University of Massachusetts Amherst

ScholarWorks@UMass Amherst

Doctoral Dissertations | Dissertations and Theses

November 2019

Tools for Tutoring Theoretical Computer Science Topics

Mark McCartin-Lim
Wabash College

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

Part of the Artificial Intelligence and Robotics Commons, Graphics and Human Computer Interfaces Commons, Other Computer Sciences Commons, and the Theory and Algorithms Commons

Recommended Citation

McCartin-Lim, Mark, "Tools for Tutoring Theoretical Computer Science Topics" (2019). *Doctoral Dissertations*. 1797.
https://scholarworks.umass.edu/dissertations_2/1797

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

TOOLS FOR TUTORING THEORETICAL COMPUTER SCIENCE TOPICS

A Dissertation Presented

by

MARK MCCARTIN-LIM

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2019

College of Information and Computer Sciences

© Copyright by Mark McCartin-Lim 2019

All Rights Reserved

TOOLS FOR TUTORING THEORETICAL COMPUTER SCIENCE TOPICS

A Dissertation Presented

by

MARK MCCARTIN-LIM

Approved as to style and content by:

Andrew McGregor, Co-chair

Beverly Woolf, Co-chair

David Mix Barrington, Member

Siman Wong, Member

James Allan, Department Chair
College of Information and Computer Sciences

DEDICATION

To the students who faithfully came to my office hours, whose struggles and perseverance inspired this dissertation.

To my good friend Lucas, who provided much needed moral support during the hardest times, and who gave me the courage to choose this dissertation topic.

To my parents, who patiently waited for me to finish my Ph.D., and supported me, often financially, during that time.

To Marina, who convinced me to keep going when I was almost about to quit.

ACKNOWLEDGMENTS

I acknowledge my long and challenging journey to produce this dissertation required assistance and encouragement from many people. I want to thank them.

My sincere thanks goes to my co-advisors Beverly Woolf and Andrew McGregor for their guidance, help, and forbearance with my journey. Professor McGregor was my advisor when I started my Ph.D. program in 2009 working on research projects under his supervision. When I decided to do my own research on intelligent tutoring systems in 2012, Professor McGregor was willing to remain a co-advisor to me even though his experience with intelligent tutoring systems was limited. He always found time from his busy schedule, even during his sabbatical leave, to help me when I needed it. He generously gave his vast knowledge, wisdom, and guidance to enable me to complete my journey. My interest in intelligent tutoring systems developed with my attendance of Professor Woolf's course on the subject. Her course inspired me to create the computerized tutoring system that is in my dissertation. Professor Woolf became my co-advisor in 2012. She used her expertise to help me develop the cumbersome protocol required by the IRB (Institutional Review Board) for the experimental testing of my tutoring system. Professor Woolf also advised me how to deal with the many time consuming issues imposed by the IRB. My journey could not be completed without her help and guidance.

I greatly appreciate the participation of Professors David Barrington, Robert Moll, and Simon Wong, in addition to Andrew McGregor and Beverly Woolf, as members of my review committee. Professor Barrington pointed me to the work of Gries and Schneider which I referenced in my dissertation. Professor Moll loaned me a book he co-authored which I cited in my dissertation. Professor Moll is no longer on the

committee because he has retired. The guidance provided by the committee has facilitated my journey.

I thank Professors Barrington, McGregor, and Krishnamurthy for providing the opportunities and assistance for me to conduct my experimental testing in conjunction with portions of the courses they taught. The experimental testing is a critical element of my dissertation.

I am grateful to Alistair Sinclair for motivating me to start my journey. Professor Sinclair took me under his wing when I was a visiting undergraduate student at the University of California, Berkeley in 2005. Professor Sinclair gave me one on one mentoring sessions every week for more than a year. He exposed me to the joys of making discoveries through research, and convinced me to pursue a Ph.D. degree in computer science.

I am also grateful to many colleagues, friends, family members, and my parents for their advice and encouraging messages. They gave me the morale boosting sustenance I needed to complete my long and challenging journey.

ABSTRACT

TOOLS FOR TUTORING THEORETICAL COMPUTER SCIENCE TOPICS

SEPTEMBER 2019

MARK MCCARTIN-LIM

B.S., UNIVERSITY OF CALIFORNIA, SANTA BARBARA

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Andrew McGregor and Professor Beverly Woolf

This thesis introduces Complexity Tutor, a tutoring system to assist in learning abstract proof-based topics, which has been specifically targeted towards the population of computer science students studying theoretical computer science. Existing literature has shown tremendous educational benefits produced by active learning techniques, student-centered pedagogy, gamification and intelligent tutoring systems. However, previously, there had been almost no research on adapting these ideas to the domain of theoretical computer science. As a population, computer science students receive immediate feedback from compilers and debuggers, but receive no similar level of guidance for theoretical coursework. One hypothesis of this thesis is that immediate feedback while working on theoretical problems would be particularly well-received by students, and this hypothesis has been supported by the feedback of students who used the system.

This thesis makes several contributions to the field. It provides assistance for teaching proof construction in theoretical computer science. A second contribution is a framework that can be readily adapted to many other domains with abstract mathematical content. Exercises can be constructed in natural language and instructors with limited programming knowledge can quickly develop new subject material for Complexity Tutor. A third contribution is a platform for writing algorithms in Python code that has been integrated into this framework, for constructive proofs in computer science. A fourth contribution is development of an interactive environment that uses a novel graphical puzzle-like platform and gamification ideas to teach proof concepts. The learning curve for students is reduced, in comparison to other systems that use a formal language or complex interface.

A multi-semester evaluation of 101 computer science students using Complexity Tutor was conducted. An additional 98 students participated in the study as part of control groups. Complexity Tutor was used to help students learn the topics of NP-completeness in algorithms classes and prepositional logic proofs in discrete math classes. Since this is the first significant study of using a computerized tutoring system in theoretical computer science, results from the study not only provide evidence to support the suitability of using tutoring systems in theoretical computer science, but also provide insights for future research directions.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

ABSTRACT vii

LIST OF TABLES xv

LIST OF FIGURES xvi

CHAPTER

1. INTRODUCTION **1**

1.1 Challenges in teaching theoretical topics 3

1.1.1 Student-centered pedagogy versus traditional lecture formats 3
1.1.2 Abstract concepts disconnected from other computer science courses 6

1.2 Insights on meta-skills of students who successfully learn theoretical computer science concepts 7

1.2.1 Acquiring and applying factual knowledge 7
1.2.2 Translating informal intuition into formal reasoning 8
1.2.3 Understanding notation and semantics 9
1.2.4 Applying analogy / proof schema 11
1.2.5 Generating examples / inductive reasoning 11

1.3 Components of Complexity Tutor 14
1.4 Potential advantages of Complexity Tutor 15

2. RELATED WORK USING COMPUTERS TO AID IN TEACHING THEOREM PROVING **17**

2.1 Pedagogical issues with automated proof verifiers 17

2.1.1 Formal logical systems 18
2.1.2 Differences between proofs constructed from formal logical systems and informal narrative proofs 20
2.1.3 Can informal narrative proofs be substituted by those constructed from a formal logical system? 21

2.1.3.1 Assertion level reasoning 23

2.1.4 Consideration of proof granularity 25

2.2 A brief introduction to automated theorem proving 27

2.2.1 Resolution-based theorem provers 29
2.2.2 Peter B. Andrews' theory of matings 32
2.2.3 More references on automated theorem proving 34

2.3 Theories of how humans naturally reason 34
2.4 Survey of systems and guiding principles 37

2.4.1 EXCHECK and early work from Patrick Suppes' group 38
2.4.2 The EPGY Theorem Proving Environment 40
2.4.3 Cognitive tutors for geometry theorem proving 42

2.4.3.1 ACT theories of cognition 43
2.4.3.2 Anderson's first tutor for geometry theorem proving 45
2.4.3.3 A New Geometry Learning Enviornment (ANGLE) 47

2.4.4 Automated Proof Search (AProS) 49
2.4.5 The Dialog project and Ωmega-Tutor 51
2.4.6 Deep Thought 55
2.4.7 Gries and Schneider's formal logical system for computer science 57

3. ARCHITECTURE AND SYSTEM DESIGN CHOICES 60

3.1 Theorem Proving Environment 60

3.1.1 Proof Space 60
3.1.2 Assumptions Box 62
3.1.3 Assertions Box 62
3.1.4 Demonstration of solving a simple proof problem 63

3.1.4.1 Choosing assumptions and assertions 64
3.1.4.2 Justifying assertions 64

3.1.4.3 Completing the proof 66

3.1.5 Comparison to standard proof writing exercises 67

3.1.5.1 Proofs as graphs 68

3.1.6 Requirements for domain knowledge representation 72
3.1.7 New features added to the Theorem Proving Environment after preliminary experimental testing 73

3.1.7.1 Visual hint mechanism for coarse-grained inferences 73
3.1.7.2 Search Box for finding assertions 78

3.2 Algorithm Environment 82

3.2.1 Using the Algorithm Environment to tutor NP-completeness reductions 83
3.2.2 Using Levin reductions rather than Karp or Cook reductions 88

3.2.2.1 Technical motivation for using Levin reductions 92

3.2.3 How the Algorithm Environment works under the hood 94

3.2.3.1 Limitations of the black-box testing framework and alternatives 97

3.3 Authoring new problems 98
3.4 Developing a complete intelligent tutoring system 101

4. DESIGN OF EXPERIMENTS 104

4.1 General study procedures 104

4.1.1 Soliciting volunteers 104
4.1.2 Logistics of the experiments 107
4.1.3 Procedures for the experimental group 109
4.1.4 Procedures for the control group 111
4.1.5 Evaluation of submissions and follow-up 112

4.2 NP-completeness tutoring experiments 113

4.2.1 Fall 2016 115
4.2.2 Spring 2018 117

4.3 Prepositional logic proof tutoring experiments 121

4.3.1 Spring 2017 123
4.3.2 Fall 2017 125

4.4 Limitations of the study 128

5. EXPERIMENTAL RESULTS AND ANALYSIS 131

5.1 Evaluation of the Theorem Proving Environment 131

5.1.1 Theories to explain differences between the results for the logic and NP-completeness experiments 132
5.1.2 For the NP-completeness experiments, did the Theorem Proving Environment correct students' misconceptions? 138
5.1.3 Analysis of the relationship between performance on the practice problems and exam improvement in the logic experiments 146

5.1.3.1 Second analysis with Goodman and Kruskal's γ 154
5.1.3.2 Analysis of Fall 2017 results 155

5.2 Evaluation of the Algorithm Environment 158

5.2.1 Evaluation of interactions with the Algorithm Environment in the Fall 2016 experiment 160

5.2.1.1 Did programming ability or prior familiarity with Python affect the results? 160
5.2.1.2 Were there any programming errors not identified by the Algorithm Environment? 165
5.2.1.3 Summary of findings about substituting Python programming for writing pseudocode 168
5.2.1.4 Why the Algorithm Environment did not help students successfully produce NP-completeness reductions in Fall 2016 168

5.2.2 Scaffolding of hints for the BIN-PACKING Reduction Problem 175
5.2.3 Improvement in Spring 2018 results 178

5.3 Feedback about Complexity Tutor 180

5.3.1 Would the questionnaire responses reflect the sentiment of students who did not participate in the study? 181
5.3.2 What students liked about Complexity Tutor 183
5.3.3 Constructive criticism from students 185

5.3.4 Was the hint-line feature beneficial to students or not? 187

5.4 Conclusion of study results 189

6. FUTURE WORK 192

6.1 Correcting student misconceptions 194

6.1.1 Delayed Feedback Mode 194

6.1.1.1 Minimal Feedback 195
6.1.1.2 Graphical Error Report 196

6.1.2 Debug Mode 197
6.1.3 Find-the-Bug Mode 197
6.1.4 Freestyle Mode 197

6.2 Developing a graphical interface that can represent a wide range of proof strategies 198

6.2.1 Interface challenges with using subproof boxes in the Proof Space 199
6.2.2 Alternative possible subproof representations 202

6.2.2.1 Labeling assertions to denote the subproof they belong to 202
6.2.2.2 Using a separate Proof Space for each subproof 202

6.2.3 How proof strategies would be applied in the Theorem Proving Environment 204

6.3 Other general interface issues 205
6.4 Inductive Example Construction Aid 206
6.5 Hypothesis about problem description types 207
6.6 Using machine learning to develop a student model 209
6.7 Automation of problem generation 212
6.8 Going beyond Levin reductions 213

APPENDICES

A. MATERIALS USED TO PRESENT AND DIRECT EMPIRICAL STUDY 215
B. NP-COMPLETENESS PROBLEMS USED IN COMPLEXITY TUTOR EXPERIMENTS 240

C. LOGIC PROBLEMS USED IN COMPLEXITY TUTOR EXPERIMENTS **243**
D. EXAMS QUESTIONS USED IN EMPIRICAL STUDY **246**
E. POSTTEST QUIZZES **253**

BIBLIOGRAPHY **258**

LIST OF TABLES

Table **Page**

4.1 Participation in study. 105

5.1 Summary statistics for the prepositional logic problems 133

5.2 Summary statistics for the conceptual NP-completeness problems 134

5.3 Results of Spring 2018 posttest quiz, questions 1–4 139

5.4 Results of Spring 2018 posttest quiz, question 5 140

5.5 Pearson correlations between individual practice problems and exam improvement in Spring 2017. 153

5.6 Summary statistics for the NP-completeness reduction problems 159

5.7 Did familiarity with Python affect a subject's willingness to attempt using the Algorithm Environment? 163

5.8 Did you find the Complexity Tutor tutoring system helpful in your learning how to construct proofs? 180

5.9 Do you think the Complexity Tutor tutoring system trains you to be meticulous (careful about not skipping or overlooking obvious or evident assertions) when you construct proofs? 181

5.10 Do you think the Complexity Tutor tutoring system helps you to develop the skills needed to construct proofs when you use the traditional pen and paper method for proof construction? 181

5.11 Do you want the Complexity Tutor tutoring system to be available in other courses with proof construction? 182

5.12 Would you recommend the Complexity Tutor tutoring system to others learning proof construction? 182

LIST OF FIGURES

Figure **Page**

1.1 Two flowcharts comparing how computer science students may learn from their programming assignments and theory assignments. 5

