Title

Using Case-Based Reasoning to Improve the Quality of Feedback Provided by Automated Assessment Systems for Programming Exercises

Author

Kyrilov, Angelo

Publication Date

2017

UNIVERSITY OF CALIFORNIA, MERCED

Using Case-Based Reasoning to Improve the Quality of Feedback Provided by Automated Assessment Systems for Programming Exercises

A dissertation submitted in partial satisfaction of the requirements

for the degree of Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Angelo Kyrilov

Committee in charge:

Professor David Noelle, Chair
Professor Stephanie August
Professor Marcelo Kallmann
Professor Shawn Newsam

2017

The dissertation of Angelo Kirilov Kyrilov is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Stephanie August

Marcelo Kallmann

Shawn Newsam

David C. Noelle

Chair

University of California, Merced

2017

Abstract

Information technology is now ubiquitous in higher education institutions worldwide. More than 85% of American universities use e-learning systems to supplement traditional classroom activities. An obvious benefit of these online tools is their ability to automatically grade exercises submitted by students and provide immediate feedback. Most of these systems, however, provide binary (correct/incorrect) feedback to students.

While some educators find such feedback is useful, we have found that binary instant feedback causes plagiarism and disengagement from the exercises as some students may need additional guidance in order to successfully overcome obstacles to understanding.

In an effort to address the shortcomings of binary feedback, we designed a Case-Based Reasoning (CBR) framework for generating detailed feedback on programming exercises by reusing existing knowledge provided by human instructors. A crucial component of the CBR framework is the ability to recognize *incorrectness similarity* between programs. Two programs are considered to be similarly incorrect, if they contain similar bugs, which ensures that corrective feedback generated for one program, is equally appropriate for the other.

We investigated several approaches for computing incorrectness similarity, including static analysis of source code, execution traces of running programs, and comparing outputs from test cases. We found that, given the kind of errors committed by our students, the dynamic approach of comparing outputs from test cases proved to be the most accurate method of computing incorrectness similarity.

We built an e-learning system, called Compass, on top of the CBR platform that we developed. Compass was deployed in a live classroom environment at the University of California, Merced, in the Spring 2017 semester. We compared data collected from this class to data from previous instances of the course, where students were completing the same exercises but received binary instant feedback.

We found that the introduction of Compass, and the detailed feedback it is able to generate on programming exercises, led to a statistically significant decrease in plagiarism and disengagement rates. In addition, we found that students were able to complete exercises faster, with fewer errors. All these factors are associated with improved student learning.

Another significant aspect of Compass is that it scales well to large class sizes. This is because the number of different mistakes made by students is relatively small and the number of students making the same mistake as other students is large. These two conditions enable the CBR engine of Compass to handle a large number of students with minimal instructor intervention.

Work is currently underway to incorporate Compass into other undergraduate courses at the University of California, Merced. As future work, we are planning to investigate the effects of Compass on underrepresented student populations. We have reasons to believe that Compass can provide much needed help to students who may lack confidence to seek such assistance on their own.

To my wife and my son.

Acknowledgments

This dissertation would not have been possible without the help and support I have received from many people. First and foremost, I would like to thank my adviser David Noelle for everything he has done for me throughout my time at UC Merced. In my weekly meetings with David we did not only discuss matters related to my research, but a wide variety of topics that contributed significantly to my academic development. David was always there to help, through all the ups and downs, showing great care and understanding no matter what obstacles I was facing. I will always owe a debt of gratitude to David for everything he has done for me.

I would also like to recognize the other members of my committee: Stephanie August, Marcelo Kallmann, and Shawn Newsam. I want to thank each of them for attending my qualifying exam and my final defense and for reading my dissertation and providing constructive feedback.

I also received invaluable advice from my lab mates at the Computational Cognitive Neuroscience Lab at UC Merced. I would like to thank William St. Clair, Jeff Rodny, Jacob Rafati, Tim Shea, and Narjes Tahaei for all the stimulating discussions. I am also grateful to my lab mates for sitting through practice rounds of conference presentations and posters, and for giving me helpful suggestions.

I would like to thank my parents, Violeta and Kiril, for all the sacrifices they made to help me get where I am today. They have supported me through good times and bad, and I have always been able to count on them. I hope that I have been able to make them proud.

My brother Michael made significant contributions to the development of the Compass e-learning system described in this dissertation. I would like to thank him for all the advice on software engineering best practices, and for simply being there when I was facing technical challenges. The modular and scalable system design, and the fact that the development process went smoothly, is in large part thanks to Michael's advice.

Last but not least, I would like to thank my wife Mayya and my son Kiki for everything they had to endure over the years to help me complete my degree. Their love and support is what got me through my studies at UC Merced. I apologize to them for all the evenings and weekends when I had to work and was not able to spend time with them. As I move onto the next chapter of my career, I hope I can be the husband and father that they deserve.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Challenges in Computer Science Education

In addition to the many benefits of higher education, degree holders enjoy more employment opportunities and greater financial stability. A recent report by the National Center for Education Statistics shows that, on average, the annual salaries of university graduates are $25,000 higher than those of people who never went to college. A study by the Bureau of Labor Statistics looked at employment rates in the United States over the period 1990 - 2012. They report that in 1990, the employment rate was 90% for college graduates, and 75% for people with no university education. In 2012 these figures were 84% and 63%, respectively, showing a disproportionate decrease in the employability of people without university education. This is explained by the fact that much of the economic growth over the last two decades has been driven by industries with high entrance requirements, such as the IT industry, which is projected to grow 22% by 2018. It is also estimated that, over the next decade, universities in the United States will only produce 60% of the Computer Science graduates needed to meet industry demands. In an increasingly hostile global economic climate, the United States can not afford to fall behind in the production of employees for the fastest growing industry. Actions that can be taken in an attempt to resolve this situation include: introducing Computer Science in K-12 curricula, encouraging women and underrepresented groups to participate in Computer Science, and reducing the attrition rates in undergraduate Computer Science, which is the focus of this proposal.

Beaubouef and Mason (2005) suggest that high attrition rates in Computer Science can be attributed to factors such as poor student advising on the part of university administration, inadequate math and problem solving skills of incoming students, insufficient amounts of practice and feedback, and poorly designed laboratory sessions. Since the first issue is administrative in nature and the second one is related to Computer Science in K-12, they are not dealt with here. Instead this proposal aims to address the issue of insufficient amounts of practical work in undergraduate Computer Science courses as well as to offer strategies for designing a better laboratory experience for students.

Freshman Computer Science courses typically have large enrollment numbers, so if instructors wish to include a significant laboratory component for students to practice their programming skills, members of the teaching team will either have to incur the high labor cost associated with grading, or they will have to use automated grading systems. Beaubouef and Mason (2005), as well as many other instructors, have expressed strong opposition to automated grading systems because they believe students deserve to have their code evaluated by a competent human programmer, who can provide meaningful feedback. This is currently a valid concern, since automated grading systems for Computer Science usually produce binary feedback. One of the major contributions of the proposed work will be to address this concern by using Case-based Reasoning to enable automated grading systems for Computer Science to provide high-quality feedback to students, while keeping instructors in the loop, without forcing them to do significant amounts of additional work.

Many undergraduate Computer Science instructors try to provide their students with practical experience through weekly lab sessions, where students are normally expected to complete programming assignments, possibly with the help of teaching assistants. Walker (2004) believes that running labs in such a way is detrimental to students. He argues that lab sessions turn into "teacher-assisted debugging sessions" where learning and development of programming skills is not fostered because students are preoccupied with trying to get the output of their programs to match the expected output. Students will often passively wait for a teaching assistant to offer advice. This process is frustrating for students and teaching assistants alike. A student may need to wait a long time before his/her question gets answered, while teaching assistants may find themselves explaining the same concept numerous times. While waiting for the teaching assistant, some students would attempt to perform an Internet search to find a solution to their problem, while others would resort to randomly changing their code in the hopes of arriving at the correct output. It is clear that good programming and problem solving practices are not followed in the typical Computer Science lab session, which is another problem addressed in this proposal.

A possible reason for the ineffectiveness of typical Computer Science lab sessions is the fact that they follow a *synchronous learning* approach, meaning that learning takes place at a specific time and place, the pace is set by the instructor, and assessment takes place at the same time (and place) for every student. In short, whenever learning happens according to a schedule, the learning is synchronous. There are many disadvantages to synchronous learning, some of which have existed since the dawn of education. One such disadvantage is the use of *classroom* environments. In order for learning to take place, students have to be present at a location at a specific time. This strict requirement can make education inaccessible to many individuals, resulting in an underutilization of teachers. The invention of the printing press in the fifteenth century made books more widely available than ever before. This invention was significant for education because it lifted the restrictions imposed by classroom learning. This was the beginning of *asynchronous education* because learners could read the subjects that interested them and they could do so in their own pace. This only made *informal education* asynchronous, because student-teacher interactions and assessment, two key components of formal education, were still synchronous, and therefore formal education was still synchronous.

1.2 The Promise of E-Learning

In order for students to enjoy all the benefits of asynchronous learning in their formal education, there would have to be a way for students and instructors to communicate reliably and quickly from remote locations, and there would have to be no scheduled assessments. Students should be allowed to cover relevant materials in any order they wish, taking as much time as needed. They should also be able to ask instructors for clarifications at any time (and from any place), and assessments should be taken when students are ready. Historically, this has been difficult to implement, but modern technologies, such as e-learning, have already made it possible to provide at least partially asynchronous learning, with the potential for more.

1.2.1 Learning Management Systems

Since its emergence, the Internet has had the potential to impact education in similar proportions to the printing press, a potential which is still largely unrealized because many current e-learning systems do not take full advantage of the available technology. Most e-learning efforts amount to deployment of Learning Management Systems (LMS). These systems mainly serve as repositories for course materials. They also provide the means for students to communicate with each other and with instructors, which is an important component of asynchronous learning. Students can also use the LMS to submit assignments and keep track of grades. Well known examples of such systems, also known as *virtual classroom* environments, are Moodle and Sakai.

A drawback of LMSs is that they do not provide automatic assessment, which is needed for asynchronous education. Automated grading is difficult to do in general because assessment methods that are easy to automate, such as multiple choice questions, are not suited to testing higher-order problem solving and reasoning skills, while techniques that are suited to testing these skills, such as essay-type questions, are hard to automate. Many virtual classroom environments make use of multiple choice questions, graded automatically, as well as essay-type questions, graded through peer-grading systems.

1.2.2 Automated Assessment for Computer Science

In Computer Science, it is reasonable to ask students to write computer programs in order for their knowledge and skills to be assessed. It is relatively easy to automatically determine whether or not a computer program is correct. In the process of designing the programming assignment, the instructor can provide a set of test cases, where each test case is a set of inputs and their associated output. A program submitted by a student can be evaluated by compiling and running it on the test cases provided and seeing if its outputs match those provided by the instructor. A grade for the programming assignment can be computed as a function of the test cases.

Grading systems of this kind allow instructors to offer *formative assessment*, which takes place during the learning process and its main purpose is to provide informative feedback to students and enable them to rectify problems with the material in a timely fashion. Teachers can also use the results of formative assessment in order to make modifications to their teaching strategy as necessary. This is contrasted with summative assessment, which is done at the end of a course, such as assigning a course grade. It is meant to determine whether or not a student has sufficiently mastered the necessary skills and knowledge in order to proceed to the next phase of the education process.

Many existing e-learning systems for Computer Science make use of automated grading schemes to provide formative assessment. Well known examples of such systems include Coursera and EdX, which are both platforms for Massive Online Open Courses (MOOCs). In addition, there are thousands of smaller-scale systems that are designed to supplement classroom instruction of Computer Science. One such example is SATS (Student Administration and Testing System), developed in the University of California, Merced, and used in several undergraduate Computer Science courses.

Systems like the ones above facilitate true asynchronous learning, especially in MOOCs where there is no classroom. Despite this fact, current e-learning systems for Computer Science still have shortcomings. This is exemplified by the poor pass rates and retention rates in MOOCs and the non-significant effect e-learning systems are having in traditional universities. A possible explanation for this is related to the binary nature of the feedback produced by the systems. In the case of incorrect submissions, e-learning systems are unable to offer the student any guidance.

1.3 Dissertation Contributions

1.3.1 Problems with Binary Instant Feedback

Ridgway et al. (2007) points out that poorly designed formative assessment systems often have negative effects on students. Binary feedback of the form "Correct/Incorrect" may be sufficient for some, but the majority of students often need more guidance. It is not difficult to see that getting feedback that only says "Incorrect" can be frustrating and demotivating for students. It is Beaubouef and Mason's main argument against automated assessment in Computer Science. We investigated the effects of binary instant feedback on student performance and behaviors and found that it causes disengagement and plagiarism. As far as we know, this was the first study exploring binary feedback for computer programming exercises, and it exposed a serious flaw of automated assessment systems, of which many educators were unaware. The full details of the study appear in Kyrilov and Noelle (2015a).

1.3.2 Common Programming Errors Committed by Students

In order to be able to provide better feedback to students on programming exercises, it was important to characterize the kinds mistakes students were making in their solutions. Existing literature on this topic focuses predominantly on compiler errors (Brown and Altadmri, 2014). We ask our students to test their code locally, before submitting it for evaluation, so compiler errors are caught and addressed by students before they make their first submission. We conducted a study to find the most common reasons for students submitting incorrect solutions for programming exercises in our laboratory sessions, and found that incorrect formatting of output was the most common problem. Other issues included hardcoding of inputs, instead of reading them from the user, and not testing beyond the sample input-output pair provided as part of the exercise statement. The full details of this study can be found in Kyrilov and Noelle (2016). The results of this study enabled us to prevent the most common error from occurring in the future, by simply verifying the formatting of the output. The instructor effort required for this step is minimal, and the benefits to students are significant.

1.3.3 Novel Application of Case-Based Reasoning

Case-based reasoning is a machine learning technique that has been successfully applied in several domains, such as medical diagnoses and generation of legal advice (Begum et al., 2011). We were the first to apply the technique to automated assessment of computer programming exercises, and the design for our framework first appeared in Kyrilov and Noelle (2014). The technique relies on reusing knowledge gathered from past student-instructor interactions in order to generate meaningful feedback for students who submit incorrect solutions to programming exercises. Case-based reasoning is highly suited in situations where similar problems occur often, and can be solved in the same way. This is true of undergraduate Computer Science laboratories, where many students make the same programming mistakes.

1.3.4 Innovative E-Learning System

We developed an e-learning system, called Compass, which is specifically designed to address the issue of feedback quality in automated grading systems for programming exercises by using case-based reasoning. In addition to this, Compass has features for preventing common errors, such as incorrect output formatting. Compass is built as a web application, available to students any time from any location. It uses modern standards, such as HTML5, CSS3, and JavaScript, making it compatible on all modern platforms and devices. The backend is implemented as a distributed application accessible through RESTful API. This design allows other front-end interfaces, such as Moodle or Sakai, to easily integrate with Compass, making it possible for multiple institutions to use the automated assessment components of the software, without having to adopt our front-end application. This is an important consideration, since many schools have institution-wide policies governing their choice of Learning Management Systems.

1.3.5 Improved Student Performance in Programming Exercises

We performed a number of statistical analyses to compare the effects the Compass system had on student performance and behavior. We found that it sufficiently addresses the problems with binary instant feedback, that we uncovered in an earlier study. The introduction of high-quality feedback to programming exercises led to a statistically significant decrease in the time, and attempts, taken by students to correct an initially flawed solution to a programming exercise. There were also statistically significant reductions in the plagiarism and disengagement rates, after Compass was introduced.

This could prove to be a significant factor in Massive Online Open Courses (MOOCs), where low completion rates continue to be an issue. Current MOOCs generally rely on multiple choice tests, and binary feedback, which have been shown to have negative effects in a typical classroom environment. The case-based reasoning module of Compass can be integrated in MOOCs to provide timely, detailed feedback to students, which may lead to higher retention rates in these courses.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents relevant background material on e-learning in general and automated assessment in particular. Chapter 3 presents a study on the negative implications associated with binary instant feedback, which is the typical form of feedback received by students for programming exercises. Chapter 4 introduces the reader to the case-based reasoning framework and the way it is applied to an e-learning environment. Chapter 5 provides an overall description of the Compass e-learning system that we designed and built. Chapter 6 describes the study that we conducted in order to determine whether or not the Compass system adequately addresses the problems with traditional automated assessment systems, identified in chapter 3. Chapter 7 contains some concluding remarks. Appendix A presents a study of an alternative automated assessment system that we built, focusing on exercises in first-order logic. It is not officially a part of the Compass system but it had a statistically significant effect on student performance in exercises involving translating English sentences to first-order logic. Appendix B contains the raw data for all the experiments performed in chapter 6.

Chapter 2

Background and Related Work

2.1 E-Learning

The use of any modern technology to aid the education process is referred to as e-learning. This broad definition encompasses educational technologies that existed before the internet. In those days it was believed that television would revolutionize education, replacing classrooms with educational TV programs (Rosenberg, 2001). With the exception of a few educational programs for young children, such as Sesame Street, TV education was not successful. The main reason for this was the fact that people underestimated the difficulties and costs associated with creating educational content for television. Other hurdles, such as the high cost of distributing course materials via conventional mail, and using telephone and telegraph services for communication would have hindered the process of TV education. Despite the fact that television has been ubiquitous since the mid-twentieth century, it failed to transform education in the way the printing press had done five centuries earlier. The internet offers effective solutions to the distribution and communication problems experienced in television education, which led to the development of modern e-learning systems. These systems make use of the internet in order to deliver education. Rosenberg (2001) identifies the many benefits of modern e-learning systems, such as:

Lower cost. Since e-learning takes place on the web, costs associated with the traditional classroom are eliminated. In the long run e-learning is the most cost effective way to deliver education, easily offsetting the initial costs involved with development and deployment of the e-learning system.

Responsiveness. E-learning systems can handle much larger class sizes than the traditional classroom. This allows more people to be trained more quickly, allowing educators to respond to rapid changes in curricula.

