APPROVAL SHEET

Title of Thesis: Identifying Extraneous Elements of Novice Source Code

Name of Candidate: Michael Patrick Neary III
Master of Science, 2021

Thesis and Abstract Approved: _____

Dr. Cynthia Matuszek
Assistant Professor
Department of Computer Science
and
Electrical Engineering

Date Approved: _____

ABSTRACT

Title of Thesis: Identifying Extraneous Elements of Novice Source Code

Michael Patrick Neary III, Master of Science, 2021

Thesis directed by: Cynthia Matuszek, Assistant Professor
Department of Computer Science and
Electrical Engineering

This work analyzes elements of novice source code to support the hint generation component of Intelligent Tutoring Systems (ITS). The purpose of an ITS is to provide one-on-one guidance in a specific subject area. An important component of an ITS is the generation of constructive feedback to enable a student to correct mistakes in their work. An ITS applied within a computer science classroom can generate feedback by looking for similarities with the student's code and one or more correct programming solutions. The system would give a hint based on any similarities or lack thereof to guide the student towards a correct solution.

The state-of-the-art hint generation techniques do not take into account extraneous (unnecessary within the context of the problem) lines of code in a student's solution. Current systems are not capable of meaningfully identifying these lines, or reasoning about them to give constructive feedback. In this work, I first show that extraneous lines of code exist through analysis of novice source code to show this is a problem worth contributing to. Then, I introduce a method for their identification, and report mixed results when testing this method. This work represents the first steps towards a useful system, and I argue the importance of continued development of this system for the identification of extraneous lines of code.

Identifying Extraneous Elements of Novice Source Code

by

Michael Patrick Neary III

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2021

Acknowledgment

This thesis would not have been possible without a significant number of people in my life. First and foremost, I would like to thank my advisor Dr. Marie desJardins for her boundless wisdom and endless patience with me. She gave me many opportunities to advance my career, particularly by hiring me to work on the CS Matters project in the MAPLE Lab. She helped me to pare down my original thesis idea into something more manageable. She carved time out of her incredibly busy schedule to meet weekly to talk about progress. Even when I dropped the ball on deadlines, and found myself ABD, she offered a kind helping hand in spite of her increased responsibilities in a new position at a new university. I will be eternally grateful for all that she did.

I would also like to thank my co-advisor Dr. Cynthia Matuszek for all that she did in the last few weeks before I submitted this document. She met with me weekly to ensure that I was on the right track, offering words of advice that I took to heart. She set up a scholarship in my name so that I would be able to pay for the final semester's worth of credits. I am incredibly thankful for her kindness in an otherwise difficult time for me.

Thanks to all of the faculty at UMBC for being excellent teachers, friends, and colleagues. Thanks to Dr. Katherine Gibson, Dr. Ben Johnson, and Mr. John Park for participating in a mock thesis committee the day before my defense so that I could get the chance to practice my presentation. Thanks to members of the MAPLE lab for hanging out and helping to bounce ideas around, including but not

limited to Matt Landen, Stephanie Milani, Shawn Squire, Nicholay Topin, and Dr. John Winder.

This thesis was the the hardest thing I have accomplished to date. As it turns out, working on a self directed research project with undiagnosed ADHD is incredibly difficult. Thanks to all of my friends in Banana Phone, Dog Collar Comedy, and the UMBC Musical Theatre Club; I would would have gone absolutely crazy without an outlet. Thanks to Mom, Dad, and my brother Matthew for supporting me in all of my endeavors. Thanks to Grace Chandler for being my person.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1: Introduction

1.1 Overview

Intelligent Tutoring Systems (ITS) are a method of computer-based reinforcement of knowledge. They are capable of tracking student progress on a topic, and reinforcing that knowledge with constructive feedback. Recently, these systems have been used in introductory computer science classes. Intelligent Programming Tutors (IPT) are capable of helping students solidify their knowledge of computer programming by reinforcing concepts taught in a traditional lecture setting. The generation of meaningful hints in an IPT is difficult to do correctly. A successful IPT is capable of generating constructive feedback to guide the student towards the correct solution of a problem that they are working on.

Every problem has a particular goal in mind. Feedback to help reach that goal should be infrequent enough as not to coddle the student, but it should also be given exactly at the right time it is needed. Feedback should be informative enough to give insight into a misconception, but should not hand out the answer for free. Current methods to produce feedback include the comparison of the student's answer to past known solutions, or the generation of all possible solutions and searching through that space to find a similar solution. Each of these methods relies on a comparison

to other programs, but does not inspect the student's program on its own merit. An IPT will miss vital information hidden within the unnecessary lines of code by comparing a student's program solely to other known solutions. Therefore, it is beneficial to provide a method of identifying these extraneous lines of code so a hint generation system can use that knowledge to generate better feedback.

1.2 Motivation

If a student is given a problem to solve, they could use a variety of methods to produce a solution. Many different programs can produce the same solution to a problem. Therefore, generating hints based on the difference between a correct program and the student's current attempt can be limiting in the types of hints that are generated. This method of providing feedback guides students towards one particular method of solving the problem. However, the student's attempted solution could contain lines of code that are necessary for their solution to function, but are not necessary in what is considered correct. In fact, they could have lines of code that are unnecessary regardless of the correct solution used in current methods. I believe this to be an oversight in current hint generation systems: most methods simply ignore the potentially important information contained in unnecessary code.

1.3 Research Problem

I focus on the detection of extraneous lines of code in programs written by novice students. An extraneous line of code is a line of code written in a program

that does not have an effect on the desired solution to a problem. The majority of errors that students make when learning to program are syntactic [1]; such errors are easily detectable by the compiler or interpreter of the language. An extraneous line of code is, by contrast, a semantic error: the student may believe that writing that line of code has affected their desired solution, when in reality it has not. Novice students are often unable to accurately trace their programs linearly [2], which may contribute to the addition of meaningless lines of code. An unimportant line of code can contain valuable information about the misconception(s) that a student has on a particular topic, which is useful for generating meaningful hints. Identifying such lines of code is important to the acquisition of programming knowledge.

1.4 Contribution

In order to feasibly produce feedback from extraneous lines of code, I must be able to identify them programmatically and within a reasonable amount of time. I propose a system that is designed with the intent of identifying extraneous lines of code in a target program. The system utilizes artificial intelligence techniques and syntactical analysis to construct a graph of dependencies among lines of code. The single source, shortest path (SSSP) algorithm is applied to that graph to determine the lines that are not used between the start and end of a program. My system yields mixed results when tested against novice source code with extraneous lines labeled by domain experts. I explore a few potential reasons for these mixed results ranging from expert labeling error to the underlying difficulty of the research problem. These

results represent an initial exploration of this particular space, and I argue that future work will advance Intelligent Programming Tutors.

Chapter 2: Background

The purpose of this chapter is to describe the components of Intelligent Tutoring Systems (ITS), and to give motivation for using ITS in programming courses. This chapter provides the necessary background knowledge on ITS, first-order logic, abstract syntax trees, and graph theory in order to understand the methodology in Chapter 4.

2.1 Intelligent Tutoring

An Intelligent Tutoring System (ITS) provides students with one-on-one guided instruction in a particular subject, typically after they have had prior exposure to that subject in a traditional classroom setting. A typical ITS mimics the interaction between a human tutor and a student. The goal of this interaction is to bring the student from a state of confusion in a subject to mastery of that subject. The ITS develops an internal model of what the student understands, providing the student with feedback based on that model to help guide the student towards mastery. The successful implementation of a tutoring system can be rewarding for both student and teacher, relieving some of the burden of teaching and learning in particularly difficult courses.

A majority of the content in introductory programming courses is difficult to grasp for the novice student, especially considering that the material is almost entirely novel. Those who fall behind in an introductory course are likely to suffer learned helplessness because each new concept builds upon previous concepts [3]. Students also find it difficult to combine the basic structures of programming to form solutions to larger problems. If a student is able to think of a programmatic solution, they may still have trouble fixing bugs within that solution [4]. Students learn different concepts at different rates, which makes deciding what programming concepts to teach in what sequence particularly hard [5]. The programming instructor must find solutions for each these problems in order to be effective. The difficulty of programming for first-time learners and their instructors justifies the application of ITS to introductory programming.

An Intelligent Programming Tutor (IPT) is a specific implementation of an ITS for introductory programming. There are three important components of an IPT: domain knowledge, student knowledge, and tutoring knowledge. An IPT needs domain knowledge, or a method of representing concepts related to programming (e.g., language syntax), in order to identify define content mastery. It needs a model of student knowledge, or a method of encoding student understanding based on observation (e.g., code samples, multiple choice answers). An IPT also needs to know how to tutor, or how to provide meaningful feedback to a student in a timely manner. Exceptional IPTs must have all three components, as constructive feedback is best generated from the combination of an accurate student model and sound domain knowledge. In this chapter, we discuss methods of modeling domain

and student knowledge. The next chapter focuses on constructive feedback in-depth.

2.1.1 Domain Knowledge

It is imperative that an ITS has expert-level domain knowledge in the subject area it is applied to. An IPT needs knowledge of programming concepts in order to model what a student knows and to give feedback to that student. The necessary programming concepts for an IPT are how to write programs, how to identify and describe errors, and declarative information about programming structures [6]. These concepts should be encoded in an easily accessible manner, such that the student model and feedback generator can leverage the information. Programming domain knowledge need not be a separate entity in an IPT, although there are methods that rely on domain knowledge as a separate entity. One can easily incorporate domain knowledge into a student model or feedback generation method.

One direct method of domain knowledge representation utilizes first-order logic to create a knowledge base of programming concepts. This method views a solved programming problem as a conjunction of predicates that describe the expected constructs in an ideal solution. Weregama and Reye [7] translated a set of student code examples into a knowledge base of correct solutions. They leverage this method of domain knowledge encoding to administer tutoring through simple planning algorithms. This method creates an accurate tutoring system, and is suited for handling the differences in solutions to problems.

Some IPTs encode domain knowledge as a graph to capture the relationship

between programming concepts. Understanding the relationship among programming concepts is crucial for navigating the tutoring process in an IPT. If the system understands what the student has not yet mastered, and it knows the similar concepts that they do understand, it can generate feedback to bridge that gap. Kumar extends the graph of programming concepts, adding learning objectives to each concept [8]. Expert domain knowledge is the foundation of feedback generation and the internal student model.

2.1.2 Student Knowledge

It is important for the interaction between an intelligent tutor and a human to be as personalized as possible, since such a system must have an understanding of what the current user is capable of. It should reason from what it observes of the student's actions to determine the content that student has learned. This is a difficult task due to the inherent uncertainty in the observation of a student. A variety of methods currently exist for the representation of student knowledge given this uncertainty. A few of these methods utilize classic techniques such as Bayesian networks and Markov Decision Processes, while others employ novel methods. I first discuss the classic techniques applied to this problem, then more recent methods.

A Bayesian network is a directed acyclic graph used to encode conditional dependencies among random variables. These networks are the most utilized method for the modeling of student knowledge because they are easy to implement and they can determine the probability that some concept is understood given evidence, or

lack thereof, for that hypothesis. Butz et al. use a Bayesian network to model the student who is using the system, and update their model based on the student's self-assessed understanding [9]. Chang et al. go a step further by treating the student modeling problem as a function of time. This approach allows for the application of a dynamic Bayesian network. The model constructed is better than standard knowledge tracing, at the cost of an increase in training time [10]. A Markov Decision Process can also be used as a model for student behavior in a tutoring system. This technique determines an optimal policy for the intelligent tutor given a set of actions it can take (feedback to the student) and the states that result from taking those actions (student understanding of the material), with some probability that action A results in a transition from state S to S′. A student can be modeled using a Partially Observable Markov Decision Process (POMDP) because there is a level of uncertainty in the result of an action taken by a student. The model must do what it can with unreliable evidence, since a piece of feedback the student receives may either help or hurt their understanding of the topic. A student's response to a problem can vary, and it is not certain what state of understanding they are in until after their response is analyzed. A POMDP is a powerful modeling method, but the application of POMDPs can be intractable. Folsom-Kovarik et al. propose two variations—state queues and observation chains—of POMDPs to solve the problem of increased complexity in modeling student knowledge [11]. Both methods use properties of tutoring tasks to compress the information needed to reliably model a student with a POMDP. Their results show that their compression methods do not have any negative effect on the student model, but there was no substantial

improvement over existing modeling methods.