1.2 An example NP-completeness problem, where students are asked to reduce the language PARTITION to the language KNAPSACK. 12

2.1 Wason selection task with abstract theme, from [45]. 35

2.2 Wason selection task with familiar social theme, from [45]. 36

2.3 Euclidean geometry proof in the EPGY Theorem Proving Environment 41

2.4 Screenshot of Anderson's first geometry tutor, from his book [9] 46

2.5 Screenshot of proof tutoring environment in ActiveMath, from Marvin Schiller's Ph.D. thesis [136]. 55

2.6 Screenshot of Deep Thought, from the Game2Learn Lab website [21]. 56

3.1 Screenshot of completed Pizza Proof Problem in the Theorem Proving Environment. The Assumptions Box is on the top left, the Assertions Box is on the bottom left, and the Proof Space is on the right. 61

3.2 Status indicators used in the Theorem Proving Environment 62

3.3 Moving assumptions and assertions into the Proof Space, assumptions are indicated by complete dots and assertions are indicated by partial dots. 65

3.4 Connecting the dots, validated assertions become complete dots, toggling text and relocating dots. 66

3.5 Screenshot of a student producing a low-grained proof sketch in the Theorem Proving Environment, to show that 0/1-PROG is NP-Hard. 74

3.6 Hint-lines help to refine a proof sketch into a finer-grained proof. 75

(a) Initial proof sketch 75

(b) First step of refinement 75

(c) Second step of refinement 75

(d) Third step of refinement 75

(e) Forth step of refinement 75

(f) Fifth step of refinement 75

3.7 Screenshot of a symbolic formula being entered in the Search Box. Assertions in the Assertions Box are ordered by their relevance to what has been typed. 79

3.8 Screenshot of partially completed NP-completeness proof in the Theorem Proving Environment. The problem asks students to prove that BIN-PACKING is NP-complete. Assertions “8” and “9” remain unjustified. 84

3.9 Screenshot of the Algorithms Environment. Students are to write Python code for `Reduce_Partition_to_BinPacking` and `Cert_Partition_to_BinPacking`, to construct a reduction from PARTITION to BIN-PACKING. 86

3.10 An incorrect reduction. The student is given feedback about `Reduce_Partition_to_BinPacking` producing incorrect output. 87

3.11 A correct reduction from PARTITION to BIN-PACKING. 88

3.12 Screenshot of completed NP-completeness proof in the Theorem Proving Environment. Assertions “8” and “9” are now justified. 89

4.1 Proof graph of Murder Mystery Problem in Complexity Tutor. 122

5.1 Boxplot of exam improvement by group from Spring 2017. The variance for the control group was 47.5 and the variance for the experimental group was 34.4. 147

5.2 Normal Q-Q Plot of exam improvement scores from Spring 2017. 148

5.3 Normal Q-Q Plot of extra credit scores from Spring 2017. 149

5.4 Scatterplot of all subjects from Spring 2017. Overall, there was no statistically significant correlation between extra credit earned in the study and exam improvement, $r(66) = .07$, $p = .566$. 150

5.5 Scatterplot of experimental subjects from Spring 2017. For the experimental group, there was a mild positive correlation between extra credit earned in the study and exam improvement, $r(30) = .28$, $p = .120$. 151

5.6 Scatterplot of control subjects from Spring 2017. For the control group, there was a very weak negative correlation between extra credit earned in the study and exam improvement that is not statistically significant, $r(34) = -.13$, $p = .440$. 152

5.7 Scatterplot of control subjects from Fall 2017. For the control group, there was a weak negative correlation between extra credit earned in the study and exam improvement that is not statistically significant, $r(20) = -.25$, $p = .266$. 156

5.8 Scatterplot of experimental subjects from Fall 2017. For the experimental group, there was no statistically significant correlation between extra credit earned in the study and exam improvement, with the null hypothesis having a high probability of being true, $r(18) = .04$, $p = .880$. 157

5.9 Survey results on familiarity with Python amongst Fall 2016 subjects from the experimental group. 162

5.10 How long did it take for a programming error to be corrected once identified by the Algorithm Environment? 166

5.11 Error message in the Algorithm Environment indicating that a reduction for the BIN-PACKING Reduction Problem fails to pass the test case `[4,4]`. 172

6.1 Fitch-style diagram. 199

6.2 Graphical version of a Fitch-style diagram. Boxes indicate subproofs with their own scope. The dot with the disjunction symbol is used to indicate a split between the cases $x \in B$ and $x \in C$. 200

6.3 Alternative graphical representation of subproofs. The dots for each proof statement are labeled according to the subproof they belong to. In this example, there are three subproofs. Statements that belong to the outer subproof get 'A' labels. Statements that belong to the two inner subproofs get 'B' and 'C' labels respectively. The dot '10' is outside all the subproofs. 203

CHAPTER 1
INTRODUCTION

Limited mathematical proficiency is a barrier for many computer science students to learning core topics such as automata, algorithms, computability and computational complexity. These topics form the theoretical foundation of the field. Some computer science educators worry that their curriculum has become "math phobic" over time, removing much of the mathematical and theoretical content that students struggle with instead of encouraging students to understand and appreciate the mathematical foundations of computer science [158]. This phenomenon may be particularly pronounced in the United States—an international study of 500,000 software developers found that while American developers are stronger than their Chinese counterparts in programming skills, they lag, comparatively, in general math skills [156].

However, it is noticed that computer science students elsewhere have difficulty with theoretical concepts as well. In Finland, at the University of Joensuu, it was reported that over the course of several years of teaching a theory of computation course, only at most a third of the students who registered for the course ever passed it, and many had to repeat the course several times before they could pass it [70].

This dissertation focuses on addressing a particular Achilles' heel hindering learning outcomes in theoretical computer science courses—proofs. While proofs are the lingua franca for mathematical and theoretical computer science, many students—even those in upper-year courses—find it difficult to discern and produce rigorous proofs. Likewise, computer science instructors lament the difficulty of teaching proof construction [65, 97].

But even though computer science students may feel that proofs are alien to them, the author of this dissertation surmises that proof construction and programming depend on similar cognitive abilities. After all, proofs and programs share a lot in common. Both involve applying strict rules in a sequence to reach a desired outcome. Both use modular techniques to organize their structure—proofs use lemmas and theorems, while programs use functions and classes. Even proof tactics often have analogies in programming. For instance, it has been argued that proof by induction and recursion are two sides of the same coin [3].

Why then are so few computer science students able to transfer their skill in programming to proof construction? The author surmises that pedagogical challenges in teaching theoretical computer science topics are the main reason that students struggle with learning those topics, and with learning proof construction in particular. His investigation of those challenges has led to the development of COMPLEXITY TUTOR, a computerized tutoring system.

COMPLEXITY TUTOR gets its name from the author's endeavor to make NP-completeness and intractability, cornerstone concepts from computational complexity theory, more accessible to undergraduates. In the preface to a popular theoretical computer science textbook [75], it is noted that even at Stanford University—widely regarded as having one of the top Computer Science departments in the world—many of their incoming graduate students have a poor understanding of NP-completeness.

There are two components to COMPLEXITY TUTOR. The first is the Theorem Proving Environment, which by itself is versatile enough to aid in teaching many topics involving mathematical proofs, not just those in computer science. For instance, it could be used for set theory, point-set topology, abstract algebra and real analysis—these subjects are not the focus of this dissertation but the author has explicitly designed COMPLEXITY TUTOR so that it would be easy for an instructor to adapt it to that purpose.

The second component is more specific to the proofs one encounters in theoretical computer science, which tend to be *constructive proofs* that present algorithms as part of the proof. This is the Algorithm Environment, which is used to help students produce *reductions* for NP-complete problems. In the NP-completeness proofs one is likely to encounter in an undergraduate course, not only is the *reduction* itself algorithmic but the justification of the *reduction* is implicitly algorithmic as well.

1.1 Challenges in teaching theoretical topics

Here are two pedagogical issues to consider, which may explain the difficulty computer science students have learning theoretical material.

1.1.1 Student-centered pedagogy versus traditional lecture formats

Various student-centered teaching methodologies—sometimes referred to as *active learning*—have become a popular replacement for the traditional lecture format of teaching. These methodologies are supported by a theory called *constructivism*, which presupposes that learning occurs when learners actively construct knowledge rather than passively receive information [80]. Ideas from this school of thought can be traced back to Socrates, and were cemented by educational research done by John Dewey and Jean Piaget in the early 20th century. Examples of specific methodologies that follow this trend include *inquiry-based learning* and *problem-based learning*.

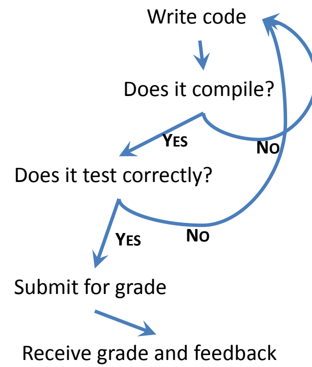
In the field of mathematics, the *Moore method* [81] refers to a specific way of teaching where the instructor proves nothing for the student, and students must learn the course material by proving everything on their own. It is named after the mathematician Robert Lee Moore, who used it to teach topology to his doctoral students, and became famous in the mathematical community with his teaching style producing 50 Ph.D. students and 3,739 doctoral descendants to date [119]. *Inquiry-based learning* methods used in mathematics are often referred to as *modified Moore*

methods, which share the central premise of Moore's teaching philosophy—that students be in the driver's seat as much as possible when they are learning mathematical theorems, actively discovering and developing the proofs of those theorems on their own. A multi-institutional study found that *inquiry-based learning* courses had better outcomes over lecture-based courses when teaching undergraduate mathematics [94].

However, it is difficult to scale *Moore's method* and *inquiry-based learning* methods to courses with large student enrollment, where time constraints make it unfeasible for every student to frequently present and receive feedback on proof problems they have attempted on their own. Typically, in such courses, lecture-based teaching becomes the primary method of instruction, and most important proofs that the instructor expects the student to understand are presented verbatim rather than having the student discover them on their own, as Moore would have done. Students may practice problem solving in their homework, but the feedback they receive is not likely to be timely nor extensive, and they are not given the opportunity to make more than a single attempt on any given problem. Thus, the student's ability to succeed in such a course critically hinges on their ability to rote learn from lectures and textbooks. This is a limitation that holds true for most courses in mathematics and theoretical computer science.

However, in many non-theoretical computer science courses, the situation is different. While those courses may still follow the traditional lecture format, it is often possible for self-directed learners to learn the material without even paying attention to the lectures or textbooks, simply by doing the homework and projects. The student's direct interaction with their computer naturally facilitates an *inquiry-based learning* model, as they learn from compiler errors and from debugging the output of the programs they write. Computer science students who are accustomed to this way of learning may feel particularly disenchanted in their theory courses—the problem sets in a theory course may require no more time and patience than the programming

How CS students learn from their programming assignments



How CS students learn from their theory assignments

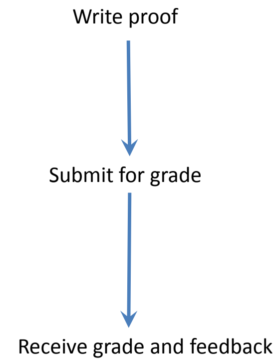


Figure 1.1: Two flowcharts comparing how computer science students may learn from their programming assignments and theory assignments.

projects in their other courses, but they often don't know if the time they spent on a problem set is going to pay off until after it has been graded since they get no feedback while working on it. Figure 1.1 illustrates this dichotomy.

One of the main benefits COMPLEXITY TUTOR will provide computer science students is that it gives immediate feedback to students while they are constructing proofs, just as they would expect when they are programming. This reduces the dependency of students learning from lectures and textbooks, and permits student-driven teaching methods even when course enrollment is large.

At the University of Joensuu, educators experimented with using a *problem-based learning* format to teach standard computing theory topics ranging from finite automata to computability theory, in place of the standard lecture format they had previously been using, and the results were dramatic. The drop-out rate for the *problem-based learning* version of the course was only 7% and 90% of the students

passed, compared to 50% dropping out and only a third passing when the course was taught in the traditional lecture format. Furthermore, enrollment was three times larger than in previous years where the course had only been offered in the traditional lecture format [70].

1.1.2 Abstract concepts disconnected from other computer science courses

An explanation given for why computer science students struggle more in their theoretical courses is that they find the material too abstract, and not closely related to what they are learning in other courses [164]. One approach to making abstract concepts like *automata* and *formal languages* more concrete is to introduce the student to simulation applets where they can directly interact with different kinds of automata like *finite automata*, *push-down automata*, *Turing Machines*, as well as explore parse trees generated by *formal languages* [164].

Another approach is to change the curriculum to recast the concepts of theoretical computer science in term of programming concepts that students would be familiar with, rather than the esoteric automata models that were originally used when the theoretical concepts were introduced. The textbook *A Programming Approach to Computability*, written by Kfoury, Moll and Arbib [86], takes this approach. It replaces all references of Turing Machines with meta-programs written in a language similar to Pascal. A benefit of this approach is that students will see that a seemingly foreign idea like that of a "universal Turing Machine" is in fact not conceptually different than the compilers they are already familiar with. Their book covers all the major results from computability theory using programming language concepts.

Neil Jones' *Computability and Complexity: From a Programming Perspective* [82] also takes this approach. Jones uses subsets of Pascal and LISP like programming

languages as basic models to teach computability theory. He then uses those same models to explain modern results in computational complexity.

Complexity Tutor is inspired by this idea, using a programming model of computation to train students in ideas of computational complexity like NP-completeness, but extends the concreteness further by giving students a real programming language to write their *reductions* in (Python) and evaluating them with a real compiler.

1.2 Insights on meta-skills of students who successfully learn theoretical computer science concepts

Based on the author's experience as a teaching assistant, the author conjectures that there are certain meta-skills that largely define a student's chance of success in theoretical computer science courses. These meta-skills are listed below.