Personalization. In a traditional classroom, all students see the same content but with e-learning systems it is possible to personalize the experience on the basis of individual learners.

Timeliness. Users of e-learning systems allow instructors to author new content and to modify existing materials easily, ensuring students always have access to the most up-to-date information.

Unlimited access. E-learning systems can be accessed from anywhere at any time, a critical component of asynchronous learning.

Familiarity. Since a large percentage of students are comfortable with computers and the internet, there is no need to offer any specialized training on using e-learning systems.

Universality. Since e-learning systems typically run in web browsers, all users get the same experience, irrespective of what computer platform they use.

Scalability. E-learning systems are much more scalable than traditional classrooms. The difference between having 10 students and 10,000 students is not significant, as long as proper infrastructure is in place.

2.1.1 Learning Management Systems

Since the turn of the century there have been countless attempts at developing e-learning systems. Most of them were developed by instructors and used within the organization they were developed, while a few became commercial or open-source products. Blackboard is an example of a commercial e-learning system, while Moodle and Sakai are open-source. These mainstream systems are commonly referred to as Learning Management Systems (LMSs). In general, an LMS is an online platform that offers the following features:

Access to course materials. Instructors can upload educational resources, in the form of documents, presentation slides, and videos, which students can access anytime from a device connected to the internet.

Communication. In addition to regular email, an LMS offers discussion forums where students can post questions and get answers from their peers or from the teaching team.

There are also instant chat systems in place for real time communication, allowing efficient resolution of problems.

Assignment submissions. An LMS allows students to digitally submit solutions to assignments. Having solutions in digital form allows the instructor to use many other e-learning tools, such as plagiarism detection software and automated grading software, which are not typically part of LMSs because they are usually domain specific.

Administrative tools. LMSs allow students to keep track of their grades and to manage their enrollments online. Administrative functions for instructors are more elaborate, allowing the instructor to author content, set deadlines and modify the appearance of the interface in a way that is best suited for the course.

Rössling et al. (2008) present technological and pedagogical guidelines for developing Learning Management Systems. They stress issues such as platform independence, licensing, dissemination, security and customizability. Most LMSs that exist today are web-based applications, which ensures platform independence, as long as users have access to the internet. The authors note that the vast majority of LMSs developed never get used outside the institution they are developed in, which is inefficient as it results in a lot of duplication of previous work. Developers are therefore encouraged to make their systems available to as wide an audience as possible. Customizability becomes a key issue as adoption rates grow because different organizations will have specific requirements that an LMS should meet. Finally, security is an ever growing concern with online systems due to the increased rates of collection of personal data. Learning management systems collect large amounts of such data, making them an attractive target for malicious users. Proper authentication methods as well as encrypted communications between users and the LMS are important features to have in place.

It is also important to consider pedagogical issues when designing an LMS. There should be tools to support the instructional process, which includes the setting of goals, presenting plans on how to achieve the goals, presenting the actual material, highlighting the learning outcomes, and providing feedback. An illustration of the instructional process, adapted from Rössling et al. (2008) appears in figure 2.1

Another area of consideration is individual learning styles. There are different learning style frameworks in the literature, including Felder and Silverman's model (Felder and Silverman, 1988), where students are classified as active vs reflective, sensing vs intuitive, verbal vs visual, and sequential vs global. Kolb's model of learning styles (Kolb, 1984), suggests

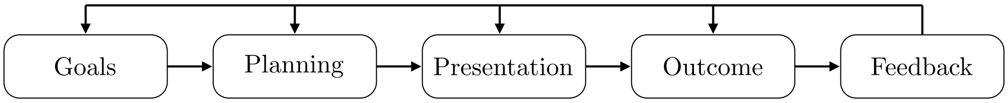


Figure 2.1: The instructional process

four learning styles, namely, diverging, accommodating, converging, and assimilating. A well designed LMS will enable instructors to cater to different learning styles at the same time. It is useful to note that students can change their learning styles depending on context, so presenting material in different ways, suited to different learning styles, is beneficial.

Keeping students motivated is a major challenge in LMS design. Rössling et al. (2008) highlight two kinds of motivation, namely internal and external. Internal motivation stems from the learners' own interest in the subject material, whereas external motivation is derived by factors such as receiving positive feedback, reaching a milestone, or getting a good grade. Other external motivating factors include high levels of active engagement and the availability of additional resources to support the learning process.

2.1.2 Computing Augmented Learning Management Systems

Rössling et al. (2008) published guidelines for augmenting a regular LMS to be better suited for Computer Science education. Such LMSs are called Computing Augmented Learning Management Systems (CALMSs). Their main recommendation is to add automated assessment capabilities, specifically for programming exercises. Techniques for automatically grading programs have been around since the 1960s (Forsythe and Wirth, 1965). The basic idea has always been to compare outputs produced by student programs to model outputs provided by the instructor. It is up to the instructor to provide appropriate test cases that establish the correctness of programs. If a program bug is not covered by a test case, it will not be detected, resulting in an incorrect program possibly declared correct. There should also be enough test cases to ensure students can not arrive at a correct output by guessing.

An assessment scheme of this nature will produce binary feedback, meaning that it will only be able to detect whether a program passes a set of test cases or not. The system will not be able to suggest corrective actions to students who submit incorrect solutions.

2.1.3 Massive Online Open Courses

In recent years, organizations such as Coursera and EdX have started offering Massive Online Open Courses (MOOCs). They use e-learning technology to deliver high-quality education to anyone with an internet connection. Content, in the form of video lectures, course notes, and assessment materials, is provided by universities around the world and can be accessed free of charge. The first MOOCs to be offered in 2012 were Computer Science courses from Stanford University. MOOCs are delivered via state-of-the-art e-learning systems, which include all the features of CALMs discusses above. Coursework is usually divided into modules that students can complete, with the assessment built into the module. Since the assessment is automatic, EdX and Coursera offer truly asynchronous learning. Both of these platforms have recorded lectures and course notes available online. Participants can communicate with each other and with the instructors through online forums. They can cover the material in whatever order they want at a time of their choosing. Their homework and other assessments are graded automatically as they are submitted. Other than preparation work and participation in discussion forums during the course, there is no other work the instructor needs to do. This frees instructors to focus on improving course materials and interact with students, which are the most important tasks of an instructor. Repetitive and mundane tasks, such as lecturing, grading exams and calculating grades are no longer required. This also allows instructors to reach a far greater number of students. EdX and Coursera have course enrollments in the hundreds of thousands. Many higher education institutions around the world are embracing the MOOC platform and are making their courses available on EdX and Coursera.

2.2 Automated Assessment of Programming Exercises

Automated assessment of computer programming exercises has been studied extensively. Douce et al. (2005) present a historical account of influential automated assessment systems, spanning from the earliest systems developed in the 1960s to modern web-based systems that are in use today. There are numerous benefits associated with automated assessment systems, such as their ability to maintain objectivity and consistency when grading students' work. This is difficult for human graders to achieve (Ala-Mutka, 2005).

Due to their speed and ease-of-use, automated assessment systems offer substantial relief to educators facing highly demanding grading processes. By reducing the time cost of grading, these systems allow instructors to devote more time to other activities that benefit students.

Also, there is empirical support for the idea that giving students more practice with programming exercises assists learning (Woit and Mason, 2003). By leveraging the reduction in grading overhead provided by automated assessment systems, instructors are able to assign more programming exercises while ensuring that the students will receive some form of feedback on each of them.

Automated assessment systems can also provide instructors with valuable information regarding student performance on programming exercises, potentially identifying the skills and concepts that students are finding most challenging. Since such student performance information is available rapidly and continuously, instructors have the opportunity to make adjustments to their lesson plans in order to respond to common difficulties and misunderstandings.

Numerous studies have investigated the effects of automated assessment with instant feedback on student performance. Falkner has suggested that immediate feedback helps students build confidence and improves their understanding of programming concepts (Falkner et al., 2014). Ala-Mutka has reported a similar finding but also warns that the design of the exercises can play a significant role in the overall effectiveness of an automated assessment system (Ala-Mutka, 2005).

Researchers have raised concerns with automated grading systems for programming exercises, including the fact that they may encourage students to engage in dishonest behavior, such as trying to trick the grader by hard-coding program output to match system test cases. Another concern is that students may start using the grader as a debugging tool, avoiding the learning experiences associated with testing their own code (Ala-Mutka, 2005). This also leads to *bricolage*: the practice of mindlessly modifying incorrect code in the hopes that the grader will eventually accept it (Ben-Ari, 1998).

When deciding whether to employ an automated grading system for programming exercises, instructors should carefully consider the advantages and disadvantages listed above. Most educators believe that the benefits outweigh the disadvantages, which explains why automated assessment systems are so widely used in undergraduate computer science education. Still, some instructors find the quality of feedback unacceptable and are not willing to use automated assessment systems (Beaubouef and Mason, 2005). Most systems that evaluate the functionality of programs do so by following a test-based approach. Such approaches only allow very coarse grained (usually binary) feedback. There is evidence that increased feedback granularity leads to better student performance (Falkner et al., 2014) . Many instructors who use automated grading systems for programming exercises agree that the quality of the generated feedback is not as good as that generated by human instructors, but they consider the automated feedback to be better than no feedback, at all.

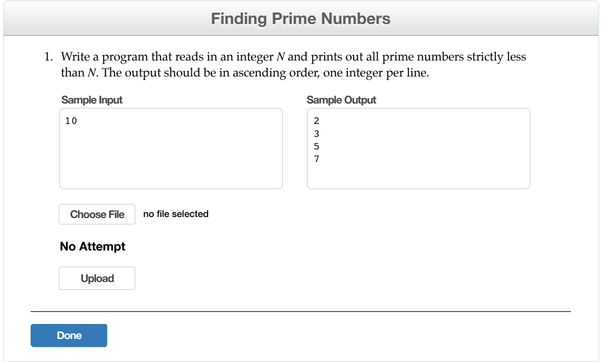
We have used an automated assessment system, described in the next section, for several years. We have suspected that many students have been cheating on their exercises by copying solutions from their peers. This suspicion is supported by a study that found that, when surveyed, about 80% of students admitted to some form of cheating, and 30% specifically admitted to submitting someone else's work as their own (Sheard et al., 2003). The authors of this previous study also investigated the reasons behind cheating. They found that one of the most frequent reasons given for cheating involved a belief that the student will fail the course if they don't cheat. We suggest that the excessive amounts of negative feedback delivered to students by systems using binary instant feedback would reinforce their self-doubt and increase the likelihood of them cheating.

2.2.1 Our e-Learning Environment

The course which served as the vehicle for this study is an upper-division undergraduate class called *Introduction to Object-Oriented Programming*. It has been offered every year for some time. The programming language used in this course has been C++. Every week, students were expected to attend a 3-hour laboratory session, during which they were given a set of programming exercises. Students had one week to complete the set, with each set typically containing about 8 exercises.

Student solutions to exercises were submitted to our online grading system for evaluation. The system used a test-based approach to grade students' programs. Thus, it only gave feedback on the functionality of the programs. Feedback was binary in nature. If a program passed all of a set of test cases (which were kept secret from the students), the student received a "Correct Answer" message. If the program failed one or more test cases, the author received a "Wrong Answer" message. The submission system stored the source code of each submission, as well as relevant time stamps and information from the grading module, such as the results from different test cases and the number of attempts the student had made at the time of the given submission. A screenshot of the web interface appears in Figure 2.2.

By design, this system could generate feedback only for complete and executable programs, so the feedback it provided does not fully meet the criteria of formative feedback.



Finding Prime Numbers

1. Write a program that reads in an integer *N* and prints out all prime numbers strictly less than *N*. The output should be in ascending order, one integer per line.

Sample Input

```
10
```

Sample Output

```
2
3
5
7
```

Figure 2.2: Automated grading system interface

To address this concern, we designed each exercise in a given set to ask students to build on work completed in previous exercises, leading towards an overall goal. In this way feedback given for the first exercise in a set can be seen as formative feedback for the overall task. For example, students were asked to build a 2D graphical application in OpenGL that had several rectangles drawn on the canvas. They were provided with code that detected the coordinates of mouse clicks, and they were asked to find the rectangle that was clicked on, if any, and change its color. This task was broken up into (1) an exercise involving the creation of a rectangle class with the appropriate instance variables, (2) an exercise that asked students to implement appropriate accessor and mutator methods, (3) an exercise involving the creation of an ordered pair object, and (4) an exercise to determine if a given set of $< x, y >$-coordinates lies inside a rectangle.

Chapter 3

Adverse Effects of Binary Instant Feedback

3.1 Introduction

Teaching students to write computer programs is a key aspect of undergraduate computer science education. Supervised programming practice is often provided through laboratory sessions, during which students are presented with sets of programming problems and are asked to generate working solutions for them. The large enrollments of introductory computer science courses frequently make it difficult to provide timely feedback to students on the programs that they write. In order to address this difficulty, many educators have turned to the use of automated assessment systems for computer programming exercises.

Automated assessment systems offer substantial benefits to both students and instructors. They provide significant labor savings to members of the teaching team, allowing them to spend less time grading students' exercise solutions and more time on other educational activities. This reduction in grading time makes it possible to increase the number of exercises assigned during a term, providing students with more practice using programming skills and, thus, supporting the learning of associated concepts (Woit and Mason, 2003). Automated assessment systems are often made available through the web, allowing students to interact with the system at any time, from anywhere, and receive instant feedback on their work. As an additional benefit, automated assessment systems are objective and consistent (Ala-Mutka, 2005), which is difficult to achieve when using human graders.

There is evidence that instant feedback serves to motivate students by alerting them to their mistakes at an early point in their efforts, providing them with opportunities to correct their programs and re-submit them for evaluation (Woit and Mason, 2003). While the feedback provided by automated assessment systems for computer programming exercises is typically timely, its quality is often questionable. Many automated assessment systems used in practice generate binary feedback, indicating only that a given submission is “correct” or “incorrect”. Some computer science educators, however, are strongly opposed to providing this kind of feedback to their students (Beaubouef and Mason, 2005). They contend that students who struggle with the material, and consequently produce incorrect solutions to programming exercises, should receive guidance from an expert instructor.

Educators generally agree that feedback produced by automated grading systems is not as useful as feedback generated by human instructors. Still, because of its speed, objectivity, and consistency, automatically generated feedback is considered to be far better than providing no prompt feedback at all, perhaps relying, instead, on the traditional practice of simply providing model solutions after submission deadlines have passed.

3.2 Plagiarism and Disengagement Due to Binary Instant Feedback

Our classroom experience with using binary instant feedback on computer programming exercises has led us to suspect that the delivery of such feedback has negative effects for some students, making it unclear that the advantages of binary instant feedback outweigh its disadvantages. Ala-Mutka has suggested that automated assessment can lead to cheating (Ala-Mutka and Jarvinen, 2004). We hypothesize that this is especially true when instant feedback is binary in nature. If a student is struggling with the material and produces an incorrect solution, it is often insufficient to indicate that the solution is wrong. The student may need some detailed guidance in order to understand the flaws in their solution, the conceptual misunderstandings that led to the exhibited errors, and productive ways to go about correcting the program.

A variety of educational theories are consistent with our hypothesis. For example, self-efficacy theory suggests that students who are not sufficiently confident in their abilities will be affected negatively by binary instant feedback (Bandura, 1977). Since students in introductory computer programming classes often exhibit low self-efficacy with regard to their programming skills (Ramalingam et al., 2004), and binary instant feedback in typical situations is more frequently negative than positive, it is reasonable to suspect that the learning of many students in these classes will be hindered by such automatic feedback. We conjecture that the excessive negative feedback students receive in these learning environments causes some of them to attempt fewer exercises, while others resort to academically dishonest practices. Clearly, both of these responses to binary instant feedback may lead to poor performance on formal summative assessments of student understanding.

In this chapter, we investigate the following questions:

1. Does binary instant feedback on computer programming exercises lead students to cheat more than when no instant feedback is provided?
2. Does binary instant feedback on computer programming exercises lead students to attempt and/or complete fewer exercises than when no instant feedback is provided?

3.3 Investigative Study

The course which served as the vehicle for this study was an upper-division undergraduate class called *Introduction to Object-Oriented Programming*. It is offered every year. Each week, students were expected to attend a 3-hour laboratory session during which they were given a set of approximately 8 programming exercises involving writing code in `C++`. An example of a programming exercise given early in the semester is: "Write a program that reads in an integer N and prints out all prime numbers strictly less than N. Sample input: 23, expected output: 2, 3, 5, 7, 11, 13, 17, 19." Students had one week to complete each set.

In 2013 and 2014, student solutions to exercises were submitted to our online grading system for evaluation. The system used a test-based approach to grade students' programs. Thus, it gave feedback only on the functionality of the programs. Feedback was binary in nature. If a program passed all test cases (which were kept secret from the students), the student received a "Correct Answer" message. If the program failed one or more test cases, the author received a "Wrong Answer" message and was allowed to try again. Thus, each student could generate a sequence of submissions for each exercise. The system recorded the source code of each submission, as well as submission times, the output produced on test cases, and the number of attempts on the given exercise that the student had made at the time of submission.

By design, this system could generate feedback only for complete and executable programs, so the feedback it provided does not fully meet the criteria for formative feedback. To address this concern, we designed each exercise in a given set to ask students to build on work completed in previous exercises, leading towards an overall goal. In this way feedback given for early exercises in a set could be seen as providing opportunities for formative feedback with regard to the overall task.

By the end of 2014, we had collected data from two iterations of this course, with binary instant feedback provided for every exercise. Students were allowed to resubmit solutions for each exercise as many times as desired, until a due date one week from the introduction of the set of exercises. Given concerns over the use of binary instant feedback, the course was modified during its 2015 offering. In 2015, all exercises were identical to the preceding years, but the automated grading system delayed the delivery of feedback until after the weekly deadline had passed. All other aspects of the course remained unchanged. It was taught by the same instructor, with the help of the same teaching assistant, using the same set of lecture notes. There were no students present in more than one offering of the course. For this study, we examined results from exercises assigned during the first four laboratory sessions of each year.