2.2 First-Order Logic

My thesis uses the concept of first-order logic (FOL) to represent a particular kind of dependency among lines of code. First-order logic statements utilize variables in order to describe objects and relationships in the world. These variables can be quantified: that is, one can write a statement that applies to every object or at least one object without naming anything specifically. A term in an FOL statement can be a constant, a variable, or a function. A constant is a symbol that represents a specific thing in the world. A variable can represent any object in the world. A function is a rule applied to one or more terms. Terms are used to write sentences, a predicate of n terms that evaluates to some truth, in FOL.

Complex sentences are created by joining terms with logical connectors, including conjunction ($\wedge$), disjunction ($\vee$), or implication ($\rightarrow$). Sentences can be quantified either universally ($\forall$, "for all") or existentially ($\exists$, "there exists"). The universal quantifier applied to a variable in a sentence is an assertion that the sentence is satisfiable by all members of the variable's domain. The existential quantifier on the same variable and sentence is an assertion that the sentence is satisfied by at least one member of its domain. As an example, consider the following English sentence: "All graduate students are tired." This is a blanket statement about all graduate students; therefore, a universally quantified sentence is necessary. The translation of this English sentence in FOL could look like this: $\forall x\ GradStudent(x) \Rightarrow\ Tired(x)$.

In this sentence, the variable x has the domain of people. This sentence is read as, "For all people, if that person is a graduate student, then they are tired." A group of sentences can be written to describe a system, and subsequently reasoned over to prove different sentences related to that system.

FOL is used to prove statements about the world based on what is already known or observable. Observed facts and known rules can be combined into a collection of sentences known as a knowledge base (KB). The KB is a collection of statements about the world. A model of a KB is an assignment of True/False values to each symbol in the KB. Given a model, the conjunction of each evaluated sentence under that model results in an over all KB evaluation to True or False. KB evaluation is important to determining logical entailment.

The logical entailment of a sentence S by a KB, denoted by $KB \models S$, occurs when every model that evaluates KB to True also evaluates S to be True. That is to say that the sentence S is a logical consequence of the knowledge base. Resolution is used to check if $KB \models S$. This process attempts to prove that no model exists such that $KB \wedge \neg S$ evaluates to True. Resolution is performed by applying logical inference rules to the previous conjunction. If a contradiction arises during the application of logical inference,then that proves $KB \models S$.

2.3 Graphs

A graph, sometimes referred to as a network, is a mathematical construct used to describe a relationship among objects. A graph G is comprised of two sets

(V, E), which correspond to vertices and edges. Each vertice, or node, represents one particular object. Each edge is a pair of vertices (u, v) that represents a connection between two objects. The edges of a directed graph go one way from the source node (u) to the destination node (v), whereas an undirected graph's edges are bidirectional. A relevant problem to this thesis is that of shortest paths.

A number of relationships exist within a computer program, where single instructions interact with data or with other instructions to accomplish a particular goal. A program can be represented as a graph where nodes correspond to line numbers of individual instructions and edges correspond to an interaction that takes place between two instructions.

2.3.1 Single Source Shortest Path

Given a directed graph G = (V, E), a path from a starting node to a destination node is a sequence of nodes in V such that each pair of adjacent nodes in the sequence are connected by an edge in E. The problem of shortest paths is finding the path of minimal length from a starting node to a goal node. The problem of single source shortest paths (SSSP) is finding every shortest path from some beginning node to every other node in the graph.

2.4 Abstract Syntax Trees

Formal representation of language syntax and structure is useful at compile time and for general analysis of program structure. An abstract syntax tree (AST)

encodes the syntactical structure of source code as a tree; i.e., an undirected graph where any pair of vertices have exactly one path that connects them. Each node of an AST represents an abstract programming construct within the source, such as statements, expressions, or decisions. An AST disregards portions of the actual syntax since some constructs, like grouping of parentheses, can be derived from the tree structure itself.

Chapter 3: Related Work

In this chapter, I provide related work in the generation of constructive feedback for Intelligent Programming Tutors. I discuss the strengths and drawbacks of each method.

3.1 Constructive Feedback

Quality feedback is important to facilitate the acquisition of new knowledge. The majority of feedback that a new programmer receives comes from the compiler or interpreter of the language they are using, which can be hard to decipher. This type of feedback is useless to the novice beyond highlighting syntax errors. It is therefore up to the instructor to provide meaningful feedback on student programming work, ranging from proper syntax use to walking through flawed logic. An IPT removes the need for the instructor to intervene and correct a flaw in student thinking. It should provide meaningful hints towards valid solutions and correct explanations of errors that arise when programming.

Stamper et al. laid the groundwork for constructive feedback in an intelligent tutor with the Hint Factory [12]. They created a system for a logic tutor that could generate step-by-step hints as a student was attempting to solve a problem. This

system used data from past student problem solving attempts to create a Markov Decision Process, effectively creating a student model that was not individualized. Using this model, the system would figure out what the next best problem solving state was, and generate a hint to get the student into that state. Through the use of this hint generation method, Stamper found that students increased the number of times they attempted to answer questions, and increased the number of questions they solved overall. The students who used the hint generation system achieved higher scores on post-tests in their experiments and received higher course grades overall. The Hint Factory was meant to be generalizable to all sorts of intelligent tutors, but there are limitations to using this system in a tutoring environment for programming.

Generating hints for a student trying to solve a programming problem can be difficult. The solution space for a single programming problem can be infinitely large unless a bound is placed on the syntax or standard library functions that are allowed in a solution. Searching through a solution space to find the most similar solution to what a student has is therefore impractical. Current methods for hint generation in this space either circumvent the need for looking at the solution space, or take steps to pare down its size.

There have been a few approaches for improving hint generation and programming feedback in an IPT. Recently, there has been a push towards the use of programming data to generate feedback. One method was developed by Lazar and Bratko without the need for a state-space representation of the problem solving process, instead analyzing student programs by looking for common "edits" between

them [13]. Once these edits were found, the authors applied them to a student submission until a proper solution was reached. From the sequence of applied edits, they derived hints they would show to the student. Using this approach, they were able to fix 70% of student submissions, independent of language choice, without the need to manually enter common mistakes for the system to be aware of.

Other approaches treat the problem solving environment of learning to program with a state-space representation method. Koedinger et al. generate a solution space using Abstract Syntax Trees created from correct student code samples, and they reduce the size of this solution space by applying certain transformations on these ASTs [14]. Using the reduced solution space, they create the AST for a student's intermediate submission, and then compute the difference between the intermediate solution and the known solutions. The goal is then to reverse the differences to generate feedback from the closest solution found.

Expanding on their previous results, Rivers and Koedinger describe a path construction method, where instead of comparing all possible solutions, they develop a space of possible paths that a student may take in order to get to a solution, and generate hints based on the most likely path that a student could take, given the previous attempts [5]. This method was more computationally feasible than generating the solution space, and they found most hints to be generated fairly quickly (in less than one minute). However, they are not certain how helpful the hints generated were for the students interacting with the system.

Chapter 4: Methodology

My process for detecting extraneous lines of code is composed of two parts: the detection of line dependencies and the determination of program correctness. This chapter discusses the methods I use to uncover dependencies among lines of code in a given program. I also discuss the process of determining the correctness of a solution. Combining both line dependencies and program correctness, I describe a method for identifying extraneous lines of code.

4.1 Line Dependencies

The execution of a single line of code in a program depends on the outcome of at least one line of code that came before it. This is a quality shared by all programs, and it can be leveraged to find extraneous lines of code. The following section outlines three different types of line dependencies necessary to find such extraneous code.

Understanding how data flows through a program helps to determine what computations are necessary to fulfill the program goal.

4.1.1 Data Dependencies

The first type of line dependency is the data dependency. This line dependency is defined as a relationship between two lines of code where each line uses the same piece of information in a specific manner. As a program executes, it stores data in variables and uses those variables for various computations. This dependency occurs between a line that assigns or alters the value of a variable, and a line that uses the contents of that variable in some manner. The line that edits a value must come before a line that uses that value. Two lines that use the same value in different computations do not constitute a data dependency, since data isn't flowing between those two lines: it is only being looked up. When a variable is updated, any line that uses that value later would be dependent upon the line that performed the update and not on any previous lines that altered that same variable.

Formally, a data dependency exists between two lines (a, b) if and only if the following is true: line a comes before line b in the source code; there is an identifier x that exists on both line a and line b; line a is an assignment of a value to identifier x; and line b uses identifier x is a way that is not an assignment. This definition is easily written in FOL in Equation 4.1:

$$\exists a, b, id \; Depends(a, b) \leftrightarrow isBefore(a, b) \wedge Assigns(a, id) \wedge Uses(b, id). \quad (4.1)$$

The domains of a, b are all possible line numbers (as integers) in the target program. The domain of id is all possible variables that are initialized at any point in the target

program. Each relationship between a, b represents a fact about the interaction between those two lines. $isBefore(a, b)$ encodes the fact that line a comes before line b in a piece of source code. $Assigns(a, id)$ encodes the fact that line a contains the assignment of a value to identifier id. $Uses(b, id)$ encodes the fact that line b needs id id in order to execute.

4.1.2 Structural Dependencies

The second type of line dependency is the structural dependency. This type of line dependency is defined as the relationship between a line of code and the programming construct that encloses it. There are a few programming structures that enclose blocks of code, such as the if statement, for loop, and while loop. The only way that a line of code enclosed by one of these programming structures will execute is when the entry point of that structure is triggered. For example, the lines of code enclosed by an if statement will only execute if the condition of that statement is True. Therefore, a structural dependency exists between the line of that condition and each line of the if statement's body, as the body would not execute without first testing the condition. A similar argument can be made for the iterative structures For and While.

4.1.3 Execution Dependencies

This is the most straightforward type of line dependency. A program is simply a sequence of lines of code that execute one after another. It can be said that a

dependency exists between each the each pair of lines in the chain of execution. That is to say that if line A is executed in a program's life cycle, followed by line B, then a dependency exists between line A and line B. This type of line dependency is used to determine the usefulness of other types of line dependencies. It is important to note that using only execution dependencies to detect extraneous lines is not enough.

4.1.4 Code Coverage and Dead Code

An analysis of only the executed lines of a program does not provide enough information for the detection of extraneous lines of code. Discovery of dead code — i.e., lines of code that are never executed — depends on the analysis of program execution. Tools for the discovery of dead code are an important part of the software development life cycle, but are not sufficient for extraneous line detection. In the IPT domain, novice programmers often write lines of code that are executed by the machine but do not contribute to the overall goal of the program. As seen in Figure 1, if a student initializes a variable in a global scope that they fail to use in the remainder of their program, that code would not be caught by analyzing the executed lines of that program. The line that contains that variable's assignment will be executed every time, and thus not be deemed extraneous even though it does not affect the overall goal of that program.

```
1 CONST_VAR = 45
2 def main():
3     x = int(input('Enter an integer:'))
4     print(':) ' * x)
5 main()
```

Listing 1: Line 1 is executed, but it is not used anywhere else in the program. If execution dependencies are used, Line 1 would be incorrectly identified as not extraneous.

4.2 Uncovering Line Dependencies

In order to identify extraneous lines of code, we must first detect each of the three types of line dependencies. Data dependencies are the most computationally difficult to detect, followed by structural and execution dependencies. The process of detecting each of the described dependencies is discussed in the following sections.

4.2.1 Detecting Data Dependencies

The detection of data dependencies logically follows from the formal definition. First order logic resolution is applied to a knowledge base in order to determine the lines of code that are dependent upon each other. The target program must first be converted into a knowledge base. To begin this process, the $isBefore$ relationship is added to the knowledge base for every line number with every other line number that occurs afterwards. The target program is then transformed into an AST. This encoding standardizes the target program to make certain relationships easier to find than processing the original source.