1.2.1 Acquiring and applying factual knowledge

Given the problem, "Is the set of Real Numbers in P?", the answer can be derived just by applying a short chain of facts, "P is a subclass of the set of computable languages" and "computable languages are a subclass of countable sets" and "Real Numbers are not countable". Therefore, "Real Numbers are not in P". This problem would be considered easy by most instructors if given on an exam–it follows directly from facts and definitions given in the course. However, if a student forgot or never acquired one of the facts that they were taught early in the course, such as that computable languages are countable, they may get stuck on this problem and think that it is much harder to answer than it was intended to be. As a teaching assistant, the author discovered that students did not always retain knowledge they had learned earlier in the course, which would indeed cause them difficulty on these kinds of problems.

An interesting observation to make about the kinds of facts that one needs to acquire in the course of studying the theory of computation and complexity theory is that many of the facts are of the forms **X is a subclass of Y** or **x is a member of X**. This kind of information can be modeled with a Venn Diagram. In a review session, an exercise the author gave to students was to have them draw the relationships between all the computability and complexity classes they had learned about as a Venn Diagram. Students mentioned that this was very helpful as it helped them synthesize and obtain a big picture of all the knowledge they had previously been taught.

1.2.2 Translating informal intuition into formal reasoning

One of the biggest challenges in teaching advanced theoretical computer science courses is that many of the proofs the instructor gives students would be too long if written out formally with every single little detail. For instance, using formal logic to prove the existence of Universal Turing Machines requires writing up the exact description for the construction of a UTM, and then proving that every component of that constructions works the way it is supposed to. Such a laborious proof can be found in [33], where it takes up an entire chapter, as well as in Turing's original paper on the subject [159].

In a typical theory of computation course, with a large number of topics to cover, there is not enough time to cover such detailed proofs. Also, since those details are repetitive and not very insightful, they would not engage most students. Instead, what instructors want to do is impart a high-level understanding of the proof--to give an informal argument or condensed proof that contains enough information that the students would be able to reproduce the full proof on their own if they wanted to.

As one can imagine, that task is somewhat difficult when the class being taught is very diverse. Some students will require more details than others to reach a level

where they fully understand how a proof works. Also, some students will have a different intuition than the instructor does, so they may not understand the instructor's argument for that reason.

Another reason why an instructor would expect students to appeal to informal intuition in this course, is specifically because the instructor knows the students do not have a strong background in abstract mathematics.

The author recalls an instance where an instructor thought that students would find the abstract definitions in the textbook meaningless, and thus specifically chose to instead explain everything much more informally without ever referencing a definition. Unfortunately, this had the effect of confusing the students even more, because as far as they were concerned, what they were learning in the lectures was completely disconnected from the textbook.

The author found that the students often attempted to write their proofs in the same informal rhetoric used in the lectures, and ended up making a lot of logical fallacies that led to completely incorrect proofs. Math education researchers believe that confusion over the subtle differences between the rules of everyday discourse and those of logical discourse is often what causes students to make these errors [50].

The situation is likely to be even worse for non-native speakers of the language the class is being taught in, since they are likely to struggle with nuances of particular idioms that a native speaker would be more comfortable with.

1.2.3 Understanding notation and semantics

Many times, students do not even understand what a problem is asking them to do because they do not understand the notation being used in the problem. A student might be given the following problem on a homework assignment or exam:

> Is $\{(P, Q, k) \mid$ there are fewer than k natural numbers for which: $P(x)$, $Q(x)$ both halt and the output of P is at least twice the output of $Q\}$ decidable? Is it recognizable?

But many of the students will not have a good enough grasp of this so-called *set builder notation* to know the difference between the above problem and this one:

> For fixed natural number k, is $\{(P, Q) \mid$ there are fewer than k natural numbers for which: $P(x)$, $Q(x)$ both halt and the output of P is at least twice the output of $Q\}$ decidable? Is it recognizable?

The important difference is that in the second problem, k is interpreted to be some constant for every element in the language. In the first problem, it is not.

Additionally, the author has observed that students have trouble differentiating the semantic types used in theoretical computer science–*languages* (sets), *classes of languages* (collections of sets), *strings* (elements in languages), *alphabets*, *functions*, *algorithms* and *Turing Machines*.

So for instance, a student might mistakenly state that a given algorithm belongs to the class NP, which makes no sense.

However, the same students may have a good understanding of the semantic types used in programming languages, like Java—such as the difference between *primitive data structures*, *objects*, *classes*, *functions* and *literals*. They also probably recognize the difference between passing-by-value and passing-by-reference. These semantic concepts do not seem simpler than the ones used in the "language of mathematics".

A key difference is that while computer science students have been given ample opportunity to learn Java by writing code in it, they have not had much opportunity to write in the "language of mathematics", i.e., naive set theory. Furthermore, they do not have a compiler that interacts with them and tells them whenever they are making

even a slight mistake. The interaction the student has with their Java compiler is a form of dialogue, and this is very important for mastering any new language.

1.2.4 Applying analogy / proof schema

Once one has evaluated a lot of problems in a theoretical computer science course, patterns begin to emerge. The solutions to these problems, which are usually in the form of proofs, look very similar. Presumably, this is by design, because the goal of an introductory course is to make the students comfortable with some important abstract concepts and to learn some key theorems, not to test their ingenuity at coming up with brilliant proofs.

An example would be the standard "diagonalization argument" used to prove that a language is not computable.

One could imagine a database of "proof templates" given to students in a theory of computation course. There would be three steps required for a student to successfully apply a proof template to solving a problem.

First, they must correctly identify which proof template corresponds to the proof problem they are trying to solve. Second, they must apply the proof template to the specific problem, which generally means substituting variables specific to the problem into the template. Depending upon the proof template being used and the difficulty of the problem, this may in fact be enough to solve the problem.

If not, the final step is to add the missing details of the proof that are not contained in the template. In some problems, the template will only give the student a very rough outline of what needs to be done in the proof, and the student will have to add the rest. In other problems, the template practically is the solution itself.

1.2.5 Generating examples / inductive reasoning

Many of the problems encountered in theory of computation courses require the student to prove something about one or more formal languages. The author has

<table>
<tr><td colspan="2">Show that PARTITION $\leq_p$ KNAPSACK. In other words, show that if KNAPSACK has a polynomial time algorithm, so does PARTITION.</td></tr>
<tr><td>

<u>KNAPSACK</u>

Instance: (U, w, v, B, K) where U is a finite set of objects, and for each object u in U, $w(u)$ is the weight of the object and $v(u)$ is the value of the object, and where B and K are both natural numbers.

Question: Is there a subset $U' \subseteq U$ such that

$$\sum_{u \in U'} v(u) \geq K \text{ and } \sum_{u \in U'} w(u) \leq B$$

</td><td>

<u>PARTITION</u>

Instance: A set X of positive integers.

Question: Can you partition X into disjoint sets X_1 and X_2 such that:

$$\sum_{x \in X_1} x = \sum_{y \in X_2} y$$

</td></tr>
</table>

Figure 1.2: An example NP-completeness problem, where students are asked to reduce the language PARTITION to the language KNAPSACK.

found that a particular strategy, which is most effective when the solution does not immediately jump out, is to start by generating specific concrete examples from the language in consideration. Often times, it will be easier to reason about the specific concrete examples than the abstract language as a whole.

Another reason students should be in the habit of generating examples is that it is the only way they can really convince themselves that they understand the problem. If they cannot generate a concrete example for a given problem, it probably means that they still do not understand the problem well enough.

Consider the problem in Figure 1.2 to see how generating examples can help students find the solution.

Assuming one understands the general framework for reductions, a natural way one might try to find the solution to this problem is to first try to come up with a

specific positive instance of PARTITION and then try to translate that into a specific positive instance of KNAPSACK.

So, a student might consider $X = \{1, 2, 2, 3\}$, which is a positive instance of PARTITION, because if we partition it into $X_1 = \{1, 3\}$ and $X_2 = \{2, 2\}$ then $\sum_{x \in X_1} x = \sum_{y \in X_2} y = 4$.

Next, in thinking about how to translate this into an instance of KNAPSACK, the student conjectures that it's possible the instance variable X in PARTITION maps to one of the five instance variables in KNAPSACK (U, w, v, B, K). From those variables, it would seem that U might be the best candidate, because unlike the other variables, both X and U are set variables.

However, X and U correspond to different types of sets—one is a set of objects and the other is a set of positive integers. An integer cannot directly be mapped to an object, but each integer in X could correspond to a property of each object in U. As it so happens, KNAPSACK specifies that each object in U has two properties, weight and value, corresponding to the function variables w and u.

Out of these two possibilities, assume that the student hypothesizes that $w(U)$ maps to X. In other words, in the specific example the student is considering, $\{1, 2, 2, 3\}$ would correspond to the weights of objects in U.

Next, the student looks at the U' mentioned in the constraints of KNAPSACK. Based on the hypothesis that $w(U)$ maps to X, the student can deduce that $w(U')$ must map to some subset of X. Which subset? There are only two subsets of X mentioned in the description of PARTITION, X_1 and X_2. Both are equally good candidate hypotheses.

Assume the student hypothesizes that $w(U')$ maps to X_1. This implies that $\sum_{u \in U'} w(u) = \sum_{x \in X_1} x = 4$, from the example the student has chosen. Since one of the constraints of KNAPSACK is that $\sum_{u \in U'} w(u) \leq B$, it follows that $B \geq 4$. So the student hypothesizes that $B = 4$.

Now, there are only two remaining variables of the KNAPSACK instance that are still unspecified—v and K. And there is only one constraint that has to be satisfied for these two variables to create a positive instance of KNAPSACK, which is $\sum_{u \in U'} v(u) \geq K$. So far, the student's strategy has been to try to map unspecified variables to variables they have previously specified. If the student continues with this strategy, they will realize that if they set $v = w$ and $K = B$, then the constraint $\sum_{u \in U'} v(u) \geq K$ is satisfied.

Hence, the student now has a complete mapping from a positive instance of PARTITION to a positive instance of KNAPSACK. Generalizing, the hypothesized reduction defines U as a set of $|X|$ objects with $w(U)$ and $v(U)$ both mapping to X, and with $B = K = \frac{\sum_{x \in X} x}{2}$. The student can now check different positive and negative instances of PARTITION to see if the hypothesized reduction creates corresponding positive and negative instances of KNAPSACK.

The process used to develop this hypothesis may seem arbitrary, since a lot of good guesses had to made to get the right examples that would lead us to a good hypothesis for the reduction. In general, one will probably have to make a number of guesses before they get the right examples. So this whole process could be called "guess and check".

But if enough possibilities are considered, one should eventually find the right examples. In fact, this "guess and check" methodology is similar to how a computer algorithm would most likely solve this problem—checking as many examples as possible by brute force. One can only hope that the student will have better intuition than a computer does and thus not need to make as many guesses. It is hoped that the student's intuition will also get better over time by following this process.

1.3 Components of Complexity Tutor

Complexity Tutor has two components, described in detail in Chapter 3.

First, there is the Theorem Proving Environment, where students can construct proofs that are similar to the narrative proofs used in their classes. Proofs in the Theorem Proving Environment can be expressed in natural language, and there is no special syntax to learn. In this regard, COMPLEXITY TUTOR's Theorem Proving Environment is pedagogically more flexible than other tutoring systems that have been developed for theorem proving.

Second, since many proofs in computer science are constructive in nature and have an algorithmic component, there is the Algorithm Environment. Currently, this is being used for the construction of NP-completeness reductions.

1.4 Potential advantages of Complexity Tutor

The Theorem Proving Environment presents a novel way to learn to construct proofs, whereby students generate proofs by connecting assertions together in a puzzle-like game. The system can provide the following advantages to students and instructors:

- **The puzzle-like game structure should increase engagement.** Puzzles have a rich history of being used to teach mathematics [108], and more recently Puzzle-Based Learning has been adopted in computer science and engineering pedagogy [53]. Digital game-based learning has also been shown to effectively motivate learners in numerous fields [129].

- **Students are encouraged to be precise and meticulous.** COMPLEXITY TUTOR constrains students to only construct rigorous proofs. That way, students are guided away from the common pitfalls that frequently plague informal student proofs such as semantic ambiguity, insufficient detail, gaps in justification, and logical inconsistency.

- **Immediate feedback is provided to the student.** Traditional homework practice is non-ideal because it does not provide timely feedback for students to rectify learning deficiencies. It may be weeks before students get their corrected homework back, and some students may find themselves pushed to the next topic in the course before they have mastered the previous topic.

- **Homework can be graded automatically.** When Complexity Tutor is used to replace some or all of the traditional written homework, it reduces the amount of resources needed for manually grading student work. This could be a major benefit for teaching very large courses such as *massive open online courses* (MOOCs).

- **It is easy to adapt Complexity Tutor to a wide variety of contexts.** Instructors can easily adapt it to any topic that involves proof instruction. The simplicity of the interface makes it equally suitable for introducing logic problems to K-12 students or teaching highly abstract mathematics to college students.

CHAPTER 2

RELATED WORK USING COMPUTERS TO AID IN TEACHING THEOREM PROVING

This chapter surveys other software systems that have been used pedagogically in proof-based courses, mostly outside of computer science. Many of these systems could be considered *intelligent tutoring systems*, specifically designed to help students improve their understanding of course material. In other cases, the systems were designed primarily to help the instructors, such as by automating grading.

However, all the systems we will consider in this chapter share one thing in common—on the backend, they all use a rule-based automated proof verifier to model expert domain knowledge. As such, the pedagogical utility of each of these systems is limited by what its associated proof verifier can accomplish.

Uniquely, Complexity Tutor does not have this limitation, since it models expert domain knowledge in a very different way, not requiring an automated proof verifier for a given domain, but instead requiring problem-specific hypergraph structures to be pre-constructed. Complexity Tutor's domain knowledge model is explained in Section 3.1.6.

This chapter begins by introducing some of the issues that must be taken into consideration when using automated proof verifiers in an educational setting.