To answer the first research question, we tested all of the submissions for plagiarism. A submission was marked as being plagiarized if it was identical to another student's submission for the same exercise from the same year. This is a very conservative measure of plagiarism, but it ensures that the likelihood of false positives is very small. All of the programming exercises required the writing of a sufficient number of lines of code so as to ensure that two students would not have generated identical files independently. We also attempted to use more sophisticated plagiarism detection tools, such as MOSS, but we found that the number of false positives was too large to make the measure reliable. This is likely due to the fact that each exercise was small enough that many students could have structured their code in a similar way.

After each submission was classified as plagiarized or plagiarism free, we tallied the number of exercises that each student had cheated on. We also counted the number of *honest sequences* of submissions produced by each student, across exercises. We define an honest sequence as a sequence of submissions for a given exercise for which all of the submissions in the sequence are free of plagiarism.

We performed an analysis of variance between students in different years with respect to the number of times they had cheated. A similar analysis was performed with respect to the number of honest sequences produced by each student.

3.4 Results of Study

We examined data from a total of 33 programming exercises in each year, spread over the first four laboratory sessions. We looked at the sequences of submissions from each student for each exercise. Table 3.1 provides summary information about the length of submission sequences over the three years of this study.

Table 3.1: Submission sequence length statistics

Year	**Students**	**Mean Length**	**Variance**	**Longest**	**Length=1**
2013	62	2.32	6.36	22	56%
2014	62	2.70	11.99	44	53%
2015	90	1.73	1.37	19	58%

Observe that in 2013 and 2014, when binary instant feedback was provided, students were making extensive use of the resubmission feature of the system. This suggests that a large number of students could have been *iterators*, that is, students who search for a correct solution by submitting numerous programs to the system, with only small, often poorly thought out modifications between consecutive submissions (Karavirta et al., 2006). Employing such a strategy is an ineffective use of the grading system, and binary instant feedback seems to encourage it. We considered a submission sequence of length 10 or more as an iteration sequence. In 2013 and 2014 there were 45 and 71 such sequences, respectively, while in 2015, when binary instant feedback was not offered, there were only 2 such sequences.

To address our first research question, we looked at the number of exercises for which a student had submitted at least one plagiarized solution. Typically, plagiarized solutions appeared at the end of a submission sequence. The average number of plagiarized exercises per student, for each year, is shown in Figure 3.1.

It is clear from Figure 3.1 that, in the years when binary feedback was provided, students tended to cheat more. An analysis of variance between 2013 and 2014 showed no significant difference ($t(122) = 0.337,\ p = 0.736$), so the two groups were collapsed into one control group. We performed an analysis of variance (using a weighted means approach to addressing unequal sample sizes) between the control group and the test group, which was composed of the students from 2015. This analysis showed that there was a statistically significant difference between the test and control groups ($t(212) = 3.873,\ p < 0.001$). This strongly suggests that binary instant feedback had promoted cheating.

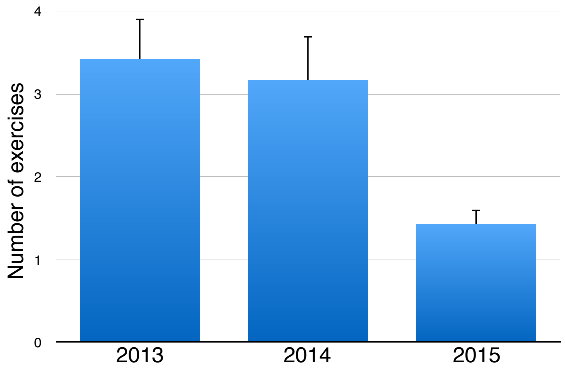


Figure 3.1: Mean number of plagiarized exercises per student (with standard errors of the mean)

Our second question was whether binary instant feedback leads students to reduce the number of exercises that they attempt. Such a reduction might be taken as a sign of disengagement. We looked at the number of *honest sequences* per student, shown in Figure 3.2.

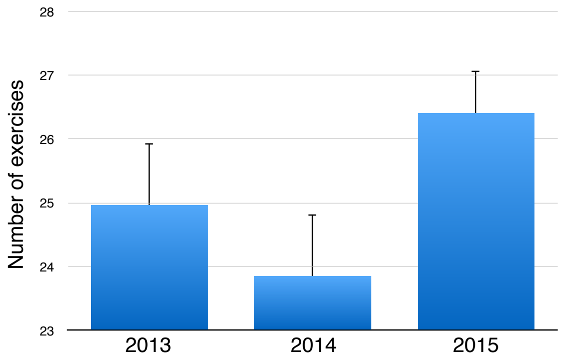


Figure 3.2: Mean number of honest sequences per student (with standard errors of the mean)

The means for 2013 and 2014 were not significantly different ($t(122) = 0.805,\ p = 0.422$), so they were collapsed into a control group. The students from 2015 made up the test group. The analysis of variance between the two groups (using a weighted means solution to the problem of unequal sample sizes) showed that there is a statistically significant difference between the two groups ($t(212) = 2.032,\ p = 0.043$). This strongly suggests that binary instant feedback led students to make fewer honest attempts at the exercises.

In hopes of further understanding the situations in which a student failed to submit a solution for an exercise (a potential sign of disengagement), we also computed the probability that a student would not attempt an exercise given that he/she failed the previous one. This probability was 0.3 in both 2013 and 2014, while, in 2015, it was only 0.1.

The results of our analyses are summarized in Table 3.2.

Table 3.2: Summary of analysis results

Number of plagiarized exercises per student

Group	*N*	Mean	SD	SE	*t*	*p*
control	124	3.161	4.421	0.562	3.873	0.000
test	90	1.433	1.914	0.202		

Number of honest sequences per student

Group	*N*	Mean	SD	SE	*t*	*p*
control	124	24.411	7.683	0.690	2.032	0.043
test	90	26.411	6.222	0.656		

3.5 Discussion

We found a statistically significant difference in cheating between the group of students who had access to instant binary feedback and the group of students who did not receive feedback until after the programming exercises were due. It is important to note that our conservative measure of plagiarism, involving only perfectly identical submissions, certainly did not detect all of the cheating cases. It would have been sufficient for a student to rename an identifier or modify a comment in the code in order to avoid detection, with regard to our analysis. The fact that we found significant differences despite the conservative nature of this measure strengthens our findings. Laboratory sessions exist for students to get practice in programming, allowing them to improve their skills. If they are submitting other students' work without modification, it is very unlikely that they are obtaining the full educational benefit of the exercises.

Similarly, our analysis of the number of honest sequences shows significant differences between conditions when binary instant feedback is present and when it is not. Students receiving binary instant feedback made honest attempts on roughly 24 of 33 exercises, on average, while students “deprived” of such feedback attempted about 26 exercises, on average, without committing plagiarism.

These results demonstrate that instant binary feedback can introduce educational hazards, suggesting that instructors should not necessarily see this form of feedback as, at worst, “harmless” with regard to student learning. Our data are not sufficient to determine the mechanisms through which instant binary feedback might promote cheating and a reduction in attempted exercises. However, one possible explanation is rooted in self-efficacy theory.

Self-efficacy theory holds that people with low self-efficacy for a given task are likely to avoid it. Schunk has pointed out that success raises one’s self-efficacy while failure lowers it (Schunk, 1991). In our data, the automated grading system sent positive feedback to students only 30% of the time. On average, students received much more failure feedback than success feedback, potentially diminishing their confidence in their abilities. For students who began the course with low self-efficacy, the negative feedback could have been crushing, driving them to cheat or leave the laboratory without further attempts at exercises. This is consistent with the findings of Sheard et al. (2003), where self-doubt is identified as the second most likely reason for students to cheat.

Self-efficacy theory also explains why excessive negative feedback should not be expected to reduce every student’s confidence in their programming abilities. According to the theory, if a person has already developed a strong sense of self-efficacy, then failure on a task actually serves to motivate the person because he/she has confidence in his/her abilities to correct the problem. This could explain why some students did well on the programming exercises without cheating. To them, a negative binary signal provided enough motivation to go back and correct their problem.

3.6 Conclusion

Many computer science instructors make use of automated assessment systems for programming exercises, especially in introductory undergraduate courses. Many of the systems that are in daily use grade programming exercises by inspecting the output generated by students’ submissions when run on predefined test cases. In many systems, the feedback received by students is limited to a binary (“correct” or “incorrect”) signal. Despite this fact, many computer science educators believe that any formative feedback must be better than no feedback at all.

We investigated the research questions of whether binary instant feedback promotes cheating and whether it causes students to more easily disengage from the laboratory exercises. We analyzed data collected from an undergraduate computer science course over a period of 3 years. In the first two years, students received binary instant feedback (control group), and, in the final year, students did not receive binary instant feedback (test group). Our analyses found statistically significant differences between the two groups with respect to cheating, as well as with regard to tendencies to attempt subsequent exercises.

We have hypothesized that providing instant binary feedback may be harmful to students because they are novice programmers with low self-efficacy. Excessive amounts of negative feedback can be highly demoralizing to them, especially since there is no explanation of what went wrong or guidance on how to correct it. This may leave students with two options: give up on the exercise, and by extension the laboratory session, or obtain the solution from a friend who has got it right.

It is worth noting that we do not see these results as advocating against automated assessment, in general. We recognize the previously demonstrated benefits of instant feedback, but we have demonstrated that there are also potential educational costs when that feedback is binary. In our future work, we will investigate ways of mitigating these costs by automatically providing more elaborate feedback to students on their programming exercises.

Many researchers have tried to address the shortcomings of binary feedback. Falkner showed that increasing granularity of assessment increases the effectiveness of instant feedback (Falkner et al., 2014). Some automated assessment systems provide students with the test cases that their code failed on. While this might help to support student learning, it also might encourage students to write code that is inappropriately specialized to handle the test cases used by the feedback system (Ala-Mutka, 2005).

In chapter 4, we explore the idea of using case-based reasoning to improve the quality of feedback generated by automated grading systems for programming exercises, by addressing the disadvantages of binary instant feedback demonstrated here.

Acknowledgement

This chapter is largely based on Kyrilov and Noelle (2015a).

Chapter 4

Case-Based Reasoning for Automated Assessment

4.1 Introduction to Case-Based Reasoning

Case-based reasoning (CBR), first introduced by Schank (1982), is a problem solving framework that uses past experiences to solve problems. Past experiences, referred to as *cases*, are stored in a database, known as the *case base*. A single case consists of a problem description and a solution. When a new problem, or a *query*, is encountered, the CBR system retrieves past cases whose problem descriptions are similar to the new problem, and uses the past solutions to generate instructions on how to solve the query. If executing the instructions does not lead to a solution of the problem, then the instructions are revised and evaluated again. Revisions may take place multiple times, until the solution generated by the system is accepted. At this point a new case, made up of the query and the accepted solution, is stored in the case base, making additional knowledge available for future queries. Due to its ability to create new knowledge in this way, CBR is considered a machine learning technique. More specifically, CBR is a *lazy* machine learning technique because training a CBR system involves merely storing past experiences in a database, and learning only happens during query time. This approach differs significantly from rule-based machine learning techniques, which are usually *eager*, meaning that learning happens during training, which involves generalizing information into rules.

Case-based reasoning systems are usually designed to be domain-specific, therefore, the way knowledge is represented, the retrieval methods for similar cases, and the evaluation procedures for suggested solutions, are all dependent on the system domain. For example, the technical support department of a printer manufacturer could use a CBR system to help employees with phone support. When a customer calls for support, the technician generates a list of symptoms (the query), and searches the database for records of previous support calls with similar symptoms. The system returns one or more past cases and the employee uses the information to make suggestions to the customer. If these suggestions lead to a successful solution of the problem, the call record is stored in the case base. In this system, a problem description is a list of sentences, each describing a symptom. A solution is also a list of phrases, each giving an instruction. A typical case for such a system can be seen in figure 4.1. The retrieval of similar cases amounts to searching through the case base for past instances containing some or all of the items in the problem description. If a past case with the exact same problem description is found, it is likely that the past solution will be applicable to the new problem. If however no such past case exists, the system can still return the closest matching one, which may prove useful in solving the problem. If the system fails to find a similar case, the technician will have to use his/her own knowledge to assist the customer. Upon successful resolution of the problem, a new case will be created, and all future encounters of the same problem will be dealt with effectively.

Problem description	**Suggested actions**
Green power LED is on, POST completes, cartridges are installed, cables are plugged in, drivers installed, Not printing	Download and install latest drivers Restart computer

Figure 4.1: A typical case from printer manufacturer help desk

The CBR process explained above can be summarized as the following four stages, illustrated graphically in figure 4.2:

1. **Retrieve:** Retrieve past cases that are similar to the query.
2. **Reuse:** The retrieved cases are used to generate a solution to the query.
3. **Revise:** The solution generated in the last step is evaluated and modified if necessary.
4. **Retain:** A new case, made up of the query and the solution are stored in the case base.

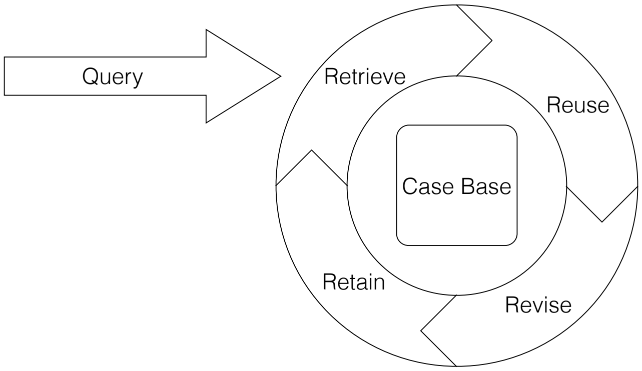


Figure 4.2: The case-based reasoning methodology

4.1.1 Knowledge Representation

The knowledge base of a CBR system is the case base. In general, a case $c = (c^d, c^s)$ is a pair, made up of a problem description $c^d \in D$ and a corresponding solution $c^s \in S$, where D is a *problem description space* and S is a *solution space*. A query $q \in D$, is a problem problem description, and the goal of a CBR system is to find a solution for q.

4.1.2 Case Retrieval

The process begins by retrieving a set of cases $T_q = \{c_1, c_2, \dots, c_k \ : \ f(c_i^d, q) < \theta\}$, from the case base, where $f : D \times D \rightarrow \mathbb{R}$ is a distance metric between two problem descriptions and θ is a user-defined threshold. The set T_q is a set of cases retrieved from memory, whose problem descriptions are similar to q in the relevant dimensions of D, and the set R_q is the set of solutions corresponding to T_q.

4.1.3 Case Reuse

The case-based reasoner uses T_q in order to learn a function $g : S^k \rightarrow S$, which transforms a set of k solutions into a single solution, $s' = g(R_q)$, which is then suggested as a solution to q.

4.1.4 Case Revision

The solution s' may or may not be an acceptable one. It is therefore evaluated and possibly modified. The accepted solution is defined as $s* = h(s')$, where $h : S \rightarrow S$ is a function that modifies a given solution, which is evaluated again, until an acceptable solution is found.

4.1.5 Case Retainment

Once $s*$ has been accepted as a correct solution to q, the case $(q, s*)$ is stored in the case base. The entire process is illustrated graphically in figure 4.3.

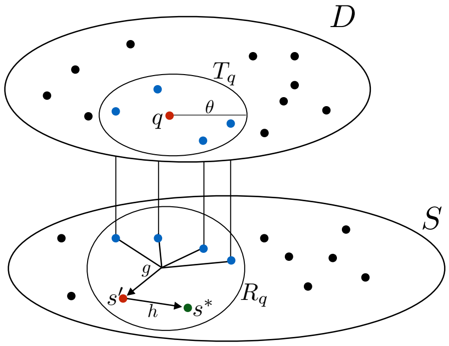


Figure 4.3: The CBR process

It is clear that the choice of functions f, g, and h will completely govern a system's behavior and abilities. The functions can be as simple or as complicated as necessary, depending on the domain of the system. In some cases functions may need to be learnt by the system, which would require additional machine learning techniques. For example, in the retrieval process, the function f measures the distance between two vectors in the multi-dimensional space of problem descriptions D. Some dimensions of D may not be as relevant as others, in the context of the specific problem. A machine learning algorithm could be used to learn what the relevant dimensions are, thereby allowing the system to complete the retrieval step more efficiently. Certain problem domains allow functions to be very simple. For example, the function g which constructs a solution from a set of previous solutions could be made to simply return a previous solution unchanged. In addition, human intervention is allowed to play a part in these functions. For example, the function h, which revises a proposed solution, could be delegated to a human.

CBR systems have been successfully applied in customer support centers, as shown in Allen (1994), while Jo et al. (1997) used a CBR approach to predict bankruptcy in Korea. Begum et al. (2011) surveyed a number of CBR systems used in the health sciences and found that these systems are being used for a wide variety of tasks, such as diagnosis, treatment planning, and training of medical personnel. CBR has also been applied to the educational sector, although it has not received as much attention as it has in other domains. Jonassen and Hernandez-Serrano (2002) proposed a CBR system to support problem solving using stories, and Ballera et al. (2013) proposed a CBR system to personalize the e-learning experience of students by sequencing the topics being presented. Wiratunga et al. (2011) presented RubricAce, a CBR system designed to assist instructors who use rubrics for grading students work. RubricAce suggests feedback comments to instructors once they have assigned grades according to a rubric. Instructors then decide how to use the suggested feedback to provide summative evaluations to students. The following section describes methods for using case-based reasoning to improve the automated assessment process.

4.2 Applications to Automated Assessment

Case-based reasoning typically works well in domains where similar problems occur frequently and similar problems have similar solutions (López, 2013). These conditions are necessary since CBR builds knowledge from past experiences, and the solutions learnt from these need to be relevant for future occurrences of a particular problem.

