groups have been formulated and studied, many other studies have considered programming strategies as binary and there is a need to consider programming strategies outside of this binary classification.

In an effort to analyze the student coding process and to identify and measure new coding methods and strategies, we developed the TASTE feature. By visualizing and analyzing keystroke data, we aim to uncover more knowledge about student coding and how to categorize and demonstrate different effective coding strategies. In the following sections, we define how TASTE utilizes the TAST model and its derived visualizations and metrics followed by an analysis of how the methods we have developed can be applied to real-world data.

3.3 Methods

In this section, we describe the development of the TASTE tool. We define the different metrics currently implemented within TASTE. To fully understand the metrics provided, we revisit the TAST model and its derived visualizations. We describe how TAST can be derived from keystroke data rather than from a Code Process Chart (CPC) (the latter having been defined in Chapter 2).

3.3.1 TAST Extension

KeystrokeExplorer[1], developed by Shrestha et al. [21], provides a web-based tool for exploring keystroke data from student code submissions. Our TASTE extension[2] adds the ability to visualize meta-level data for individual submissions and groups of submissions to provide insight into the student coding process via interactive scatterplots. An image of Keystroke-Explorer without TASTE can be seen in Figure 3.3. The TASTE extension visualizes data derived from TAST models. Currently, TAST models and derived data are pregenerated using Python scripts. However, we anticipate the models and data will be computed dynamically by TASTE in the future. JavaScript and D3, a visualization library for JavaScript, were used to integrate TASTE into KeystrokeExplorer. The D3 library provides functionality to create well-formatted interactive scatterplots within the web tool. A figure of TASTE incorporated in KeystrokeExplorer can be seen in Figure 3.4.

[1] `https://edwardsjohnmartin.github.io/KeystrokeExplorer/`

[2] source code at `https//:github.com/edwardsjohnmartin/KeystrokeExplorer/tree/delaney`

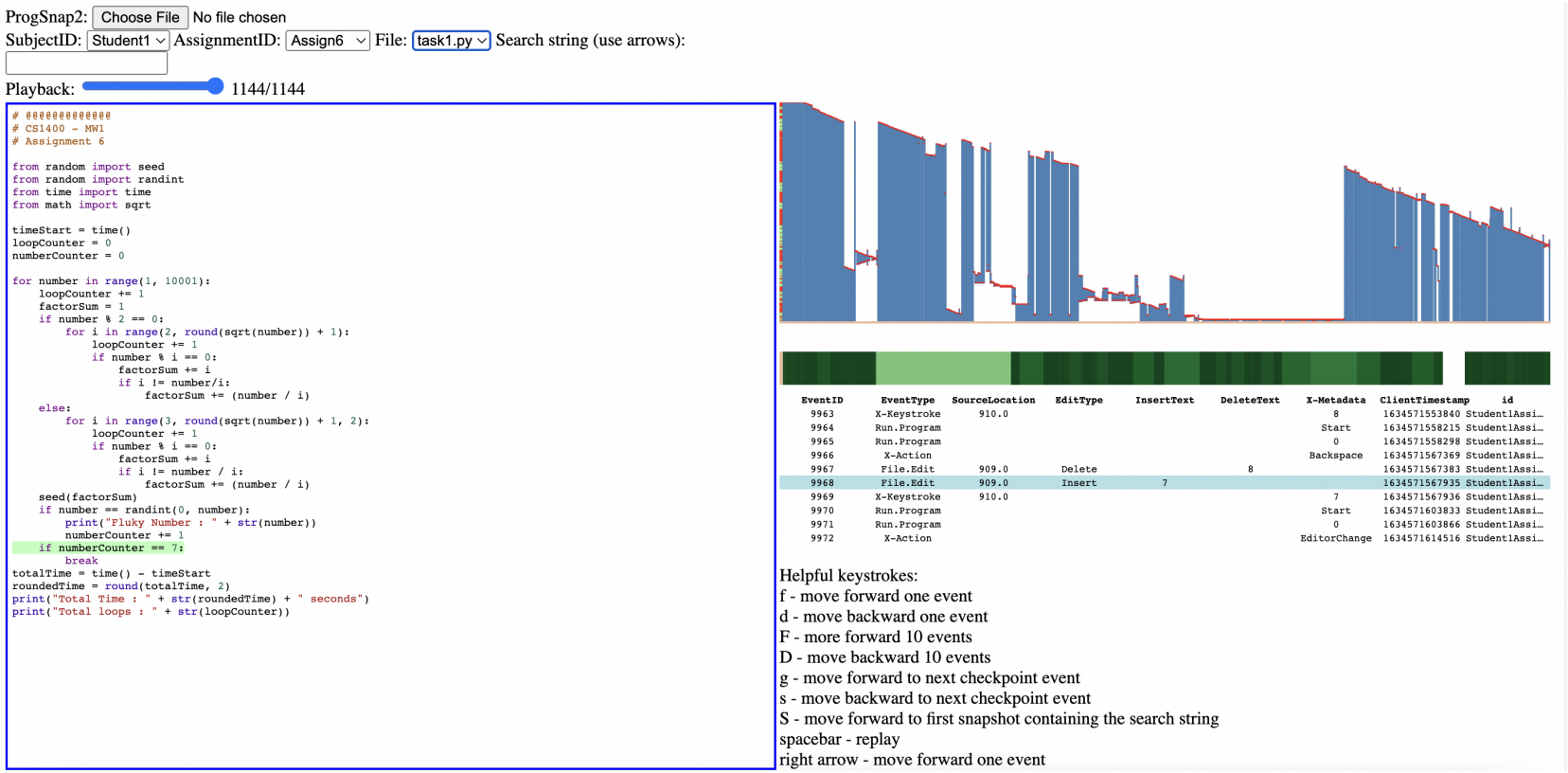


Figure 3.3: **KeystrokeExplorer Screenshot.** Pictured is an example of the web tool known as KeystrokeExplorer [21]. The tool depicts source code of the selected student assignment and visualizes the corresponding keystroke data as a Code Process Chart (CPC). This specific example corresponds with Figure 3.1. In the top left of the page there is a code playback slider that allows the user to walk through how the submission was formulated by the student.