2.1 Pedagogical issues with automated proof verifiers

Artificial intelligence pioneer John McCarthy once wrote,

> "Checking mathematical proofs is potentially one of the most interesting and useful applications of automatic computers." [102]

Paul Abrahams, one of McCarthy's Ph.D. students, developed Proofchecker, one of the first automated proof verification tools. His dissertation [1] explains that he originally sought to develop a system that could verify ordinary textbook proofs but later realized that was too ambitious of a goal. Thus, Proofchecker was limited to only being able to verify proofs constructed in a formal logical system, where the proof is written in a precise technical language and each step of the proof follows from a limited set of rules.

Since then, numerous tools for automatically verifying and generating proofs have been developed, such as Coq [29], Mizar [113], and Isabelle [118]. But like Proofchecker, they are also all limited to verifying proofs constructed in formal logical systems.

2.1.1 Formal logical systems

Formal logical systems were originally developed by logicians for formalizing logic and mathematics. A formal logical system is defined by the following four components:

1. A finite set of *symbols*.

2. A *language*, which is a set of *formulas*. Each *formula* is a string constructed from the symbols. Typically, the *language* will be generated syntactically from a grammar, such as a context-free grammar.

3. A set of *axioms*, which is a subset of formulas in the language. The set of *axioms* may be infinite, but in such case will typically be generated syntactically from a finite set of *axiom schemata*, which are also described by grammars.

4. A finite set of *inference rules*. Each *inference rule* maps a sequence of one or more *premise* formulas from the language to a single *conclusion* formula from the language. Formally, each *inference rule* r would be a function of form $r : P_1 \times \ldots \times P_k \to L$ where L is the language, and $P_1, \ldots, P_k$ are subsets of L. Typically, r can be implemented with a simple deterministic algorithm.

A *proof* produced from this formal logical system is a finite sequence of *proof steps*, corresponding to a sequence of *justified assertion* formula sets, $A_1 \ldots A_n$.

A_1 is the set of *assumption* formulas, which contain the axioms and any other formulas that are to be assumed for the proof. For every $1 < i \leq n$, $A_i = A_{i-1} + \{\alpha\}$ where α is a conclusion of one of the inference rules, using some subset of A_{i-1} as premises. In the special case of A_n, we say that α is the *goal* of the proof.

Notice that two different formal logical systems can define the same *logic*. This occurs when both systems share the same language, and every set of assumptions in the shared language prove the exact same set of goals. For instance, *first-order logic* can be defined either by a *Hilbert-style system* (many axioms but only one inference rule) or a *natural deduction system* (many inference rules but no axioms).

It is also worth pointing out the close connection between formal logical systems and *term rewriting systems* [19]. Assuming that all the inference rules in a formal logical system are Turing computable functions, each of those inference rules can be composed of term rewriting rules, since term rewriting systems are Turing complete. Many logics are defined by formal logical systems that directly correspond to term rewriting systems, and special attention is paid to such systems by computer scientists, since there are results from the theory of term rewriting that are useful to automated theorem proving. For instance, the Knuth-Bendix procedure [20] can be used to prove equality results using a term rewriting system.

2.1.2 Differences between proofs constructed from formal logical systems and informal narrative proofs

Traditionally, mathematicians have used narrative to communicate proofs, whether by delivering a live talk with the aid of a blackboard, or by writing the narrative in a publication or textbook. As such, they also train their students to present proofs in narrative form.

There are two significant differences between the narrative proofs that are presented by mathematicians and their students in ordinary informal mathematical discourse, and the proofs constructed from formal logical systems:

1. The narrative proofs use natural language rather than a formal language to express assertions. Statements expressed in a natural language may be underspecified or ambiguous in terms of their precise meaning, unlike those produced from a formal language.

2. Inferences made in an ordinary narrative proof do not have to directly map to the limited set of inference rules found in any formal system. Since narrative proofs are intended specifically for the purpose of communicating proofs between humans, the inferences in a narrative proof will be considered valid if the human audience of the narrative agrees that the inference is correct.

In other words, both of these differences imply that narrative proofs lack a formal structure, and without that structure, intelligence must be used to fill in the gaps. Human intelligence is required to make a good judgment about whether a narrative proof should be considered correct or not. The ability to parse and discern the correctness of arbitrary narrative proofs is clearly an *AI-complete problem* [140], since in the general case, it is as hard as any other open-ended natural language problem.

Note that the problem of recognizing specific types of narrative proofs, such as those that you would likely find published in a textbook, might be more tractable.

Textbook proofs are written to be as clear and unambiguous to the reader as possible, and generally use very consistent language conventions. Even though Paul Abrahams did not succeed at developing a program to recognize textbook proofs, more recent work on this problem has yielded a small amount of progress. For instance, Donald Simon's Nthchecker [144] system was able to parse some proofs from a number theory textbook. Also, Claus Werner Zinn wrote a dissertation [173] on a theoretical framework for parsing the kind of narrative proofs one would find in a textbook.

However, the narrative proofs that students tend to write for class assignments are often not nearly as clearly written as textbook proofs. A student's narrative proof may be ambiguous, imprecise and at times even use terminology incorrectly. It then becomes a judgment call for the person evaluating the student's proof to determine if the student understood a correct proof but explained it inadequately, or if they did not even understand a correct proof at all. The author of this dissertation has himself many times struggled with this discernment when grading student proofs, and a machine is likely to have an even harder time at this task.

2.1.3 Can informal narrative proofs be substituted by those constructed from a formal logical system?

In principle, all informal proofs should map to proofs constructed from a single formal logical system, such as a system that uses either the Zermelo-Fraenkel set theory axioms or the von Neumann-Bernays-Gödel set theory axioms [150].

Why then don't mathematicians regularly construct their proofs in formal logical systems, since this would make automated verification easier?

There are two possible reasons to consider. First, while there may be a mapping from informal proofs to proofs in formal logical systems, the mapping is usually a very cumbersome one, where each assertion in the informal proof may map to a long sequence of formal proof steps. Second, a narrative proof can be used to add

commentary, explaining the intuition behind the proof, whereas a proof from a formal logical system cannot.

There are additional pedagogical considerations that must be taken into account when deciding whether it is appropriate to use informal narrative proofs or formal logical systems in teaching a proof-based course. Consider that a proof starts out as a cognitive construction—there is some abstract representation of it in the mind. That abstract representation is then translated into some medium that can be communicated. In the case of informal proofs, the medium is a narrative.

When a student is learning proofs, they are developing two skills simultaneously—both the skill of cognitively constructing the proof and the skill of communicating it in a given medium. The first skill is of primary importance, so it is desirable that the medium of the final communicated output align closely with the intended abstract representation in the mind, so as to avoid extraneous cognitive load.

Therefore, the chief pedagogical advantage of using narrative as a medium for proof communication is that its flexibility permits it to express proof ideas as conceived by the mind, with little overhead. Note though that narrative is not necessarily the only medium with this advantage or even the best one—a graphical medium may be even more likely to align with the cognitive construction of proofs than narrative will, as argued in Section 3.1.5.1. Note also that despite the lack of flexibility in formal logical systems, there is no reason to preclude the possibility of developing a formal logical system that aligns closely with how proofs are conceived in the mind.

In fact, for the domain of geometry, the cognitive model developed for Koedinger's ANGLE tutoring system (Section 2.4.3.3) can be thought of as a formal logical system that aligns with how geometry experts conceive of proofs.

Even without a cognitive model, pedagogical arguments can be made favoring formal logical systems over standard textbook narratives in particular domains, as David Gries and Fred Schneider do for their formal logical system (Section 2.4.7).

2.1.3.1 Assertion level reasoning

It is worth briefly mentioning one particularly interesting area of research on formal logical systems that looks at the construction of proofs that are structurally similar at an inference level to proofs one would find in a textbook, although not necessarily as close to student proofs. Xiaorong Huang observed [76] that there are three levels of verbal justification for inferences used in textbook proofs:

Logic level — justification is a simple logic rule from a natural deduction calculus.

Assertion level — justification involves applying an axiom, definition or theorem.

Proof level — justification is by analogy to something earlier in the proof.

Huang discovered that the assertion level justifications were the most common in the textbook proofs he studied, far outnumbering the times that textbook authors had to resort to explaining a proof step with either a lower-level justification from logic or a "proof level" justification. He developed a system called PROVERB, which could take a proof in a formal natural deduction system and shorten it to a proof that only uses *assertion level inferences* [76].

This work inspired other researchers to develop formal logical systems that simulate assertion level reasoning. Serge Autexier's *contextual reasoning* (CoRe) calculus [16] is an example of such a system. These assertion level reasoning systems can be thought of as "meta systems" because they have the following two properties:

1. There is not a fixed set of inference rules, as one would find in a classical formal logical system such as natural deduction. Instead, inference rules are automatically generated from definitions, theorems or axioms. A logical formula representing a definition, theorem or axiom is transformed using an underlying logical calculus into new inference rules.

2. The assertion level system is defined on top of a lower-level formal logical system, and the assertion level inference rules subsume several levels of inference rules in the lower-level system that are chained together. The application of the newly formed inference rules to a lower-level formal logical system has been referred to as *superdeduction* [35]. In the specific case where natural deduction is used as the lower-level system, the term *supernatural deduction* [166] was coined. More recently, *atomic metadeduction* [17] was introduced by Serge Autexier and Dominik Dietrich.

For instance, assume that composition of binary relations has been defined by the following higher-order logic formula:

$$\forall R, S, x, y : (x, y) \in R \circ S \iff \exists z : (x, z) \in R \wedge (z, y) \in S$$

In Autexier's CoRe calculus, this formula could generate two new inference rule schemata, corresponding to $(x, y) \in R \circ S \vdash (x, z) \in R, (z, y) \in S$ and its inverse, $(x, z) \in R, (z, y) \in S \vdash (x, y) \in R \circ S$. The application of these schemata is far looser than in a classical formal logical system—in the first schema, the precondition $(x, y) \in R \circ S$ would be matched against all subformula in a given premise formula, not just the premise itself. Such flexibility has been referred to as *deep inference* [68]. In CoRe, it is accomplished through *higher-order unification* [136].

Autexier based CoRe calculus on the "expansion proofs" that were originally used in TPS, Peter B. Andrews' theorem proving system for higher-order logic, mentioned in Section 2.2.2. CoRe calculus was implemented in the ΩMEGA$^{\text{CoRe}}$ *proof planning* system developed at Saarland University in Germany, and which has been used in intelligent tutoring research projects at that institution, described in Section 2.4.5.

2.1.4 Consideration of proof granularity

Consider the following two hypothetical narrative proofs that might come up in a class teaching NP-completeness, where a student is trying to prove that a language X is NP-hard, on the assumption that SAT $\leq_p X$:

Proof 1. Since SAT is NP-hard, every language $L \in$ NP is polynomial-time reducible to SAT. Since SAT is polynomial-time reducible to X, every language $L \in$ NP must also be polynomial-time reducible to X. Therefore, X is NP-hard.

Proof 2. SAT is NP-hard and polynomial-time reducible to X. Therefore, X is NP-hard.

Both proofs use the same argument structure. However, Proof 1 provides more details for that argument than Proof 2. These two proofs have different levels of *granularity*—a notion of the level of detail in describing something. Proof 2 is thought of as a *coarse-grained* version of Proof 1, because it abstracts away certain details from Proof 1 while maintaining the same argument structure.

Is one proof preferable to the other? This largely depends upon context. When an instructor is teaching NP-completeness for the first time, they may want to go over proofs with the level of granularity of Proof 1, to ensure that students understand how the argument follows directly from definitions. Later, to avoid getting drowned in detail, they may abbreviate to the coarse-grained Proof 2, once they believe that their students have the ability to internally map Proof 2 to Proof 1. Similarly, when a student communicates proofs back to the instructor, it must be decided what level of granularity the instructor will accept from the student. The instructor should accept a coarse-grained proof only when they are reasonably confident that the student understands how to transform it to a finer-grained proof.

Knowledge representation theorists have produced abstract frameworks for analyzing granularity [73, 85] that are applicable beyond the domain of proofs. Jerry

Hobbs' granularity framework [73] has previously been applied to intelligent tutoring systems research by Jim Greer and Jordan McCalla, who used it in the development of a LISP tutor [62]. They suggested two pedagogical reasons for caring about granularity. First, the granularity level at which a topic is presented will influence success in understanding it—tutoring success depends in part on choosing the right granularity level for a student. Second, recognition of coarse-grained problem solving strategies leads to a tutor providing more helpful advice to a student, when it is not possible to precisely trace a student's thinking at a fine-grained level.

Note that the approach that Greer and McCalla take to differentiating granularity requires a very explicit ontological mapping of the domain. However, a looser characterization of granularity for the domain of proofs shall suffice to illustrate the challenges to automatically recognizing proofs of different granularity levels.

Assume for a moment that Proof 1 could be closely represented in some formal logical system, for instance a system that uses assertion level reasoning (Section 2.1.3.1). Then Proof 2 can be conceptualized as omitting or combining some of the proof steps from the formal analogue of Proof 1. In general, a proof in a formal logical system is implicitly the finest-grained representation of itself, and any coarse-grained abstraction of that proof will be produced by omitting or combining proof steps.

In the context of a formal logical system, the task of automatically recognizing these coarse-grained proofs is difficult. One approach is to use an automated theorem prover to attempt to verify coarse-level assertions. This is the approach chosen by the EPGY Theorem Proving Environment (Section 2.4.2), but such an approach is limited by the capabilities of automated theorem provers. Another approach is to explicitly model coarse-grained proofs. This is in essence equivalent to developing a new formal logical system that captures the coarse-grained inferences at a particular level of granularity, which is what ANGLE (Section 2.4.3.3) does for the domain of geometry. However, this approach is inflexible and extremely cumbersome to scale,

because one would need to produce a formal logical system that rigidly defines each distinct level of granularity.

For more observations about proof granularity, including an analysis of granularity in student proofs, the reader is referred to Marvin Schiller's Ph.D. dissertation on the topic [136].

2.2 A brief introduction to automated theorem proving

While all automated proof verifiers can verify the correctness of proofs constructed in a formal logical system, *automated theorem provers* have the additional capability of being able to fully construct some proofs from scratch.