To determine whether our laboratory environment satisfies these conditions, we arbitrarily selected 5 exercises, and we manually clustered the incorrect submissions by grouping together solutions that had the same problems. Section 4.3 provides the details on computing *incorrectness similarity*, the metric that was used to cluster the submissions. Table 4.1 summarizes the results of this analysis. The first column shows the exercise number, the second column shows the number of incorrect submissions that have been made for that particular exercise. The "Number of clusters" column indicates the number of distinct errors made by students for a particular exercise. The column "Largest cluster" indicates the number of students who had committed the most frequently occurring mistake, while "Smallest cluster" shows how many students made the least common mistake for a given exercise.

Of the five exercises examined, the first was less challenging than the others. A total of 111 incorrect programs were submitted, but these contained only 4 distinct errors. The other exercises exhibited 8-10 distinct errors, which was still substantially lower than the number

Exercise	Incorrect submissions	Number of clusters	Largest cluster	Smallest cluster
1	111	4	54	2
2	82	10	18	1
3	73	11	19	1
4	28	8	15	1
5	26	8	13	1

Table 4.1: Summary of data from manual clustering

of incorrect submissions. Each of these exercises had clusters of size 1, indicating that there were errors made by only one student. It is clear from these data that a large fraction of the errors made were shared by a large number of students. This analysis illustrates the suitability of Case-based reasoning as a framework for providing feedback to incorrect solutions of programming exercises.

The case-based reasoning framework for automated assessment maintains a database of cases. A case is defined as an incorrect solution for a particular exercise, together with instructor generated feedback on how to correct the submission. The case base is said to be parametrized by exercise, as cases for a given exercise would not be appropriate for other exercises.

When a newly submitted incorrect solution, call it S', for a particular exercise arrives, the CBR process begins with the retrieval stage. The system finds all the cases for that exercise, that are *similarly incorrect* to S'. The process of computing incorrectness similarity, described in detail in Section 4.3, involves finding programs that contain the same bugs as a given piece of code.

Once a case has been retrieved, the feedback stored in it is given to the student who authored S'. If this feedback leads the student to submit a correct solution then S' together with the feedback, is stored as a new case in the database, thereby creating new knowledge in the system. If, however, the student submits another incorrect solution after receiving the initial feedback, the case is again forwarded to the instructor, who revises the feedback before it is sent back to the student for another attempt. This process may iterate several times until the instructor is satisfied with the feedback.

The new knowledge created, after possible refinements, is stored in the database for future use. If no appropriate cases could be found at the retrieval stage, then S' is forwarded to members of the instructional team, whose job it is to examine the incorrect solution and generate appropriate feedback for it. This feedback, together with S' is stored in the database as a case, thereby creating the first piece of knowledge for solving the particular problem that exists in S'. The entire process is illustrated in Figure 4.4.

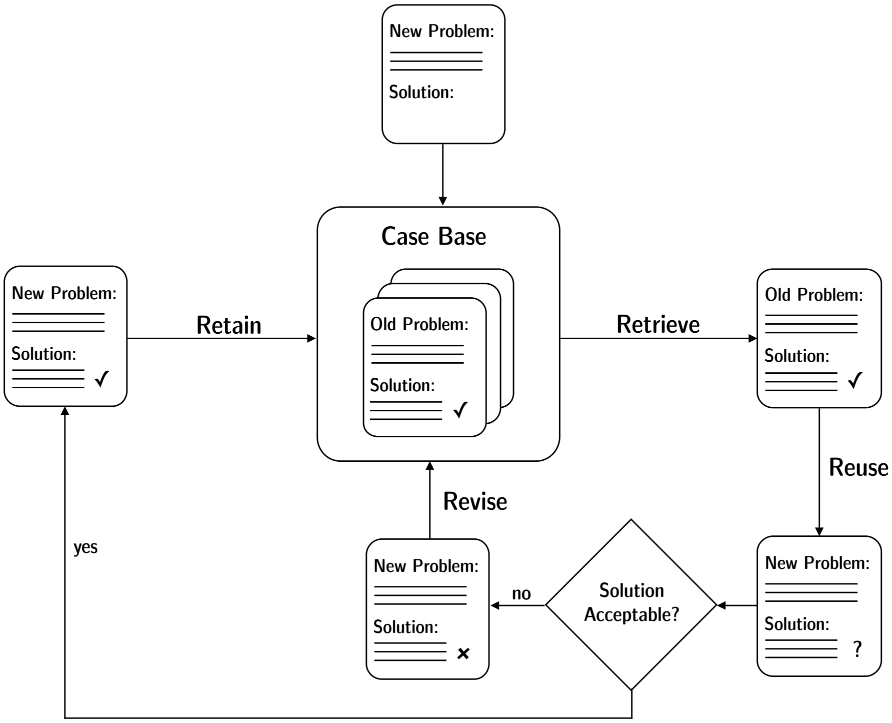


Figure 4.4: A case-based reasoning framework for automated feedback generation on programming exercises

4.3 Similarity With Respect to Incorrectness

4.3.1 Introduction

In order for the retrieval stage of the case-based framework to be successful, there needs to be a method for finding programs that are *similarly incorrect* to other programs. That is to say, both programs are incorrect solutions to a given exercise, and moreover they both have the same mistakes. Note that this measure of incorrectness similarity is distinct from a general measure of similarity appropriate for, say, detecting plagiarism. Two programs can be similar in overall structure but contain distinctly different errors. Also, two programs can be structured differently (e.g., iterative versus recursive solutions) while containing the same error (e.g., failure to initialize variables). Figure 4.5 shows an example of two incorrect implementations of the factorial function, that differ in structure but are similar with respect to incorrectness. Both programs make the assumption that $0! = 0$, when in fact $0! = 1$, which is highlighted in the code.

```
int factorial ( int n ) {
  int result = 0;

  for ( int i = 1; i <=n; i++ ) {
    result *= i;
  }

  return result;
}
```

```
int factorial ( int n ) {
  if ( n == 0 ) {
    return 0;
  }
  else {
    return n * factorial ( n - 1 );
  }
}
```

Figure 4.5: An example of two similarly incorrect programs.

Formally stated, incorrectness similarity is defined as follows. Given two incorrect solutions to a particular programming exercise, say s_1 and s_2, if corrective feedback for s_1 is appropriate for s_2, then s_1 is similarly incorrect to s_2, denoted by $s_1 \overset{\times}{\sim} s_2$.

In general, a prerequisite for computing incorrectness similarity between two programs is confirming that both programs are incorrect solutions to the same programming exercise. This can be done using traditional automated assessment techniques, described in Section 2.2. Once the incorrectness part has been established, the full incorrectness similarity measure can be computed by utilizing bug finding techniques and seeing whether both programs contain the same bugs. There are static and dynamic methods for doing this, where dynamic methods require the system to compile and run submitted programs, and static methods do not. Both of these approaches are discussed in detail in the following sections.

4.3.2 Dynamic Methods

This section describes methods of finding bugs, and consequently incorrectness similarity, that are based on compiling and running submitted code on a suite of unit tests. The output from the unit tests of two different submissions is used to compute their incorrectness similarity.

The first step in the process is to determine whether a given program, call it s, is correct or incorrect.

Definition 1. $t = (u, v)$ is a unit test, where u is the input and v is the expected output.

Definition 2. $P(s, t)$ is the output produced by program s when executed against test t.

Definition 3. $E(s, t)$ as the expected output of program s on test t.

Given a suite of tests $T = t_1, t_2, \ldots, t_n$, where t_i is a unit test, a program s is correct if and only if $\forall t \in T, P(s, t) = E(s, t)$, which is to say that a program is correct if it passes all the tests.

Conversely, a program is said to be incorrect if and only if $\exists t \in T \mid P(s, t) \neq E(s, t)$, which is to say that there is at least one unit test, for which the program does not produce the expected output.

Definition 4. $F(s) = \{t \in T \mid P(s, t) \neq E(s, t)\}$ is the set of all tests failed by s.

Consider two programs, s_1 and s_2, that are both incorrect solutions to a given exercise. By definition $F(s_1) \neq \emptyset$ and $F(s_2) \neq \emptyset$

Definition 5. $K(s, F(s)) = \{P(s, t) \mid t \in F(s)\}$ is the set of outputs for all the failed tests.

Definition 6. Given two incorrect programs, s_1 and s_2, we say that s_1 is similarly incorrect to s_2, denoted by $s_1 \overset{\times}{\sim} s_2$, if and only if $F(s_1) = F(s_2)$ and $K(s_1) = K(s_2)$.

A program s_1 is similarly incorrect to a program s_2 if they both fail on the same set of tests, and for each failed test, both programs produce the same outputs. For example, the two programs in Figure 4.5, are similarly incorrect because for any test suite, both programs will output 0 for every test.

The effectiveness of this method is dependent upon the quality of the test suite. If there are specific test cases designed to catch particular common problems, and all or most of these common misconceptions have a corresponding test case, then the approach works well. If on the other hand, the test suite does not have adequate coverage of the possible mistakes, the test cases will fail to differentiate between different bugs, leading to inaccurate measures of incorrectness similarity. The system described in Chapter 5 is designed to allow instructors to add test cases to exercises at any time if it is discovered that the original suite fails to recognize specific bugs.

4.3.3 Static Methods

This section presents an overview of static analysis techniques that can be used for computing incorrectness similarity between programs. As with the dynamic method outlined earlier, the process of computing incorrectness similarity will involve finding common bugs in programs, but while dynamic methods rely on the outputs produced by running the programs, the methods described here operate on the source code, or some intermediate representation of the source code, such as Control Flow Graphs (CFG).

A CFG represents all paths that can be traversed during program execution. A node in a CFG is a *basic block*, which is a straight-line sequence of code, without any branching or jumps. A bug-finding technique, presented by Vujošević-Janičić et al. (2013), compares the CFGs of student programs to those of instructor generated solutions. The main idea is that a student-submitted program, call it s, has a similar CFG to a program that is know to contain a certain bug, then there is a high likelihood that s contains that bug.

A suitable similarity measure for CFGs is the *neighbor matching* method, introduced in Nikolić (2012). The notion of graph similarity based on neighbor matching is easily explained by the example of computing a similarity measure between a person's left and right hands. The left hand is similar to the right one because each finger on the left hand can be matched to a corresponding finger on the right hand. In this way, similarity between nodes on a CFG can be computed by matching their neighbors. The following is a formal description of the neighbor matching algorithm.

A directed graph $G = (V, E)$ is set of nodes V, and a set of edges E. If $(i, j) \in E$, then there is an edge between node i and node j. Since G is a directed graph, we define the in-degree of a node i, denoted as $id(i)$, as the number of edges terminating at i. Similarly, the out-degree, denoted by $od(i)$ is the number of edges originating at i. Similarity between two graphs A and B, is denoted by a matrix $X = [x_{ij}]$, called a *similarity matrix*, where element x_{ij} denotes the similarity between nodes $i \in V_A$ and $j \in V_B$.

A *matching* of nodes from graphs A and B is a set of pairs $M = \{(i, j) \mid i \in A, j \in B\}$, where no element of A is matched to more than one element of B. For a matching M, define *enumeration functions* $f : \{1, 2, \ldots, k\} \to A$ and $g : \{1, 2, \ldots, k\} \to B$, such that $M = \{(f(l), g(l)) \mid l = 1, 2, \ldots, k\}$, where $k = |M|$. Each pair in M is assigned a weight by a function $w(a, b)$. The weight of the matching M is the sum of the weights of individual elements. Similarity is computed by finding the matching with maximum weight.

Iterative solutions to finding this matching usually begin with some initial estimate of similarity and refine it using an update rule of the form $[x_{ij}^{k+1}] = f([x_{ij}^{k}])$. The update rule proposed by Nikolić (2012) is the following:

$$x_{ij}^{k+1} = \frac{s_{in}^{k+1}(i,j) + s_{out}^{k+1}(i,j)}{2}, \text{ where}$$

$$s_{in}^{k+1}(i,j) = \frac{1}{m_{in}} \sum_{l=1}^{n_{in}} x^k_{f_{ij}^{in}(l) g_{ij}^{in}(l)} \quad \text{and} \quad s_{out}^{k+1}(i,j) = \frac{1}{m_{out}} \sum_{l=1}^{n_{out}} x^k_{f_{ij}^{out}(l) g_{ij}^{out}(l)}$$

where $m_{in} = \max\{id(i), id(j)\}$, $m_{out} = \max\{od(i), od(j)\}$, $n_{in} = \min\{id(i), id(j)\}$, and $n_{out} = \min\{od(i), od(j)\}$. The functions f_{ij}^{in}, and g_{ij}^{in} are the enumeration functions of the optimal matching of in-neighbors for nodes i and j, with the weight function $w(a,b) = x_{ab}^k$. The enumeration functions for the optimal matching of out-neighbors are defined analogously. The proof of convergence appears in Nikolić (2012).

In addition to using control flow graphs as an intermediate representation of programs for the purpose of static analysis, it is also possible to use an execution trace (Paaßen et al., 2016). Generally, execution traces involve compiling and running the program, and storing information related to events of interest that occurred during execution. A possible way of producing an execution trace of a program is to step through it using a debugger and print out the values of the relevant variables at each step. Incorrectness similarity between two programs can then be computed by aligning the results of the execution traces and finding points where the traces diverge from a model trace of a correct solution. Two programs whose traces diverge from the model solution at the same point can be treated as similarly incorrect.

4.3.4 Discussion

It is important to note that each of the methods for computing incorrectness similarity, described above, will be more effective in detecting certain types of bugs, and not effective with other types of bugs. In a study conducted to determine the usefulness of binary feedback provided by automated assessment for programming exercises (Kyrilov and Noelle, 2016), we investigated the most frequently occurring types of errors, that caused student submissions to be treated as incorrect. Contrary to other studies, that found missing semicolons to be the most frequent error, we found that a general failure to follow instructions was the biggest problem. Many students' solutions were marked as incorrect simply because they had not formatted their output as the instructions required. Others had hard-coded the sample inputs provided in the instructions, unaware that their programs would be tested on different inputs. The failure to address corner cases, such as the fact that 1 is not a prime number, was also prevalent.

In our setting, students are given programming exercises to complete during their laboratory sessions. A student is free to use any development environment to generate working code that solves the problem described in the exercise instructions. Students will typically submit their programs for evaluation once they feel that they have solved the problem correctly. Since we encourage students to test their code locally before they submit it, it is very unusual for us to receive a submission containing compile errors.

In general, it was found that the static analysis methods are effective at finding bugs that cause runtime errors, such as division by zero and buffer overflows. Dynamic methods are not effective at finding these problems because they rely on the output of the program. If a runtime error occurs, the program produces incomplete output, which is not useful to the method.

On the other hand, dynamic methods are effective at finding logic errors, provided that a unit test that triggers the error is present. Static analysis methods struggle here because there is nothing illegal in the code, and the problem is related to the student's conceptual understanding of the problem, and not of the programming language being used to solve it.

We therefore decided to implement a metric for incorrectness similarity based on the unit-testing dynamic method, since this method is more effective at finding the bugs we commonly encounter. In order for this to work, it is necessary to ensure that each exercise has a suite of unit tests that covers all the different possible logical errors. In general, it would not be practical to try and predict all the possible ways that a program could go wrong, and design unit tests for each scenario, but given the size of the exercises that we are assigning to our students, the number of different mistakes that are made is fairly small. In a proof-of-concept study, we looked at several exercises from our labs and manually clustered the programs based on incorrectness similarity. There were, on average, 5 - 6 clusters, meaning that the instructor would only have to design less than 5 unit tests for each exercise. This is because some of the errors happened due to poorly formatted outputs. There is no need to design unit tests for that problem. Instead, a simple regular expression can be assigned to each exercise, which describes the format of the expected output. Then, before any unit tests are executed, the system checks whether the output matches the regular expression. In cases where it does, the unit-testing can proceed, and if the output does not match the given regular expression, the system can generate appropriate feedback for the student, informing them that their code was not tested for validity because the mis-formatted output would have resulted in the submission failing the unit tests, regardless of algorithmic correctness.

The resulting automated assessment system is described in chapter 5 and the study of its effectiveness on improving student learning is presented in chapter 6.

Chapter 5

The Compass E-Learning System

5.1 Introduction

The main product of this dissertation was to produce an automated assessment system for programming exercises that is able to generate meaningful feedback to students who submit incorrect solutions. As described in chapter 3, automated assessment systems that produce binary feedback can reduce student engagement and promote cheating. A case-based reasoning framework, described in chapter 4, was designed to address this problem. The completed system, called Compass, is described in this chapter.

The goal of Compass is to be a modern e-learning system for Computer Science courses, specifically for administering programming exercises. It is designed to be available online any time from any location, via the Internet, so students can interact with it whenever it suits them. There is a user management system, where each user can log into the system and access materials for courses that he/she is enrolled.

Once a student has logged into a course, he/she is presented with the laboratory assignments for that course. Each laboratory assignment is a collection of programming exercises, where each exercise consists of a problem description, a sample input-output pair, and a collection of support files that may be needed for the exercise. Students may read the instructions on-screen while implementing the solution for the exercise in a text editor of their choice. When the student is satisfied with his/her solution, the source code can be uploaded to Compass for evaluation.

The server-side component of Compass is responsible for compiling and running the code against instructor provided test cases to determine its correctness. If the submission passes all tests, then positive feedback is returned to the student. If, however, one or more of the test fail, the feedback generation module of Compass is invoked. The first step is to determine if the tests failed because of poorly formatted output. During exercise creation time, the instructor is asked to provide a description of the expected output formatting as a regular expression. Compass matches the student-produced output to the regular expression. If the output does not match, then appropriate feedback is immediately returned to the student, informing him/her that the output is incorrectly formatted and lets them know how to correctly format it. If the format of the output is correct, then the case-based reasoning module tries to find a similarly incorrect prior solution from its knowledge base. If such a case exists in the knowledge base, the feedback from it is sent to the student. If no matching case is found, the submission is flagged for instructor intervention and an appropriate message is sent, informing the student that an instructor is looking at his/her submission and that feedback will be provided soon. It is important to note that the student is not told that their solution is wrong at this time, because the system is not able to provide more detailed feedback. The knowledge that a human instructor is looking at the code should be motivation enough for the student to wait for that feedback before giving up on the exercise or resorting to academically dishonest practices.