The AST is traversed node-by-node to find the lines that contain data assignment and usage. Each time that an identifier is assigned a value at a node on the

AST, the *Assigns* relationship between the line number of that node and the referenced identifier is added to the knowledge base. This relationship signifies that the identifier is assigned a value on that line, but the actual value of the assignment is not relevant to the dependency. It only matters that the identifier becomes a part of the program at that line. Each time that an identifier appears on a line other than when is it assigned a value, such as in an expression or in an argument to a function call, the *Uses* relationship is added to the knowledge base. This relationship between the line number and the identifier signifies that the identifier is used in some manner on that line. The actual usage of that identifier is not relevant to the dependency; only the fact that is was used in some manner is relevant. After traversing the AST, the knowledge base contains every instance where an identifier is assigned a value or used in some manner in the form of a fact.

The knowledge base is now usable as it contains all the necessary facts to determine the data dependencies. First-order logic resolution is applied to determine all possible pairs of line numbers (a, b) that satisfy Equation 4.1. This process results in a list of every data dependency that exists in the target program.

4.2.2 Detecting Structural Dependencies

A structural dependency exists among two lines (a, b) if they are both used in the same programming structure, independent of the contents of each line. The structural dependencies of the programming structures available to the novice student look similar. It is possible to define this type of dependency for all programming

structures, but I limit the definition to three basic structures known to all students within the first few weeks of an introductory course. The dependencies within Conditionals, For Loops, and While Loops are defined in the next sections.

The structural dependencies in conditional statements are straightforward. The body of an If statement will only execute if the condition of that statement is true. Regardless of how that condition evaluates at run time, every line in the body of an If statement is structurally dependent on the line that contains the condition. Similarly, if there is an attached Else clause, every line in the body of that clause is dependent on the condition. The Else If clause is similar, except every line in the body of that clause is only dependent upon the condition of the Else If and not any preceding conditions.

4.2.3 Detecting Execution Dependencies

The detection of execution dependencies can be performed with existing code tracing solutions, usually used for debugging software. A code tracer executes source code one line at a time, reporting the time and order in which lines are executed. This report is used to say that an execution dependency exists when a line b is executed immediately after another line a.

4.2.4 Directed Dependency Graph

A method to successfully detect each type of line dependency has been described, but that detection alone is not sufficient for finding extraneous lines. The

next step to find extraneous lines is the selection of an appropriate data structure. A line dependency is a one-way relationship between two line numbers, resembling an edge in a directed graph. Intuition suggests that a directed graph of line dependencies would allow the discovery of a path through dependencies as a program executes. The idea of tracing through dependencies to find extraneous lines motivates the construction of a directed dependency graph (DDG) from every known line dependency in the target program.

To build the DDG, start from the target program itself. For each valid line number in the target program, add a node to the graph. Add an edge for each detected dependency (a, b) among lines. The source of this edge must be b, and the destination of this edge must be a. It is now possible to perform searches and path-finding over the underlying dependencies in the target program.

```
1 var1 = 3
2 var2 = 15
3 val = var1 + var2
4 if val % 2 == 0:
5     print('even')
6 else:
7     print('odd')
```

Listing 2: A simple program used to demonstrate the concept of a DDG in this work.

Consider the short program above. The DDG without execution dependencies is constructed in 4.1.

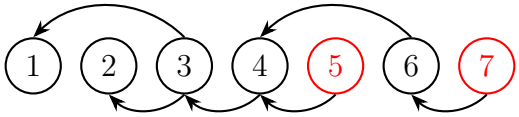

Figure 4.1: A short program, and the derived DDG without execution dependencies.

Here is that same program, and the resulting DDG with execution dependencies:

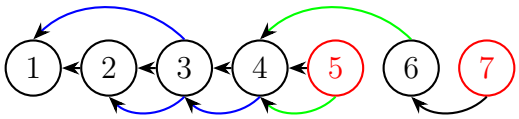

Figure 4.2: A short program, and the derived DDG with execution dependencies.

4.3 Program correctness

The next step in detecting extraneous lines of code is determining the line in the target program that is responsible for outputting the correct solution. For this step, I assume that the target program is in a final state. A final state proposes a solution to the problem it was written for, producing an output for every input

given to it. I assume a final state due to the uncertain nature of extraneous lines of code. A program may contain lines of code that do not contribute to the output before it is complete. These lines may end up tying into the solution after more code is written. Also, computing the dependencies can be costly for large programs as it involves both program tracing and recursion over an AST. Therefore, it is not reasonable to detect extraneous lines of code unless the target program is in a final state. With this assumption, a target program is correct if it is in a final state, and it satisfies every goal of the problem it is intended to solve

The goal of a particular target program is defined prior to performing the extraneous line detection process. A goal defines what that program is trying to accomplish in order to determine if a given solution is correct. Typically, the goal of a small program written by a novice is to output some value given some input. The exact mapping from input to output is usually given in an assignment description. For example, a student may be asked to take an integer as an input, and output whether or not that integer is even. Programs can have multiple goals if there are multiple tasks for a student to perform. In the previous example, the student could also have to write a value to the screen a certain number of times. Multiple goals need not be satisfied for the same input as different inputs might illicit different outputs. For example, the target program could be a currency converter that takes an amount in US Dollars and outputs that amount in Canadian Dollars. The goals for a program should be defined on an input-by-input basis in order to reflect the correct result given a particular input. Defining the program goals allow us to determine where in the target program that solution is finalized.

4.3.1 Matching Code Lines to Output

After defining the goals of the target program, it is necessary to determine which lines of the source code are responsible for producing its output. This step is done by running the student's code through a program tracer. The output of the tracer in a combination of the output of the program immediately preceded by the line of code that executed before the output was produced. For each line that produces an output as determined by the tracer, I add a relationship between the line number and the contents of the output as a fact to the KB. Each line number and string content pair as given the *hasOutput* relationship.

4.4 Reporting Extraneous Lines

A target program has been manipulated in such a manner that makes the detection of extraneous lines simple. If the line responsible for a particular output is known, this method is able to trace backwards through the dependencies of that line in the DDG. The dependency edges were added with b as the source to help with this backwards tracing step. An extraneous line of code does not appear on a path from the solution line to any node above that line. That is, an extraneous line of code was not used to arrive at a particular output. To ensure accuracy, the path from the solution line to every line that occurs before it is checked. This is done by performing the single source shortest path algorithm for each identified output line.

Chapter 5: Experimentation

This chapter discusses the methods of experimentation that are used to analyze student solutions to programming problems, including the human labelers, their qualifications, and their overall agreement on the data sets.

5.1 Data

The data that I use to test my method of extraneous code detection originates from the Computer Science I for Majors (CMSC 201) course at the University of Maryland, Baltimore County (UMBC). This is the first course in programming, required for Computer Science majors at the university. The course is taught in Python, a suitable beginner's programming language. It is assumed that the students in this course do not have prior programming experience. Homework assignments during this semester were given weekly, with four to seven individual problems per homework. I have collected several hundred code samples for two different problems that were given as homework assignments during the Fall semester of 2016. Each of these problems have been selected for their simplicity; they can be solved by applying a few fundamental programming structures. The following paragraphs are short descriptions of each of the problems used in this work.

```
def main():
    temp = float(input("Please enter the temperature: "))
    unit = input("Enter 'C' for Celsius, or 'K' for Kelvin: ")
    if unit == "K":
        temp -= 273.15

    if temp >= 100:
        print("At this temperature, water is a gas.")
    elif temp <= 0:
        print("At this temperature, water is a solid.")
    else:
        print("At this temperature, water is a liquid.")

main()
```

Listing 3: A correct solution for Problem 1.

5.1.1 Problem 1: State of Matter

The goal of this problem is to output the state of matter that water would be in at a certain temperature. This problem asks the student to accept two inputs from the user: a floating point number that represents a temperature and a character to determine what unit that temperature is in ('C' for Celsius or 'K' for Kelvin). Acceptable output should contain one of the following strings: 'liquid', 'solid', or 'gas.' The output format was not strictly enforced, as long as it included the correct string for the given input. An appropriate solution uses two input statements, followed by a sequence of condition statements, to produce correct output. Refer to Figure 3 for a correct solution, bearing in mind that this problem's complexity gives rise to many different methods of solving this problem.

5.1.2 Problem 2: Box Display

The goal of this problem is to display a rectangular box on the screen constructed with any character on the keyboard. The problem asks the student to accept two integer inputs that represent the height and width of the box, one character input that fills the inside of the box, and one character input that makes up the border of the box. Their program must output a box with the given dimensions, constructed with the given characters in the correct places. A student must combine multiple programming structures to give a correct output as they must consider a few edge cases in order to produce a correct solution. Refer to Figure 4 for a correct solution.

5.1.3 Data Preprocessing

Each of the data sets must be standardized before my extraneous line detection method is evaluated. Standardization of each file helps streamline the process of labeling the extraneous lines, both manually and automatically. The contents of each file in the same assignment group are different for each student, but the name of each submitted file is the same within each data set (i.e., hw3.txt). To distinguish between unique solutions within each data set, every file name is appended with a different number (e.g., hw3_123.txt). Every file is stripped of the file header and comments to protect the identity of the student who submitted that file. Identity was not preserved between the datasets, meaning that hw3_123.txt and hw5_123.txt may not be the same student's code. All blank lines are removed from the file to

```
def main():
    width = int(input("Enter the width of the box: "))
    height = int(input("Enter the height of the box: "))
    outline = input("Enter a symbol for the box outline:")
    fill = input("Enter a symbol for the box fill:")

    for i in range(width):
        print(outline, end = "")
    print()

    if height > 1:
        for i in range(height - 2):
            print(outline, end = "")

            if width > 1:
                for j in range(width - 2):
                    print(fill, end = "")

                print(outline, end = "")

            print()

    for i in range(width):
        print(outline, end = "")

    print()

main()
```

Listing 4: A correct solution for Problem 2.

remove the possibility of reporting such lines as extraneous. All empty files are removed from the dataset.

5.2 Data Labeling

Each individual assignment must be manually checked for extraneous lines of code in order to properly evaluate my detection method. In other words, a baseline of "true labels" that describe whether or not a line of code is extraneous needs to be created in order to understand if this detection method is valid. This process involves reading through each program line by line, scanning for lines of code that do not make sense or do not help the student solve the problem. This is a simple task that I could perform on my own, so I independently produced a set of labels for each assignment in the data. However, if I were the only person creating this baseline, any produced results would be of questionable integrity. I am well aware of the intricacies and limitations of my system. I could easily label lines that I know would be detected and not label lines that I know would not be detected, either deliberately or subconsciously. I need to take precaution to deter bias in the results of my experiments. Therefore, I enlisted the help of four experienced undergraduate computer science students as domain experts to label these data independently of each other.

It was important to be thoughtful when getting the experts ready to help label the data. Each expert needs to have a few important things explained to them first. The set up process involved an explanation of the definition of an extraneous

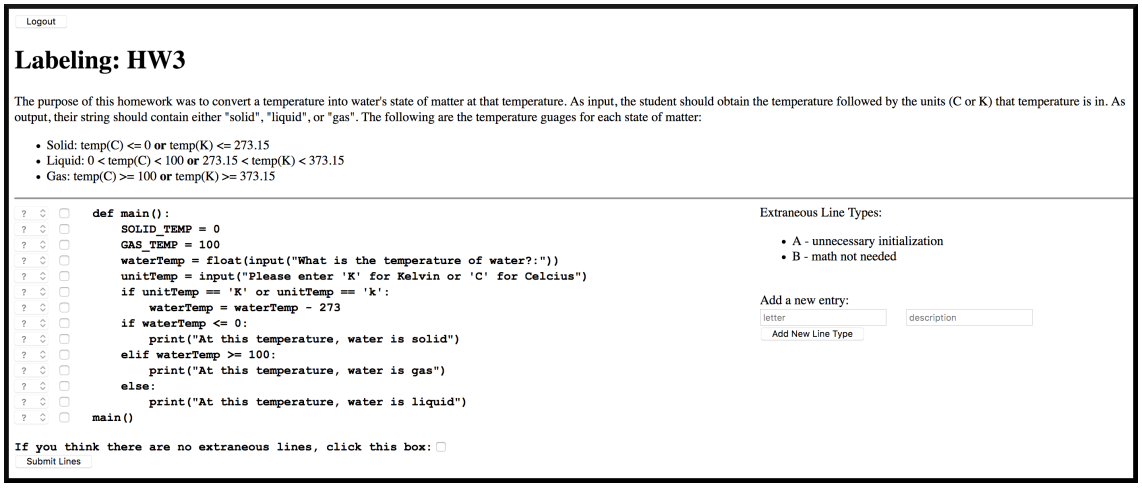

Figure 5.1: Sample screen of a student labeling an assignment.

line of code, an explanation of the two problems, and a brief explanation of the labeling web application used for collecting labels from each expert. I explained the definition of "extraneous lines of code" used in this work to each of the experts. I did so in a uniform manner as not to bias the labels produced by anyone. The experts were allowed to ask for clarification on the definition at any time, but I did not comment on any question they had regarding a specific line of code. I did not have a role in the labeling process of each individual expert, apart from getting them started. While I did produce my own set of labels, I did so after all of the experts had completed their set of labels.