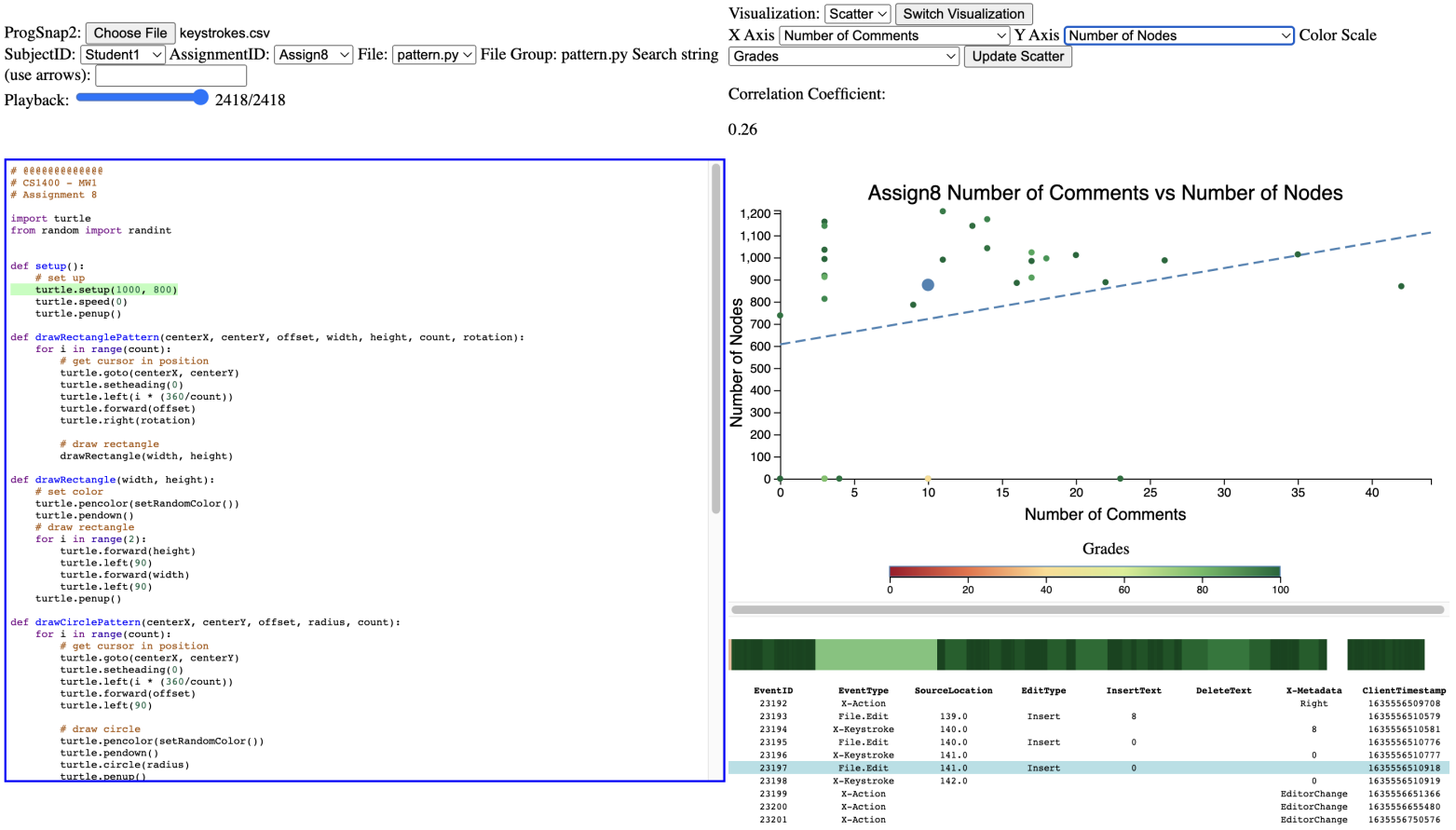


Figure 3.4: **TAST Extension (TASTE) Screenshot.** Pictured is a screenshot of the KeystrokeExplorer tool with the *TAST extension* (TASTE) incorporated. The right-hand side of the KeystrokeExplorer is where TASTE is implemented. The scatter plot currently visualizes the number of comments against the number of nodes for Assignment 8 and file "pattern.py" in the real-world dataset. The color scale represents the grade each student received, where red represents grades lower on the scale and green represents grades higher on the scale. Each point represents a student and is colored based on the grade color scale. This example highlights Student 1 Assignment 8, which is depicted in blue within the scatter plot on the right. The selected point has also been enlarged to be easier to see. This screenshot demonstrates how someone can use TASTE and the code playback feature together. TASTE also allows the user to select a point and see the associated submission visualized in the code playback window.

3.3.2 Temporal Abstract Syntax Tree From Keystroke Data

An AST is a graphical model of the abstract syntax that represents a piece of source code. These trees are most often used by the compiler to interpret source code. In general, higher-level programming features, such as control flow structures, will form the topmost vertices and simpler structures such as variables or print statements will form the bottom-most vertices or leaf vertices.

In the rest of the section, we define a TAST which is an AST that also represents the creation value for each vertex. The creation value is the keystroke timestamp that corresponds with the code segment responsible for the creation of the vertex. The creation value of each vertex is represented by color visually. This visual can identify which parts of the tree are developed first and which are developed last and give us insight into the thought process the student exercised while developing the final submission. TAST is redefined here because our new method of deriving TAST uses a keystroke dataset instead of a CPC. A CPC is a visualization derived from a keystroke dataset that compares each state of the program to the final version of the code. The ability to create a TAST from a keystroke dataset allows us to create TAST without needing to create a CPC. Before we can provide a mathematical definition for TAST, we must first provide mathematical definitions for an AST and a keystroke dataset.

3.3.2.1 Abstract Syntax Tree

Given a source program X, we define an AST $A_X = (V, E)$ where V is a set of vertices denoting constructs in X and E is a set of edges denoting the relationships between constructs in V as manifest in X.

3.3.2.2 Keystroke Dataset

Shrestha et al. [21] developed a PyCharm plugin that keeps a log of the character c_i typed at time step i. Given a source program X, we define the keystroke data for X as a sequence

$K_X = ((c_0, l_0, k_0), \ldots, (c_i, l_i, k_i), \ldots, (c_n, l_n, k_n))$ where c_i is the character typed at time step i, l_i is the location of c_i in X, k_i is the original keystroke creation value of c_i, and n is the number of total characters in X. We simplify K to only include characters appearing in the final source program.

Now that we have defined an AST and a keystroke dataset, we can define a TAST. Given an AST $A_X = (V, E)$ for a given code submission X and a keystroke dataset $K_X = ((c_0, l_0, k_0), \ldots, (c_i, l_i, k_i), \ldots, (c_n, l_n, k_n))$ which also represents a given code submission X, we can define a TAST as $T = (V', E)$ where T is a copy of A_X and each $v \in V'$ has an attribute of *creation_value*. An example of a TAST model can be seen in Figure 3.1.

The algorithm for assigning vertex creation values to an AST to create a TAST is described in Algorithm 4. It should be noted that we use the Python library *ast* in order to build A_X. We also assume that the creation value for each node is accurately described by the keystroke timestamp of the first character of the corresponding code segment for simplicity. We use the vertex attributes *lineno* and *col_offset* which provide the location for the first character in the code segment related to a specific vertex to determine the corresponding creation value.

3.3.3 Depth Chart

To extrapolate more information from TAST, we developed a bar chart that represents the tree depth of each vertex in order from the first vertex created to the last vertex created. The tree depth of a vertex is defined as the number of edges between a given vertex and the root vertex. Given a TAST $T = (V', E)$, we define a depth chart as $D = [d_0, \ldots, d_i, \ldots, d_m]$ where d_i is the depth of the vertex that was created ith. The ith bar represents the vertex $v \in V'$ whose temporal order is i. The depth of this bar is derived from the depth of v in T. An example of a depth chart can be seen in Figure 3.5. The x-axis represents the order in which nodes were implemented, with the leftmost bar representing the root of the TAST

Algorithm 4 Create TAST

Inputs: *source code* X *as a string and keystroke data* $K_X = ((c_0, l_0, k_0), \ldots, (c_i, l_i, k_i), \ldots, (c_n, l_n, k_n))$.
Outputs: *a temporal abstract syntax tree T=(V',E).*
$T_X = ast.parse(X)$ ▷ Initialize $T_X = (V, E)$ as the AST for X
$new\_lines = [0]$ ▷ This list will be responsible for holding the index values of the first character of each new line.
for *i in range(len(X))* **do**
    if $X[i] ==$ "\n" **then**
        $new\_lines.append(i + 1)$
for v *in* V **do**
    $lineno = v.lineno - 1$ ▷ The attribute *lineno* is 1-based.
    $col = v.col_offset$
    $index = new_lines[lineno] + col$
    for (c_i, l_i, k_i) *in* K_X **do**
        if $index == l_i$ **then**
            $v.creation\_value = k_i$
            break
return T_X

(which represents the submission as a whole). The height of each bar corresponds with the depth of the associated node in the TAST.

3.3.4 Metrics for Characterizing Student Coding Strategies

We define several metrics by which to measure a student's coding submission and a student's coding strategy. The metrics we define are only a portion of all possible metrics that can be used to evaluate student performance and the programming process. The TASTE tool is designed to be easily extended with additional metrics. For our research purposes, we look at several basic metrics as well as two custom metrics. We use the term *custom metrics* to refer to metrics that a user defines that attempt to represent the measurement of a subjective, abstract, qualitative aspect of the coding process. The two custom metrics that we have defined are *attentiveness*—which we define as the act of programming a programmatic element to completion)—and *foresight*, which we define as the ability to see from the beginning what key programmatic elements the final submission will need to contain in order to solve the

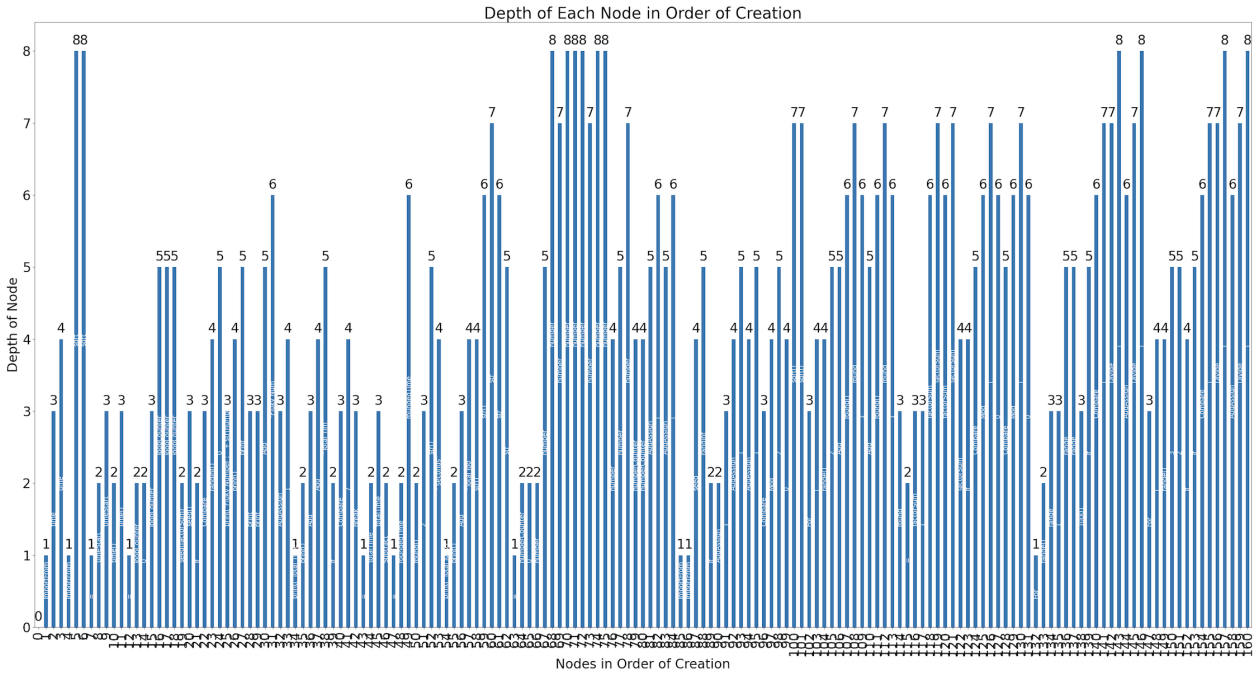


Figure 3.5: **Depth Chart for Student 1 Assignment 6.** This depth chart is for Student 1 Assignment 6, which correlates with Figure 3.1 and Figure 3.3. At a glance, we can see that there are shorter bars on the left-hand side of the graph and taller bars on the right-hand side of the graph. This suggests that nodes closer to the root node were coded first and that the leaf nodes were not coded until later in the development of the solution.

problem at hand. As an example of high attentiveness, we envision a student who starts coding a function and finishes coding the function before moving on to other parts of the code. As an example of high levels of foresight, we envision a student who writes out all their function signatures prior to implementing the bodies of those functions.