When the instructor generates feedback and sends it to the student, Compass will monitor the next submission from the student to determine if the feedback was useful or not. If the next submission from the same student for the same exercise no longer contains the original error, then the original submission, together with the newly generated feedback is stored in the knowledge base as a case. Any future occurrences of the same error, will be handled automatically by Compass. There could be scenarios where a student does not understand the feedback message received from Compass, in which case the instructor or a teaching assistant will be asked to manually examine the submission. Section 7.3 puts forward a proposal for extending the system capability to maintain multiple cases per error, so that the system can offer alternative explanations to students before getting the instructor involved.

5.2 System Organization

This section describes the various components of the Compass system and how they are connected together.

5.2.1 Server-side Components

The server-side components of Compass are responsible for managing users, administering exercises, storing student submissions, automated assessment of the submissions, generating feedback for incorrect submissions, maintaining administrative data, such as grades and deadlines, log data, such as timestamps for each student login and submission attempt, as well as number of attempts by each student for each exercise, and facilitating communication between students and instructors via instant messages.

Compass uses a MySQL database engine for storing and managing data. All server-side modules rely on the database layer for storage. The data stored includes user details, course information, individual submissions, and log data. In addition the knowledge base of the feedback generation module is stored in MySQL as well as the messages sent to students by the instructor.

There is a RESTful API layer that can be used to interact with the backend components. Any user interface can connect to the API layer and send messages in order to request data or write to the database. Appropriate user permissions need to be granted to agents reading or writing data. This design allows the server-side components of Compass to be used by any frontend interface, even though the Compass system comes bundled with an interface.

The automated assessment module is another server-side component of Compass. It accepts one or more source code files, and a suite of test cases, and returns a message indicating whether or not all test cases were successful. Possible messages are: "compile error", "runtime error", "time limit exceeded", "incorrect answer", and "correct answer". In the case of "compile error", and "runtime error", the actual error message is also returned, if the program is taking more than a specified amount of time, usually set to 5 seconds, the "time limit exceeded" message is returned. If all test cases pass, the "correct answer" message is returned, and if any of the test cases fail, then the "wrong answer" message is returned.

The automated assessment module is responsible for managing the processes that it spawns, such as killing those that have exceeded their time limit. In addition to this, the module also provides a sandbox environment for student processes to run, meaning they are not allowed to execute any system calls, spawn sub-processes, or do anything else outside the scope of the exercise. Usually student processes are only allowed read and write access to the temporary folder created for the test, in an effort to ensure that malicious code does not do damage to the system.

The final server-side component is the case-based reasoning module, responsible for generating appropriate feedback for incorrect solutions to exercises. This module is only invoked if the automated assessment module returns a "wrong answer" result. In this case, the first step is to verify the output formatting by matching it against an instructor-provided regular expression. If the format is correct, then the module retrieves a case from the knowledge base, according to a given similarity metric with respect to incorrectness. This module also uses the database to store relevant case information. Once feedback has been provided to a student, this module also monitors future submissions by the same student in order to determine whether the feedback was useful or instructor intervention is needed. The server-side components of the system are graphically represented in figure 5.1.

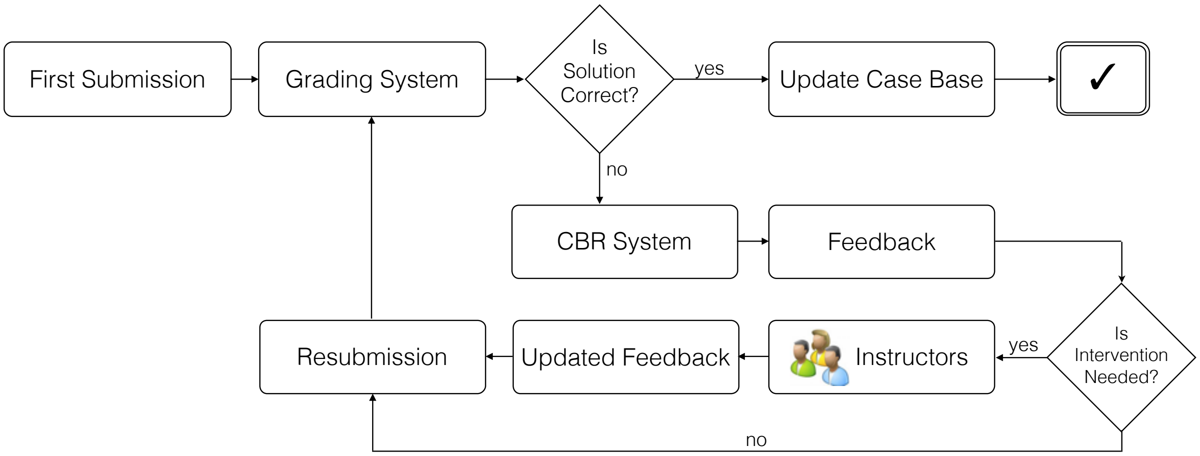


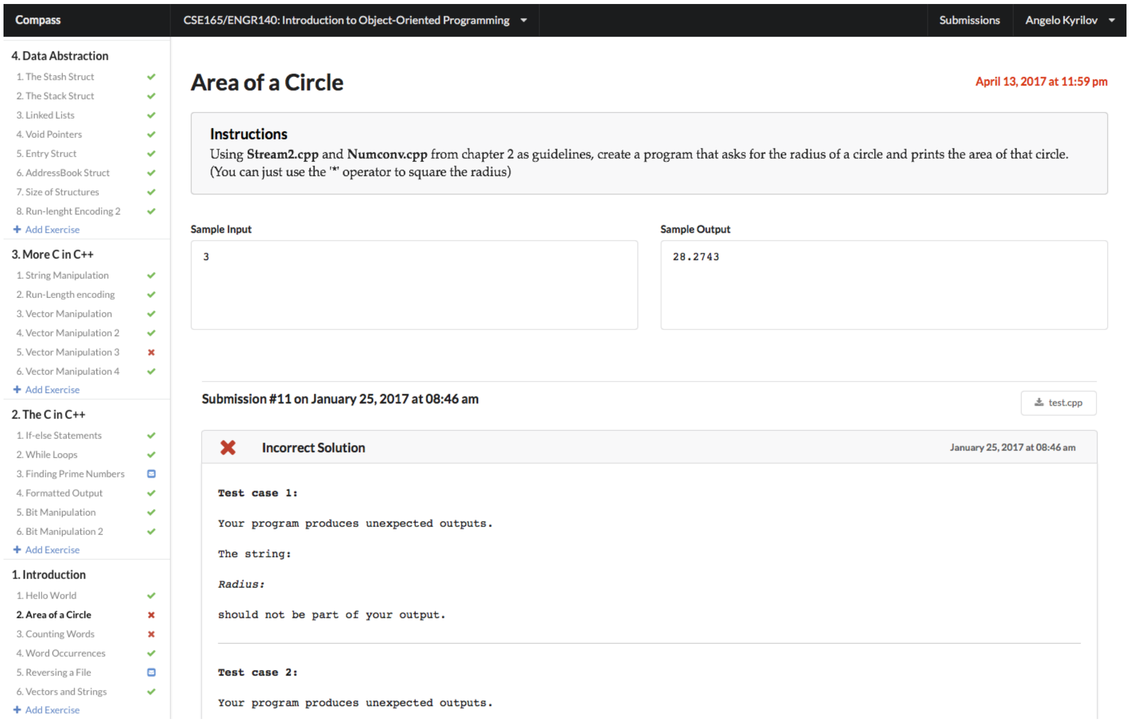
Figure 5.1: A flowchart of the automated grading component of the proposed system

5.2.2 User Interface

It is important for students to be able to interact with the system in a non-obtrusive way so as to not hinder the learning process. Early systems that required students to upload their submission to a server via the UNIX secure copy (scp) utility. Students who lacked familiarity with the UNIX command line were therefore disadvantaged.

The user interface of Compass was built using modern web standards, including HTML5, CSS3, and JavaScript, as well as a number of libraries such as SemanticUI for a consistent look of all the visual components, jQuery for managing and manipulating HTML elements with code and various others. There are no proprietary components, ensuring that Compass is compatible with all modern platforms and devices.

There is an interface for logging in, changing or resetting a password, and selecting a course to log into. The main portion of the interface is devoted displaying the lab assignments and exercises. The side panel is for selecting an exercise from a particular lab assignment, while the main portion of the screen is used to display the exercise instructions, sample input-output pairs, an interface for uploading source code, and feedback messages for each submission. A screenshot of the interface appears in Figure 5.2.



Area of a Circle

April 13, 2017 at 11:59 pm

Instructions

Using **Stream2.cpp** and **Numconv.cpp** from chapter 2 as guidelines, create a program that asks for the radius of a circle and prints the area of that circle. (You can just use the '*' operator to square the radius)

Sample Input

```
3
```

Sample Output

```
28.2743
```

Submission #11 on January 25, 2017 at 08:46 am

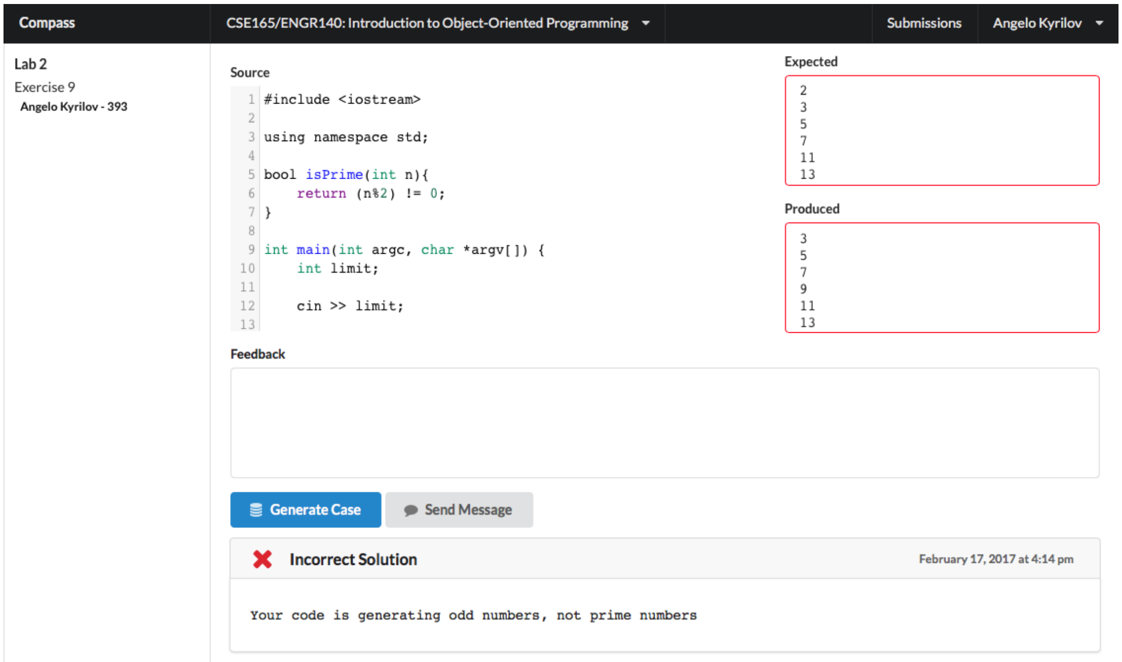
✗ Incorrect Solution — January 25, 2017 at 08:46 am

```
Test case 1:

Your program produces unexpected outputs.

The string:

Radius:

should not be part of your output.

Test case 2:

Your program produces unexpected outputs.
```

Figure 5.2: The student interface of Compass

All visual elements are updated via XML HTTP requests (AJAX), so that no page reloading is necessary when new information needs to be displayed. When a student submits a solution, the feedback message appears on the interface within seconds, which informs the student on what they should do next.

In addition to providing an interface for students to complete programming exercises, the Compass system also provides useful utilities for instructors. There are standard exercise creation and assignment tools, but the most important instructor tool is the interface for the refinement process of the case-based reasoning framework. Recall that when a student receives feedback for an incorrect submission, the system checks to see if the student ends up submitting a correct solution. If that is not the case, the submission is flagged and added to the refinement interface, where the instructor sees the code, the unit tests and the previous feedback given to the student. The instructor has the opportunity to update the feedback in order to clarify concepts that may have originally been vague. The instructor interface appears in Figure 5.3.



```
#include <iostream>

using namespace std;

bool isPrime(int n){
    return (n%2) != 0;
}

int main(int argc, char *argv[]) {
    int limit;

    cin >> limit;
```

Expected

```
2
3
5
7
11
13
```

Produced

```
3
5
7
9
11
13
```

✖ Incorrect Solution February 17, 2017 at 4:14 pm

```
Your code is generating odd numbers, not prime numbers
```

Figure 5.3: The instructor interface of Compass

5.3 User Experience

This section illustrates a typical interaction a student may have with the system, starting with an incorrect solution to an exercise, receiving feedback by the system, and correcting the original solution.

The exercise in this example asks students to read in an integer N and print out all prime numbers strictly less than N. The following sample input-output pair was provided to the students.

Input

10

Expected Output

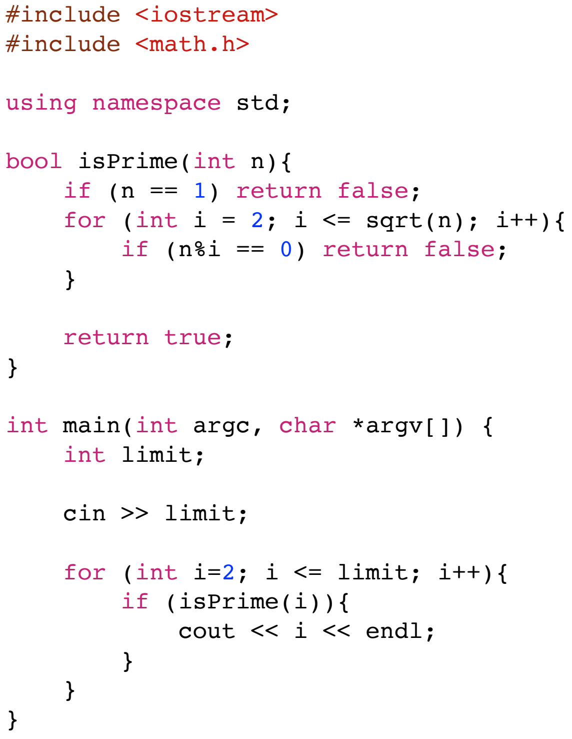
2

3

5

7

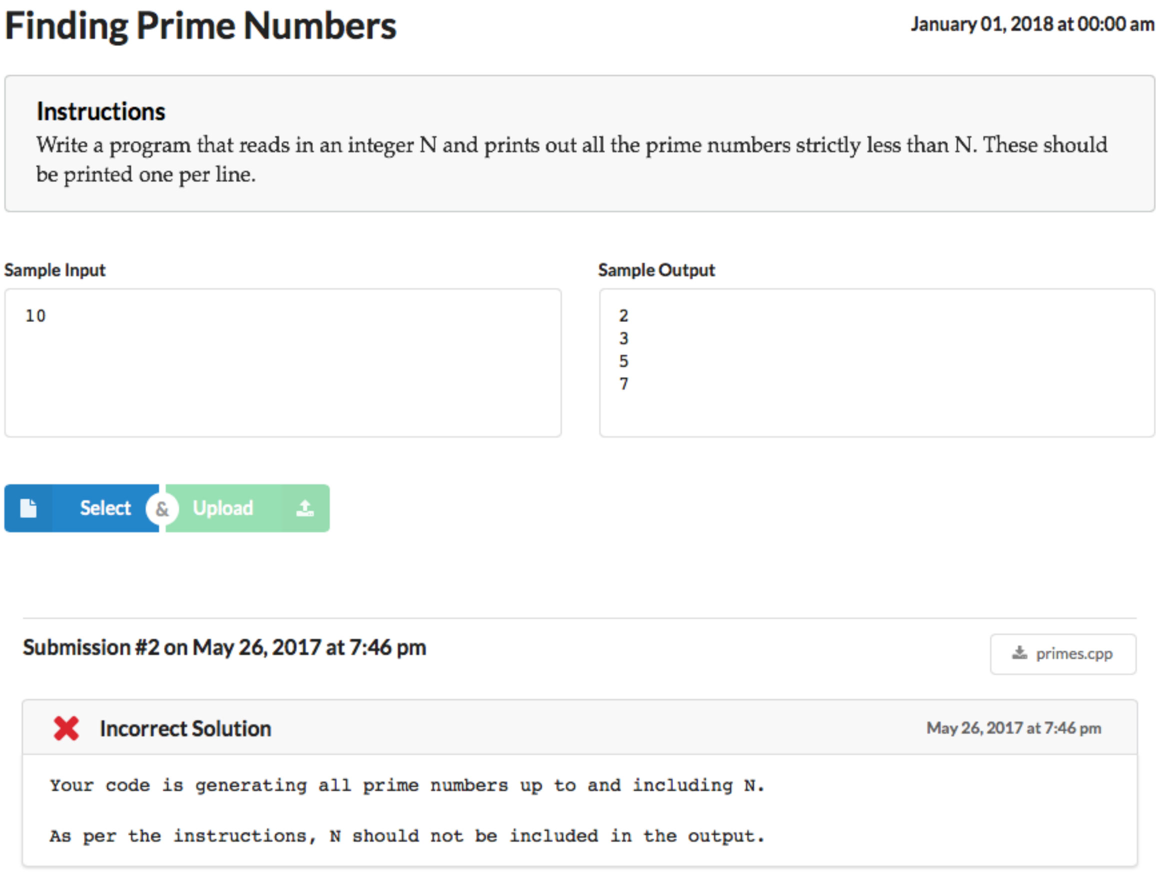
A typical student solution would involve creating a function that tests primality of integers, and a loop in the main function that iterates from 2 to N and tests each number for primality, printing out the ones that pass. An example implementation in the C++ language appears in Figure 5.4.



```
#include <iostream>
#include <math.h>

using namespace std;

bool isPrime(int n){
    if (n == 1) return false;
    for (int i = 2; i <= sqrt(n); i++){
        if (n%i == 0) return false;
    }

    return true;
}

int main(int argc, char *argv[]) {
    int limit;

    cin >> limit;

    for (int i=2; i <= limit; i++){
        if (isPrime(i)){
            cout << i << endl;
        }
    }
}
```

Figure 5.4: An incorrect C++ solution

If a student submits this solution to a typical binary feedback system, he/she will simply be told that the solution is incorrect. This could cast doubt in the mind of the student about the correctness of the `isPrime` function, which in this case is correct, and lead the student down a wrong path, which would frustrate the student when he/she finds out that the mistake is altogether different. Submitting this solution to Compass would yield a completely different result. This is shown in Figure 5.5.