I built a custom web application to standardize the collection of labels. It allows me to easily collect different sets of labels from different people. Each expert reads each program, and determines which lines, if any, are extraneous. The application expedites the process of labeling, since navigating between assignments and keeping track of what has already been labeled is taken care of for the user. Figure 5.1 shows a sample screen for a single assignment. On the top of the page, there is a

description of the problem, including any guidelines as to what constitutes a correct solution, for their reference. On the left-hand side of the screen, they can view the source code line-by-line. A line is marked as extraneous by clicking a checkbox next to the line in this area and choosing a letter from a drop-down menu. The experts were required to supply a short description of why they marked a line as extraneous. Each expert developed a unique mapping of letter to description to save time in the labeling process. On the right-hand side of the screen they can create a new mapping and view any mappings they have already made. The short description helps me to determine the reliability of each expert labeler. I can compare the lines that multiple experts marked as extraneous, and use the descriptions that the experts provide to verify whether or not that label makes sense. This labeling system contains a few shortcomings that were not discovered or mentioned until after most experts had completed their set of labels. In the interest of full transparency, these shortcomings are explained in the following section.

5.2.1 Labeling System Shortcomings

In this system, it was not possible for an expert to label a line with more than one label that they had created. If an expert believed that a line contained multiple transgressions, they had to select the one that they felt was the most relevant, instead of marking it with as many labels as possible. This was partly by design, as I did not want an expert to go overboard with the labeling of any one particular line, but it does prevent the few genuine cases where multiple transgressions were present

```
1  def main():
2      height = int(input("Please enter the height of your box "))
3      width = int(input("Please enter the width of your box "))
4      OUTLINE = input("Please enter a symbol to outline your box ")
5      FILL = input("Please enter a symbol to fill your box ")
6      listWidth = list(range(width))
7      listHeight = list(range(0, height - 2))
8      print(len(listWidth) * OUTLINE)
9      for h in listHeight:
10         str(OUTLINE)
11         fillMiddle = str((len(listWidth) - 2) * FILL)
12         print((OUTLINE + fillMiddle +  OUTLINE))
13     print(len(listWidth) * OUTLINE)
14 main()
```

Listing 5: Line 10 in this submission for Problem 2 is extraneous, but the experts gave different reasons why.

from being accurately labeled. By my count, fewer than five such occurrences exist between both problems in my data, e.g., the example in figure 5. The labeling system also did not allow for existing label letter and description pairs to be edited after they are created. This was less of an issue, as there was no limit on how many pairs that an expert could generate. However, it may have led to errors during the labeling process if a typo was made or if two lines were similar in fault and the expert did not believe that they needed two different labels. Finally, the design of the webpage itself may have led to some mislabeling. There was some error-checking built into the page itself, such as preventing an expert from submitting a file with no labels without acknowledging that they intend to leave that file unlabeled, and preventing an expert from submitting a line of code as extraneous without a corresponding label. However, these software checks are unable to catch a mistake in labeling such as accidentally labeling the line directly above or directly underneath the intended

line, so the experts had to be careful in their labeling process.

If I were to redesign the system, I would be sure to include some more data gathering, such as the amount of time that a particular expert spends on a single page between arriving on the page and clicking the submit button. This would provide an interesting metric that would allow me to gauge how much "thought" an expert put into the labeling processing: did they spend a significant amount of time reading the solution and thinking about where the extraneous lines, if any, are? Did they mistakenly click on any lines before moving on to the next file? This is one of the reasons why the selection of trustworthy and capable experts was so important, as this type of data collection is beyond the scope of this work; still, to understand how well my system performs, the labels I am using must be reliable. The following section discusses the reliability and integrity of the expert-produced labels.

5.2.2 Label Reliability

The results of each experiment are heavily influenced by the reliability of the experts who labeled the data. The experts were carefully chosen such that their overall experience would help to inform their selection of extraneous lines. To protect the identity of these experts, I will refer to them only as Expert A, B, C, or D. Experts A and D are female, while experts B and C are male. Each expert is an undergraduate computer science major who has completed the gateway requirements, meaning that they have all passed the first three introductory courses in the computer science major at UMBC (CMSC 201 - Introductory Programming

in Python, CMSC 202 - Object Oriented Programming, and CMSC 341 - Data Structures). Each expert has also served as a teaching assistant for at least one of these introductory courses. In the teaching assistant position, they helped many inexperienced students with homework assignments, and were also responsible for grading those assignments. Having passed the gateway for the major and held a TA position in an introductory course, these experts have experience with identifying lines of code that are not necessary to satisfy the goal of a programming assignment and subsequently giving feedback on those lines.

	Lines	Files	LPF
Raw Values			
P1	8435	465	18.14
P2	6305	466	13.53
Expert A	478	295	1.62
P1	138	96	1.43
P2	347	199	1.73
Expert B	1276	466	2.73
P1	906	263	3.44
P2	370	203	1.82
Expert C	774	369	2.10
P1	575	261	2.20
P2	199	107	1.86
Expert D	1092	423	2.58
P1	722	211	3.42
P2	370	212	1.75
My Labels	893	423	2.11
P1	481	238	2.02
P2	412	223	1.85

Table 5.1: This shows the number of lines and files labeled by each of the five experts. LPF stands for Lines Per File. At the top, there is the overall amount of lines and files in the entire data set for comparison.

Each set of labels produced by an expert will be discussed in the following paragraphs. Table 5.1 shows the breakdown of the average number of lines labeled per problem, per expert, and the average number of lines labeled per problem, per

solution, for each set of expert labels. These numbers offer key insights into how each person handled the labeling process. I can compare these numbers to each of the other expert labels, and my own labels, to determine how well each expert created their labels. I should note here that this table can be interpreted to mean that each expert, including myself, did not label every single file in each problem. This is not the case, as each expert read through all of the programs in both data sets. The file numbers being reported per expert, per problem, include only the files that the expert labeled with at least one extraneous line. Lines that contain an error but were not not labeled by an expert may exist in the other files. The omission of these files in Table 5.1 does not mean that they were not included in my analysis later on in this chapter.

In the following discussion, I validate some labels while also offering criticism of other labels. This serves to offer insight into the difficulty and the subjectivity of the labeling process. Each table of expert labels in the following analysis highlights the labels that I believe are the closest in description to labels that I believe should be detectable by my system. All labels produced by the experts are used in the analysis of the experimental results; the criticism of a label does not equate to a removal of that label from the set in order to bolster or diminish the experimental results.

Letter	P1	P2	Description
A	73	114	unnecessary conversion of input to string
B	**1**	**3**	**statement has no effect**
C	**19**	**0**	**conditional that has no effect**

D	1	0	unnecessary print inside of input
E	**18**	**0**	**statement never reached**
F	10	10	unnecessary conversion of a variable type
G	**1**	**0**	**unnecessary variable**
H	2	0	while loop should've been if statement
I	13	0	duplicate code
Y	0	202	unnecessary start and/or step for range function
Z	0	18	loop that only executes once

Table 5.2: Set of labels produced by Expert A.

Expert A produced the set of eleven different labels described in Table 5.2. Of these labels, I believe that four match up well with my definition of extraneous line of code. This expert describes the lines she found very concisely. She does describe several different cases of statements that don't have any effect, which is exactly the kind of line that I intended for the experts to find. However, this expert labeled the fewest lines per file (LPF) overall. This expert labeled very few files in problem one when compared to each of the other experts, leading to this overall decline in LPF. Accounting for both problems, the type of line that was most identified by this expert concerned an unnecessary start/stop with Python's range() function. This is a common mistake made by novice Python programmers, because there are multiple default behaviors of this function. Some of the lines identified by expert A might not actually be extraneous, such as those with label H or Z. The descriptions of these labels identify a programming construct that was used incorrectly, but that does not mean that the usage of that construct at that particular point was entirely unnecessary to solve the problem.

Expert B produced the set of twenty different labels described in Table 5.3. Half of the labels produced by this expert I believe are well-aligned with my definition

of extraneous and should be easily identified by the system. This expert clearly put a substantial amount of effort into the labeling process. They identified the most types of extraneous lines in the data, and they had the highest LPF overall. They were more relaxed in what they understood to be extraneous, as their most identified extraneous line type is simply "extraneous syntax." This label included lines that had unnecessary parentheses, semicolons, and a few other pieces of "syntactic sugar" that Python understands. At the same time, they were very specific with a selection of their labels. The labels generated by this expert were usually more informative than the other three students, sometimes including information related to a specific problem description. Such labels, however, are not general and will reduce the alignment of this expert with the others. The most interesting thing about this expert's labeling is the choice of granularity. They lumped together all extraneous syntax into one category, but split up the same general range() function mistakes into several different categories.

Letter	P1	P2	Description
A	24	0	Constant not used
B	19	0	Extraneous conditional where one or more conditionals have no affect in execution
C	117	122	Extraneous type cast
D	23	0	Extraneous line duplication
E	86	26	Unnecessary line because it is extra information that does not achieve the program's core functionality requirement
F	3	0	Self Assignment
G	453	44	Extraneous syntax
H	1	0	Unnecessary return
I	159	1	No conditional required or should be default case (i.e. elif is not required if it is the base case)

J	1	0	This line assumes that water is frozen if the temperature is less than K_BOIL, which is not true. Water is frozen if below 273
L	3	20	This operation should not be required
M	6	18	Unnecessary declaration
N	5	0	Unnecessary operation exit() because using elif will ignore following cases if a previous if statement is true
O	6	0	This value is always overwritten in the following control flow, and is therefore always defined when it is actually used
P	0	1	Range defaults from 0 to n and the 0 is not required
Q	0	13	Unnecessary for loop
R	0	2	Counter not used
S	0	11	The default step of range is 1
T	0	108	The default start of range is 0
U	0	4	This function call / declaration does not accomplish anything

Table 5.3: Set of labels produced by Expert B.

Letter	P1	P2	Description
A	71	37	unnecessary print
B	18	44	unnecessary initialization
C	66	0	error-handling that does not contribute to solution
D	18	6	unnecessary conditional
E	324	1	an if/elif should be an elif/else
F	71	111	str() used on input()
G	1	0	unnecessary input
I	6	0	exit() not necessary

Table 5.4: Set of labels created by Expert C

Expert C produced the set of eight different labels described in Table 5.4. Of these labels, I believe that six match up very well with my definition of extraneous line of code. This expert understood my definition of extraneous line of code without

question, as this ratio is the highest among all of the experts and the difference between their LPF and mine is only .01. However, there is still a point of criticism to be made. The type of line that they identified the most in the data has to due with a mistakenly used programming construct. The extraneous line marked here is a case of incorrectly applied logic, where the program author is performing an unnecessary test of a condition that could be accomplished with less code. This is a very specific extraneous line that my system may not be able to identify, as the construct is likely still necessary at that point in the solution even though there is a better way to write it that does not require that extra test. The next most identified line by this expert describe an unnecessary type cast on an input statement, similar to each of the other students. Based on their generated labels, I believe that this student had the most accurate internal definition of what it means for a line of code to be extraneous.