Many of these metrics attempt to directly measure a sense of *linearity* in the student's thought process. For example, once a student defines a function signature or the heading of a while loop, do they immediately code the body of that function or loop from top to bottom, or do they jump around and develop other parts of the solution first? Perhaps the student encoded the body of the loop *first* and only later came back to make it a loop. We are able to measure (to some extent) this notion of linearity in code development by looking at patterns in the height and depth charts produced for a particular TAST. For example, for a student that codes the heading of a loop and then immediately implements the body of the

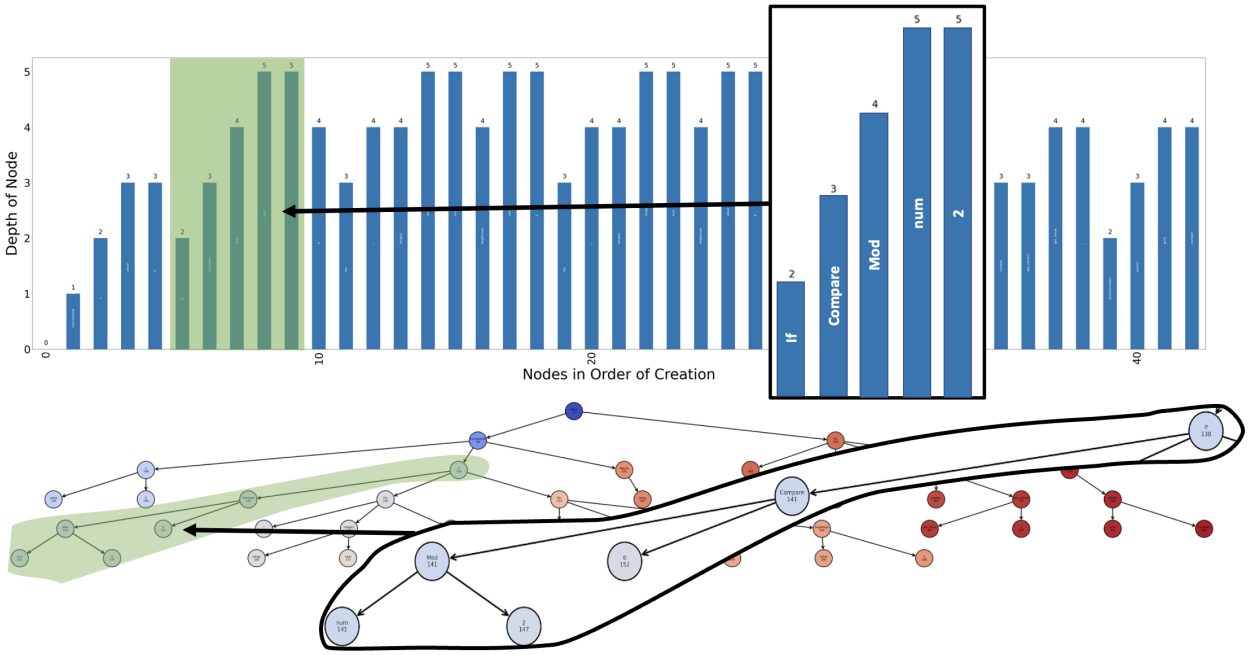


Figure 3.6: **Example of Monotonicity.** This figure features a depth chart and a TAST model for the same assignment. The section highlighted in green in the depth chart and in the TAST represents the same segment of the code. The bars for the depth chart corresponding with the segment of code are monotonically increasing, meaning that the bars are either growing in height or staying the same height from left to right. Once the segment of code was started, it was coded to completion.

loop from top to bottom, this behavior creates a sequence of monotonically increasing bars in the depth chart (see Figure 3.6). Thus, we define the *monotonicity* of a TAST depth chart as the percentage of bars (in order of increasing x value) that have a greater or equal y value than the bar immediately preceding it. With this definition, a monotonicity score of 1 would be assigned to a TAST in which function headings and control flow structures are all encoded prior to their bodies being implemented. The goal of this metric is to create a spectrum along which to measure solutions based on the order in which instructions are implemented rather than on the final resulting artifact. Monotonicity largely serves to determine whether a student generally implements code associated with high-level nodes in the AST *before* code associated with the lower-level nodes (all else held equal).

An *AST behavior graph* is a visualization that looks at the student coding process over time and was developed by Shrestha et al.[21]. The x-axis represents the cumulative number of keystrokes and the y-axis represents the number of nodes in the AST for the code state after a given number of keystrokes. A point only exists if the code state is compilable. An example of an AST behavior graph can be seen in Figure 3.7. We define the monotonicity for an AST behavior graph in the same way as it is defined for a TAST depth chart (ignoring uncompilable states). In the context of an AST behavior graph, the monotonicity metric serves to measure how often a student is deleting code that results in the removal of AST nodes. This again helps to create a spectrum along which to measure solutions based on coding *behavior* rather than on the final resulting artifact.

The following list itemizes the metrics that are currently implemented within the TASTE tool:

- *grade* - The grade a student received for the specified assignment
- *height* - Represents the TAST tree height for the given student assignment. Tree height is defined as the number of vertices in the longest branch of a tree graph.
- *number of nodes* - The total number of nodes within a TAST for a given student assignment
- *number of keystrokes* - The total number of keystrokes included in the keystroke dataset for a given student and assignment
- *number of comments* - The total number of lines that include comments within the final submitted version of the student assignment
- *percentage of compilable states* - The percentage of keystrokes for which the code state is compilable (i.e., the number of states the program is compilable divided by the total number of keystrokes)
- *monotonicity of AST behavior* - The monotonicity of the AST behavior graph of a given student assignment

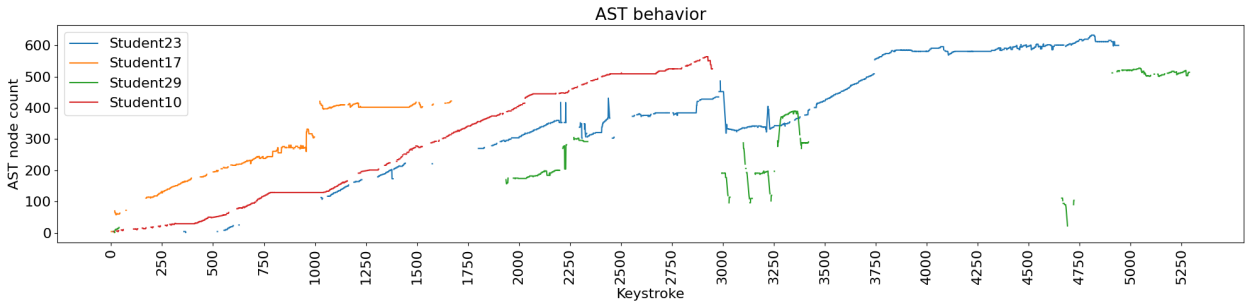


Figure 3.7: **AST Behavior Graph Example.** This AST behavior graph compares the number of keystrokes to the number of nodes in the AST of every compilable state. The AST behaviors are for students 10, 17, 23, and 29 from Assignment 8 and file "pattern.py". The goal of the AST behavior graph is to visualize the compilable states over the entire development of a student's assignment. The ability to compare the compilable states to the number of nodes in a given state allows insight into the number of deletions that occur over the development of the solution. We use this visualization to calculate two different metrics: the percentage of compilable states and the monotonicity of AST behavior.

- *attentiveness* - The monotonicity score of an assignment's depth chart. The goal of this metric is to measure if a student codes a branch of the TAST to completion or codes in a nonlinear fashion.
- *foresight* - The weight of the right half of the depth chart divided by the weight of the left half of the depth chart of a given student assignment. The goal of this metric is to measure the amount of planning and foresight the student demonstrates when coding a solution.

3.4 Results

In this section, we demonstrate results and insights gained from applying the TASTE extension to different datasets—a dataset of real-world student coding solutions and two handcrafted validation sets. These results demonstrate how the TASTE tool can be used a) to help instructors see at a glance the various approaches that students take to solving particular assignments and b) as a tool for teaching students about diverse aspects of programming

strategies and correlations between strategic behaviors. We discuss the results from basic metrics and the results from our custom metrics used within TASTE.