Automated theorem proving was mentioned in Section 2.1.4 as a potential way to address the proof granularity problem, and one might assume additional benefits to using automated theorem provers in the design of a proof tutoring system. For instance, hypothetically, an automated theorem prover might provide a student guidance about the next step they should take in a proof when they get stuck, like a chess engine recommending the next move that should be made. An intelligent tutoring system could then use the automated theorem prover to supply the student with suggestions of proof tactics and strategies they should consider in any given situation—this was the main goal behind the AProS project (Section 2.4.4).

Unfortunately, most advances in the field of automated reasoning with computers have made use of proof techniques that are foreign to humans. In fact, the formal logical systems used by computers usually differ significantly from those used by humans, even when proving theorems in the same logic. This inhibits the pedagogical utility of most automated theorem provers.

To understand the reason for this dichotomy between human proofs and computer proofs, consider the concrete case of first-order logic, an area where much progress in automated theorem proving has been accomplished.

How does one go about computationally searching for a proof in first-order logic? The naive strategy would be to indiscriminately apply all possible inference rules to all possible assumptions to produce all valid sequences of proof steps until a valid proof is found. Early researchers in automated theorem proving referred to this strategy as the "British Museum algorithm" [116] and knew it to be hopelessly inefficient for formal logical systems that humans commonly use, like natural deduction. Here are some observations to make about why this is the case:

1. The process may never terminate. It will terminate if a proof by some luck happens to be found. However, if there is no proof to be found, the process will continue indefinitely.

2. With a large number of inference rules to choose from, the combinatorial explosion of the search tree is humongous even at short depths. This means that practically, it will be impossible to find proofs with more than a few steps.

3. Some inference rules are increasing the size of formulas, and as formulas increase in size and number, the combinatorial explosion of ways to apply the inference rules to them gets worse.

4. It is possible to create cycles, where a sequence of inference rules applied to a set of formulas results in the identity function, returning you the formulas you started with. In order to make the search productive, you need to have a way of preventing this.

5. Many formulas that can be produced in first-order logic are semantically equivalent to each other. For any given formula, there are an infinite number of logically equivalent formulas. Applying the inference rules mindlessly is guaranteed to produce some formulas that are syntactically different from existing formulas that have been produced, but still semantically equivalent, thus mak-

ing the search even less productive. *Normalization* is one strategy for mitigating this problem.

From the perspective of theoretical computer science, the situation is even less optimistic. It is possible to encode Turing computations in first-order logic, and thereby show that first-order logic is *undecidable* by reduction from the halting problem [32]. Even proving statements in propositional calculus in *NP-hard*, by reduction from the Cook's satisfiability problem [59].

Why then do humans succeed at theorem proving? Humans develop an intuition that helps them decide heuristically how to navigate the proof search tree. Replicating that intuition is a challenging artificial intelligence problem, so researchers in automated reasoning have instead focused on designing formal logical systems that play to a computer's strengths.

In comparison to humans, machines are innately poor at strategizing but much better at brute-force search. Therefore, it makes sense to design formal logical systems where simple search strategies that can easily be mechanized are more likely to lead to a proof.

2.2.1 Resolution-based theorem provers

In the early 1960's, John Alan Robinson made a breakthrough in automated theorem proving in first-order logic, using a method referred to as *resolution* [134]. This method was developed from earlier insights by Jacques Herbrand [71]. It produces indirect proofs using the proof by contradiction strategy applied to a formal logical system that primarily uses a single inference rule, the *resolution inference rule*, to do heavy lifting. Resolution-based theorem provers (e.g., Otter [170], Prover9 [104], Vampire [131]) are one of the main triumphs of automated theorem proving—they have been demonstrated to be very successful in theorem proving competitions [153] and have even solved numerous open problems in mathematics [170].

To use resolution, all the assumption formulas and the negation of the goal formula must first be put in a specific normal form. The normalization process is as follows:

1. Put the formula in *prenex-normal form*: All quantifiers are moved to the front of the formula. For example, $\forall x \exists y((\exists z(x \vee \neg z)) \vee ((\forall z(z \vee \neg x)) \rightarrow y))$ becomes $\forall x \forall y \exists z_1 \exists z_2((x \vee \neg z_1) \vee ((z_2 \vee \neg x) \rightarrow y))$.

2. Put the *matrix* (quantifier-free part) of the formula in *conjunctive-normal form*: The matrix becomes a conjunction of several clauses. Note that this process often greatly expands the length of the formula. For example, $\forall x \forall y \exists z_1 \exists z_2((x \vee \neg z_1) \vee ((z_2 \vee \neg x) \rightarrow y))$ becomes $\forall x \forall y \exists z_1 \exists z_2((x \vee \neg z_1 \vee \neg z_2 \vee y) \wedge (x \vee \neg z_1 \vee x \vee y))$.

3. *Skolemize* the formula to remove existential quantifiers: For every existentially quantified variable, introduce a *Skolem function*, which is simply a new predicate in terms of the universally quantified variables that are within the scope of the existential quantifier (appear earlier in the formula). If there are no universally-quantified variables within the scope of an existentially quantified variable, then its Skolem function is a constant. Then replace every existentially quantified variable appearing in the formula with a term consisting of its Skolem function applied to the universally quantified variables within its scope. The existential quantifiers are no longer needed, so remove them from the formula. For example, $\forall x \forall y \exists z_1 \exists z_2((x \vee \neg z_1 \vee \neg z_2 \vee y) \wedge (x \vee \neg z_1 \vee x \vee y))$ becomes $\forall x \forall y((x \vee \neg f_1(x, y) \vee \neg f_2(x, y) \vee y) \wedge (x \vee \neg z_1 \vee x \vee y))$.

4. The formula now only has universal quantifiers. Drop the quantifier prefix entirely, leaving only the matrix. The formula can be thought of as a set of clauses. Furthermore, each clause can be thought of as the set of literals it contains. For example, $\forall x \forall y((x \vee \neg f_1(x, y) \vee \neg f_2(x, y) \vee y) \wedge (x \vee \neg z_1 \vee x \vee y))$ becomes $\{\{x, \neg f_1(x, y), \neg f_2(x, y), y\}, \{x, \neg z_1, y\}\}$.

Through this process, each formula produces multiple clauses. So, after converting all the formulas, a lot of clauses are produced. The combined set of all those clauses can be thought of as being the assumptions used by resolution. The resolution inference rule takes as input two individual clauses C_i and C_j and produces a new clause C_k. To describe this inference rule, first define a function of these two clauses, $M(C_i, C_j) = \{l_i \in C_i \mid \exists l_j \in C_j \text{ such that } l_j \text{ is the negation of } l_i\} + \{l_j \in C_j \mid \exists l_i \in C_i$ such that l_i is the negation of $l_j\}$. Then the new clause produced by resolution would be $C_k = C_i \cup C_j - M(C_i, C_k)$.

Using terminology invented by Peter B. Andrews [11], the literals that belong to $M(C_i, C_j)$ are said to be *mated*, which means that a literal is paired with its negation. On the other hand, literals that are duplicated in C_i and C_j are said to be *merged*. Resolution removes all mated literals and one copy of each merged literal from a pair of clauses, and produces a new clause from the remaining literals. The resolution inference rule can be viewed as a generalization of *modus ponens*. Whenever resolution produces an empty clause, a logical contradiction has been found.

Since the goal of resolution is to produce contradictions, thereby showing a goal to be true by proving its negation false, one wants to apply resolution in such a way to cancel out as many literals as possible, by mating and merging them. One other inference rule is permitted to help with this goal, which is *substitution*. Free variables in a clause can be substituted with other terms. For every pair of clauses C_i and C_j, the strategy then is to find substitutions for C_i and substitutions for C_j such that when resolution is applied to the resulting clauses, the number of mated and merged literals that get cancelled out is maximized. The process of finding these substitutions is referred to as *unification*, and efficient algorithms exist to perform the task [100].

Proofs produced with resolution tend to be significantly longer than their natural deduction counter-parts [26, 31]. Thus, resolution will never be considered a better way for humans to do proofs, since it is tedious and requires extensive iteration. With

proper intuition, humans can produce short proofs in natural deduction much more quickly than they would by following an algorithm that uses resolution. However, as a search process to be used by machines, resolution is ideal since it has significantly less combinatorial blow-up than the aforementioned "British museum algorithm" on natural deduction.

2.2.2 Peter B. Andrews' theory of matings

Peter B. Andrews developed a structural characterization of when resolution proofs exist that is useful since it permits one to determine if such a proof exists without doing a full search for it [11]. Similar insights were also independently discovered by Wolfgang Bibel [30].

The characterization is based on the observation that the entire process of resolution is essentially trying to find mates for literals. In a deduction produced by the system of resolution, each assumption clause may be used as a premise for the resolution rule (prior to substitutions) multiple times. Hence, there are multiple occurrences of each given assumption clause in the deduction, which can be indexed as unique entities. For a given resolution deduction, let S be the set of clause occurrences that appear in the deduction, and let $L(S)$ be the set of *literal occurrences* that appear in those clauses. An *abstract mating* M is defined to be a symmetric binary relation over $L(T)$ where for any two literal occurrences l_i and l_j, the relation $M(l_i, l_j)$ holds if some substitutions will make l_i the complement of l_j.

For a given abstract mating M, a *mating cycle* is defined as an even length sequence of literal occurrences from $L(T)$, $l_1, \ldots, l_n$, where the following conditions hold:

1. l_1 and l_n are distinct literal occurrences belonging to the same clause occurrence.

2. If i is an even number between 1 and n, then M(l_i, l_{i+1}) holds.

3. If i is an odd number between 1 and n, then l_i and l_{i+1} are distinct literal occurrences belonging to the same clause occurrence.

Two distinct literal occurrences are said to be *merged* by a mating cycle if they share a common mate in the mating cycle. I.e., if there are three distinct literal occurrence l_i, i_j and l_k belonging to the mating cycle where $M(l_i, l_k)$ and $M(l_j, l_k)$ both hold, then the mating cycle *contains a merge*.

Andrews' characterization is that for a resolution deduction to be a proof (i.e., for it to lead to a contradiction), there must exist an abstract mating M where every literal occurrence in $L(S)$ has a mate, and where every mating cycle produced by M contains a merge. The abstract matings meeting these conditions are referred to as *acceptable matings*. Andrews proved that a set of clauses in first-order logic will have a *model* if and only if there is no acceptable mating of a non-empty finite set of clause occurrences from that set of clauses [11].

There is a particular interest of this mating theory to tutoring. Not only does Andrews' characterization potentially lead to a more efficient search process than resolution itself, but Andrews later realized it could be used to guide the search process for natural deduction proofs. One could first find an acceptable mating and then use that as a plan for finding a natural deduction proof [12]. This resulted in the first version of an automated theorem prover called TPS. The mating method was later extended to higher-order logic by two of Andrews' students, Dale Miller [109] and Frank Pfenning [125], resulting in a generalization of matings referred to as "expansion proofs" and one of the first successful automated theorem provers for Church's simple type theory [13].

Unfortunately, Andrews and his students never completely capitalized on these methods, with respect to how they might be used for proof tutoring. This is odd considering that Andrews did develop ETPS [13], an educational platform that shares the same code base as his theorem prover. However, ETPS specifically disables the

powerful theorem proving functionality, and there is no attempt to use automated theorem proving to give the student intelligent guidance in proof construction.

2.2.3 More references on automated theorem proving

A brief background on automated theorem proving has been covered here to show why most of these techniques are not easy to adapt to proof tutoring. Readers who would like to learn more about resolution-based theorem proving are recommended to read Larry Wos' gentle introduction to the subject [170], which discusses his journey in developing the theorem prover Otter. A two-volume *Handbook of Automated Reasoning* [133] is also available, which explains most of the state-of-the-art research in the field.

2.3 Theories of how humans naturally reason

Humans obviously have some innate reasoning capabilities, but cognitive psychologists debate the mechanism that provides those capabilities. On one side of the debate, Lance Rips and Martin Braine have proposed theories [34, 132] that explain the performance of human reasoning as a general application and chaining of abstract rules, i.e., they suggest that there is a formal logical system underlying human reasoning independent of context.

However, a psychological experiment known as the *Wason selection task* [167] would seemingly poke holes in those theories. The general setup for the experiment is that subjects are given a reasoning problem involving four cards on a table along with a description of what the four cards represent. There are different versions of the experiment, each with a different semantic interpretation of what the cards represent. In all versions, subjects can only see the side of the cards facing them and are tasked with determining which cards need to be flipped over to verify a logical assertion about the cards. Furthermore, each version of the experiment uses

a logically equivalent assertion. Consequently, all versions of the experiment can in principle be solved using the exact same logical inference rules.

Surprisingly, performance on the Wason selection task varies considerably depending on the context given in the reasoning problem. When given an abstract version of the problem (Figure 2.1), less than 25% of subjects from a college student population could correctly solve it. However, approximately 75% of subjects from the same population could correctly solve the problem when it was themed with a social situation (Figure 2.2) they were likely to be familiar with [45].

Part of your new clerical job at the local high school is to make sure that student documents have been processed correctly. Your job is to make sure the documents conform to the following alphanumeric rule:

"If a person has a 'D' rating, then his documents must be marked code '3'."

You suspect the secretary you replaced did not categorize the students' documents correctly. The cards below have information about the documents of four people who are enrolled at this high school. Each card represents one person. One side of a card tells a person's letter rating and the other side of the card tells that person's number code.

Indicate only those card(s) you definitely need to turn over to see if the documents of any of these people violate this rule.

D	F	3	7

Figure 2.1: Wason selection task with abstract theme, from [45].

These experiments suggest that the cognitive mechanism for reasoning is somehow context dependent. Various theories have been proposed to account for this. Richard Griggs and James Cox were proponents of a *memory-cueing hypothesis* [67, 98] to

In its crackdown against drunk drivers, Massachusetts law enforcement officials are revoking liquor licenses left and right. You are a bouncer in a Boston bar, and you'll lose your job unless you enforce the following law:

"If a person is drinking beer, then he must be over 20 years old."

The cards below have information about four people sitting at a table in your bar. Each card represents one person. One side of a card tells what a person is drinking and the other side of the card tells that person's age.