Finding Prime Numbers

January 01, 2018 at 00:00 am

Instructions

Write a program that reads in an integer N and prints out all the prime numbers strictly less than N. These should be printed one per line.

Sample Input

```
10
```

Sample Output

```
2
3
5
7
```

Submission #2 on May 26, 2017 at 7:46 pm

✗ Incorrect Solution

May 26, 2017 at 7:46 pm

```
Your code is generating all prime numbers up to and including N.

As per the instructions, N should not be included in the output.
```

Figure 5.5: The feedback provided by Compass

The student is informed that his/her code is producing the prime numbers up to and including N, and not strictly less than N, so all the student would need to do is change the terminating condition of the *for-loop* in the main function. With a message of this kind, the student is much more likely to succeed on the next attempt, whereas without this feedback the student would have found it more difficult to debug because the problem only reveals itself on prime inputs. In addition to leading the student to succeed more quickly, this feedback message may also make the student realize the importance of testing code with different test cases, and think about corner cases, which is a good programming practice.

5.4 Conclusion

In summary, the Compass system is a complete, asynchronous e-learning system for Computer Science, referred to as a Computing Augmented Learning Management System (CALMS). It can be used as a standalone educational platform or as a supplement to classroom instruction. The feature that makes this system different from existing systems is its ability to provide appropriate feedback to programming exercises, which is made possible by using case-based reasoning techniques. Binary feedback provided by existing automated grading system is a serious flaw in the formative assessment process. Binary feedback affects students negatively because it only serves to reinforce negative perceptions students have of their own abilities. This can be highly demotivating for students, causing them to disengage from the material and perform poorly in the course.

By addressing the binary feedback problem, the Compass system allows instructors to focus on important activities such as teaching concepts, discussing ideas and helping students learn. Current e-learning systems do not allow instructors to do that, even if automated grading with binary feedback is in place. Instructors are most likely debugging programs of students who are struggling, leaving no time for the more useful activities discussed above.

Chapter 6

System Evaluation

6.1 Introduction

The goal of this work was to create an automated assessment system for programming exercises, capable of providing meaningful feedback to students who submit incorrect solutions. While there are many automated assessment systems for programming exercises, most of them are limited to providing binary feedback to students. This is a serious flaw and causes disengagement and cheating on the part of the students, described in detail in chapter 3. The case-based reasoning framework, described in chapter 4, was developed to address the issue of binary feedback and the negative effects it has on students. The resulting system, named Compass is described in chapter 5. This chapter focuses on evaluating whether the detailed feedback provided to students by the Compass system has been helpful to their learning of programming.

As is the case with other environments where automated assessment is used for programming exercises, almost all students end up with correct solutions for all the exercises they attempt. This is because there is usually no limit on the amount of times a student can resubmit a solution to a given exercise. Therefore, looking at student scores on their lab exercises is not a good measure to determine the effectiveness of the system. Instead, this study focuses on whether the problems of binary feedback have been adequately addressed by the Compass system and whether students are finding the feedback helpful.

Since Compass is a system for formative evaluation, students are supposed to attempt exercises, perhaps make mistakes along the way but ultimately learn from those mistakes. This can not happen if students are not attempting the exercises, or are submitting the work of others as their own.

The following are the three research questions we aim to answer in this chapter:

1. Do students find the feedback generated by Compass to be useful for their programming exercises?
2. Does Compass adequately address the problem of plagiarism caused by binary feedback?
3. Does Compass adequately address the problem of disengagement caused by binary feedback?

The following section describes the methods that were employed to answer these questions. The results from our analyses appear in section 6.3. The chapter ends with a discussion, explaining the results that were obtained.

6.2 Methods

To test the research questions presented in the last section we collected data from two instantiations of the Introduction to Object-Oriented Programming course at the University of California, Merced. In the first instance, which took place in Spring 2016, the students were assigned weekly programming exercises, but the Compass system was programmed to provide binary instant feedback only. The following year, Spring 2017, the students were assigned the same programming exercises, and the case-based reasoning component of Compass was turned on. The knowledge base of Compass was initially empty, meaning that the first incorrect submission of each exercise was handled by a human instructor, which was the first step of creating knowledge in the system. There were 90 students enrolled in 2016 and 120 in 2017. We examined student submission data from 31 programming exercises, spanning 5 laboratory sessions.

The exercises ranged in difficulty level, from very basic at the beginning to intermediate towards the end. Interesting examples include an exercise where the students were asked to read a text file and count the number of times a given word occurs. This was an interesting exercise because of the wider variety of errors that could have occurred. Common mistakes made by students were producing superfluous output. This was usually done by creating prompts for entering input, such as “Please enter the word you wish to count:”. Students did not realize at first that such prompts are treated as part of the output of the program, so their code would have necessarily been marked as incorrect, if not for the Compass feature of verifying output format. Solutions with this mistake were easily detected by Compass, and the students who authored them were given detailed explanations of how to correctly format their output. An example of such feedback was: "Your program produces unexpected outputs. The string "Please enter the word you wish to count:" should not be part of your output". This output was generated by Compass automatically, without using case-based reasoning, because the output produced by the program does not match the regular expression provided by the instructor at exercise creation time. Furthermore, Compass extracts the part that does not match and advises the student that it should not be part of the output.

The second interesting error made by students in this exercise was hardcoding of inputs and outputs. The exercise instructions included a sample run of the program, with a text file called "words.txt" and "many" as the input to the program. Some students hardcoded the file name and/or the input word, so their programs always produced the incorrect outputs because the test cases in the assessment module used different inputs. Delivering negative feedback to students who have made this mistake can be particularly damaging since their algorithm for counting the number of occurrences of a substring within a string may have been correct, and the negative feedback could shed doubt in their confidence, prompting them to start fixing something that is not broken. The case-based reasoning module of Compass was particularly effective in catching this kind of mistake, and students who had committed it were immediately alerted to this fact, which prevented entire episodes of unfortunate events.

The final type of mistake students commonly made in this exercise was ignoring corner cases. These include case insensitivity, that is failing to recognize different capitalizations of the same word as matches, and punctuation, where some students treated punctuation symbols read from the text file as part of the word that was being considered. It is a very straightforward process to design test cases that detect the presence of these errors, and the Compass system was able to effectively let students know what they were doing wrong.

For the first research question, we consider the number of attempts students needed in order to arrive at a correct solution. The intuition behind this approach is that if a student submits an incorrect solution to an exercise and receives useful feedback, then the student will arrive at a correct solution faster, namely in fewer attempts compared to a situation where the student did not find the feedback useful.

In order to mitigate the possibility of students arriving at a correct solution quickly by committing plagiarism, we consider honest sequences, defined as the number of plagiarism-free attempts by a student to get to a correct solution. We also exclude students who submitted a correct solution on their first attempt, as these students could not have benefited from the detailed feedback generated by Compass.

Once we had computed the honest sequences for each student for each one of the exercises of interest, we performed analysis of variance in order to determine whether there was a statistically significant difference between the two groups.

To answer the second research question we computed the number of dishonest submissions made by students for each exercise. As it was defined in chapter 3, a dishonest submission is a submission that is identical to a submission from another student. This is an underestimation of the actual number of academically dishonest submissions but we are highly confident that there are no false positives, that is to say, there are no submissions marked as dishonest that are actually plagiarism-free. The idea behind this analysis is that if a student receives detailed feedback on what exactly is wrong with the submission, the student is less likely to cheat. We performed analysis of variance between the 2016 and 2017 groups in order to determine whether the rate of cheating has decreased because of the introduction of non-binary feedback, and whether the reduction is statistically significant.

For the third research question, we considered the number of exercises students did not attempt. The intuition here is that success on earlier exercises in a given lab, will result in higher likelihood of the student attempting later exercises, thereby increasing the overall number of exercises attempted. Once again, we performed analysis of variance between 2016 and 2017 to determine if the increase in number of exercises attempted is statistically significant.

6.3 Results

To answer the first research question, we computed the length of honest sequences for each student, averaged over the 31 exercises of interest. That is, for each student, we computed the number of plagiarism-free submissions the student makes in order to arrive at a successful solution for an exercise. Table 6.1 shows the number of students in each year, with the average honest sequence length, including standard deviation and standard error. For the full data set, refer to appendix B.

It is interesting to note that in both years almost every student needed more than one attempt for at least one of the exercises. It is also clear that the average number of attempts made by students for an exercise has gone down from 4.7789 in 2016 to 3.6253 in 2017. We performed analysis of variance (ANOVA) and found $F(1, 201) = 21.578$, and $p < 0,0001{*}{*}{*}$, which is a strong indication that the feedback generated by Compass was useful to students,

year	N	MEAN	SD	SE
2016	87	4.7789	2.2290	0.2390
2017	116	3.6253	1.2822	0.1191

Table 6.1: Honest Sequence Length Statistics

as they were able to solve problems significantly faster than students who did not have access to the detailed feedback generated by Compass.

For the second research question we looked at plagiarism rates. More specifically, for each exercise, we counted the number of plagiarized submissions. The full data set is available in appendix B. Table 6.2 shows the average number of dishonest submissions per exercise for both 2016 and 2017. Standard deviation and standard error values are included.

year	N	MEAN	SD	SE
2016	31	14.3548	12.3977	2.2267
2017	31	5.9677	7.6441	1.3729

Table 6.2: Average number of plagiarized submissions per exercise

It is immediately clear from the table that the number of dishonest submissions per exercise has decreased dramatically in 2017, when students were provided with detailed feedback. We performed ANOVA and found $F(1,30) = 32.205$, and $p < 0.0001{*}{*}{*}$, indicating a strong statistical significance, which allows us to answer the second research question positively.

The third research question has to do with how engaged students are with the exercise material. We computed, for each exercise, the number of students that did not attempt the exercise. This gives a measure of disengagement of students from the material. We expect that detailed feedback generated by Compass will encourage students to stay in the lab and attempt more exercises. Table 6.3 shows the average percentage of students that do not attempt an exercise. The full data set is available in B.

It is obvious that the number of disengaged students is reduced from 11.7919 in 2016 to 8.1994 in 2017. ANOVA revealed that the reduction is statistically significant, with $F(1,30) = 64.603$, and $p < 0.0001 * **$. This is strong evidence that detailed feedback, provided by Compass, encourages students to attempt more exercises than those students who had no access to the Compass system.

year	N	MEAN	SD	SE
2016	31	11.7919	7.2847	1.3084
2017	31	8.1994	8.1792	1.4690

Table 6.3: The average number of students who do not attempt an exercise, expressed as a percentage of the class

6.4 Discussion

The statistical analysis performed in the preceding section provides solid evidence that all three research questions can be answered positively. The first experiment investigated whether or not the feedback provided by Compass helps students solve problems faster (in fewer attempts), thereby increasing their rate of learning. We make the assumption that if a student goes from having an incorrect solution, to a plagiarism-free correct one, then learning has taken place, and Compass makes the process more efficient. It is not a surprising result, since with negative binary feedback a student will have to spend time finding the problem on their own, and they may not be successful at finding it right away, attempting to rectify an issue which does not exist. We have seen a lot of empirical evidence of this from earlier courses, where binary instant feedback was the only form of feedback students received on their programming exercises. The detailed feedback provided by Compass was able to point students in the right direction, as evidenced by the statistical analysis, which led to increased learning rates.

A similar argument can be made for the experiments related to plagiarism and disengagement rates. If a student receives binary negative feedback, there is no information there to guide the student to correcting the solution. This is particularly detrimental to novice programmers, whose self-efficacy is low. Bombarding them with negative feedback only diminishes their already low level of self-confidence. At that point, many of them see no alternative other than disengaging from the material, or resorting to academically dishonest practices. Both of these are counter productive to learning and should be avoided. The detailed feedback provided by Compass, opens up a third avenue to explore, namely follow the advice given in the feedback. Depending on the exercise and the instructor who generated it, this feedback could point out the error in the code or at least give the student a hint on how to find it. This is when learning happens.

The Compass system also protects students from the possibility of lowering their self-efficacy. Since binary feedback offers no explanations as to why a submitted solution is incorrect, students with low self-efficacy will naturally assume that it is because they are not good programmers, a belief which is only reinforced by binary feedback. Compass has the ability to point out the actual source of the problem, making it clear to students that it is not their programming skills, but rather the fact that they are not following the instructions of the exercise. This is the most common reason for a submission to be marked as incorrect, and Compass is very effective in pointing out these errors to students.

In addition to the statistical analysis described above, we ran a pilot study with a prototype version of the Compass system, with the feedback generation module working. We collected survey responses from 10 participants, who generally had positive comments for the Compass system. For example, students described it as “more interactive”,“more detailed”, and “friendlier” than standard lab submission practices. They also noted that they received more timely feedback: “It was much better to get feedback on grading immediately rather than having to wait”. Further, some students noted that it influenced their interactions with the TA, stating: “I was able to ask my TA more specific questions [as] to what was wrong with my code”, and “It helped me find the problem and think of questions to ask the TA”. Finally, several students said that using the system influenced their confidence in the exercise, with one student noting, “It made me feel confident that the code was actually correct and functional”. Another simply noted, “It made me feel more confident”. This is simply more evidence that students find detailed feedback generated by Compass to be useful. The small sample size in this survey did not allow for the results to be published on their own.

It is also worth examining the amount of additional work instructors had to do in order to populate the knowledge base of Compass. On average, instructors were required to generate 4 knowledge cases per exercise. The effort involved in that was to examine the source code of a student submission, and write a constructive feedback comment, which was then stored as a case in the case-based reasoning framework. All subsequent submissions that were recognized to have the same errors, were handled automatically by Compass. The overall effort per exercise is equivalent to composing 4 email messages to students explaining why their code is not correct. Also note that this effort is only required in the first year. If the same exercises are used in future instantiations of the course, then they will already have a populated knowledge base and it will not be necessary for instructors to do any manual grading, unless a submission arrives with an error that has never been encountered before by the system.

The Compass system was specifically designed to be agnostic to the actual content of the feedback messages, relying on the instructor to provide high quality feedback, and the system is only responsible reproducing it to different students when appropriate. This means that if the feedback generated by the instructor is of low-quality, there is no way for the system to improve the quality, so the overall effect on student learning can be negative, if the instructors populate the knowledge base with low-quality samples. Assuming, however, that instructors will be able to provide good, or at least moderately high-quality feedback, using Compass is expected to lead to benefits for student learning.

We also posit that the system is highly scalable in the number of students it can support, without significant increases in instructor workload. This is because the number of cases instructors have to generate by hand is a function of the problem, not the number of students. The exercise we typically assign to our students have around 5 distinct errors that need to be recognized by the system. This number will stay the same even as the number of students dramatically increases. It will simply be the case that more students are making the same mistakes that the system has already encountered, allowing it to handle them automatically, without instructor intervention. The only time instructors will be required to put in effort is in the knowledge refinement stage of the case-based reasoning framework, which involves making existing knowledge more clear and more understandable to students.

Chapter 7

Conclusion

7.1 Dissertation Summary

As the number of students enrolled in introductory Computer Science courses continues to increase, there is an ever-present need for instructors in these classes to adopt automated assessment systems for programming exercises. This is because of the widely accepted principle that students learn to program by doing programming exercises. Therefore providing more practice to them is beneficial for their learning but the labor costs of manually grading the exercises quickly becomes unmanageable.

Automated assessment systems for programming exercises have existed for as long as people have taught programming. Early systems lacked many of the features of today's systems and required students to have knowledge of the UNIX command line in order for them to submit their exercises. Modern automated assessment systems for programming exercises are always part of a Learning Management System (LMS), which provides user account management, course material repositories, remote communication tools and gradebooks. LMSs are available online any time from any location, accessible on any device connected to the internet.

The automated assessment component of the systems usually works by compiling and running student solutions to programming exercises on a suite of unit tests, which were provided by the instructor at the time of exercise creation. If the student solution passed all the unit tests, then it is considered to be correct and the student is given the appropriate positive feedback. If, however, the submitted program fails one or more of the tests, then the student is given a message saying that something is wrong with their code and that they should rectify the problem and re-submit their solution.

While many educators will acknowledge that this kind of binary feedback is not as good as feedback provided by an expert human programmer, they feel that it is better than providing no feedback at all. We decided to test this hypothesis by looking at data collected by our automated assessment system. The experiment we set up was to compare behaviors of two groups of students enrolled in the same course, in different semesters. The programming exercises assigned were the same for both groups, but one group received binary instant feedback, generated by our automated assessment system and the other group received no feedback at all. Their solutions were graded after the exercise deadline had passed and their grades were published on the LMS.

Through statistical analysis, we found that the group receiving binary instant feedback was more likely to commit plagiarism or give up on the exercises completely. We found statistical significance when we analyzed the variance between the two groups in terms of the number of plagiarized submissions for each exercise, and the number of exercises students were leaving unattempted. A possible explanation for these results comes from self-efficacy theory, which states that students with low self-efficacy, that is to say students who do not believe in their own abilities, are likely to get demotivated by negative feedback. The opposite is true for students with high self-efficacy. Negative feedback encourages them to work harder because they want to maintain their belief that they have the skills required to solve the problem.