3.4.1 Basic Metric Results

Our visualizations were created from a keystroke dataset that we will refer to as the real world dataset, designed and implemented by Dr. John Edwards at Utah State University using the PyCharm plugin called ShowYourWork [21]. The ShowYourWork plugin records and formats the keystroke data of a project as a student codes. The plugin creates a CSV of the compiled keystroke data within the project environment. The real world dataset was collected in an introductory CS course in the Fall of 2021. The dataset includes eight assignments and forty-five students and contains each keystroke for each student assignment.

3.4.1.1 Regular Commenting

A common practice that instructors of beginning programming courses teach their students is to comment often. Regular commenting suggests that a student understands how the code functions or has planned for specific programmatic functionality to be implemented. Regular commenting helps with maintainability and readability by creating an outline for the program and providing a log of the logic behind different programmatic implementations for future reference. Regular commenting can be an abstract concept for novice programmers. The ability to understand the benefits of regular commenting and how to apply commenting in practice requires the ability to conceptualize the programmatic solution as a whole rather than just syntactically.

An instructor is able to visually see the positive correlation between number of comments and grade for the example seen in Figure 3.8 due to the functionality provided by TASTE. Using TASTE allows an instructor to show a graph such as this to start a dialogue with students about the effect of regular commenting for this assignment. Within the extension, TASTE allows an instructor to then select an assignment from the scatterplot

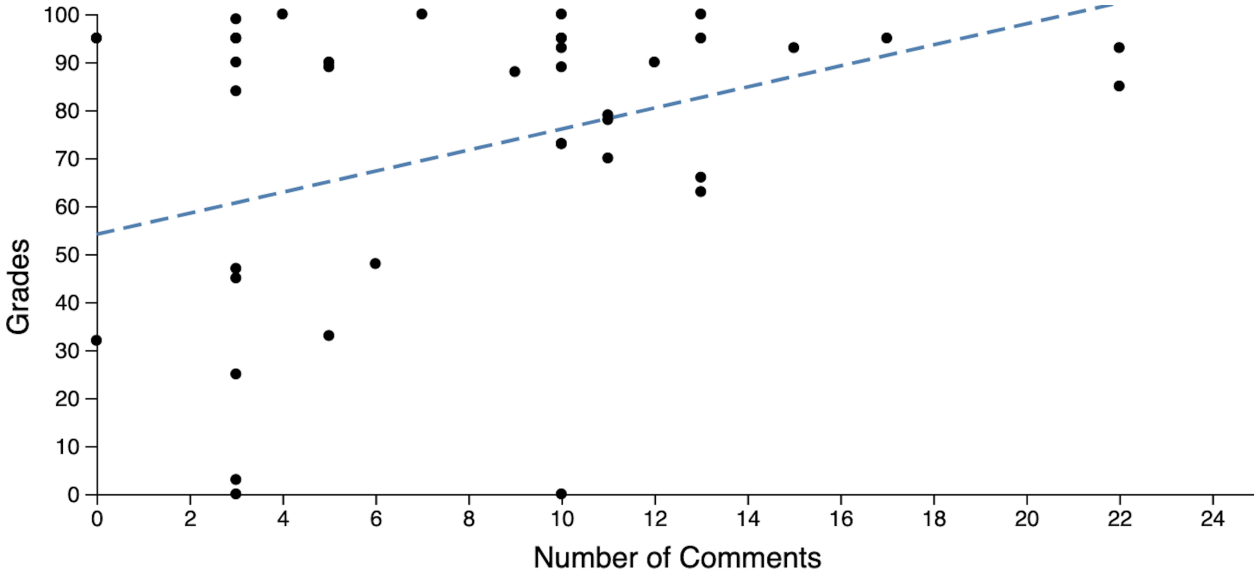


Figure 3.8: **Number of Comments vs Grades for Assignment 6.** This scatterplot is a screenshot from the TASTE tool and graphs number of comments against grades for real-world student implementations (correlation coefficient $r = 0.35$). The file name for this figure is "task1.py". This example illustrates how an instructor might use TASTE to emphasize the importance and effectiveness of commenting source code. An instructor trying to demonstrate the importance of commenting to their students can use an example such as this one to help demonstrate their point.

that has a high grade and number of comments. The assignment is then displayed in the left-hand portion of KeystrokeExplorer and allows the instructor to discuss the benefits of each comment included within the assignment. The tool facilitates the ability to playback the assignment to discuss the process of writing the comments during the development of the code.

3.4.1.2 Keeping a Working Program

Another common practice taught in introductory CS classes is to keep a working program. This practice teaches students to compile their code often to ensure that their current code is working as expected. Keeping a working program helps with catching bugs in the program early. This is possible because a student can isolate the parts of the code that are responsible for a new bug. For example, if a student writes 100 lines of code without ever verifying that

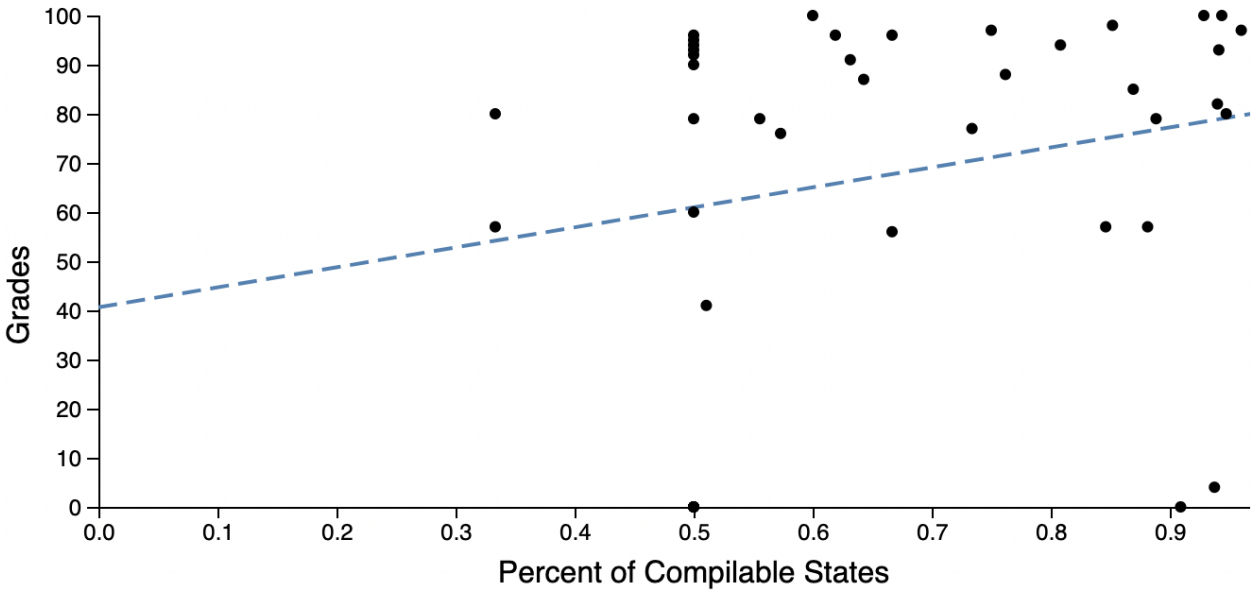


Figure 3.9: **Percent of Compilable States vs Grades for Assignment 10.** This scatterplot was visualized using the TASTE tool and graphs the percent of compilable states against the grades of real-world student implementations (correlation coefficient $r = 0.22$). This graph is specific for the file name "task1.py". An instructor might use a figure such as this to help demonstrate that keeping a working program is an effective strategy.

the program is working and receives an error when compiling after writing the 100 lines, they must investigate all 100 lines of code to find the bug. Consider alternatively that if a student writes 100 lines of code but checks that the program is working every time 5 lines are written, the student is more easily able to locate the bug when it arises.

An instructor can use the TASTE feature to identify a graph such as Figure 3.9 that visualizes the effectiveness of keeping a working program and can choose to share this result with students to help demonstrate the concept. The scatterplot within TASTE allows an instructor to select student submissions where many of the states were compilable and the student received a high grade. The code playback feature enables the ability to walk through the code which can be utilized to discuss the decisions made throughout the coding process that allowed the student to be organized and also kept the code in a working state.

3.4.1.3 Understanding the Problem

The monotonicity of AST behavior gives us insight into the growth of a program over time. If the monotonicity of AST behavior is high, the AST of the student's code was constantly growing without many deletions that caused the tree to decrease in size. We might attribute a student who deletes significant sections of code that affects their AST size more often to being less confident, and that a more confident student does not delete large portions of code often. A technique some instructors teach students is to understand the problem before starting to code. Prior understanding can help improve a student's ability to understand what is necessary in order to implement a correct programmatic solution and will also give a student confidence in their ability to implement the solution correctly.

The TASTE tool allows for the creation of graphs such as Figure 3.10 that visualize the effectiveness of understanding the problem before starting to code. This might be a situation where an instructor wants to identify ways of helping students who are not performing as well as others. An instructor can select the students with the lowest confidence according to the monotonicity of AST behavior in the scatterplot via TASTE and walk through the code using the code playback feature in KeystrokeExplorer to see where the student is deleting large sections of code to identify what concepts are keeping the student from making progress. When addressing a struggling student, an instructor can utilize TASTE to select a student submission with high confidence to walk through with the struggling student.