Indicate only those card(s) you definitely need to turn over to see if any of these people are breaking this law.

drinking beer	drinking coke	25 years old	16 years old

Figure 2.2: Wason selection task with familiar social theme, from [45].

account for performance discrepancies on the Wason selection task—they suggested that reasoning proceeds by recalling specific instances of past reasoning events that are directly related to the context of the situation, and that those instances are used to determine appropriate conclusions. Patricia Cheng and Keith Holyoak proposed a broader theory involving *pragmatic reasoning schemas* [40], whereby people develop *schemas* based on past experiences and adapt those schemas to new reasoning situations. A schema may align with a context-dependent formal logical system. For instance, experience with the United States drinking culture may lead to a schema that effectively reasons about problems like the one in Figure 2.2, but that schema can easily be adapted to other contexts involving granting permission and legal obligation [40].

Finally, Phillip Johnson-Laird's *mental models* [78] theory is quite popular and was used to model reasoning in Allen Newell's cognitive architecture, Soar [126]. This

theory presupposes that humans have domain-independent comprehension procedures to construct concrete models of different scenarios, and that they use those models to produce deductions. These mental models are analogous to the finite models that logicians use for studying semantics and are in direct contrast to the syntactic proofs produced by formal logical systems. The mental models theory has been used to successfully predict systematic reasoning errors that humans make [79].

Note that while all these theories attempt to explain how novice reasoning works, they say little about the cognitive skills one must develop to have expertise in theorem proving. For instance, even if human reasoning were to follow from mental models rather than adhere to a formal logical system, that does not mean it is pointless to teach formal logical systems to students. In fact, one could argue the opposite—that a student's lack of skill with deduction using logical inference rules justifies teaching it. However, having a correct theory of native reasoning would be very useful for building an accurate *student model* for an intelligent tutoring system, to predict how students will struggle and to develop pedagogical strategies to mitigate their misconceptions.

For instance, one conjecture is that many students use their native reasoning when writing informal proof narratives, even if they have previously been taught to reason using a formal logical system. If informal narrative is to continue to be used as the preferred medium for students to communicate their proofs, then students may need a better bridge between structured proofs from formal logic and informal narrative proofs, to ensure they successfully transfer skill from the former to the latter. Complexity Tutor might be that bridge.

2.4 Survey of systems and guiding principles

In this section, several systems are surveyed that help students construct proofs in formal logical systems. The systems chosen for this survey are notable for their historical importance and also because they are exemplars of different design method-

ologies that have been applied to proof tutoring. These various methodologies include controlled natural language (EXCHECK), cognitive modeling (GPT and ANGLE), proof search (AProS), assertion level reasoning (ΩMEGA-Tutor), and machine learning (Deep Thought). Additionally, a formal logical system developed by David Gries and Fred Schneider is covered, given its relevance to computer science.

2.4.1 EXCHECK and early work from Patrick Suppes' group

Patrick Suppes' research group at the Stanford University Institute for Mathematical Studies in the Social Sciences was an early pioneer in Computer Assisted Instruction (CAI) systems. Much of the initial focus from this group revolved around developing systems that could aid students in learning to write logical proofs.

Their earliest system from 1963 was developed to teach rules of logic to elementary school students. In an example [151] of how this system worked, students are asked to prove that "Jack and Bill are not the same height" from the following numbered assumptions:

1. If Jack is taller than Bob, then Sally is shorter than Mavis.
2. Sally is not shorter than Mavis.
3. If Jack and Bill are the same height, then Jack is taller than Bob.

The student could then type the command `DC 1.2` at the terminal, which would apply the *modus tollendo tollens* inference rule to assumptions 1 and 2, to produce a new assertion (assigned number 4):

4. Jack is not taller than Bob.

The student could then type the command `DC 3.4` to apply the same inference rule to assumption 3 and the new assertion, producing the desired result:

5. Jack and Bill are not the same height.

Suppes' group then developed EXCHECK, a more ambitious system, which would interactively verify proofs that students wrote. During the 1970's, EXCHECK was used to teach numerous proof-based courses at Stanford University, including elementary logic (predicate calculus), axiomatic set theory, proof theory and probability theory [105, 152]. EXCHECK was also a very modular system, and it was later adapted to teaching topics unrelated to mathematical proofs, such as teaching the Armenian language [105].

In verifying proofs, EXCHECK made use of both an internal theorem prover and the computer algebra system, REDUCE [105].

EXCHECK was notable for its extensive language capabilities, which allowed students to type assertions for their proofs using both symbolic expressions and natural language statements [148]. For instance, EXCHECK would recognize the following:

```
For all x,y
     if x is a set and y is a set then
        For all z,
           z is in x if and only if z is in y
```

EXCHECK also gave students the flexibility to type either a mathematical expression like `{x : x neq x} = 0` or its natural language equivalent:

```
The set of all x such that x is not equal to x is empty.
```

The back-end was a *context-free language* parser with 700 grammar rules [147] and corresponding macro templates to transform parse outputs to an internal representation, similar to the way modern compilers work. The proofs that students could write using this context-free language would resemble proofs they might find in a textbook.

However, note that even with the 700 grammar rules, EXCHECK is not able to recognize all natural language statements that might be written in a proof. For

instance, it is observed [148] that the system would not be able to recognize the sentence, "Two sets are equal just in case they have the same elements." Students are thus confined to a *controlled natural language*, a strict subset of a natural language with unambiguous semantic interpretations. Students might struggle to learn the precise rules of this language, which would potentially cause extraneous cognitive load when they are constructing proofs.

Furthermore, considerable effort may be required to adapt a system like EX-CHECK to new course topics, which would require adding new grammar rules to the parser, even though philosophy professors and graduate students who are not professional programmers were able to successfully author content for the system [148].

2.4.2 The EPGY Theorem Proving Environment

The Education Program for Gifted Youth (EPGY) was a program offered by Stanford University until 2013, when it was spun-off as a start-up that ended up getting purchased by McGraw-Hill and later abandoned [106]. EPGY provided college-level computer-based distance learning courses to gifted K-12 students. The research behind EPGY was a continuation of the earlier work conducted by Patrick Suppes' group. Some of the courses offered by EPGY were proof-based and made use of the EPGY Theorem Proving Environment [107]. This included courses in linear algebra, Euclidean geometry and logic.

In the EPGY Theorem Proving Environment, students use a graphical interface to construct Fitch-style diagrams of their proofs, as shown in Figure 2.3. Students can search from a library of existing definitions, axioms and theorems that they might want to use in their proof. To create individual assertions in the proof, students use the EPGY Derivation System [130]. Students can also apply different proof strategies, including conditional proof, biconditional proof, proof by contradiction, and proof by induction. For instance, the conditional proof strategy allows students

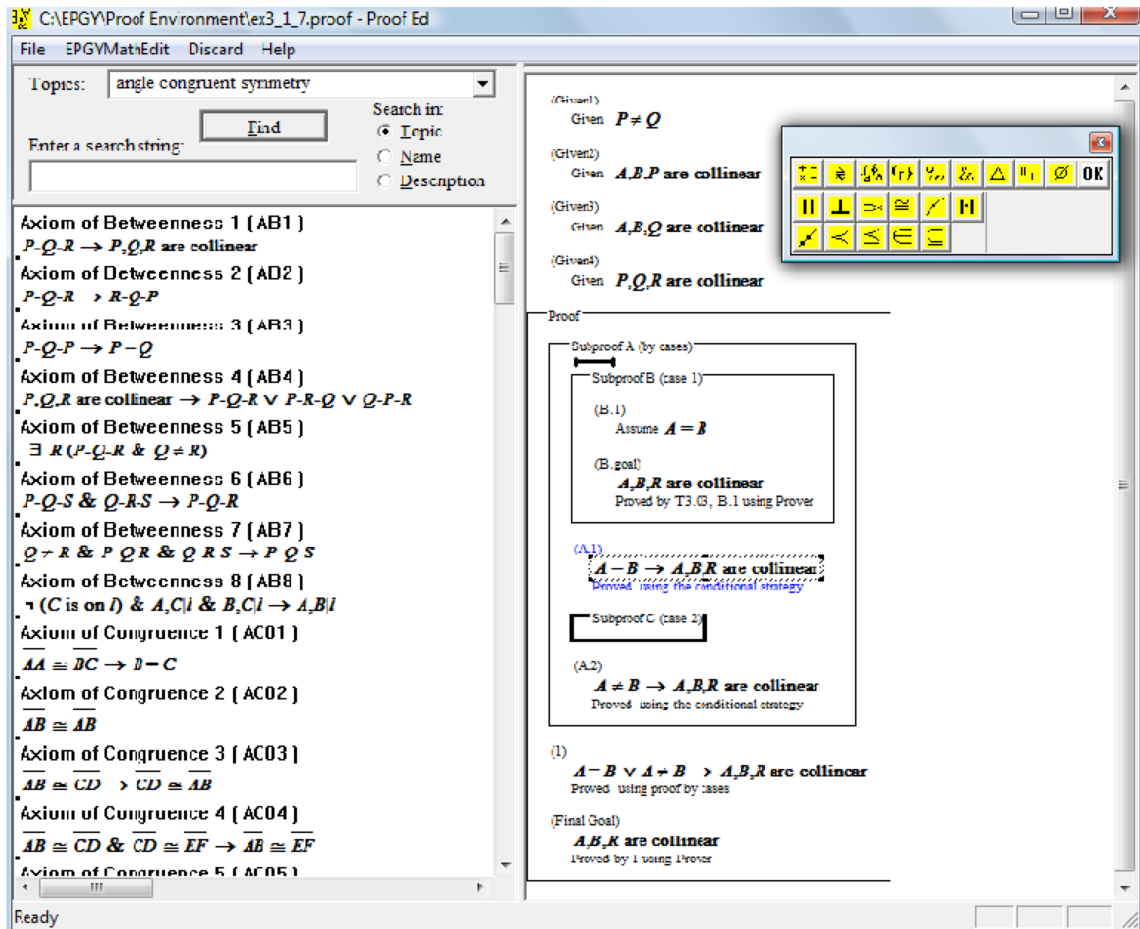


Figure 2.3: Euclidean geometry proof in the EPGY Theorem Proving Environment

to introduce a sub-proof (i.e., a lemma), having its own separate assumption A and goal G. Completing this sub-proof justifies a new assertion of the form $A \to G$.

One of the main potential advantages of the EPGY Theorem Proving Environment is that it is one of the only systems that attempts to verify the correctness of coarse-grained proofs that skip extraneous proof steps students would be likely to omit. The intent is to produce an experience for the students that is similar to how they would normally construct proofs on paper.

The system attempts to accomplish this using a theorem prover, Otter [170], on the backend to do inference resolution [107], but in a rather ad-hoc way. Otter is run with a fixed timer of 5 seconds, to see if it can prove a given step in the student's

proof from the previous steps. If Otter is not able to prove the inference in the allocated amount of time, the EPGY system assumes that the student's inference is either incorrect or the student has not supplied enough steps in their proof.

The reason for placing the time limit on the theorem prover is that otherwise it might accept a proof inference that is too coarse-grained, i.e., that the student is making too big of a leap of logic. Nevertheless, using a timer is a very ad-hoc heuristic for measuring granularity. Otter tries to prove inferences very differently from how a human would prove them, as explained in Section 2.2. This is compounded by the fact that Otter can only prove assertions in first-order logic, and the student proofs are expressed natively in a multi-sorted logic, so assertions must be translated from the multi-sorted logic to the first-order logic before Otter can deal with them [107]. The reason for using a multi-sorted logic is to be able to differentiate the types of variables. For instance, in a linear algebra course, you might have some variables that represent matrices, others representing scalars, etc.

Ultimately, when the system fails to verify a student's inference, it does not know why it failed. Maybe the inference wasn't correct. Maybe the granularity was too coarse. Maybe Otter just failed when it shouldn't, since no automated theorem prover is foolproof. And when the system succeeds, it may know that the student's inference is correct, but the system still has no sense of whether the student understands why the inference is correct or not.

2.4.3 Cognitive tutors for geometry theorem proving

Cognitive tutors are intelligent tutoring systems that are designed from principles of cognitive learning theories. John Anderson, a cognitive psychology researcher at Carnegie Mellon University, pioneered the development of cognitive tutors based on a series of cognitive theories he developed. The first of these theories was the Adaptive Control of Thought (ACT) theory [7], which was followed by subsequent refinements—

the ACT* theory [8] and the ACT-R theory [9]. The tutoring systems based on these theories were very successful and led Anderson's research group to create a company called Carnegie Learning, which sold a high school math curriculum based around the tutoring systems that they developed for algebra and geometry. Hundreds of thousands of students have been exposed to this curriculum. In 2007, about 10% of high school math classes in the United States had used the Carnegie Learning curriculum [169].

Cognitive learning theories presuppose that the mind learns and processes information in accordance with a specific *cognitive architecture*—Allen Newell developed this concept from his background in computer architecture [24]. Thus, research in cognitive psychology notably shares many of the same principles that guide research in artificial intelligence, with the distinction that the latter does not attempt to accurately model the human mind.

2.4.3.1 ACT theories of cognition

ACT cognitive architectures are framed in terms of modular *production rules*. Each production rule consists of a *condition* as well as an *action* that can be performed if that condition is met. To illustrate, consider the following production rule that was presented [9] for the task of geometry theorem proving:

```
IF there exists Triangle ABC and Triangle DEF
    and Edge AB is congruent to Edge DE
    and Edge BC is congruent to Edge EF
    and Edge AC is congruent to Edge DF
THEN conclude Triangle ABD is congruent to Triangle DEF
```

In this rule, the condition checks for the existence of two triangles and whether their edges are congruent. The action is to determine that the two triangles are congruent. The rule can be thought of as taking certain knowledge as input and producing

certain knowledge as output. Anderson refers to the input and output of a production rule as *declarative knowledge*—loosely speaking, this is knowledge that people can describe. In addition to declarative knowledge, there is also *procedural knowledge*, which is the kind of knowledge you may infer that someone knows by watching their behavior. Production rules themselves are defined to represent procedural knowledge. In his book [9], Anderson gives several justifications for theoretically differentiating declarative knowledge from procedural knowledge. One example he gives is of a typist who cannot explain the keyboard layout—they retain the procedural knowledge of how to type but do not retain the declarative knowledge needed to describe where the keys are located in relation to each other.