Since the students enrolled in introductory courses are novice programmers, they have low self-efficacy and the negative feedback is really detrimental to them. Since the binary negative feedback offers no explanation of what the problem is, or how to fix it, students with low self-efficacy are unable to find it on their own, leaving them only three options. Ask someone for assistance, be it the instructor or a fellow student, get a working solution from a friend and submit it as their own, or simply leave the lab without attempting any more exercises.

We have reason to believe that members of underrepresented groups are less likely to ask for help on a programming exercise because they are already uncomfortable enough just being in this environment without having to do things that would potentially cause them embarrassment, as they may be under the impression that the concept they are struggling with is very basic and they should be expected to have mastered it, so they do not want to reveal this perceived weakness. We are currently involved in a project to study these effects at the University of California, Merced, whose student population includes a large number of underrepresented population groups and first-generation college students.

To address the serious problems with binary feedback that we uncovered, we set out to design an automated assessment system capable of providing detailed feedback to incorrect programs, which is comparable to feedback generated by expert human programmers. We built a case-based reasoning framework for reusing knowledge from past interactions between students and instructors. Our new automated assessment system, called Compass, stores these interactions in a database and links them to the incorrect source code submitted by the students. When a different student submits a program that is similarly incorrect to the one stored in the database, that is to say both programs contain the same bugs, the system has human-generated knowledge that it can provide to the student as feedback.

The key to making the system work is the ability to compute incorrectness similarity. We investigated multiple approaches to accomplishing this goal, including both dynamic and static analysis methods. Static methods we considered included analyzing abstract syntax trees, control flow graphs, execution traces, and other kinds of intermediate source code representation. We found that these methods are more suited to uncovering the bugs that cause run-time errors, such as buffer overflows and division by zero. Dynamic methods, which rely on the outputs from test cases, are more suited to finding logical errors, where students did not do anything illegal, but were simply not following the instructions or lacked understanding of the problem. Designing unit tests is the best way to discover these problems.

We investigated the kinds of problems students most often experience when submitting solutions to our automated assessment system, and we found that the vast majority of them are caused by students not following the instructions correctly and formatting their outputs wrong. These are most easily caught by dynamic methods for program similarity so the Compass system computes incorrectness similarity based on dynamic analysis methods. Future work will involve incorporating the static analysis methods into Compass for the fewer cases where students submit code that causes runtime errors.

We deployed Compass in an undergraduate course at the University of California, Merced in Spring 2017. The previous instantiation of the course was in Spring 2016, and a binary feedback assessment system was in use then. We kept the same exercises as in 2016 but administered them through Compass, which is capable of providing detailed feedback to incorrect submissions. We performed a study to determine whether the problems we identified with binary instant feedback are adequately addressed by Compass.

We found that the average number of attempts students make on a single exercise had gone down significantly with the introduction of Compass. Students were now able to arrive at a correct solution faster, and in fewer attempts than students who had binary instant feedback. This is likely because the detailed feedback generated by Compass provides enough guidance for students to be able to correct the problem quickly, and there is no need for them to resort to trial and error tactics that we have seen with binary feedback. This is where a student does not know the reason why their code is failing the unit tests, so the student makes small random changes hoping to stumble on the correct solution. This is counterproductive to their learning so it is a very positive result to see this practice diminish.

The other undesirable effects of binary instant feedback, which are increased levels of plagiarism and disengagement from the course material, were also successfully addressed by Compass. We saw substantial and significant decreases in both of these practices with the introduction of Compass. Once again, giving the students reasons as to why their code is wrong, or at least some guidance on how to find the problem, means that they now have an alternative option to cheating and giving up. If they follow the suggestions in the feedback messages they are likely to succeed, eliminating the need for plagiarism and disengagement. It is exactly when students make mistakes, and they understand the reasons why what they did is wrong, and they know a way to correct the problem, that learning is actually taking place.

7.2 Discussion

The dissertation presents several new discoveries and introduces new approaches to automated assessment, that have not been considered before. Specifically, we show that binary instant feedback is detrimental to student learning and performance, as it can demotivate students and increase their propensity to cheat or disengage from the material. This is an important discovery as it debunks a widely held belief in the Computer Science Education community that automated assessment systems limited to binary feedback are “good enough” and there is no pressing need to improve the quality of feedback. A possible reason for this incorrect perception is that studies of automated assessment systems have focused on success rate of students on the exercises administered by these systems. Due to the infinite resubmission policy, where students are allowed to resubmit solutions for an exercise until they get it right, almost all students receive the maximum number of points for the exercises, which leads to high success rates among students. This, however, does not mean that learning has taken place, because if students use dishonest methods of obtaining the correct solution, they learn nothing. If students make random changes to their code, a practice that has been observed in undergraduate Computer Science labs, the students learn little, or nothing. If a teaching assistant corrects the code of a student, again, there is little learning on the student’s part. All these factors are significant motivation for pursuing the design and development of improved automated assessment methods.

Case-based reasoning, which is a machine learning technique successfully employed in various domains, was selected as a framework for reusing previous student-instructor interactions to generate feedback for students who submit incorrect solutions to programming exercises. It was the first time that case-based reasoning has been used in the domain of automated assessment of programming exercises. We saw it as promising because of the repetitive nature of problems with programming exercises, which is necessary for a successful implementation of a case-based reasoning system.

We also coined the term “incorrectness similarity” of programs. Unlike traditional measures of software similarity, incorrectness similarity ignores the structure of programs and instead focuses on bugs. Two programs are considered similarly incorrect, when they both have the same bugs. This concept, which can lead to an entire research discipline in its own right, has not been explored before. It is a crucial component of the case-based reasoning framework as it is necessary for the system to recognize when two programs have similar problems, so that it can provide similar feedback to both of them. Beyond case-based reasoning, incorrectness similarity can be useful in a number of other applications, such as informing the instructor, in real time, of the kinds of mistakes students are making, allowing the instructor to modify lecture materials on the fly in order to address the mistakes early.

All the effort in this dissertation ultimately culminated in the Compass e-learning system. It is a system specifically designed to provide feedback comparable to the feedback generated by expert human programmers. Compass is implemented as a web application, based on modern standards, which make it compatible with all platforms and devices. It can be installed at any institution and used by instructors who are not familiar with its inner workings. There are web interfaces for students to view exercise instructions and upload solutions. There are instructor interfaces for populating the knowledge base of the feedback generation system and to communicate with individual students. These communications are also used as prior knowledge in the feedback system. The system is an illustration that the case-based reasoning technique works well for this application and student learning is improved as a result.

As the system is used by more instructors over time, and the knowledge base matures, the workload of instructors will be reduced further as there will be less need for them to intervene. This could be a game-changer in MOOCs, which currently suffer from low completion rates. A significant reason for this could be that the instructional team does not have the time to address the problem of every student individually. Compass will allow them to address the problem only once (the first time it occurs), and let the system handle all future cases. This will result in more students getting the help they need, in a timely manner, thereby reducing the probability of the student dropping out.

7.3 Limitations and Future Work

We have designed and implemented an automated assessment system for computer programming exercises that provides feedback to incorrect submissions, which is comparable to human-generated feedback. It has been shown to address the serious flaws of binary instant feedback, that we uncovered. Even though it successfully addresses the problems of cheating and disengagement, and it seems to accelerate the rate at which students learn, there is still room for improvement.

A possible shortcoming of the current system is that it always provides the same feedback to all students who have made a particular mistake because it stores a single case per error. The system could be modified to store multiple cases, each of which will have a different level of detail in its explanation of how to correct the problem. Different students may respond better or worse to varying levels of abstraction in the explanations. This mechanism will allow the system to offer alternative explanations for a given mistake, before bringing the instructor into the loop. There can also be a mechanism for determining which one of the multiple cases is most appropriate for a particular student, given past submission history for the exercise, timestamp data, and other information known about the student. This will further increase the utility of the system and decrease instructor involvement. It will prove necessary if the system scales to thousands of users or higher, since that will increase the likelihood of a student not understanding a particular feedback message and needing an alternative explanation.

The current version of Compass only considers correctness of programs, established by running test cases. Issues such as efficiency, and coding style are ignored. It is inherently more difficult to design test cases for these issues and approaches such as static code analysis and execution traces will prove more useful for this. Static analysis can be used for checking coding style and ensuring students adhere to software engineering best practices, while execution traces can help to determine the algorithm that is used for a particular problem, which is an indication of efficiency. Test cases focusing on efficiency can also be designed, but this approach is less desirable as such test cases would essentially try to force the program to run out of time, or memory. Having to do that for all submission would place unnecessary burden on the system, which could result in delays in getting feedback to the students.

Another area of future research is to implement a mechanism for providing feedback at earlier stages of solution development. Currently, Compass requires students to submit code that compiles and runs, in order for it to be effective. There are students who struggle with that aspect, and could benefit greatly from some feedback in cases where they do not know how to start coding the solution. The static analysis techniques that we investigated could prove really useful for this.

Overall, this dissertation achieved its desired goals of improving student learning of programming, while maintaining instructor workloads at a minimum. The system we designed is highly scalable due to the recognition of the fact that the number of different mistakes posible in an exercise is a function of the exercise and not the number of students. Throughout the development of the Compass system, which has been thoroughly tested and proven effective, we uncovered many promising avenues for future research. Pursuing those will result in more success in broadening participation in Computer Science as well as improving retention rates, which are cornerstones of Computer Science Education research.

Bibliography

Ala-Mutka, K. and Jarvinen, H.-M. (2004). Assessment process for programming assignments. In *Proceedings of the IEEE International Conference on Advanced Learning Technologies*, ICALT '04, pages 181–185, Washington, DC, USA. IEEE Computer Society.

Ala-Mutka, K. M. (2005). A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102.

Allen, B. P. (1994). Case-based reasoning: Business applications. *Communications of the ACM*, 37(3):40–42.

Ballera, M., Lukandu, I. A., and Radwan, A. (2013). Personalizing and improving e-learning system using roulette wheel selection algorithm, reinforcement learning and case-based reasoning approach. In *The Fourth International Conference on e-Learning (ICEL2013)*, pages 184–193.

Bandura, A. (1977). Self-efficacy: toward a unifying theory of behavioral change. *Psychological review*, 84(2):191.

Beaubouef, T. and Mason, J. (2005). Why the high attrition rate for computer science students: Some thoughts and observations. *SIGCSE Bull.*, 37(2):103–106.

Begum, S., Ahmed, M. U., Funk, P., Xiong, N., and Folke, M. (2011). Case-based reasoning systems in the health sciences: A survey of recent trends and developments. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 41(4):421–434.

Ben-Ari, M. (1998). Constructivism in computer science education. *SIGCSE Bull.*, 30(1):257–261.

Brown, N. C. and Altadmri, A. (2014). Investigating novice programming mistakes: Educator beliefs vs. student data. In *Proceedings of the Tenth Annual Conference on Inter-*

national Computing Education Research, ICER '14, pages 43–50, New York, NY, USA. ACM.

Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3).

Falkner, N., Vivian, R., Piper, D., and Falkner, K. (2014). Increasing the effectiveness of automated assessment by increasing marking granularity and feedback units. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 9–14, New York, NY, USA. ACM.

Felder, R. M. and Silverman, L. K. (1988). Learning and teaching styles in engineering education. *Engineering education*, 78(7):674–681.

Forsythe, G. E. and Wirth, N. (1965). Automatic grading programs. *Communications of the ACM*, 8(5):275–278.

Jo, H., Han, I., and Lee, H. (1997). Bankruptcy prediction using case-based reasoning, neural networks, and discriminant analysis. *Expert Systems with Applications*, 13(2):97–108.

Jonassen, D. H. and Hernandez-Serrano, J. (2002). Case-based reasoning and instructional design: Using stories to support problem solving. *Educational Technology Research and Development*, 50(2):65–77.

Karavirta, V., Korhonen, A., and Malmi, L. (2006). On the use of resubmissions in automatic assessment systems. *Computer science education*, 16(3):229–240.

Kolb, D. A. (1984). Experiential learning: Experience as the source of learning and development.

Kyrilov, A. and Noelle, D. C. (2014). Using case-based reasoning to improve the quality of feedback provided by automated grading systems. In *Proceedings of the International Conference on E-Learning*, pages 384–388.

Kyrilov, A. and Noelle, D. C. (2015a). Binary instant feedback on programming exercises can reduce student engagement and promote cheating. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 122–126, New York, NY, USA. ACM.

Kyrilov, A. and Noelle, D. C. (2015b). Using automated theorem provers to teach knowledge representation in first-order logic. In *Proceedings of the Fourth International Conference on Tools for Teaching Logic*, TTL 2015.

Kyrilov, A. and Noelle, D. C. (2016). Do students need detailed feedback on programming exercises and can automated assessment systems provide it? *J. Comput. Sci. Coll.*, 31(4):115–121.

López, B. (2013). Case-based reasoning: a concise introduction. *Synthesis lectures on artificial intelligence and machine learning*, 7(1):1–103.

Nikolić, M. (2012). Measuring similarity of graph nodes by neighbor matching. *Intelligent Data Analysis*, 16(6):865–878.

Paaßen, B., Jensen, J., and Hammer, B. (2016). Execution traces as a powerful data representation for intelligent tutoring systems for programming. In *Proceedings of the 9th International Conference on Educational Data Mining*.

Ramalingam, V., LaBelle, D., and Wiedenbeck, S. (2004). Self-efficacy and mental models in learning to program. *SIGCSE Bulletin*, 36(3):171–175.

Ridgway, J., McCusker, S., and Pead, D. (2007). Literature review of e-assessment. Technical report, University of Durham.

Rosenberg, M. J. (2001). *E-Learning: Strategies for delivering knowledge in the digital age.* McGraw-Hill.

Rössling, G., Joy, M., Moreno, A., Radenski, A., Malmi, L., Kerren, A., Naps, T., Ross, R. J., Clancy, M., Korhonen, A., Oechsle, R., and Iturbide, J. A. V. (2008). Enhancing learning management systems to better support computer science education. *SIGCSE Bull.*, 40(4):142–166.

Schank, R. (1982). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People.* Cambridge University Press, New York, NY, USA.

Schunk, D. H. (1991). Self-efficacy and academic motivation. *Educational psychologist*, 26(3-4):207–231.

Sheard, J., Carbone, A., and Dick, M. (2003). Determination of factors which impact on it students' propensity to cheat. In *Proceedings of the Fifth Australasian Conference on*

Computing Education - Volume 20, ACE '03, pages 119–126, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Vujošević-Janičić, M., Nikolić, M., Tosić, D., and Kuncak, V. (2013). Software verification and graph similarity for automated evaluation of students assignments. *Information and Software Technology*, 55(6):1004 – 1016.

Walker, G. N. (2004). Experimentation in the computer programming lab. *Inroads*, 36(4):69–72.

Wiratunga, N., Adeyanju, I., Coghill, P., and Pera, C. (2011). RubricAce: A Case-based Feedback Recommender for Coursework Assessment. In *Proceedings of the Sixteenth UK Workshop on Case-Based Reasoning (UKCBR 2011).*

Woit, D. and Mason, D. (2003). Effectiveness of online assessment. *SIGCSE Bull.*, 35(1):137–141.

Appendix A

Automated Assessment in First-Order Logic

A.1 Introduction

Undergraduate computer science curricula often provide students with opportunities to study artificial intelligence (AI). Courses on AI frequently cover the development of intelligent systems by constructing knowledge bases composed of logical sentences and performing automated reasoning over those sentences. Computer science students regularly have little background in formal logics before attending an AI course, and this makes the learning of logic-based knowledge representation schemes particularly challenging.

In the Computer Science and Engineering program at the University of California, Merced, the "Introduction to Artificial Intelligence" class provides a broad survey of AI methods and topics, including the construction of automated reasoning systems using first-order logic to represent knowledge. This is an upper-division semester-long undergraduate course which is taught annually. Historically, students enrolled in this class have found knowledge representation to be a particularly difficult topic. When asked to translate English sentences into first-order logic, using a specified ontology, as part of a written final examination, their performance has been extremely poor. Students only score about 30% of the maximum possible credit, on average, when presented with exam questions of this kind.

These low scores are likely the result of a lack of adequate practice with first-order logic. The broad array of material covered in this survey course limits the amount of lecture time available to illustrate the construction of logical formulae, and high enrollments limit the amount of guidance and feedback each student can expect to receive from the teaching team.

While student understanding would certainly benefit from extensive practice on knowledge representation exercises, the grading of such exercises is demanding, as there are often many equally correct ways to express a proposition in first-order logic. Thus, given that students require feedback on practice exercises for them to be useful, the number of exercises that could be assigned has been highly restricted by limited human resources.

In order to address this problem, we built an online repository of exercises involving the translation of English sentences into first-order logic, and we designed and implemented an online software tool to automatically assess student solutions to these exercises. By using this tool, students received instant feedback in the form of "Correct/Incorrect" judgments, and students who submitted incorrect solutions were allowed to revise and resubmit their answers. There was no limit on the number of resubmissions permitted.

This online educational system was used in our "Introduction to Artificial Intelligence" course during the 2012, 2013, and 2014 offerings. We analyzed student performance on final examination knowledge representation questions, and we compared it to the performance of students from previous years, who had no access to our system. We found that students who used our system exhibited significantly improved scores on the first-order logic knowledge representation questions.

A.2 System Description

Our goal was to give students much more practice on knowledge representation exercises. We generated a repository of questions in which students were given an English sentence and were asked to translate it into first-order logic. Each question supplied an explicit list of predicates, functions, and constant symbols that students were allowed to use in their answers. A typical example would be:

> Translate the sentence "*All surgeons are doctors*", using the following constants: $Doctor$, $Surgeon$, and predicates: $Occupation(x, y)$.