Letter	P1	P2	Description
A	2	0	opportunity for compound assignment
B	116	124	unnecessary casting
C	13	0	semicolon to end line
D	2	0	unnecessary return
E	2	0	self assignment
F	560	208	unnecessary parenthesis
G	17	6	added to assignment
H	4	0	Debug statement
I	1	0	condition always true or always false
J	2	7	unused variable
K	2	5	renamed variable
L	1	0	print in input
M	0	6	Always loops once
N	0	3	performs no function
O	0	7	unnecessary initialization
P	0	4	unnecessary parameters

Table 5.5: Set of labels created by Expert D

Expert D produced the set of sixteen labels described in Table 5.5. Slightly over half of these labels I believe match up well with my definition of extraneous line. Their LPF is only slightly higher than mine, so I believe that this student did understand the general idea. The majority (70%) of the lines labeled by expert D were described as having "unnecessary parentheses." Again, the majority of lines marked by this student are extraneous due to a syntax choice rather than an extraneous step in the algorithm. Some of the descriptions provided by this expert are not as informative as they could be, such as "added to assignment" or "performs no function." Upon further inspection, lines with labels such as these are either just print statements that aren't directly related to the goal of the problem, or some other extraneous statement. My system may not be able to catch all of the lines labeled as such due to the large disparity in the description. However, my system should be able to pick up lines aptly labeled "unused variable" or "renamed variable," as these may be valid extraneous steps in the algorithm for a particular problem. This expert made a concerted effort to be granular in their labels, but ended up with a few line types that probably belonged in the same category.

Finally, I produced a set of labels for each of the two assignments in the data set. This baseline was made after all of the experts had created their labels, but before I performed any analysis of what they submitted. I made a few observations during the labeling process that are useful for understanding the general structure

Letter	P1	P2	Description
A	24	8	variable declared but not used
B	113	127	unnecessary type cast
C	143	0	not necessary to solve problem as described
D	2	1	self assignment
E	2	0	unreachable line of code
G	99	0	else followed immediately by if
H	76	26	extra print statement
I	4	0	same condition tested twice
J	7	1	unnecessary function call
K	9	22	separate variable initialization
L	1	0	string operation not saved
M	1	4	variable reassigned but not altered
N	0	192	range defaults not used properly
O	0	16	loop executes once
P	0	6	counting variable in for loop
Q	0	7	statement has no effect
R	0	2	unneeded function definition

Table 5.6: Labels created by Student X

and trends lying within the novice code that I used.

In the first problem, I found many first-time programmer mistakes. Several students separated the declaration and the initialization steps of creating a variable in Python, which can normally be accomplished in one line. Some students would create a variable with a relevant name, as if they wanted to use that data purposefully, but then they did not use that variable anywhere else in the program. One of the most prevalent novice issues was the redundant structure of conditional statements; a large number of students were writing an if statement immediately after an else clause, which is a sequence of statements that could easily be combined if the student consolidated the logic. There were a small number of cases, although enough to warrant a mention, of students operating on string variables but not in a way that saved the result of their operation. Strings in Python are immutable;

therefore, if you want to perform an operation you must save the result in a variable with the assignment operator. This is important when a student attempts to force a string to uppercase, or applies any general operation that changes some or all parts of a string. There were students who did not correctly use the input function in their code: instead of passing the prompt string into that function, they prompted the user in a separate print statement. A substantial fraction of the students who solved this problem included some method of error-checking, or some other code that was semi-related to the problem but not required for their solution to be correct. By the definition of extraneous code that I put forth, these lines should be marked as extraneous, since they are not directly related to outputting the correct solution for the problem. One of the most common types of extraneous lines involved the casting of data from one type to another. More often than not, the student was casting the result of an input statement to a string; this is redundant, since the return value of the input function is already a string. A handful of students were performing mathematical operations inside of the float casting function. This is also redundant as Python will handle the type conversion for you when you write an arithmetic expressions.

I found the second problem to be remarkably more difficult to label accurately. While there were still many first-time programmer mistakes, there were several more nuanced issues that ended up being marked as extraneous. One such issue is that some students wrote a loop that only executed one time, but sometimes the same student would repeatedly write a loop that executed exactly once. This is clearly extraneous, since writing that whole construct was unnecessary: the student could

remove it and the code would work the same. Many students were simply not using the power of iteration to their advantage, and ended up with an extra variable initialization beyond what the language provides syntactically (label K). Another nuanced issue was a misuse of the range function, which generates a sequence of integers based on a user-defined start, stop, and incremental step. There are default values for the start and the step components of this function, but several students did not take advantage of those values. Instead, they manually entered one or more of those same default values as arguments to the function. This is definitely extraneous, since the behavior would have been the same even if those arguments were omitted. One could argue that this was a stylistic choice taught to them: it is better to be explicit than implicit. For this problem, a few students wrote lines of code that I believe were deserving of being marked as multiple types of extraneous lines. However, this is not possible with the current version of the web based labeling application, so I chose the extraneous line type that I thought was the most relevant.

5.2.2.1 Inter-rater reliability

In this section, I explore the reliability of the expert generated labels. I compute Fleiss' kappa statistic across various permutations of my label dataset. Then, I interpret the results of the kappa calculation for each problem. I intend to show that the experts were in some form of agreement in the context of the experiments that were run, but these results also support the claim that labeling this data is difficult. I then offer qualitative explanation of the difference in expert labels in

addition to the statistical analysis.

Number of Raters	P1	P2
4 raters	0.294	0.449
5 (4 raters + me)	0.297	0.504

Table 5.7: Fleiss' kappa values for Problem 1 and Problem 2 computed with and without my set of labels.

Fleiss' kappa statistic [15] measures the agreement between more than two raters assigning categorical ratings to a set of items after factoring out any agreement that happens by chance. In the context of this thesis, Fleiss' kappa will best capture how well the experts agreed on the lines that were extraneous and how well they agreed on lines that were not extraneous. I compute the kappa statistics for two permutations of the label data that I have. First, I use the set of labels for the every line in every file. Then, I use the set of labels for every line in every file where at least one expert labeled that line as extraneous. I want to look at the difference in these permutations because there is an overwhelming number of non-extraneous lines in the data by the nature of each problem. I suspect that any random rater would be able to accurately mark a line as not extraneous. Therefore, I need to isolate the lines that were considered extraneous by at least one expert. I care more about the experts agreeing on the lines at least one of them believes to be extraneous, than on the entire set of lines. I expect the kappa statistic computed in the second permutation to be lower than the first permutation, but the magnitude of the decrease is the reason why I am making the distinction. How well do the experts agree on the set of all lines? How well do they agree on just the lines that at least one of them thought to be extraneous? Are the experts just really good

at finding the lines that are not extraneous, or is there also a decent amount of agreement on the extraneous lines in the dataset? The kappa statistics and their associated p-values are shown in Table 5.7, and are analyzed further according to the table of values presented by Landis and Koch [16].

When I consider the reliability of the labels on every possible line of code for each problem, the inter-rater reliability is decent. For the first problem, the kappa value indicates what Landis and Koch consider to be a "fair" amount of agreement among the experts. For the second problem, the kappa value indicates a "good" amount of agreement by Landis and Koch's standards. It is clear that the experts agreed more on the extraneous lines they found in problem two than they did in problem one. It is plausible that an order effect is present here, as problem one was the first set of student code the experts labeled, followed by the code for problem two. The experts had no practice before labeling problem one, so it makes sense that they there is more variance in the labels for that problem as compared to problem two. I expect that by the second problem, the experts had a better understanding of what to look for when labeling the lines. The p-values of zero for each of the permutations supports the reliability measurement being statistically significant, meaning that it is highly likely my experts did not have this level of agreement by random chance.

The kappa statistic, computed via the isolation of every line that was labeled as extraneous at least once, signifies no agreement among the experts for the first problem. It also signifies more agreement among the experts in the second problem when compared to the first; however, that agreement is poor according to Landis

and Koch. This result is unfortunate; however, it supports my exposition in the previous section on the difficulty and subjectivity of the labeling process. If each of the experts was unable to produce a set of labels that perfectly matched the definition that was explained to them, then it is understandable that there exists little to no agreement between the experts when you consider only the lines that were labeled as extraneous. The p-values support the idea that this amount of disagreement is highly unlikely to be due to random chance.

Through analysis of the actual labels produced by the experts, and the kappa statistics computed above, I can answer the questions posed earlier in this section. First, the agreement among experts suffers when you remove the lines of code that all experts believed to be non-extraneous. Again, this is not a mathematical surprise,as one would expect the reliability to drop if you remove all of the lines that all experts agreed upon; however, this can be interpreted to mean that my experts were very good at finding lines of code that were *not* extraneous. Each expert clearly had a different internal definition for code that doesn't make sense in the context of a problem. Some experts took a severe approach by including any line of code with inconsequential syntax as extraneous. For example, both sets of labels created by expert B and expert D are dominated by lines of code with unnecessary syntax. Each of the four experts identified some form of unnecessary casting of variable type. There are also instances of experts having a relaxed understanding of extraneous lines of code. One of the most important things to understand about this labeling process is the distinction between lines of code that do not help the student's solution as it is written, and the lines of code that are actually incorrect in the same context.

Table 5.9 shows the breakdown of how those lines in problem 1 were labeled by the experts. Table 5.16 shows the breakdown of how those lines were labeled by the experts in problem 2.

A noticeable trend in the expert labels is the identification of unnecessary syntax as an extraneous line of code. From the standpoint of the expert, it is clear that lines with extra characters that do not contribute to the overall solution to the problem should fall within the definition of extraneous line of code. The experts made a reasonable choice to include such lines in their choice of labels, as one would expect an ITS that reports extraneous lines to include unnecessary syntax in that report. Static code checking tools such as Pylint [17] already exist to solve this exact problem of unnecessary syntax; therefore, my system was not designed to report such lines. Instead, I am focused on logic and data flow in the novice source code. Occasionally, my system will pick up on extraneous syntax, as it might have an effect on the logic or data flow. The inclusion of expert labels that report extraneous syntax in the experimentation is important in order to preserve the integrity of the expert labels; I cannot alter their labels without introducing my own bias. It would be useful in a future work to compare the set of labels with extraneous syntax and the set of labels without extraneous syntax to get a more accurate picture of how my system does, or to instruct the experts before they begin that they are to ignore syntax.

Apart from the identification of unnecessary syntax, the experts found a similar set of lines that align with the definition of extraneous code they were provided. Each expert marked lines that contained an unnecessary usage of variable assign-

ment/initialization variables or constructs like “return” and “print.” A majority of the labels generated by the experts are related to an extraneous use of information like self assignment, or a misuse of logic in the novice source like poor error handling. Some experts even identified lines of code necessary to report errors in the input to the program, even though that was not a requirement of either assignment.

Some lines end up being marked extraneous by an expert or myself as a means of describing a larger, more fundamental, confusion on the usage of a programming construct rather than a single extraneous line. For example, many first-time programmers will confuse small differences between “for” loops and “while” loops. One very common mistake is thinking that the control variable in a Python for loop must be initialized on the line immediately before the for loop. Although Python will allow this, it is not necessary, as the “for” loop handles the creation of that variable internally. This is a failed attempt at pattern matching by the student. They would have learned that the “while” loop in Python requires that any variable used in the condition must exist before entering that loop, and they are attempting to apply the same idea to a new construct that performs the same task, repeating a sequence of statements a certain numbers of times, but behaves differently. My label K in Problem 2 was used to describe this larger confusion by marking the separate initialization of the “for” loop control variable.

It is interesting to note that while not every expert labeled more lines overall in the first problem, every expert used more of their labels to cover the first problem than the second. They used a larger spread of unique labels in the first problem, and a tighter spread in the second. For example, expert A has eleven distinct labels

that they created, using nine of them to describe lines in the first problem and four of them to describe lines in the second. This trend is true for every expert except for me: I ended up using an equal number of labels for each problem. This is an intriguing trend to discuss, because it sheds light on how each expert created their labels. This trend leads me to believe that each expert started by labeling the first problem, creating a set of labels that aligned with the extraneous lines found in that problem. When they finished that problem and moved on to the next one, which is a fundamentally different problem, they found that a majority of the labels that they had created did not fit this new problem at all. They may have also, like me, found problem two to be more difficult to label and overall not have as much room for students to place extraneous lines. So, they would either have had to generate a new label to describe an extraneous line, or figure out which bucket a line fit into best. Thus, their spread of extraneous line types for the second problem was lower than the first problem.

5.3 Experiments

I performed several experiments to test how well my system performs against the set of expert-produced labels. The main goal of these experiments is to understand whether my system is capable of accurately detecting extraneous lines of code. After testing against the experts, I should be able to answer a few questions: how well is my system able to identify lines that the experts say are extraneous? How well is my system able to identify lines that the experts say are not extraneous?