3.4.2 Foresight and Attentiveness

In an effort to both verify the accuracy of the implementation and to understand better the results when applied to real data, we created a validation dataset, which we refer to as the calculator validation dataset. The calculator validation dataset consists of handcrafted programmatic solutions designed specifically to explore the efficacy of our custom metrics in recognizing different problem-solving styles. In this controlled environment, we have prior knowledge and insight into the student coding strategy being applied because we coded the

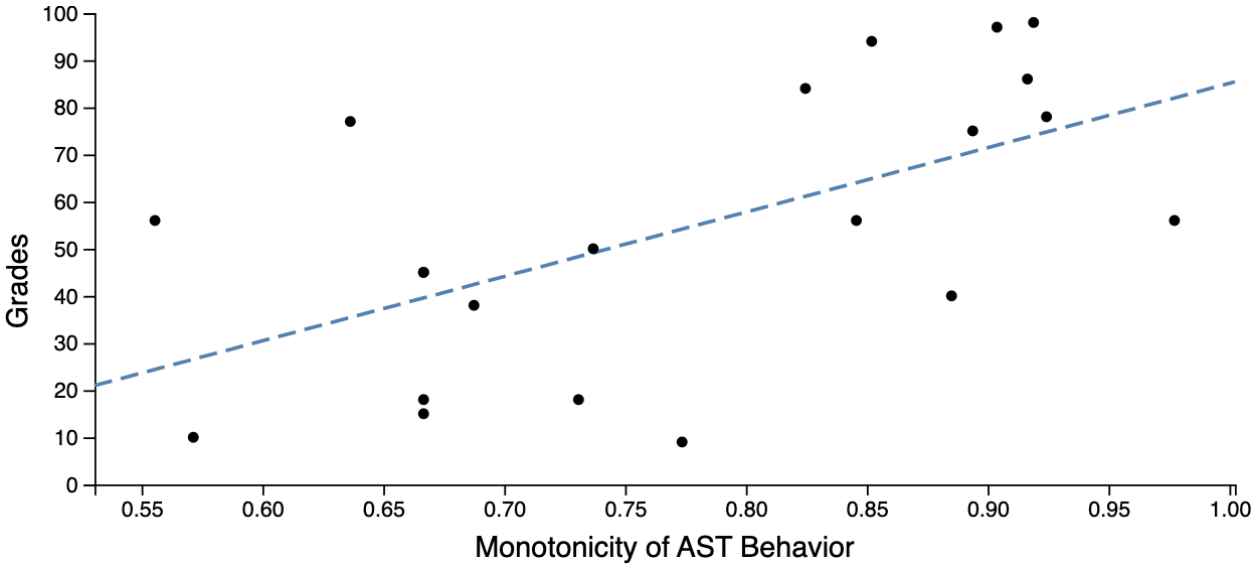


Figure 3.10: **Monotonicity of AST Behavior vs Grades for Assignment 13.** This scatterplot, generated with the TASTE tool and specific to the file name "task1.py", graphs the monotonicity of AST behavior against the grade of real world student implementations (correlation coefficient $r = 0.59$). The monotonicity of AST behavior can be used to understand the confidence, or the number of large deletions, that occurred over the development of the assignment solution for a specific student. This graph demonstrates that those with fewer major deletions received higher grades on this assignment. An instructor can use this graph to demonstrate that understanding the problem before starting to code, and therefore not needing to delete major sections of code often, is an effective strategy.

student examples ourselves. We developed an assignment that required high-level control flow structures such as loops, if statements, and functions; user input; and error checking. The assignment we implemented was a Python calculator assignment that asked a user for an operation, took in a number of variables to be applied to the operation and the actual values of the variables, and then printed the solution to the user. The program continues until the user tells the program to quit. The specification[3] we used as a reference for the assignment was an existing specification from an Idaho State University beginning programming course. The original specification calls for an implementation using the C programming language. The validation set uses Python solutions instead because the plugin for collecting keystroke data is currently only implemented for a Python development environment.

[3] `https://www2.cose.isu.edu/~bodipaul/courses/sp23/1337/labs/lab_3/c_bit_calculator.pdf`

In order to focus solely on the strategy used to produce the final product rather than on the final product itself, we created four programmatic solutions that were implemented using four different programming strategies. We created a working solution, with no intended solving strategy in mind, to use as a base solution that we refer to as the *original solution*. We created four additional solutions that had the same end product as our original solution but were implemented with a different keystroke ordering designed to be reflective of different problem-solving strategies. In particular, we defined the four unique problem-solving strategies with the aim of providing examples rather than being comprehensive. The goal here is to achieve more than merely a binary classification so as to demonstrate how the TASTE tool and metrics defined within the extension are able to characterize and identify a variety of programming and problem-solving styles. For each style, we list the prescriptive criteria that were used to define the style:

- *Specification Order* (SO)
  - Code the solution following the order of the specification given for the assignment
- *Control Flow First* (CF)
  - Call functions before they exist
  - Create functions once they are called (but after the task at hand has been completed where the call was made)
  - Leave error checking for the end
- *Arithmetic First* (AF)
  - Use static inputs first and once the correct functionality of the code has been verified then implement user input
  - Error check as we go
  - Put everything into functions last

- *Random*
  - Use a random number generator to randomly pick between the first coding methods
  - Code using those guidelines for three to five lines before randomly switching methods again

Figure 3.11 shows a scatterplot generated in the TASTE tool comparing two of our custom metrics, foresight and attentiveness, for the calculator validation dataset. Our original hypothesis was that the CF point would have the highest foresight value since there is a large amount of planning involved within the CF guidelines and that the AF point would have the lowest foresight value since control flow structures and function signatures are coded last according to the guidelines. We initially hypothesized that the CF point would have high attentiveness and the AF point would have low attentiveness. This is because the guidelines for the CF strategy require functions to be completed once their function signatures have been created and for the task at hand to be completed before focusing on a new one. This would suggest that there is a small amount of non-linear code traversal. The AF guidelines do not require the completion of given task before moving onto another task, only that correct functionality has to be verified before higher-level control flow structures can be implemented. This allows for more freedom to move from section to section of the program in an non-linear fashion.

The CF point for the calculator assignment has the highest foresight value and the second highest attentiveness value and the AF point has the lowest foresight and attentiveness value, which matches our initial hypothesis. The TAST models for the CF and the AF points, seen in Figures 3.12 and 3.13 respectively, also visualize these differences. These models visualize the order in which syntactical elements were implemented. The TAST for the CF strategy has longer branches of similar color since tasks that were started we immediately completed and the TAST for the AF strategy has less consistent colors in the branches due to non-linear traversal from task to task. The colors of the models also reveal that the nodes

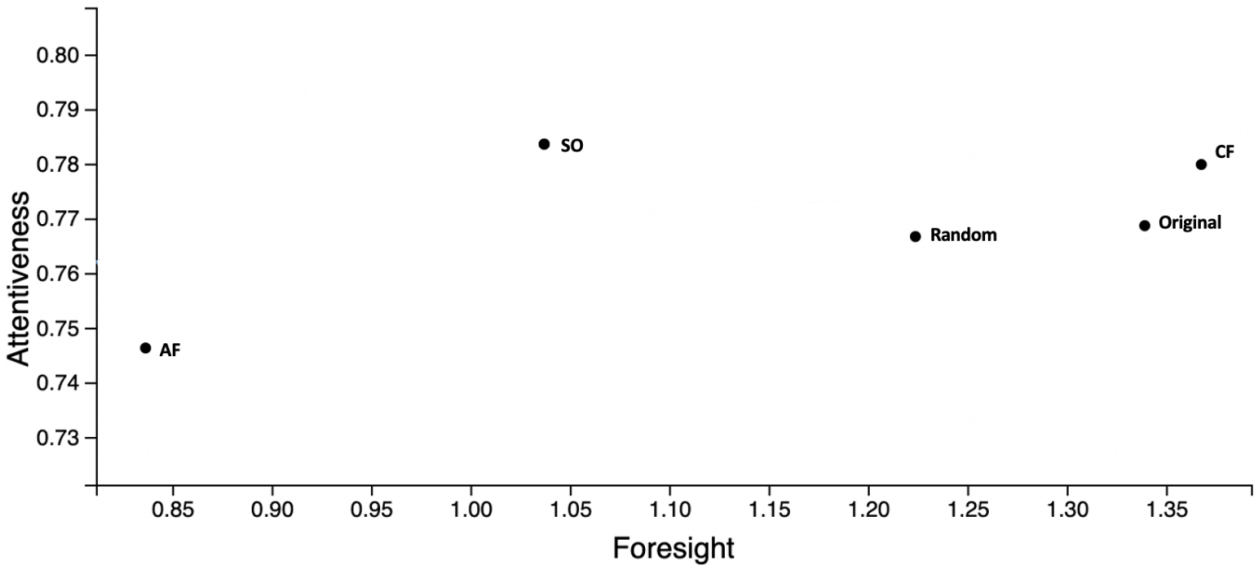


Figure 3.11: **Foresight vs Attentiveness for Calculator Assignment.** This scatterplot graphs foresight against attentiveness for implementations included in the calculator validation dataset. As predicted by our hypothesis, we see that the Arithmetic First (AF) approach results in both low foresight and low attentiveness scores, and that the Control Flow First (CF) approach results in both high foresight and high attentiveness scores. This example provides a demonstration of how TASTE can be used to objectively measure and characterize different student coding strategies.

closer to the root of the tree and the nodes closer to the leaves were implemented during different times of the coding process. This is due to the fact the CF strategy focused on implementing control flow structures such as functions first and the AF strategy focused on ensuring functionality first and defining functions last.