The claim made by Anderson is that complex cognitive skills are rendered by following a sequence of production rules. The algorithm for producing this sequence has three phases to it that run in a loop until a desired goal is reached. The first phase is *pattern matching*, where each production rule is matched against available declarative knowledge to determine if its conditions are met. The next phase is *conflict resolution*, where one of the production rules that has its conditions met is chosen. In the ACT-R architecture, a computational cost is determined for each rule in terms of memory access and other criteria, and the rule with the least cost is chosen [9]. In the final phase, one of the production rules is triggered, and its action results in new declarative knowledge being produced.

ACT cognitive architectures are not the only systems that use production rules. The idea of production systems can be traced back to Emil Post's work on term rewriting [127], and other production systems have been used in cognitive theories including PSG [114], OPS [56] and Soar [91, 115]. Unlike the ACT theories, these other theories assume that procedural knowledge is the only kind of knowledge that is permanent.

Notice that production systems can also be thought of as formal logical systems, which were defined in Section 2.1.1. The production rules are analogous to inference rules, and the declarative knowledge can be encoded in the proof steps and assumption formulas. This implies that it's easy to construct an automated verifier for any theorem proving task that can be cognitively modeled in ACT-R.

Skeptics might wonder if a symbolic production rule architecture is in fact an accurate representation of the mind, given that physical evidence of the brain suggests it functions more like an analogue neural network. Anderson addresses this skepticism by pointing out that the production rules of ACT-R might in fact be implemented at a lower-level with a neural network, and he likens ACT-R to be similar to a high-level programming language for the mind, explaining that the theory is well-supported by psychological evidence regardless of lower-level implementation details [9].

2.4.3.2 Anderson's first tutor for geometry theorem proving

One of the early cognitive tutors that Anderson's research group developed was for teaching geometry proofs [9]. Expert domain knowledge was modeled with production rules that encoded valid inferences in the domain of geometry.

Figure 2.4 shows the interface of this tutoring system, which is using a graphical structure to represent geometry proofs rather than the two-column format typically taught in high school geometry classes. Anderson's motivation for using a graphical proof representation was based on John Brown's principle of "reifying the problem space" [37,41], i.e., making the abstract features of geometry problem solving concrete. Thus, the graphical structure gives an explicit representation of the logical relation between premises and conclusions, as well as the search process by which one hunts for proofs. Students reported that they preferred this graphical proof format to the normal two-column format [9].

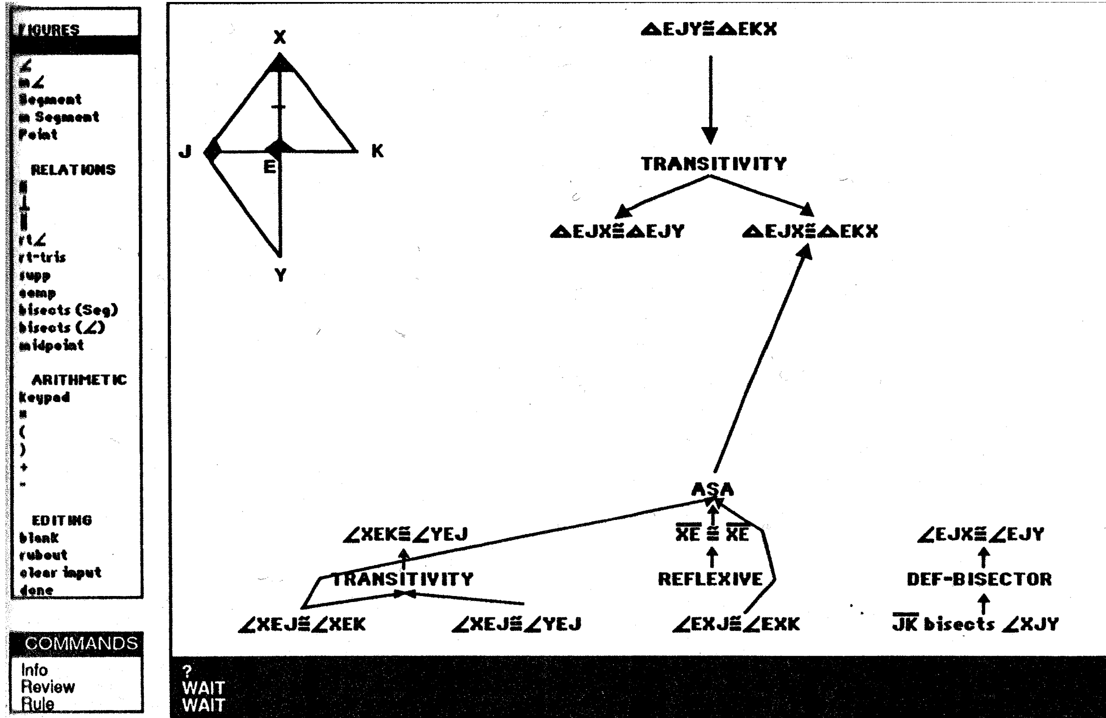


Figure 2.4: Screenshot of Anderson's first geometry tutor, from his book [9]

With respect to this dissertation, the graphical proof structure is notable because Complexity Tutor uses a similar structure to represent proofs, as explained in Section 3.1.5.1. Note that there are some significant differences. In Figure 2.4, the direction of the arrow denotes a special meaning. If the arrow is pointed upwards, then it represents *forward-chaining* from premises to conclusions. If the arrow is pointed downwards, then it represents the reverse—*backward-chaining* from conclusions to premises, i.e., finding sub-goals that will prove a goal. Premises are always displayed at the bottom of the proof and goals at the top of the proof, to keep these arrow orientations consistent.

In comparison, Complexity Tutor does not explicitly distinguish backward-chaining from forward-chaining in its interface. While students who use Complexity Tutor can implicitly use either forward-chaining or backward-chaining as a strategy, the arrows always lead from premises to conclusions.

It is possible that students who used Anderson's system may have been confused by how the interface distinguished backward-chaining from forward-chaining. Anderson reported that during his initial evaluation of the system in the 1985-1986 academic year, almost all students except those with IQ scores over 130 had great difficulty successfully applying the backward-chaining strategy. The following year, he removed backward-chaining as a feature from the system entirely [9]. Interestingly, other researchers [135] found no evidence that students struggle with backward-chaining in general, so Anderson's interface may be to blame.

2.4.3.3 A New Geometry Learning Enviornment (ANGLE)

ANGLE was a cognitive tutor developed by Kenneth Koedinger for his Ph.D., which was intended to be an improved version of Anderson's earlier geometry tutor. Koedinger used a different set of production rules in his expert model—his key insight was that when geometry experts are constructing a geometry proof, they cognitively develop proof sketches that are abstractions of concrete problem solutions. His Ph.D. dissertation [88] defined what he refers to as the Diagram Configuration Model to represent these proof sketches, where diagram configuration schemata are used to generate a proof plan for a given problem at an abstract level.

Borrowing terminology from Allen Newell and Herbert Simon [117], Koedinger explains in his dissertation that a given *problem domain* may have multiple possible *problem spaces*. For the domain of geometry theorem proving, the *execution space* refers to the search space of all possible geometry proofs that can be generated from the given assumptions for a problem. Koedinger points out that this execution space has a huge combinatorial blow-up when searching for proofs—in one instance, applying only three levels of geometry inference rules led to 100,000 possibilities. Anderson's original geometry tutor searched for solutions inside this execution space

and used heuristics that were psychologically motivated by ACT* theory to guide the search.

However, there may be a separate *abstract problem solving space* which is different from the execution space and where the actual planning of the proof occurs. This space would have significantly less combinatorial blow-up, and it is presumed that a geometry expert gets proficient in constructing geometry proofs by learning to navigate this abstract problem solving space. The evidence to support the existence of such a space for geometry problem solving comes from the observation that geometry experts skip steps when articulating how they solve a problem [88]. From prior research [52], it was known that an expert's verbalizations are likely to accurately reflect their working memory states. Therefore, this is evidence that the geometry expert is working in a problem space that allows them to skip steps in the execution space when planning the proof.

When steps from the execution space are skipped in the plan of a proof, those omitted steps can fall into two categories—*safe abstractions* and *risky abstractions*. A safe abstraction is an omitted step that provides details that will always be irrelevant to finding a correct proof. For instance, Koedinger found that it was always safe to omit details that distinguish between a congruence statement and an equality operator statement [88]. A risky abstraction, on the other hand, abstracts away details that when ignored are sometimes critical to arriving at a correct solution [117].

In addition to skipping steps, it has been found that expert problem solvers often collapse multiple consecutive steps into a single step [10, 93]. These combined steps are referred to as *macro-operators* [90], and Koedinger noticed that geometry experts follow a regular pattern in the macro-operators they use [88].

Koedinger was able to codify safe abstractions, risky abstractions and macro-operators by having experts verbalize their thinking process while solving geometry problems. From this research, Koedinger reconstructed production rules for the ab-

stract problem solving space of the geometry experts, which in turn led to the Diagram Configuration Model that ANGLE uses to tutor students. Hints given to students using ANGLE encouraged them to first search for a coarse-grained proof plan with the help of diagram configuration schemata and then refine the proofs to calculus level [88].

Empirical research was inconclusive about whether ANGLE was an improvement over Anderson's earlier geometry tutor. Both systems provided significant learning gains for the students but there was no statistically significant difference between students using ANGLE versus the older system [88].

2.4.4 Automated Proof Search (AProS)

Wilfried Sieg, a philosophy professor at Carnegie Mellon University, has a research group that has been active developing a tutoring system since 1985, to help students learn to construct proofs in formal logic [141]. The first incarnation of this work was referred to as the Carnegie Mellon Proof Tutor, and it later evolved into the Automated Proof Search (AProS) project, when Sieg began to investigate adding automated proof search capabilities to his system. Sieg sought to model not just how proofs are verified but also how they are produced, in order to provide more intelligent strategic guidance to students using his system.

Tutoring systems that only utilize a proof verifier for a formal logical system lack the insight necessary to provide hints or guidance about the process one should follow to obtain a proof. However, in general, it is not easy to affectively model or automate how humans discover proofs. Recall from Section 2.2 that most machine-oriented automated theorem proving techniques, such as resolution, search for proofs in a very different way than a human would. As such, even when an automated theorem prover is able to verify the correctness of an assertion, it generally provides

no strategic guidance that would help a student learning proof construction figure out how to construct a proof of the assertion on their own.

On the other hand, the AProS system searches for proofs in a more human-oriented way. The formal logical system AProS uses, referred to as an *intercalation calculus*, has natural deduction rules that have been restricted in how they can be applied, in order to improve efficiency of the search process [142]. Natural deduction has two kinds of inference rules, referred to as *introduction rules* and *elimination rules*. When searching for a proof, inferences can be made in either the forward direction, leading from assumptions, or in the backward direction by applying inverse inference rules to goals. However, intercalation calculi are restricted so that only elimination rules can be applied in the forward direction, whereas only inverted introduction rules can be applied in the backward direction [142]. This leads to proofs that adhere to a normalization property for natural deduction that Dag Prawitz discovered [128].

Additionally, AProS has the following heuristic ordering of *proof tactics* it follows in searching the intercalation calculus proof space for a proof [124]:

1. **Extraction** – Given a premise and a goal that is a subformula of the premise, recursively apply all elimination rules to the premise until the goal formula is reached.

2. **Inversion** – Apply one introduction rule backwards to a goal formula. Does not apply to atomic formulas.

3. **Cases** – Apply disjunction elimination to a disjunction that is strictly positively embedded in a premise, i.e., proof by cases.

4. **Refutation** – Apply negation elimination for a selected goal formula, i.e., proof by contradiction.

Later work added rules and heuristics for set theory inferences to AProS, which can prove Godel's incompleteness theorems and that $\sqrt{2}$ is not rational [143].

Douglas Perkins, one of Sieg's students, integrated AProS into a tutoring system for an online "Logic and Proofs" course offered through Carnegie Mellon's Open Learning Initiative [124]. The course covers both propositional and first-order logic, and is intended for novices without experience in formal logic. At Carnegie Mellon, it is offered as an elective requiring no prerequisites. In this course, students have exercises where they construct proofs in the Carnegie Proof Lab, an environment that allows them to construct Fitch-style diagrams[1] of natural deduction proofs.

When students are stumped about how to proceed with their proof, the tutoring system encourages them to consider using the four tactics mentioned above. Thus, students are being taught the same strategy that is being used to automate the proof search. However, students are not required to follow the AProS search strategy, and can apply any valid inference rule from natural deduction [124].

Perkins implemented the following three modes of tutoring [124]. In the first mode, students are not given any strategic guidance but the tutor gives them hints about what tactics can be applied in a given situation. The second mode is a "walkthrough" mode, where AProS constructs a proof for the problem a student is trying to solve, and then provides step-by-step guidance from beginning to end for constructing the proof. The third mode is the most interesting, because it dynamically generates a hint for a student who has produced a partial proof. To do so, the student's partial proof must first be put in Prawitz's normalized form and culled of inference rule applications that would interfere with the AProS search procedure. Then, AProS completes the proof and a recommendation is made to the student for how to proceed.

2.4.5 The Dialog project and Ωmega-Tutor

Dialog was a project undertaken by researchers at Saarland University, which had the goal of producing a mathematical proof tutoring system that could converse

[1] Stanisław Jaśkowski's convention [77] of putting boxes around subproofs is used.

with students entirely in natural language [28]. Given the ambitious scope of the project, the research was exploratory in nature and the goal of producing a full tutoring system with natural language capabilities was never actually realized.

Nevertheless, the research stands out as one of the only attempts to build a tutoring system that could analyze informal proof arguments, not just those constructed in formal logical systems. In that sense, its ambitions are closer to this dissertation than any other work that has been cited so far. However, the Dialog researchers were looking for a way to map informal narrative arguments to a formal logical system, whereas this dissertation introduces a novel way to tutor theorem proving without a formal logical system. Thus, their approach was very different.

A main contributions produced by the Dialog project were results from two *Wizard-of-Oz experiments*—participants in these experiments believed that they were interacting with a fully automated tutoring system, when in fact there was a human expert in the background controlling the responses that the system produced. As reported by Marvin Schiller, one of the researchers involved, the motivation for conducting Wizard-of-Oz experiments was to collect empirical data on natural language proof tutoring that would inform future development of the system [136].