Which combination of line dependency types is the most effective at getting accurate results, if there is a systematic variation? Does changing the combination of line dependency type used have any significant effect on the outcome of the experiment? I hypothesize that the combination of dependencies that uses all three types will be the most accurate of the permutations that I try. I believe this will be the case because of the finer detail afforded by the usage of more dependency types. This should prevent the over-identification of lines that are not actually extraneous, and help to ensure that lines that are actually extraneous are still flagged as such.

In order to answer some of these questions, I need to be able to interchange the types of line dependencies that are being used between experiments. Therefore, the ability to turn different types of dependencies on or off is built into my detection algorithm. A dependency type being "turned on" means that the detection algorithm is actively using that type of dependency to inform its detection of lines in source code. Each experiment reflects one possible combination of line dependency types that are turned on. Different permutations of line dependencies used by the system should help to visualize the impact that different dependency types have on detecting extraneous lines of code. For each problem in the data, I will only analyze seven of the eight possible combinations of line dependencies. The case where all three dependency types are turned off in the detection algorithm will be ignored, as that is equivalent to not running the detection method at all. The rest of the line dependency combinations will be tested so that I have as much information at my disposal to understand how my system performed. I will be able to see how different line dependency combinations change the ability of the system to identify

extraneous lines correctly. I intend to show how different measurements are affected in the progression that starts from the system using a single line dependency type, then adds one more type, and finally adds the last type.

In order to test the ability of my system to accurately identify extraneous lines, I will compare what it views as extraneous to what the experts view as extraneous. The experts labeled a significant number of individual lines, so in an effort to not crowd the analysis, I will focus on three versions of the labels that I have: the set of all unique lines that were labeled, the set of lines that were labeled by a majority, and the set of lines that were labeled only by myself. The set of all unique lines that were labeled is the set of lines that were labeled by any expert any number of times, so a line in this set could have been labeled by a single expert, or it could have been labeled by four different experts. The set of lines labeled by a majority of experts is smaller than the previous set, as it contains lines that were labeled by three or more experts total. I believe that a line of code that has been labeled by the experts in the same manner is more likely to actually be what the experts claim it is; therefore, it is more likely that my system will pick up on that same line. I also believe that because I have more intimate knowledge of the system, I should test my system against only the labels that I generated in order to establish how well the other experts did at labeling the data.

My extraneous line finding system is contained inside of a single class. The instantiation of this class sets up what is necessary to run an experiment with one single combination of line dependency types on a single file. Experimentation with this system therefore involves creating an object of this class with the relevant

information, running the detection algorithm on this object, and reporting what is found in the form of a log file. This is done for each assignment in the data. I keep one log file that keeps track of what happens at each step of the algorithm for each assignment that is tested, and I have another solely for reporting the results of each assignment that is experimented with. This allows for ease of creating system performance reports, and for dissecting any individual run for in depth analysis.

Experimentation with my system is limited by the correctness of each target program. My system is entirely dependent on the completeness and correctness of the target program is it analyzing. For each problem, there are specific student solutions that contain errors. These programs are left out of the following analysis. If the program that I run through my system contains any sort of errors, due to incorrect syntax or otherwise, then I cannot detect extraneous lines in that program. This is a consequence of the dependence on matching the goal output. If a program breaks for any reason, there will not be any output lines that mach the goals of that particular problem. I have defined three classes of errors that I detect and subsequently skip in my system: GoalNotFound, TraceError, and SyntaxError. The GoalNotFound error is an umbrella error thrown when the target program fails to provide output that matches the entire goal for that particular problem. The goal is divided into multiple lines, and if it is not the case that every line has been matched, then this error is signaled. The TraceError is signaled when the target program fails for one reason or another during run-time. The leading causes of this error are misinterpretation of the problem instructions by the student or a failure to submit a complete assignment. The SyntaxError is rather self explanatory: it is an error

made by the student responsible for the target program with the syntax of Python. This error prevents the student's work from being executed; therefore, the solution is incomplete.

5.3.1 Measurements

I am interested in the performance of each permutation of the algorithm on each of the data sets, and requiring metrics that can quantify how well the system is able to identify both lines that have been labeled by as expert and lines that have not been identified as extraneous by an expert. The experts were able to either label a line as extraneous (with an accompanying reason) or not label the line, meaning that is was not extraneous. My system will state that each line falls into one of these two categories: either it is extraneous or it is not. This means that my system can be treated as a binary classifier. Therefore, a confusion matrix was computed for each run of the algorithm. This captures the number of correct and incorrect matches between my system and the ground truth labels produced by the experts. The confusion matrix is comprised of four different cases: true positives, false positives, true negatives, and false negatives. Using these four pieces of information, I can calculate how accurate a combination of line dependencies was, how specific and how sensitive the results were, and the F1 score – an encapsulation of how many false positives and false negatives were in the results. I will now define the preceding terminology in the context of this line labeling problem.

I want the system to correctly identify lines that are considered extraneous by

the experts, and to correctly identify lines that are not considered extraneous by the system. Lines labeled by an expert as extraneous are referred to as the positive class, and lines that were not labeled by any expert as extraneous are referred to as the negative class. A true positive (TP) is the case where the system correctly predicts the positive class for a line, or correctly labels a particular line as extraneous. A true negative (TN) is the case where the system correctly predicts the negative class for a line, or correctly labels a line as not extraneous. Both TP and TN results are favorable in the analysis of my system.

The system may also incorrectly predict the positive or negative class for a single line. A false positive (FP) is the case where a positive class was incorrectly predicted, or a line of code that is not extraneous (per the experts) was labeled as extraneous by the system. In other words, the system says the line is not necessary in the context of the problem, when in reality it is necessary. A false negative (FN) is the case where the negative class was incorrectly predicted, or a line that was considered extraneous by an expert was not considered extraneous by the system. FP and FN results are not favorable and a model classifier should minimize the rate of both such results. The misclassification of extraneous lines would cause much confusion if this system were adopted for use with actual students.

Suppose that the end goal of this classifier were to place it in a facility where a programming student would receive feedback on their code in real time by themselves. It would process a student's solution to a problem, and inform that student of the line number(s) in their code that it believes to be extraneous. An FN result in this context is not harmful to the student: it is the equivalent of the student not

using the system in the first place, since without the system, they also would not know about the extraneous lines that they have written. If a student were alerted to a line that was an FP, that student would be very confused. They would spend an unnecessary amount of time trying to understand why the line of code being highlighted is extraneous, removing that line at times to find out that their solution breaks upon its removal. I believe an FP result is the more dangerous and harmful of the two possibilities, since the purpose of the system is to properly identify extraneous lines of code where they exist.

Now, suppose that this classifier were instead used to supplement traditional instruction. In this situation, there would be trained staff in the room who know how to help with issues related to the assignment or topic that is being worked through at any given time. In this situation, the staff go around and facilitate the interactions between the students and the system. It would be easier for the trained staff to identify a false positive from the system as opposed to a false negative. For someone who is trained to identify such lines, the false positive results (a line being marked extraneous when it should not have been) is much easier to spot, because it is fairly trivial to identify a line of code that is needed in a solution. It is not so easy to be sure of the opposite, if a line that was not labeled should actually have been labeled. The fact that this is not easy is part of the motivation for constructing this system. In any one solution, there will almost always be many more lines that are not extraneous than lines that are extraneous. The program that solves a given problem is comprised of a sequence of steps, an algorithm, where each step introduces more complexity into the program. This is evident in the set of expertly

produced labels shown in Table 5.1, where each expert labeled anywhere between 1-3 lines per file on average, but the solutions for each problem required 18 and 13 lines per file on average, respectively. Therefore, in this scenario, it is the FN result that is more harmful and will waste the time of both the student and instructor when trying to help students fix code that they did not need in a solution.

I have shown that there are situations where both the FN and FP results are harmful to the stakeholders who may use this system. Therefore, I wish to strike a balance in the number and type of these results that appear in the experiments. Ideally, of course, both values would be as close to zero as possible. The measures of sensitivity, specificity, and precision will enable me to be sure that the rates of false positives and false negatives are not too high.

Specificity in a binary classifier is the ratio of true negative results to the total number of negative results in the ground truth labels. In the context of my system, specificity is the ratio of lines that were correctly marked as not extraneous to the total number of lines that were marked as not extraneous by the experts. In other words, specificity determines how well the system was able to avoid false alarms in the data. A high specificity would mean that there was a low rate of false positives. A low specificity means the opposite as the rate of false positives would be higher. Although I argue that the false positive is less important to avoid, this metric should still prove useful when comparing the different permutations of the line dependency types.

Sensitivity in a binary classifier, otherwise known as recall, is the ratio of true positive results to the total number of positive results in the ground truth labels.

In the context of my system, sensitivity is the ratio of lines that were correctly labeled by the system to all lines that are considered extraneous by the experts. In other words, recall is a measurement of how well this classifier can detect the positive instances (expert-labeled extraneous lines) in the data. High sensitivity is equivalent to a low rate of false negatives. A low sensitivity is equivalent to a high rate of false negatives. Permutations of the algorithm used by the system that yield high sensitivity are favorable, as that particular permutation would be very certain of the extraneous lines that it marks as such, only labeling the lines that are actually extraneous. I argue that the false negative is a worse error to see in my classifier than a false positive, so achieving a higher sensitivity will be more important in my analysis than achieving a higher specificity.

In addition to sensitivity and specificity, I will measure the *precision* of each permutation of the algorithm my system uses. Precision measures the ratio of true positive results to the sum of all results labeled as true by the classifier regardless of correctness (true positives + false positives). In the context of my system, this is the ratio of system-labeled extraneous lines that are correct to the total number of system-labeled extraneous lines. This measurement is an answer to a simple question: of all the lines labeled as extraneous by the system, how many were actually extraneous? This is an important measurement to take, because it is a way of making sure that the system is very reliable in the lines it is saying are extraneous.

Not only does the nature of the data warrant the calculation of sensitivity and specificity, it also requires that some distinction be made when computing the overall accuracy of the system labels. Normally, the accuracy of a binary classifier

is computed as the ratio of correct labels to the total number of predictions made. However, I am confident that the data will be imbalanced, containing mostly lines that are not extraneous as seen in the LPF from Table 5.1. Typical accuracy measurement is useless with imbalanced data, particularly when the system is expected to perform very well on the task of identifying the side of the imbalance with more data. Therefore, I need a different method of quantifying overall how well my system performed. Overall, I would prefer to strike a balance of high sensitivity and specificity to cut down on false positives and false negatives, so I will use a balanced accuracy measure and the F1 Score to do so. The balanced accuracy is the arithmetic mean of sensitivity and specificity as shown in Equation 5.1, and the F1 score is the harmonic mean of precision and recall as shown in Equation 5.2.

$$BA = \frac{Sensitivity + Specificity}{2} \tag{5.1}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{5.2}$$

5.3.2 Problem 1

The first problem, as described in section 5.1.1, had 465 total submissions. I was only able to include 75% of those submissions in my experimentation (see Table 5.8) due to errors within these programs. The majority of those not included simply did not submit a correct solution to this problem, which is why there many of the programs show a GoalNotFound error. A significantly smaller number of students

	Total	Error	Without Error
# Files	465	120	345
# Lines	8435	2221	6214

	GoalNotFound	TraceError	SyntaxError
Error Type	90	22	8

Table 5.8: The total number of files that ran with and without errors for Problem 1, including a breakdown of what errors were seen in the files that were unable to be processed by my system.

Table 5.9: Problem 1 Breakdown

Agreement By Line	Number
Single Expert	632
Two Experts	635
Three Experts	148
Four Experts	44
Five Experts	60
Total labeled	1519

submitted code that did not work: these are the solutions in the TraceError or SyntaxError categories.

The experts labeled 1519 unique lines in this problem, of the 8435 total in Table 5.9. This includes the lines in files that would eventually cause an error in the system, such lines are not counted in any of the results further in this section. Table 5.9 serves to show how well the five experts agreed on what lines were extraneous. There were more lines marked by at least two experts than there are lines marked by only one expert. The split is roughly 60/40. This shows that, in general, the experts agreed on the lines that were marked as extraneous. I believe these labels to be accurately produced by the experts, but it is not useful to make the measurements with each set of matched labels. The measurements that I make will be on the set of all unique labels, the set of majority labels (i.e., at least three experts), and just

my set of labels. I will instead provide a breakdown of how many of the labels in each set the system got correct.