The specification used for the calculator assignment describes the functions to be implemented one at a time while flushing out a main control flow structure. We predicted that the SO point for the calculator assignment would have high attentiveness since the specification describes each function in its entirety before moving on to the next description. The SO point has a high attentiveness value and is the uppermost point in Figure 3.11 which agrees with our initial hypothesis. Since the SO point is an artifact of the specification, for any given assignment, this point will be a reflection of how the specification was written. The assignment specification clearly explained the purpose of each function one at a time which

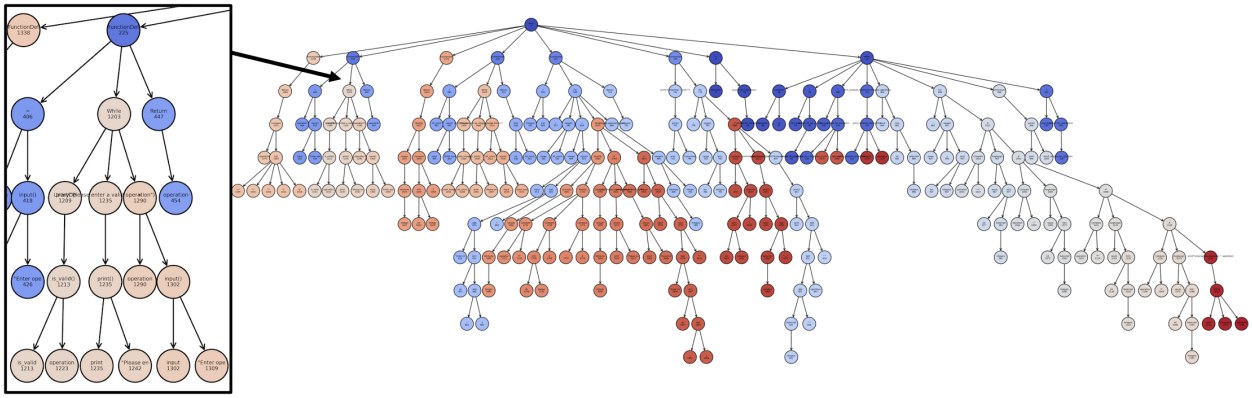


Figure 3.12: **TAST of CF Strategy for Calculator Assignment.** This TAST model is specific to the control flow first (CF) point in Figure 3.11. Darker blue nodes represent smaller creation values and were therefore coded first. Darker red nodes represent larger creation values and were therefore coded last. In this model, we can see that there are more blue nodes closer to the root and more red nodes closer to the bottom of the tree. We would expect this behavior since the CF guidelines require function definitions and other control flow structures to be created early in the development of a programmatic solution.

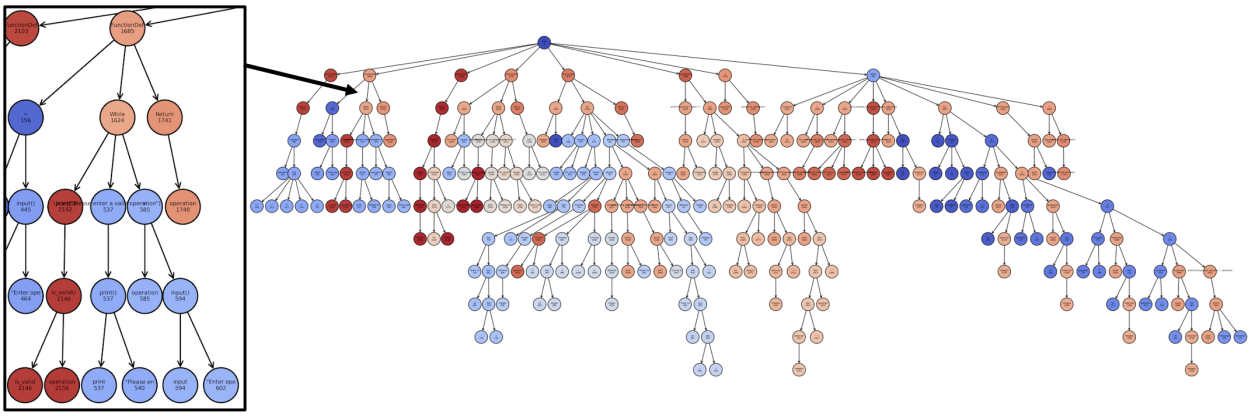


Figure 3.13: **TAST of AF Strategy for Calculator Assignment.** This TAST model is specific to the arithmetic first (AF) point in Figure 3.11. Darker blue nodes represent smaller creation values and were therefore coded first. Darker red nodes represent larger creation values and were therefore coded last. In this model, we can see that there are more red nodes closer to the root and more blue nodes closer to the bottom of the tree. This model matches our expectations since the AF guidelines require the functionality of the programmatic solution to be coded first and the organization control flow structures such as function definitions to be coded last.

guides the SO strategy to finish each function to completion and to move through the code in a linear fashion. However, if the specifications had described the functionality of a function

partially at the beginning of the instructions but then appended additional functionality to the same function in a later portion of the description, that would lower the attentiveness score of the point associated with the SO strategy for that assignment.

The point closest to the CF point represents our original solution that we coded with no intended strategy. The location of this original solution is influenced by the fact that the programmers who implemented the solution identified their own program strategies as being similar to the CF strategy. The point was also influenced by the fact the programmers occasionally traversed through the code in a non-linear fashion, in order to find and fix bugs, and thus the point does not have as high of an attentiveness value as other points. The middle point represents the random solution. Since the random strategy is a combination of the other three strategies we defined, we expect the point to be in between all the other points. The random point is influenced by strategies that resulted in both low and high attentiveness and foresight values, and those attributes should be reflected in the position of the point that represents the random strategy.

The second validation set was created in a manner similar to that used to create the calculator dataset but for Assignment 10 within the real world dataset. The specification for this assignment included building a class that required different functions and control flow structures. The assignment provided starter code that handled all of the user input and function calls. The assignment required the student to create a class with prespecified function signatures referenced from within the starter code. An original solution along with solutions for the SO, CF, AF, and random coding strategies were created and we refer to the data for this group of solutions as the Assignment 10 validation dataset.

In Figure 3.14 we can see the results of graphing the foresight and attentiveness of our five solutions for Assignment 10. Similar to the position of the CF point observed for the calculator validation dataset, the CF point for the Assignment 10 validation dataset has a high attentiveness and a high foresight value. The AF point displays a low level of attentiveness which we would expect based on our original hypothesis, but it has the largest

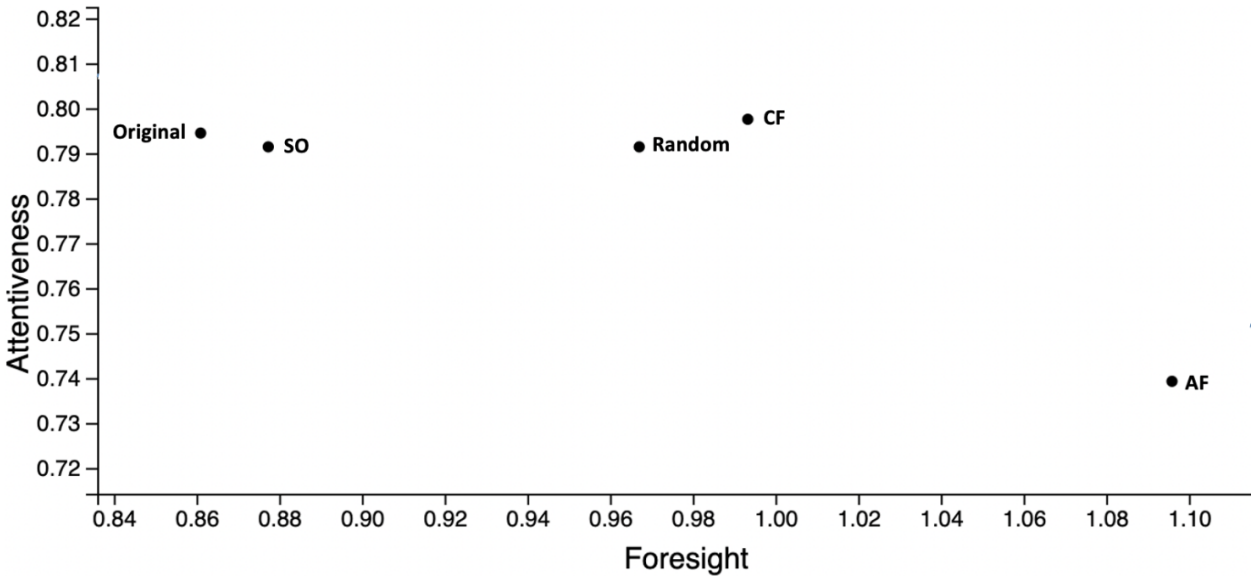


Figure 3.14: **Foresight vs Attentiveness for Assignment 10.** This scatterplot graphs foresight against attentiveness for implementations included in the Assignment 10 validation dataset and file name "wordinator.py". As predicted by our hypothesis, we see that the Arithmetic First (AF) approach results in both low foresight and low attentiveness scores, particularly relative to the Control Flow First (CF) approach. This example provides a demonstration of how TASTE can be used to objectively measure and characterize different student coding strategies.

foresight value of all the points in the Assignment 10 validation dataset. This is the opposite of what we had initially expected and is the opposite of what was observed in the case of the calculator validation dataset.