A correct solution to this exercise is the formula:

$$\forall x\ Occupation(x, Surgeon) \Rightarrow Occupation(x, Doctor)$$

It is important to note that there are usually multiple correct solutions to exercises of this kind. For example, another correct answer to the question, above, is:

$$\neg(\exists x\ Occupation(x, Surgeon) \land \neg Occupation(x, Doctor))$$

Thus, student submissions could not be assessed by performing a simple string comparison, or the like, with a correct solution provided by the instructor.

Our system does require the instructor to provide a model answer for each exercise, but it does *not* necessarily label submissions that deviate from this model answer as incorrect. Instead, any submitted formula that is found to be logically equivalent to the model answer is recognized as a correct solution to the exercise. We use the *Prover9* automated theorem prover to check for logical equivalence. If A is the model answer and B is the student solution, the solution is labeled as correct if and only if the formula $A \Leftrightarrow B$ is found to be valid. Prover9 is a resolution based automated theorem prover for first-order logic with equality. Prover9 was selected because it is very easy to use and the syntax of its interface is very similar to what students see in lectures.

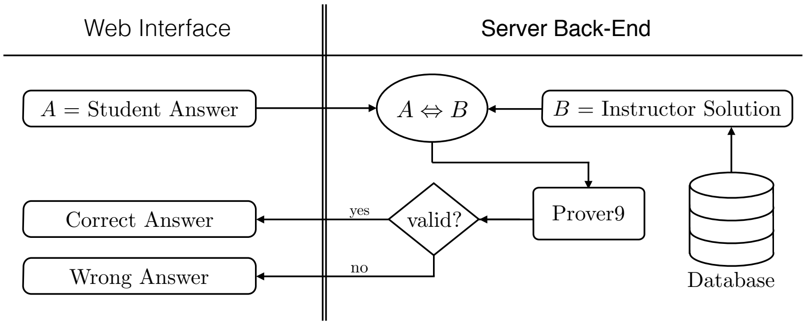
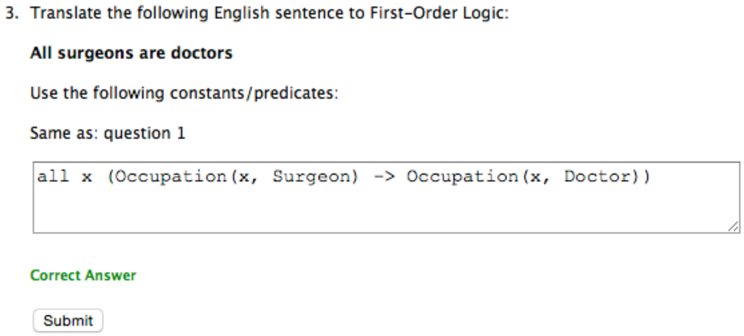


Figure A.1: Components of the Automated Grading System

Figure A.1 illustrates the automated grading system components, including the web interface and the back-end. When a student submits a solution to an exercise, the model answer is retrieved from the exercise database. Prover9 is used to determine whether the student's solution is logically equivalent to the model answer, and appropriate feedback is immediately sent to the student.

There is a restriction on the amount of time the server is allowed to spend on checking a student's submission. By default, this is set to 5 seconds but it can be adjusted on a per exercise basis. If the time limit is exceeded, an appropriate message is sent to the student informing them that the time limit has been exceeded. While this does not necessarily indicate that the student's answer is incorrect, students are encouraged to revise their solution or talk to an instructor. This takes care of the fact that the prover may run forever due to the undecidability of first-order logic.

Figure A.2 shows the user-interface of the system, which appears in a web browser window. In addition to the question listing, which is what students would see, there is also an administrative interface, allowing instructors to create and assign exercises.



3. Translate the following English sentence to First-Order Logic:

All surgeons are doctors

Use the following constants/predicates:

Same as: question 1

```
all x (Occupation(x, Surgeon) -> Occupation(x, Doctor))
```

Correct Answer

Figure A.2: The student user-interface of the automated grading system

A.3 System Evaluation

The final examination for the "Introduction to Artificial Intelligence" course is a three hour comprehensive written test that covers the full range of AI topics presented during the semester long class. Students complete the exam without access to any textbooks, notes, or other study materials. The final examination contains three questions involving the translation of English sentences into first-order logic, as well as a question asking students to produce a successor-state axiom for a given time-varying predicate. If the extensive practice afforded by our automated grading system is a benefit to student learning, then we would expect to see higher scores on these particular final examination questions when students made use of our system.

In order to evaluate our system, we collected scores on these four questions over multiple offerings of the "Introduction to Artificial Intelligence" course. Scores collected for offerings in 2007, 2008, 2010, and 2011 were produced by students who had no access to our system, as it had not yet been created. The students from these offerings acted as a control group. The automated grading system was used during offerings in 2012, 2013, and 2014, making the students enrolled during these years members of a test group. There were 113 students in the control group and 169 in the test group. The mean performance of students, as measured by the sum of scores received on all four of the relevant questions (24 points possible), is displayed in Figure A.3.

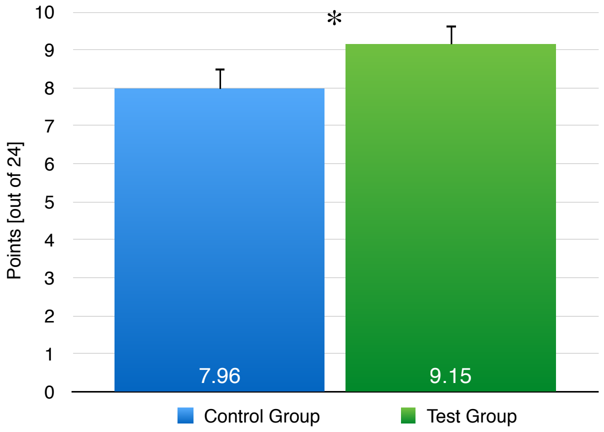


Figure A.3: Mean over students of the sum of scores on all of the relevant questions. A maximum of 24 points could be earned. Error bars display one standard error of the mean. The asterisk ($*$) indicates that the difference in mean scores is statistically significant at the $\alpha = 0.10$ level.

We performed a standard analysis of variance (ANOVA) of these data, using group and question as factors. This analysis revealed a marginally significant effect of group membership, with the group making use of our automated grading system receiving higher aggregate scores ($F(1, 280) = 96.5$; $p = 0.066$). We also conducted planned two-tailed t-tests for each of the four relevant final examination questions, assessing the impact of our automated grading system on student performance on each question type.

The first question involved a simple translation of an English sentence into first-order logic. For example, students might be asked to translate the sentence: "A block can never be on top of another block that is smaller than it." For this final examination question, we found a marginally significant benefit of use of our system ($t(280) = 1.929$; $p = 0.055$).

The second question addressed the representation of uniqueness. An example sentence would be: "There is exactly one block that is smaller than all of the others." Use of our system did not reliably influence performance on this question ($t(280) = 0.304$; $p = 0.761$).

The third question asked students to provide a definition for a predicate. Often, the question demanded the formulation of a recursive definition. For example, students might be asked to provide a definition for a simple blocks-world predicate like $Above(x, y)$ when given a predicate like $On(x, y)$. A sample solution would be:

$$\forall x \forall y \; Above(x, y) \Leftrightarrow On(x, y) \vee (\exists z \; On(x, z) \wedge Above(z, y))$$

Use of our automated grading system produced a reliable increase in scores for this question ($t(280) = 2.077$; $p = 0.039$).

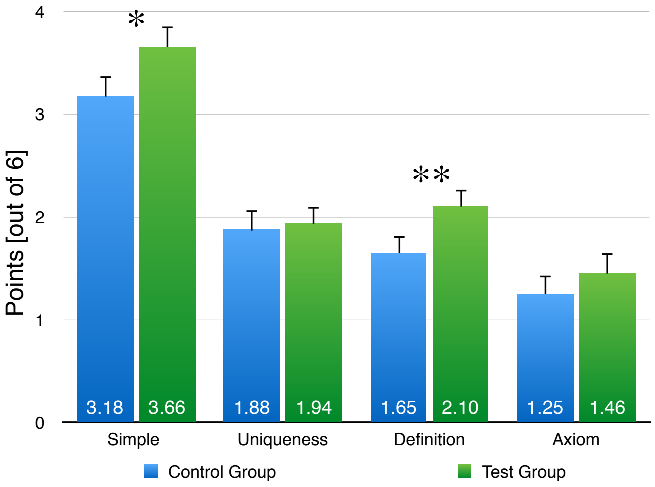


Figure A.4: Mean scores for each question type. The maximum possible score for each question was 6 points. Error bars display one standard error of the mean. An asterisk ($*$) indicates that the difference in mean scores is statistically significant at the $\alpha = 0.10$ level, and a double asterisk ($**$) marks significance at the $\alpha = 0.05$ level.

Finally, the fourth question required students to write a successor-state axiom for a given fluent using the situation calculus. Completing practice exercises using our system had no detectable impact on scores for this question ($t(280) = 0.856$; $p = 0.393$). The mean scores for each question are shown in Figure A.4.

It is worth noting that most of the exercises presented by our online system were similar to the first examination question, described above. A small number of exercises dealt with uniqueness, and there were no definition or successor-state axiom questions in the system. (Examples of definition sentences and successor-state axioms were discussed during class lectures, but the automated grading system offered no additional practice on these kinds of questions.) This observation suggests that practice on the first kind of question actually transferred to definition questions.

A.4 Conclusion

Undergraduate students of artificial intelligence regularly experience difficulties with knowledge representation exercises. This is evident in the low mean scores on relevant final examination questions that we have reported for our AI class. Reasons for this difficulty may include the limited amount of lecture hours devoted to first-order logic knowledge representation examples and students' relative inexperience with the subject matter.

We have developed an automated assessment system for exercises in which students are asked to translate English sentences into first-order logic. Our objective was to give students more practice with knowledge representation, which would lead to improved performance on final examination questions. We deployed the system in our AI class, and it has been in use over the last three instantiations of the course. An analysis of examination scores shows a statistically significant improvement in the performance of students who have used our system.

Acknowledgement

This chapter is largely based on Kyrilov and Noelle (2015b).

Appendix B

Data Sets

B.1 Honest Sequence Lengths

B.1.1 Data

Student	Year	Length
5	2016	4.2500
6	2016	3.1429
7	2016	6.2273
8	2016	18.0000
9	2016	4.3000
10	2016	3.6000
11	2016	3.5833
12	2016	7.4211
13	2016	4.5000
14	2016	3.0000
15	2016	3.0000
16	2016	3.6667
17	2016	7.8000
18	2016	3.7000
19	2016	2.8571
20	2016	5.6000
21	2016	5.2000
22	2016	3.4000
23	2016	2.4286
24	2016	6.5263
25	2016	7.0769
27	2016	6.0000
28	2016	3.2308
29	2016	5.0833
30	2016	4.1429
31	2016	3.7500
32	2016	4.8000
33	2016	4.2000
34	2016	2.5714
35	2016	5.5000
36	2016	2.5000
37	2016	3.5714
38	2016	3.0909
39	2016	4.0000
40	2016	6.7778
41	2016	5.3333
42	2016	4.6000
43	2016	3.6250
44	2016	3.8333
45	2016	4.3333
46	2016	3.0000
47	2016	4.0000
48	2016	4.5000
49	2016	9.2174
50	2016	4.2857
51	2016	7.3077
52	2016	4.2500
53	2016	5.2353

Student	Year	Length	Student	Year	Length	Student	Year	Length
54	2016	3.2222	87	2016	3.9167	1028	2017	3.9000
55	2016	4.5714	88	2016	6.3000	1029	2017	2.8667
56	2016	2.9375	89	2016	4.1250	1030	2017	4.9286
57	2016	5.3750	90	2016	4.4375	1031	2017	2.6250
58	2016	7.3333	91	2016	4.5500	1032	2017	3.3333
59	2016	3.3684	92	2016	3.6667	1034	2017	3.0000
60	2016	5.8571	93	2016	2.0000	1035	2017	3.5385
61	2016	6.3684	94	2016	4.3333	1036	2017	2.4444
62	2016	6.4375	1005	2017	5.0909	1037	2017	3.0000
63	2016	5.2778	1006	2017	3.0769	1038	2017	3.8750
64	2016	3.3333	1007	2017	3.7143	1039	2017	3.8333
65	2016	2.7857	1008	2017	2.9000	1040	2017	4.3889
66	2016	4.5000	1009	2017	2.2857	1041	2017	5.0714
67	2016	3.5000	1010	2017	9.0000	1042	2017	4.2500
68	2016	5.9286	1011	2017	3.2222	1043	2017	2.5000
69	2016	11.7619	1012	2017	3.3636	1044	2017	2.7500
70	2016	6.8571	1013	2017	4.7857	1045	2017	2.1818
71	2016	5.0000	1014	2017	5.3636	1046	2017	2.0000
72	2016	3.2000	1015	2017	2.1250	1047	2017	4.3750
73	2016	3.5556	1016	2017	3.0000	1048	2017	3.2778
74	2016	2.2500	1017	2017	4.6923	1049	2017	3.5882
75	2016	4.3333	1018	2017	2.6000	1050	2017	3.4286
76	2016	3.6667	1019	2017	3.1429	1051	2017	3.2778
78	2016	2.4545	1020	2017	5.3158	1052	2017	3.0000
79	2016	4.6667	1021	2017	3.0000	1053	2017	3.0000
81	2016	4.5000	1022	2017	2.8182	1054	2017	2.9444
82	2016	4.3750	1023	2017	4.4286	1055	2017	3.8125
83	2016	7.0000	1024	2017	3.5833	1056	2017	3.9048
84	2016	9.0000	1025	2017	2.4000	1057	2017	4.0000
85	2016	3.0000	1026	2017	3.5000	1059	2017	3.2000
86	2016	4.0000	1027	2017	3.2000	1060	2017	5.8500

Student	Year	Length	Student	Year	Length
1061	2017	3.7500	1092	2017	3.8667
1062	2017	3.0000	1093	2017	4.0000
1063	2017	2.0000	1094	2017	4.1053
1064	2017	8.8824	1096	2017	4.4737
1065	2017	2.2000	1097	2017	3.0714
1066	2017	2.7692	1098	2017	2.7778
1067	2017	2.2500	1099	2017	3.7222
1068	2017	2.6667	1100	2017	2.6154
1069	2017	3.2727	1101	2017	3.8750
1070	2017	2.6154	1102	2017	2.7143
1071	2017	4.0000	1103	2017	3.6429
1072	2017	3.3333	1104	2017	3.0000
1073	2017	2.9000	1105	2017	3.5833
1074	2017	3.3333	1106	2017	4.2857
1075	2017	2.8571	1107	2017	3.7500
1076	2017	3.7143	1108	2017	6.2609
1077	2017	4.6154	1109	2017	4.5000
1078	2017	2.8000	1110	2017	2.3750
1079	2017	4.2222	1111	2017	10.3182
1080	2017	2.5455	1112	2017	3.5385
1081	2017	3.4545	1113	2017	5.3077
1082	2017	3.2500	1114	2017	3.0833
1083	2017	4.0000	1115	2017	3.4000
1084	2017	3.2941	1116	2017	3.5625
1085	2017	2.9375	1117	2017	4.5600
1086	2017	2.8000	1119	2017	3.3125
1087	2017	3.5714	1120	2017	3.2222
1088	2017	2.6000	1121	2017	4.1250
1089	2017	5.5833	1122	2017	3.0000
1090	2017	2.3125	1123	2017	3.3750
1091	2017	2.3750	1124	2017	4.4444

B.1.2 ANOVA Results

```
SOURCE: grand mean
year       N       MEAN         SD         SE
         203     4.1197     1.8382     0.1290

SOURCE: year
year       N       MEAN         SD         SE
2016      87     4.7789     2.2290     0.2390
2017     116     3.6253     1.2822     0.1191

FACTOR  : student_id       year     length
LEVELS  :        203          2        203
TYPE    :     RANDOM    BETWEEN       DATA

SOURCE                SS     df             MS         F      p
===============================================================
mean        3445.2726      1      3445.2726  1123.534  0.000 ***
s/y         616.3588    201          3.0665

year             66.1666      1        66.1666    21.578  0.000 ***
s/y         616.3588    201          3.0665
```

B.2 Rates of Plagiarism

B.2.1 Data

Year	Exercise	Cheated
2016	1	15
2016	2	0
2016	3	0
2016	4	2
2016	5	3
2016	6	3
2016	7	0
2016	8	2
2016	9	0
2016	10	7
2016	11	7
2016	12	11
2016	13	7
2016	14	5
2016	15	10
2016	16	13
2016	17	13
2016	18	5
2016	19	16
2016	20	28
2016	21	46
2016	22	34
2016	23	18
2016	24	31
2016	25	17
2016	26	19
2016	27	20
2016	28	20
2016	29	26
2016	30	37
2016	31	30
2017	1	2
2017	2	0
2017	3	0
2017	4	0
2017	5	0
2017	6	2
2017	7	2
2017	8	0
2017	9	2
2017	10	2
2017	11	6
2017	12	8
2017	13	2
2017	14	4
2017	15	2
2017	16	4
2017	17	2
2017	18	2
2017	19	0
2017	20	2
2017	21	26
2017	22	28
2017	23	6
2017	24	10
2017	25	8
2017	26	16
2017	27	5
2017	28	8
2017	29	14
2017	30	22
2017	31	0

B.2.2 ANOVA Results

```
SOURCE: grand mean
year        N       MEAN         SD          SE
           62    10.1613    11.0545      1.4039

SOURCE: year
year        N       MEAN         SD          SE
2016       31    14.3548    12.3977      2.2267
2017       31     5.9677     7.6441      1.3729

FACTOR  :   exercise      year      total
LEVELS  :         31         2         62
TYPE    :     RANDOM    WITHIN       DATA

SOURCE                SS      df             MS          F      p
===============================================================
mean        6401.6129       1      6401.6129    35.908  0.000 ***
e/        5348.3871     30       178.2796

year            1090.3226       1      1090.3226    32.205  0.000 ***
ye/         1015.6774     30         33.8559
```