		Line Dependency Combination						
Consensus	Labels	E	D	S	D+E	E+S	D+S	All
One	423	60 (14.18%)	401 (95.02%)	115 (27.19%)	60 (14.18%)	54 (12.77%)	66 (15.60%)	54 (12.77%)
Two	505	128 (25.35%)	497 (98.42%)	196 (38.81%)	128 (25.35%)	124 (24.55%)	142 (28.12%)	124 (24.55%)
Three	94	19 (20.21%)	93 (98.94%)	61 (64.89%)	19 (20.21%)	19 (20.21%)	35 (37.23%)	19 (20.21%)
Four	28	4 (14.29%)	27 (96.43%)	27 (96.43%)	4 (14.29%)	3 (10.71%)	9 (32.14%)	3 (10.71%)
Five	46	2 (4.35%)	43 (95.56%)	45 (97.83%)	2 (4.35%)	2 (4.35%)	4 (8.70%)	2 (4.35%)
Majority	168	25 (14.88%)	**163 (97.02%)**	133 (79.17%)	25 (14.88%)	24 (14.29%)	48 (28.57%)	24 (12.29%)
Total	1096	213 (19.43%)	1061 (96.81%)	444 (40.51%)	213 (19.43%)	202 (18.43%)	256 (23.36%)	202 (18.43%)

Table 5.10: Problem 1 true positive rates for each combination of experts.

Table 5.10 shows the rate of true positives among each set of matching labels, from the set of labels that only a single expert thought was extraneous to the set of labels that every expert thought was extraneous. It also includes the set of all unique labels, and the majority set. The number of labels at each level of expert consensus differs from Table 5.9, as this table does not include lines from files that produced one of the three errors described in Section 5.3. There are a few trends of note in this table. When the system was run with only the data dependency turned on, the rate of true positives was consistently very high for every set of labels. In fact, it has the highest true positive rate among most of the permutations of the line dependency types. The data dependency TP rate is exceeded only by that of the structure-only version of the system when compared against the set of all five experts in agreement, and only by a very small margin. In some permutations, the percentage of lines that the system gets right decreases as the set of labels gets narrower. The most interesting observation is that the single dependency type permutations of the system consistently outperformed every multi-line permutation of the system, regardless of the number of experts who agreed on the line. This out performance appears only in the rate of true positives, so in order to make more

sense of this outcome, we will take a look at the overall confusion matrix values and their ratios.

5.3.2.1 System Test: All Unique Labels

Table 5.11: Problem 1 - results with all labels

Results	D	E	S	D+E	E+S	D+S	All
TP	1061	213	444	213	202	256	202
FP	2522	358	1141	358	14	25	14
TN	2531	4760	3959	4760	5104	5093	5104
FN	33	883	648	883	894	840	894
Precision	0.296	0.373	0.280	0.373	0.935	0.911	0.935
Sensitivity	0.970	0.194	0.407	0.194	0.184	0.234	0.184
Specificity	0.501	0.930	0.776	0.930	0.997	0.995	0.997
Accuracy	0.584	0.800	0.711	0.800	0.854	0.861	0.854
Balanced Accuracy	0.735	0.562	0.591	0.562	0.591	0.614	0.591
F1 Score	0.454	0.256	0.332	0.256	0.308	0.372	0.308

The confusion matrix values in Table 5.11 shows the amount of each type of match between the system and all unique labels produced by the experts. Any line that was labeled by any expert is used in this set, regardless of how many experts labeled that particular line. The number of lines is smaller that the amount presented in Tables 5.9 and 5.10 in part because these numbers exclude the files that gave an error (see Table 5.8). The permutations of line dependencies are abbreviated in these tables to save space: D stands for Data, E stands for Execution, S stands for structure, and the '+' between two of these letters means both of those line dependencies were used in that run.

The first trend to note here is the exact equivalence between the E and D+E experiments, as well as the E+S and All experiments. I did not expect any pair of

line dependency types to have perfect equivalence like this. The dependency type D had the highest sensitivity and balanced accuracy values, which seems to align with the observations in Table 5.10. However, it appears that in this configuration, the system simply labels most lines with the positive label, achieving high sensitivity because any system is more sensitive if it always says that the positive class appears more often than not. This gives way to a high rate of false positives, as seen in the low precision score.

I expected my system to perform well in this setup, and it seems to have done exactly that. With a larger number of lines labeled with the positive class, the system has a larger margin for error in its labeling. In general, the addition of more line dependency types caused the system to become more specific and precise. It got more negative classes right, and it got more positive classes right, as more line dependencies were added. It is difficult to claim exactly which combination of line dependencies worked the best, as many combinations did well, and the combinations that did well relative to each other have very small differences if any. The two combinations that performed the best performed exactly the same, and I believe it would be appropriate to call the one with fewer line dependencies the winner in this case, since that required less work to achieve the same outcome.

5.3.2.2 System Test: Majority Labels

Table 5.12 shows the confusion matrix values considering only those unique labels where at least three experts agreed that each line was extraneous. Just as

Table 5.12: Problem 1 - results with majority labels

Results	D	E	S	D+E	E+S	D+S	All
TP	163	25	133	25	24	48	24
FP	3420	546	1452	**546**	192	233	192
TN	2560	5500	4573	5500	5854	5813	5854
FN	4	143	34	143	144	120	144
Precision	0.045	0.044	0.084	0.044	0.111	0.171	0.111
Sensitivity	0.976	0.149	0.796	0.149	0.143	0.286	0.143
Specificity	0.428	0.910	0.759	0.910	0.968	0.961	0.968
Accuracy	0.443	0.889	0.760	0.889	0.946	0.943	0.946
Balanced Accuracy	0.702	0.529	0.778	0.529	0.556	0.624	0.556
F1 Score	0.087	0.068	0.152	0.068	0.125	0.214	0.125

before, there are fewer total lines that were labeled in the set, because these numbers exclude files that resulted in an error, and because these results exclude all of the lines that were labeled by only one or two experts. Recall that in Table 5.9, the majority of total unique lines fall into one of those categories. There are significantly fewer lines for the system to identify as extraneous, which means significantly more lines for the system to identify as not extraneous. The reader should notice perfect matches just as before, except this time between E and E+D, and E+S and All.

The first trend to note here is the extremely poor precision across the board. With the extreme reduction in the number of actual extraneous lines to find, the system is unreliable when it labels a line as extraneous. Curiously, sensitivity starts high, but there is a steep dropoff after one more line dependency type is added. This is likely due to the system labeling more things positively. Since this same trend appears in the previous tests with all unique labels, it is possible that when the system is run with just one line dependency, high recall is solely an artifact of the system essentially guessing. As with the previous tests, specificity improved

with the addition of line dependency types. The system got better at finding the lines that were not extraneous. Notice that accuracy improves slightly as well, this is likely due to the increase in correctly identified negative classes.

Overall, when tested against the set of majority labels my system performed very poorly. Balanced accuracy steadily declined, although the F1 scores slightly improved. This means that sensitivity and specificity steadily grew further apart, with precision and recall remaining low and close together in value. This is unfortunate, as the majority labels are a more accurate representation of the lines that are truly not needed in the solution for Problem 1.

5.3.2.3 System Test: My Labels Only

Table 5.13: Problem 1 - results with only my labels

Results	D	E	S	D+E	E+S	D+S	All
TP	336	190	328	190	188	237	188
FP	3247	381	1257	**381**	28	44	28
TN	2560	5492	4595	5492	5845	5829	5845
FN	4	151	12	151	153	104	153
Precision	0.094	0.333	**0.207**	0.333	0.870	0.843	0.870
Sensitivity	0.988	0.557	0.965	**0.557**	0.551	0.695	0.551
Specificity	0.441	0.935	**0.785**	0.935	0.995	0.993	0.995
Accuracy	0.471	0.914	.795	0.914	0.971	0.976	0.971
Balanced Accuracy	0.715	0.746	0.875	0.746	0.773	0.844	0.773
F1 Score	0.171	0.675	0.341	0.417	0.675	0.762	0.675

I can provide some insight into how the expert labelers did when I run my system against the set of labels that I produced. Table 5.13 shows the confusion matrix data for my system when run against these labels. This test largely confirms my suspicions of bias, that I know too much about how the system functions and

what its capabilities are. I cannot be sure that I produced a set of labels that accurately captures the extraneous lines in the data and not a set of labels that gets a good score when I run my system against it. This is due to the values of these tests being much larger than the values that I got for the majority class label in which I was included. If I was truly labeling all extraneous lines, I would have results closer to the ones for that set of labels.

Some of the same trends are evident here as in the previous two tests. Note that specificity increases with each additional line dependency type. Balanced accuracy was much higher, but does not fluctuate much between permutations. Sensitivity did decrease like the other tests, but it remained fairly high, meaning the the system identified a lot of the actual extraneous lines than the other sets. This makes sense, since there are not only fewer labels in this set but also more labels that should be caught by the system. Notice as well that the E+S and the All permutations are once again exact matches.

5.3.2.4 Analysis

The results from this set of experiments are promising. While the system was unable to get precise results on the set of majority labels, the labels that it produced when compared to just my labels or to the set of all labels were very good. High values for sensitivity in the single line dependency cases across the board must be due to the high rate of false negatives–the system rarely says that a line is not extraneous when single line dependency is used. The increase in specificity as

dependencies are added makes sense, as the system got better at identifying the lines that are not extraneous. This happens regardless of what set of labels is being used as the ground truth. More often than not, the system performed better when more line dependencies were added as opposed to just having a single line dependency.

	Total	Error	Without Error
# Files	466	68	398
# Lines	6305	924	5381

Table 5.14: The total number of files that ran with and without errors for Problem 2, including a breakdown of what errors were seen in the files that were unable to be run through my system.

	GoalNotFound	TraceError	SyntaxError	Other
Error Type	49	9	9	1

Table 5.15: Breakdown of errors in novice code that were unusable in my system.

In particular, the data show that the usage of data dependencies and structural dependencies almost always performs better than just the execution dependency. In other words, it is important for this system to not just run the code and see what happens, but to use the interaction of data between lines and the structural components of a program to its advantage. However, it also shows that using the data dependency provides no extra information to the system if it was already using the execution and structural dependencies. I believe that this makes sense, because lines of code that execute in sequence and the structures that are being used in that code should provide enough coverage to capture the data flow among those same lines, so there won't be line dependencies due to data that aren't also covered by either execution or structure. Essentially, the system under these tests was very good at identifying lines that were not extraneous, but its performance varied

when trying to identify the lines that were extraneous. Sometimes it did well, and sometimes it did very poorly. That depended on the ground truth labels that were being used. Therefore, the actual problem of finding extraneous lines might have too much subjectivity to actually benefit a student that uses this system.

5.3.3 Problem 2

The second problem, as described in Table 5.14, had a total of 466 files submitted for scoring. I was able to include 85% of these lines in the experimentation for this problem, as the other 15% had an error that prevented them from being used, as seen in Table 5.15. Again, the majority of the errors here were due to the program not solving the problem correctly.

Table 5.16: Problem 2 Breakdown

Agreement By Line	Number
Single Expert	303
Two Experts	167
Three Experts	132
Four Experts	40
Five Experts	101
Total labeled	743

The experts labeled 743 unique lines in this problem, of the 6305 total in Table 5.14. Just as in problem 1, this is including the files that had a detected error by the system. The breakdown in Table 5.16 shows how each of the experts agreed on the labeling of these lines. This problem shows the same 60/40 split between lines labeled by at least two experts and lines labeled by a single expert. There are half as many total lines labeled by the experts in this problem compared to the first one. This could mean one of a few things: the problem was more straightforward and

there was less room for a student to write an extraneous line of code; this problem was more complex, which makes extraneous lines harder to detect for the experts; or the students just wrote fewer actual extraneous lines of code, since they had more experience writing programs in this problem than they did in the first problem. I believe this difference is best explained by a combination of the second and third possibilities. These lines may have been labeled a little less accurately than in the first problem due to its complexity, but the 60/40 split shows that there was still more agreement among the experts than not.