An investigation into the TAST models for the CF and AF strategies, as seen in Figures 3.15 and 3.16, aided in identifying why the results differ from our hypothesis. For Assignment 10, the students were given starter code that was responsible for handling user input and function calls. All that was required of the student was to create a class with the necessary function signatures that were called in the starter code. A large portion of the mental organization and overall logic of the solution was already provided for the student. The way that the AF strategy manifests according to the foresight metric is different because the assignment forces the student into certain decisions. The student has already been given the blueprint for part of the assignment. All that is left for the student is to determine the

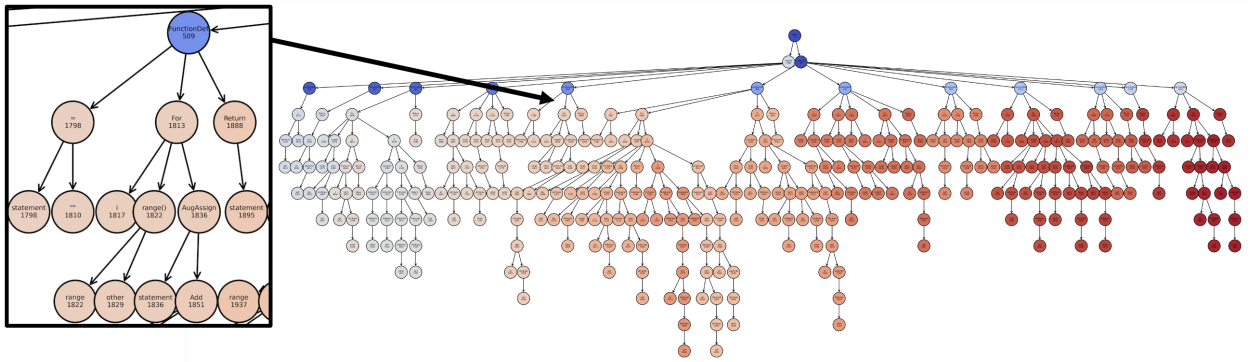


Figure 3.15: **TAST of CF Strategy for Assignment 10.** This TAST model is specific to the control flow first (CF) point in Figure 3.14. Darker blue nodes represent smaller creation values and were therefore coded first. Darker red nodes represent larger creation values and were therefore coded last. The consistency in branch color attributes to a high attentiveness score because it represents that once a programmatic element of the program was started it was completed before moving on to a new element. The darker blue nodes close to the node attributes to a relatively high foresight score since in this case the class and function definitions were coded before anything else.

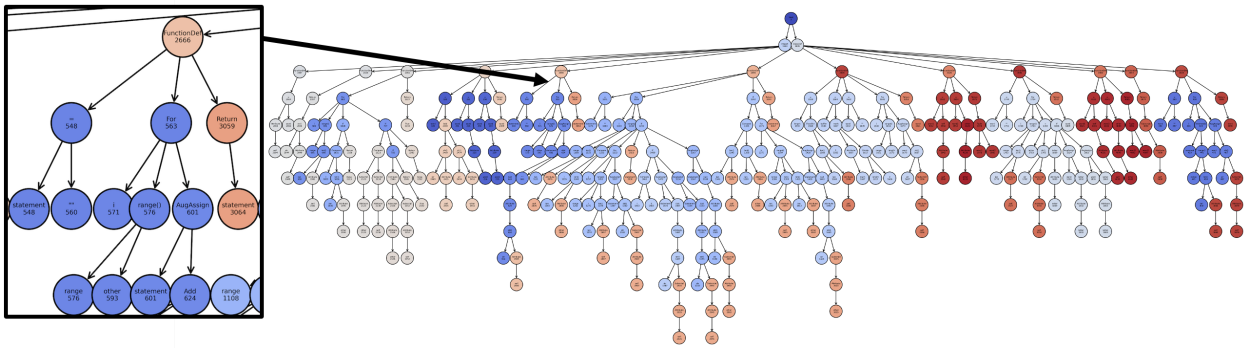


Figure 3.16: **TAST of AF Strategy for Assignment 10.** This TAST model is specific to the arithmetic first (AF) point in Figure 3.14. Darker blue nodes represents smaller creation values and were therefore coded first. Darker red nodes represent larger creation values and were therefore coded last. The inconsistent colors in the branches represents a low attentiveness score since the student worked on multiple portions of the solution at one time. Due to the nature of the assignment, this AF example, specific to Assignment 10, does not have a low foresight score. The model does not strictly have blue nodes near the bottom of the tree and red nodes near the root of the tree.

necessary logic for the functions so that they work as expected. As we can see in the CF TAST model, once the class and all of its function signatures have been defined, the CF guidelines prescribe that the next step is to implement the functions to completion. The AF

TAST model has inconsistently colored branches as it did in the calculator validation set. This contributes to a low attentiveness score, but it does not have red nodes near the root of the tree and blue nodes near the bottom of the tree as strictly as was the case for the calculator assignment (as seen in Figure 3.13). Some of the darkest red nodes are leaf nodes. In order to accurately implement the AF guidelines, the functionality of the program had to be coded first outside of the class structure. Once all the functionality was implemented correctly, the class was then created. Moving the correct functionality into its respective function required significant rewriting of variable names and control flow logic, due to the need to add return statements. The significant amount of rewrites changed many of the keystroke creation values even though the overall functionality was still the same. All of these conditions contribute to the AF point for the Assignment 10 dataset having a foresight value that did not match our hypothesis.

Many assignments chosen in introductory CS classes are selected due to the type of syntactical elements that are being tested. This result suggests that different assignments require different levels of abstract thinking, and the kinds of abstract concepts involved in an assignment are equally important to consider when choosing assignments for novice programmers. For example, in the case of Assignment 10, the instructor might have intended to take away some mental load by providing the starter code, but if the goal of the instructor is to challenge students to learn to decide how to break their solution into functions, then a better approach would be not to have specified functions within the starter code. If assignment specifications are too detailed students are led to make certain programming decisions. An instructor who wants to encourage students to have to try their own different abstract thinking strategies can provide a more broad assignment specification to push students to make their own conceptual decisions about their assignment.

3.5 Discussion

The search space of our research is much larger than defined within the work, and to cast new light on uniquely characterizable programming strategies and the metrics that properly characterize them requires a significant amount of future work. Our work only covers one real world dataset and two validation sets, nine metrics, and four predefined coding strategies within the larger search space. We have built a framework in which this future work can be effectively conducted. We have provided this subset of applications in order to supply examples of possible applications of the tool.

It should be noted that even though only four coding strategies were predefined within the scope of the validation datasets, we also acknowledged several other strategies within the basic metrics results. For example, we can consider the use of comments as a precursory task in order to outline the body of a programmatic solution as the architect strategy. We can also define the use of compiling a project often and ensuring a working program as the preventative strategy. Another strategy would be the use of many functions to organize a programmatic solution, as opposed to using minimal functions, and this strategy could be referred to as the decomposition strategy. The TASTE tool provides the ability to discover and define more of these unique programmatic strategies.

3.6 Future Work

It should be noted that the random strategy was initially selected to represent a student who was "lost" during the coding process. In creating the guidelines to be systematic and replicable, the identity of a student who is "lost" was not accurately represented within the random strategy guidelines. The strategy became more of a representation of a student who alternates between strategies rather than a student who has no strategy. A more accurate way to represent a student with no strategy might be to define guidelines that prescribe choosing random lines of code to implement and delete throughout the coding process. This

type of behavior is closer to our initial goal of the random strategy than switching between well-structured strategies.

It is worth noting that improvements to the TAST model will also contribute to improvements in the TASTE tool. As a simplifying assumption, the process of assigning keystroke values to TAST nodes uses the creation value of the first character of the associated code segment. The *ast* library within Python can access the location of the segment of code that is responsible for a node within an AST. When we access this code segment, we only look at the keystroke associated with the creation of the first character of the given code segment. There may be added benefits to using the keystroke information within the entire associated code segment. Another way to assign creation values to nodes would be to average the creation values of the entire segment and use the average as the creation value for a given node. For example, given a vertex v, v has an associated code segment $S = (s_0, ..., s_i, ..., s_l)$ where s_i is the ith character in S and l is the length of S. Instead of using the keystroke creation value of s_0, we could use the average of the keystroke creation values across all $s_i \in S$.

Our current TAST model only considers the AST of the final submitted solution of a student. In theory, an AST exists at each keystroke where the state of the code at that keystroke is compilable. A student might have created nodes that appear in intermediate ASTs throughout the development of the assignment that were not included in the final AST. For example, a student might have written an extra function to deal with error checking and then decided that they wanted to incorporate the functionality into a different function. The nodes for the error checking function existed in an intermediate tree but were not included within the final AST. Developing a way to incorporate these deleted nodes into the model or into a new metric may shed added light on student coding strategies and improve TASTE. A possible method of addressing the deleted nodes would be to look at the unique ASTs for each compilable state within the development of the solution. The intermediate ASTs can also be

used to more precisely determine when ASTs nodes are created for the purposes of creating a TAST. This method is currently being considered in the broader research community.

3.7 Conclusion

Computer science has a history of being a difficult introductory course because many novice programmers have difficulties grasping abstract coding strategies that lead to building effective solutions. New methods of categorizing and measuring student coding strategies are needed to help improve the abstract thinking processes of novice programmers. We implemented TASTE to provide a method to model the student coding process and a tool to visualize and compare different effective student coding methods.

TASTE is a tool that gives users the ability to visualize, categorize, and demonstrate different student coding strategies. The focus on student coding strategies also highlights the importance of understanding high-level thought process skills in programming. Many introductory courses focus on syntactical principles and struggle with how to effectively teach students about abstract thinking skills. A student's ability to mentally organize their solution and translate what they conceptualize into a working program is an essential skill for success. There are many possible strategies to effectively implement this skill, and the goal of TASTE is to demonstrate these strategies to students and instructors for the purpose of improving students' own programming strategies.

Chapter 4

Conclusion

TASTE was created to extend the KeystrokeExplorer with functionality to measure and visualize student coding strategies. Our goal has been to build the extension to provide a sandbox where students and instructors can identify and investigate different student coding strategies and to help improve the abstract programming skills of introductory CS students. The idea that there is only the right way and the wrong way to code a programming solution is overly simplistic; there are many paths that lead to a correct solution, and, by being able to identify and measure these strategies, we have the ability to share this knowledge with novice programmers. Only focusing on the final product of a project and not the process of developing a solution can keep introductory CS students from developing the necessary abstract thinking skills needed to become proficient programmers. The use of the TAST model and basic and custom metrics to create TASTE provides a method of identifying and demonstrating these different abstract thinking skills and processes.

Besides an instructor using the TASTE extension to examine trends and behaviors among a group of students, another possible application of the tool is that an instructor might use the tool in the classroom setting to demonstrate to students the variety of problem-solving strategies represented among their peers or among a more experienced group of programmers. The TASTE tool allows the visualization of correlations between different aspects of coding via basic and custom metrics and allows instructors to effectively demonstrate the benefit of implementing different coding strategies. A more comprehensive experimental survey of instructors using the tool in their classes and classrooms is one of several envisioned examples of future work. Another application of the tool is to investigate new metrics that can measure in a single value different programming strategies. These types of custom metrics have the potential to help instructors identify the strengths and weaknesses of students at a glance.

Thus, TASTE stands to help with the time required of instructors to extract knowledge of the strategies their students are applying in each assignment.

This body of work only examines a subset of datasets, metrics, and coding strategies within the larger scope. A substantial amount of future work is necessary to uncover and characterize more diverse coding strategies and metrics. The goal of our work was to create the TASTE extension to facilitate this type of research and future work. We have provided our subset of examples to demonstrate how TASTE can be applied to characterize and demonstrate different coding strategies with the intended purpose of aiding students in improving their own coding strategies.

References

[1] Kofi Adu-Manu, John Arthur, and Prince Amoako. Causes of failure of students in computer programming courses: The teacher learner perspective. *International Journal of Computer Applications*, 77:27–32, September 2013. doi: 10.5120/13448-1311.

[2] Vartika Agrahari and Sridhar Chimalakonda. AST[AR] – towards using augmented reality and abstract syntax trees for teaching data structures to novice programmers. In *2020 IEEE 20th International Conference on Advanced Learning Technologies (ICALT)*, pages 311–315, 2020. doi: 10.1109/ICALT49669.2020.00100.

[3] Alireza Ahadi and Raymond Lister. Geek genes, prior knowledge, stumbling points and learning edge momentum: Parts of the one elephant? *Proceedings of the ninth annual international ACM conference on International computing education research*, 2013.

[4] Alireza Ahadi, Raymond Lister, Heikki Haapala, and Arto Vihavainen. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 121–130, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336307. doi: 10.1145/2787622.2787717. URL `https://doi.org/10.1145/2787622.2787717`.

[5] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, 23:561–599, 11 2014. doi: 10.1080/10508406.2014.954750.

[6] Jeongmin Byun, Jungkook Park, and Alice Oh. Detecting contract cheaters in online programming classes with keystroke dynamics. In *Proceedings of the Seventh ACM Conference on Learning @ Scale*, L@S '20, pages 273–276, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379519. doi: 10.1145/3386527.3406726. URL `https://doi.org/10.1145/3386527.3406726`.

[7] Kevin Casey. Using keystroke analytics to improve pass-fail classifiers. *Journal of Learning Analytics*, 4(2):189–211, 2017.

[8] Michael D. Feist, Eddie Antonio Santos, Ian Watts, and Abram Hindle. Visualizing project evolution through abstract syntax tree analysis. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 11–20, 2016. doi: 10.1109/VISSOFT.2016.6.

[9] Rashina Hoda and Peter Andreae. It's not them, it's us! Why computer science fails to impress many first years. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, pages 159–162, AUS, 2014. Australian Computer Society, Inc. ISBN 9781921770319.

[10] David Hovemeyer, Arto Hellas, Andrew Petersen, and Jaime Spacco. Control-flow-only abstract syntax trees for analyzing students' programming progress. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 63–72, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344494. doi: 10.1145/2960310.2960326. URL `https://doi.org/10.1145/2960310.2960326`.

[11] Lu Jiang, Zhiyi Zhang, and Zhihong Zhao. AST based Java software evolution analysis. In *2013 10th Web Information System and Application Conference*, pages 180–183, 2013. doi: 10.1109/WISA.2013.42.

[12] Hiroshi Kikuchi, Takaaki Goto, Mitsuo Wakatsuki, and Tetsuro Nishino. A source code plagiarism detecting method using alignment with abstract syntax tree elements. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6, 2014. doi: 10.1109/SNPD.2014.6888733.

[13] Juho Leinonen et al. Keystroke data in programming courses. *Department of Computer Science, Series of Publications A*, 2019.

[14] Yu-Tzu Lin, Martin K.-C. Yeh, and Sheng-Rong Tan. Teaching programming by revealing thinking process: Watching experts' live coding videos with reflection annotations. *IEEE Transactions on Education*, 65(4):617–627, 2022. doi: 10.1109/TE.2022.3155884.

[15] Delaney Moore, John Edwards, Hamid Karimi, Rajiv Khadka, and Paul Bodily. Temporal abstract syntax trees for understanding student coding thought process. In *2022 Intermountain Engineering, Technology and Computing (IETC)*, pages 1–6, 2022. doi: 10.1109/IETC54973.2022.9796943.

[16] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083143. URL `https://doi.org/10.1145/1082983.1083143`.

[17] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 153–160, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310987. doi: 10.1145/2157136.2157182. URL https://doi.org/10.1145/2157136.2157182.

[18] Patricia Roberts. Abstract thinking: A predictor of modelling ability? In *Educators Symposium of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 753–754. Springer, 2009. ISBN 9783642044250. Educators Symposium of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems; Conference date: 01-01-2009.

[19] Kshitij Sharma, Katerina Mangaroska, Halvard Trætteberg, Serena Lee-Cultura, and Michail Giannakos. Evidence for programming strategies in university coding exercises. In Viktoria Pammer-Schindler, Mar Pérez-Sanagustín, Hendrik Drachsler, Raymond Elferink, and Maren Scheffel, editors, *Lifelong Technology-Enhanced Learning*, pages 326–339, Cham, 2018. Springer International Publishing. ISBN 978-3-319-98572-5.

[20] Duane F. Shell, Leen-Kiat Soh, Abraham E. Flanigan, Markeya S. Peteranetz, and Elizabeth Ingraham. Improving students' learning and achievement in cs classrooms through computational creativity exercises that integrate computational and creative thinking. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 543–548, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346986. doi: 10.1145/3017680.3017718. URL https://doi.org/10.1145/3017680.3017718.

[21] Raj Shrestha, Juho Leinonen, Arto Hellas, Petri Ihantola, and John Edwards. Code-process charts: Visualizing the process of writing code. In *Australasian Computing Education Conference*, ACE '22, pages 46–55, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450396431. doi: 10.1145/3511861.3511867. URL https://doi.org/10.1145/3511861.3511867.

[22] Jaime Spacco, Davide Fossati, John Stamper, and Kelly Rivers. Towards improving programming habits to create better computer science course outcomes. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '13, pages 243–248, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320788. doi: 10.1145/2462476.2465594. URL https://doi.org/10.1145/2462476.2465594.

[23] Sherry Turkle and Seymour Papert. Epistemological pluralism and the revaluation of the concrete. *The Journal of Mathematical Behavior*, 11:3–33, 1992.

[24] Daniela Zehetmeier, Axel Böttcher, Anne Brüggemann-Klein, and Veronika Thurner. Defining the competence of abstract thinking and evaluating CS-students' level of abstraction. In *HICSS*, 2019.