Table 5.17: Problem 2 True Positives

		Line Dependency Combination						
Consensus	Labels	E	D	S	D+E	E+S	D+S	All
One	260	10(3.85%)	171 (65.77%)	129 (49.62%)	8 (3.08%)	10 (3.85%)	34 (13.18%)	8 (3.08%)
Two	132	17(12.88%)	112 (84.85%)	64 (50.00%)	9 (6.82%)	17 (12.88%)	25 (19.08%)	9 (6.82%)
Three	107	5 (4.67%)	104 (97.20%)	42 (39.25%)	5 (4.67%)	5 (4.67%)	18 (16.82%)	5 (4.67%)
Four	33	14 (42.42%)	15 (45.45%)	25 (75.76%)	1 (3.03%)	14 (42.42%)	7 (22.58%)	1 (3.03%)
Five	85	83 (97.65%)	6 (7.06%)	83 (97.65%)	0	83 (97.65%)	4 (4.71%)	0
Majority	225	102 (45.33%)	125 (55.56%)	150 (66.67%)	6 (2.67%)	102 (45.33%)	29 (13%)	6 (2.67%)
Total	617	129 (20.91%)	408 (66.13%)	343 (55.95%)	23 (3.73%)	129 (20.91%)	88 (14.38%)	23 (3.73%)

Table 5.17 shows the rate of true positives among each set of matching labels. It should be immediately obvious that my system did not perform well at all on this dataset. As we determined in the previous problem, the high true positive hit rate in the single dependency cases is an artifact of the system labeling lines as extraneous more often than due to a lack of information. The drop-off whenever E is added to the mix signals that the execution dependency did not perform well with this particular problem.

5.3.3.1 System Test: All Unique Labels

In this test, we see a steady decrease in the number of true positive and false positive results as we add more line dependencies. There is a shift as the system

Results	D	E	S	D+E	E+S	D+S	All
TP	125	102	120	6	102	29	6
FP	1879	1042	2454	**343**	802	335	103
TN	3266	4114	2680	4813	4354	4791	5053
FN	100	123	75	219	123	194	219
Precision	0.062	0.089	**0.058**	0.017	0.113	0.080	0.055
Sensitivity	0.556	0.453	0.667	**0.027**	0.453	0.130	0.027
Specificity	0.635	0.798	**0.522**	0.933	0.844	0.935	0.980
Accuracy	0.631	0.783	0.528	0.896	0.828	0.901	0.940
Balanced Accuracy	0.595	0.626	0.594	0.480	0.649	0.532	0.503
F1 Score	0.112	0.149	0.106	0.021	0.181	0.099	0.036

Table 5.18: Problem 2 - results with majority labels

starts to label more lines as not extraneous as more detail is added to the permutations. True negative and false negative results increase when a second dependency is added, but remains about the same afterwards. Precision is low for each dependency, and sensitivity is typically ever lower. Specificity increases, as expected with the shift to more negative classes being predicted. The abysmal F scores across the board and the balanced accuracy hovering around 0.5 means that the system likely would not do better than a random coinflip classifier. Note also that no full column matched as occurred in problem one, showing that what makes a line extraneous is largely problem-dependent. Overall, for this problem my system did not perform well on the easiest set of labels. It performs worse for the more constricted sets of labels.

5.3.3.2 System Test: Majority Labels

The system performed worse on this set of majority labels, which is expected since the set is smaller than all unique lines. Precision is very low, meaning that

Results	D	E	S	D+E	E+S	D+S	All
TP	408	129	343	23	129	88	23
FP	1596	1015	2261	**326**	775	276	86
TN	3159	3751	2487	4440	3991	4463	4680
FN	207	486	268	592	486	522	592
Precision	0.204	0.113	**0.132**	0.066	0.143	0.242	0.211
Sensitivity	0.663	0.210	0.561	**0.037**	0.210	0.144	0.037
Specificity	0.664	0.787	**0.524**	0.932	0.837	0.942	0.982
Accuracy	0.664	0.721	0.528	0.829	0.766	0.851	0.874
Balanced Accuracy	0.664	0.498	0.543	0.484	0.524	0.543	0.510
F1 Score	0.312	0.147	0.213	0.048	0.170	0.181	0.064

Table 5.19: Problem 2 - results with all labels

the system is not accurate at all in the positive class labels that it predicts (if it says something it is extraneous, it is usually incorrect). We see the same shift from predicting positive usually to predicting negative usually. This causes sensitivity to decrease, while specificity increases. Low F score and balanced accuracy again show that the system does not perform well on this problem.

5.3.3.3 System Test: My Labels Only

Results	D	E	S	D+E	E+S	D+S	All
TP	220	108	212	12	108	59	12
FP	1784	1036	2392	**337**	796	305	97
TN	3250	4009	2635	4708	4249	4713	4948
FN	116	228	120	324	228	272	324
Precision	0.110	0.094	**0.081**	0.034	0.119	0.162	0.110
Sensitivity	0.655	0.321	0.639	**0.036**	0.321	0.178	0.036
Specificity	0.646	0.795	**0.524**	0.933	0.842	0.939	0.981
Accuracy	0.646	0.765	0.531	0.877	0.810	0.892	0.922
Balanced Accuracy	0.650	0.558	0.581	0.484	0.582	0.559	0.508
F1 Score	0.188	0.146	0.144	0.035	0.174	0.170	0.054

Table 5.20: Problem 2 - results with only my labels

The first thing to note here is that the precision, although still low, is higher

than the precision in the majority labels for the most part. This is expected but still not enough to warrant saying that there is hope for this problem. Again, sensitivity starts high with the single dependency types but sharply declines with more. Specificity increases as more negative classes are predicted. F1 and balanced accuracy are higher at first, but plateau with the addition of more dependencies.

5.3.3.4 Analysis

The system did not perform anywhere near as well as it on the first problem. The best explanation that I can give for this has to do with the fact that there is a subjective component to labeling a line as extraneous. What might be extraneous to one expert is not extraneous to another, even if they were armed with the same definition as we discussed in section 5.2.2.1. Precision was low overall, but the best results were still seen for the set of all unique lines. This makes sense, as with more ground truth extraneous lines, the system should do better even if it was just guessing. Except for the majority labels, we saw the permutation D+S perform better than E, so again there is a need for finding the data flow and structure to inform the detection system, since that does better than just running the code.

Chapter 6: Conclusion

The primary conclusion of my work is that the identification of extraneous lines of code in the source code produced by a novice student is incredibly complex. For the first problem, I showed that the hypothesis 'more dependency types will yield more accurate identifications' is not supported in all cases. There was a combination of line dependencies that balanced precision and recall well, showing that this system has promise, but there was no clear trend showing that more line dependencies increased accuracy. There were situations where adding an additional line dependency did not improve the detection at all, showing that sometimes not even every line dependency is needed. However, due to the underlying difference in problem being solved, we saw that an unnecessary dependency in one problem may not transfer to another. In problem 2, we saw the system completely fail to accomplish our goal of balancing precision and recall. There are many possible explanations for this poor result. It could be caused by an increase in complexity from the first problem to the second. The second problem required more programming constructs than the first, and it was necessary to use a combination of those constructs that were not needed in the first problem. Another potential cause could be due to the design of my system. During the creation of my software for detection, I

primarily tested with programs that were similar in complexity to the first problem. The added complexity of the second problem might be difficult for my system to parse as a result, so additional work on the system is needed. Another possible explanation could be that the difference in problem creates a differences in the type of extraneous lines that occur in each problem. It might be the case that my system is more reliable with certain "easier" types of extraneous lines like print statements that do not include goal-related output, and less reliable with other types. Further inquiry is warranted to understand these failures.

It might be helpful to understand why a line was marked as extraneous by the system so I can see the exact reason for the disparity between the system labels and the experts labels for the second problem. There is only one way to do this in the current version of the system. If you compare the results of using one set of dependencies against the results of using the same set of dependencies with either one dependency type removed or a new one added, then it can be concluded that any differences in lines marked as extraneous must be due to the one difference in dependency type. It would be more helpful if the system could attach the exact dependency that caused each line to be marked as extraneous during its labeling process. If more information is extracted out of an extraneous line, then it should be easier to generate hints that help fix the issue. This leaves open some interesting future work.

The identification of extraneous lines in novice source code is an important, yet highly complex, problem. The analysis by a group expert labelers shows that extraneous lines of code certainly exist in both of the data sets. Although the number

of experts who agreed on individual lines was not as strong as I hoped, this work this highlights the fact that extraneous lines are a problem in novice source code. This work can be expanded on in the future by performing a more robust study on the presence of extraneous lines of code in novice source, with the main change being an increase in the number of expert line labelers. The more expert labels that can be generated by more than just four people, the better my set of ground truth labels can be, which would hopefully improve my ability to measure the accuracy of my system – I would be able to more easily determine what lines the experts actually agree to be extraneous. Through my line dependency method I showed that the actual process of identification is not an impossible accomplishment. However, it is very difficult to achieve accurate results and more work is needed to refine the system in its current state. Future work may focus on approaches that do not include line dependencies, such as exploring solution trees derived from past assignments. Once improved to a point of high precision and recall on a high variety of extraneous lines of code, this work can be used to inform the hint generation component of an ITS.

Bibliography

[1] Amjad Altadmri and Neil C.C. Brown. 37 Million Compilations. Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE, pages 522–527, 2015. 3

[2] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. Identifying student misconceptions of programming. Proceedings of the 41st ACM Technical Symposium on Computer Science Education - SIGCSE, pages 107–111, 2010. 3

[3] Tony Jenkins. On the Difficulty of Learning to Program. In 3rd Annual LTSN-ICS Conference, pages 53–58, 2002. 6

[4] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. ACM SIGCSE Bulletin, 37(3):14–18, 2005. 6

[5] Kelly Rivers, Erik Harpstead, and Ken Koedinger. Learning Curve Analysis for Programming: Which Concepts do Students Struggle With? Proceedings of the 12th International Computing Education Research Conference, pages 143–151, 2016. 6, 16

[6] Nelishia Pillay. Developing Intelligent Programming Tutors for Novice Programmers. inroads – The SIGCSE Bulletin, 78(2), 2003. 7

[7] Dinesha Weragama and Jim Reye. Analysing student programs in the PHP intelligent tutoring system. International Journal of Artificial Intelligence in Education, 24(2):162–188, Jun 2014. 7

[8] Amruth N. Kumar. An Evaluation of Self-explanation in a Programming Tutor. LNCS, 8474:248–253, 2014. 8

[9] C.J. Butz, S. Hua, and R. B. Maguire. Bits: A Bayesian Intelligent Tutoring System for Computer Programming. Western Canadian Conference on Computer Education, pages 179–186, 2004. 9

[10] K-M. Chang, J Beck, J Mostow, and a Corbett. A Bayes Net toolkit for student modelling in intelligent tutoring systems. Proceedings of the 8th International Conference on Intelligent Tutoring Systems (ITS 2006), pages 104–113, 2006. 9

[11] Jeremiah Folsom-Kovarik, Gita Sukthankar, and Sae Schatz. Tractable POMDP Representations for Intelligent Tutoring Systems. ACM Transactions on Intelligent Systems and Technology, 4(2):1–22, 2013. 9

[12] John Stamper, Michael Eagle, Tiffany Barnes, and Marvin Croy. Experimental Evaluation of Automatic Hint Generation for a Logic Tutor. International Journal of Artificial Intelligence in Education, 22(1-2):3–17, 2013. 14

[13] Timotej Lazar and Ivan Bratko. Data-driven program synthesis for hint generation in programming tutors. In Lecture Notes in Computer Science, volume 8474 LNCS, pages 306–311, 2014. 16

[14] Kenneth R. Koedinger, Emma Brunskill, Ryan S.J.d. Baker, Elizabeth A. Mclaughlin, and John Stamper. New potentials for data-driven inteligent tutoring system development and optimization. AI Magazine, 34(3):27–41, 2013. 16

[15] Joseph L. Fleiss. Measuring nominal scale agreement among many raters. Psychological Bulletin, 1971. 47

[16] J. Richard Landis and Gary G. Koch. The Measurement of Observer Agreement for Categorical Data. Biometrics, 1977. 48

[17] Python Code Quality Authority (PCQA). Pylint. cmt-thttps://www.pylint.org/. 50