The first experiment involved 24 student subjects using the fake tutoring system. Data from this experiment produced a corpus of German dialogues about naive set theory proofs [28]. The subjects were asked to think aloud while working and they were video recorded [136]. Even though the domain of interaction had been limited to naive set theory proofs, the researchers were surprised to discover an "overwhelming list of key phenomena raising interesting and novel research challenges" [28] that resulted from the experiment. This motivated the second experiment.

The second experiment, described in Schiller's thesis [136], involved 37 students subjects who worked on proof problems involving binary relations. On the backend,

those 37 subjects were split between four experts who were tasked with evaluating each proof step that subjects produced according to the following criteria:

Correctness — the expert could label the proof step as either “correct”, “partially correct” or “incorrect”.

Granularity — the expert could label the granularity of the proof step as either “appropriate”, “too coarse-grained” or “too detailed”.

Relevance — the expert could label the relevance of the proof step as either “relevant”, “irrelevant” or “limited relevance”.

In conjunction with the Wizard-of-Oz experiments, ΩMEGA-Tutor was developed by Dominik Dietrich as a prototype tool for automatically diagnosing the correctness of proof steps in student proofs [47,136]. This tool uses the ΩMEGA$^{\text{CoRe}}$ proof planning system to validate the correctness of an assertion in a proof, in the same way that the EPGY Theorem Proving Environment used Otter to validate assertions. However, there's a very important distinction between Otter and ΩMEGA$^{\text{CoRe}}$. Whereas Otter uses resolution (Section 2.2) to attempt to prove first order deductions, ΩMEGA$^{\text{CoRe}}$ uses assertion level reasoning (Section 2.1.3.1) inferences, which are closer in granularity to the assertions that students would make in their proofs.

A sample of student proof attempts from 17 tutorial dialogues produced in the second Wizard-of-Oz experiment were transcribed from natural language into proof steps in a formal language that could be evaluated by ΩMEGA-Tutor. This resulted in 147 proof steps. Then, ΩMEGA-Tutor ran a *depth-limited breadth-first search* over the CoRe proof space to attempt to reconstruct a CoRe proof tree for each proof step that students produced. A depth limit of 4 on the breadth-first search proved sufficient for correctly classifying 141 or the 147 proof steps that students made [27].

It should be noted that while this result is very promising, ΩMEGA$^{\text{CoRe}}$ is likely a less robust automated theorem prover than the ones mentioned in Section 2.2.

Schiller notes that unlike most well-known theorem provers, ΩMEGA$^{\text{CoRe}}$ has never been entered in the CADE Automatic Theorem Proving System Competition [136].

Schiller used the proof trees constructed by ΩMEGA-Tutor and the granularity labels assigned by experts in the second Wizard-of-Oz experiment to build a machine learning classifier for proof granularity [136]. Thus, correctness and granularity could now be assessed for some proofs.

This prototype proof assessment functionality from ΩMEGA-Tutor and Schiller's granularity classifier was incorporated into ActiveMath, an existing web-based learning platform developed at Saarland University. A screenshot of a proof problem in ActiveMath is shown in Figure 2.5.

For each proof step, students are given multiple choice options to choose one of four assertion types, and a formula input field where they can type a symbolic formula representing the assertion. The four options for assertions types do the following [136]:

The "Let..." option allows students to create a new definition.

The "Then..." option allows students to create an inference.

The "It holds..." option allows students to introduce a lemma.

The "It's to be shown that..." option allows students to demonstrate a subgoal, i.e., backwards inference from goals.

Students can click the "Hint" button in Figure 2.5 to receive a hint for the proof step they are working on. According to Schiller's Ph.D. thesis, the hints that students receive are generated by ΩMEGA-Tutor using its "proof search/strategy mechanism" [136]. While this is rather vague in detail, it likely attempts to complete the student's proof as AProS does. A later paper by Serge Autexier, Dominik Dietrich and Marvin Schiller explains their ideas for using proof strategies to adaptively provide hints [18]. As can be seen in Figure 2.5, different levels of hints are provided ranging from "Try to work backward from the goal" to a specific bottom-out hint telling the student exactly what to do.

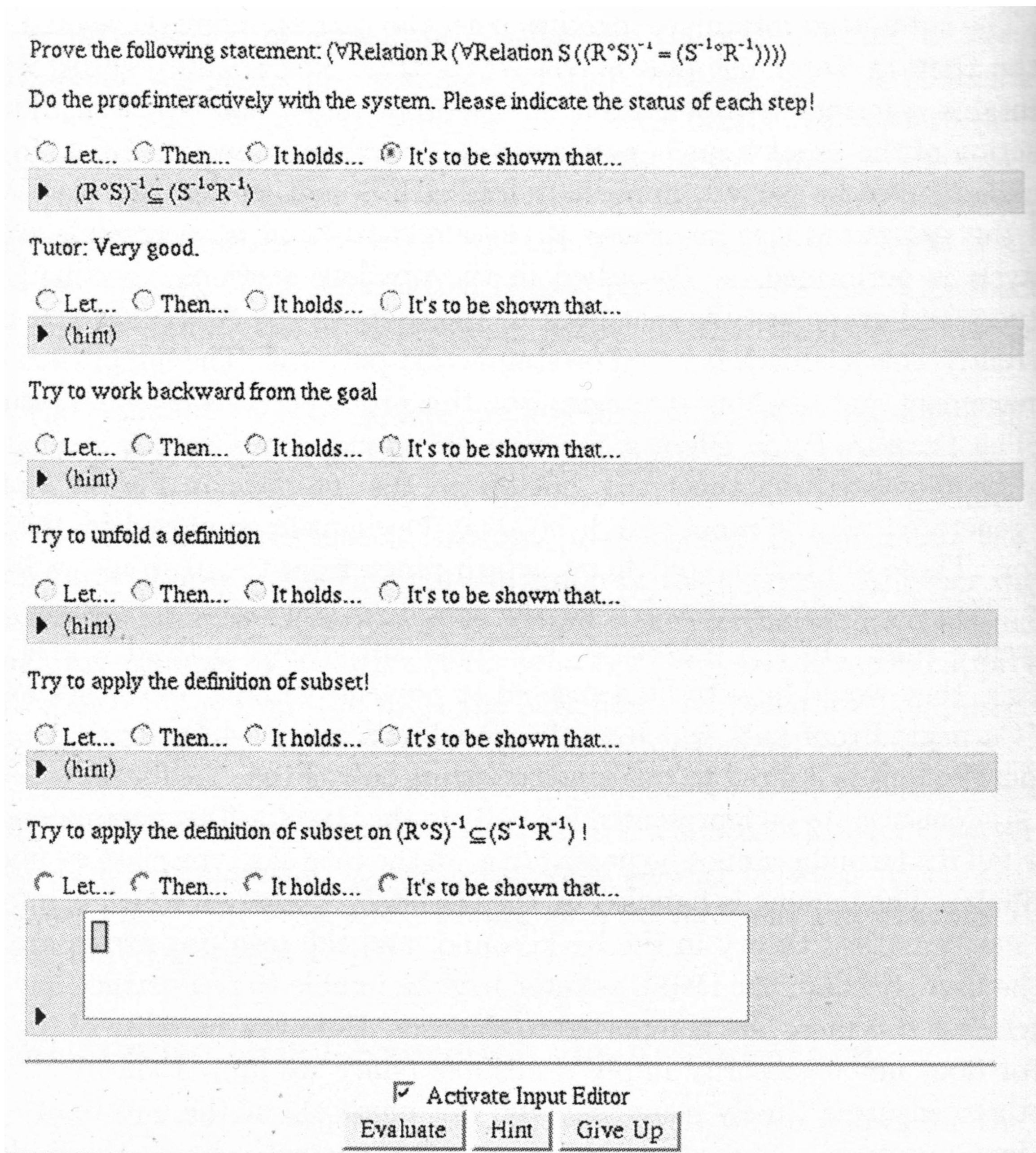


Figure 2.5: Screenshot of proof tutoring environment in ActiveMath, from Marvin Schiller's Ph.D. thesis [136].

2.4.6 Deep Thought

Deep Thought is an intelligent tutoring system, developed by the Game2Learn Lab at North Carolina State University, for teaching formal proofs to computer science

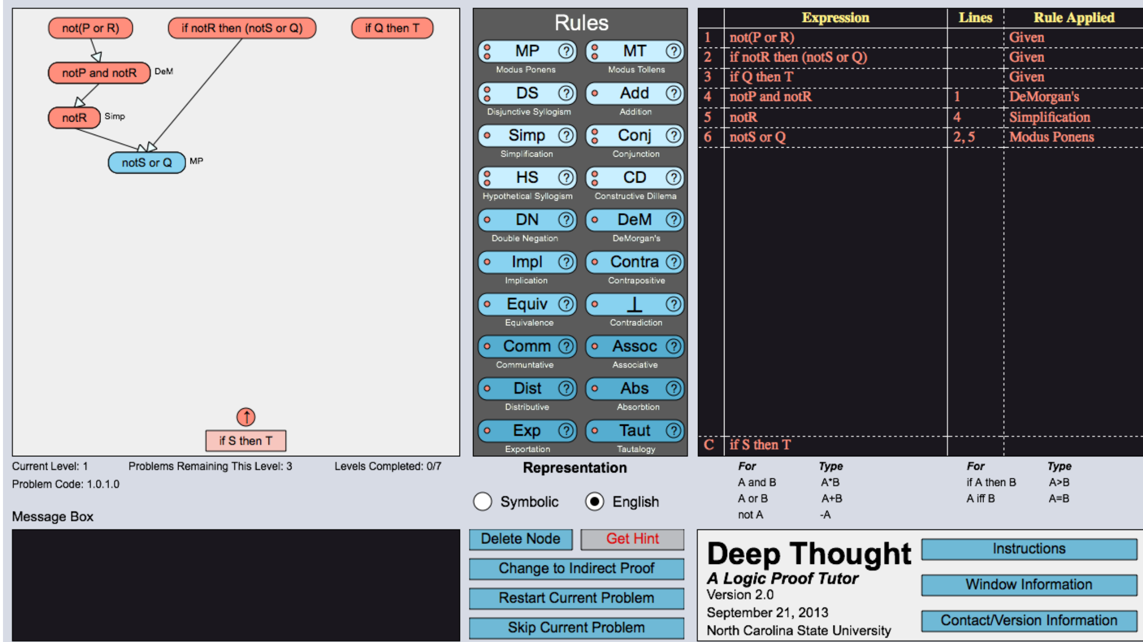


Figure 2.6: Screenshot of Deep Thought, from the Game2Learn Lab website [21].

students in their discrete math classes. Figure 2.6 shows a screenshot of the interface of Deep Thought, which uses both a traditional column format and graphical format for representing proofs.

What is most notable about Deep Thought compared to the other systems mentioned in this survey is that it uses a data-driven approach to tutoring proofs. The earliest version of Deep Thought was not an intelligent tutoring system, but merely a web-based interface for students to work on logic problems. Data from the student interactions with the earliest version was recorded, and that data was used to a develop a *Bayesian knowledge-tracing model* to be used with future versions [112]. This model was used to choose problems for students to work on that would have the right difficulty level based on their performance.

Later, Tiffany Barnes and John Stamper used machine learning on many semesters of previous data to automatically generate hints for students [22]. They used that data to develop a *Markov decision process (MDP)* for each problem, representing all

paths students have taken in trying to solve it—each state represents a partial proof construction a student produced, and each transition is a possible action a student has taken. Different reward functions were used in the MDP based on the profile of the student. For instance, a "least error-prone" reward would be used for at-risk students. Based on this reward, a desirable state for the student to go to would be chosen using the MDP. For that state, there were different levels of hints to be given:

Goal-setting hint — indicate a goal statement to derive.

Rule hint — tell the student what rule to apply next.

Pointing hint — indicate the assertions where the rule can be used.

Bottom-out hint — tell the student both the rule and the assertions to combine.

If a student landed in a state that was not part of the MDP, then the hint button would be disabled. At that point, the tutoring system was not able to provide strategic help, since it does not have proof search capabilities like AProS.

2.4.7 Gries and Schneider's formal logical system for computer science

Most of the work mentioned thus far concerns the teaching of proof topics outside of theoretical computer science, such as formal logic and geometry—topics where the usage of formal logical systems is prevalent. The formal logical system developed by David Gries and Fred Schneider [64, 65], on the other hand, is uniquely relevant in relation to the goals of this dissertation, because its design was pedagogically motivated for teaching an entire course in discrete mathematics, the foundation on which theoretical computer science is built.

Their system is a predicate calculus that gives prominence to equivalence relations over the implication relations used in classical formal logical systems. It is based on earlier research developing *equational logics*, such as the work of Edsger Dijkstra and

Carel Scholten [48]. These logics have been developed by computer scientists for the formal analysis of computer programs [63]. The formalization of an equational logic, in terms of soundness and completeness, is shown in this paper [66].

In a technical report [65], Gries and Schneider suggest that their system should be intuitive to college students since it builds off of students' pre-existing familiarity with algebraic equation manipulation, which is taught in grade school. They also argue that their system produces proofs that are simpler and more straight-forward than the narrative proofs found in many textbooks. As an example, they give the following proof, using their equational logic, of the set theory proposition $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$, i.e., that union distributes over intersection:

$$\begin{aligned} & v \in A \cup (B \cap C) \\ = \quad & v \in A \vee v \in B \cap C \text{ (by Definition of } \cup) \\ = \quad & v \in A \vee (v \in B \wedge v \in C) \text{ (by Definition of } \cap) \\ = \quad & (v \in A \vee v \in B) \wedge (v \in A \vee v \in C) \text{ (by Distributivity of } \vee \text{ over } \wedge) \\ = \quad & (v \in A \cup B) \wedge (v \in A \cup C) \text{ (by Definition of } \cup\text{, applied twice)} \\ = \quad & v \in (A \cup B) \cap (A \cup C) \text{ (by Definition of } \cap) \end{aligned}$$

Then by the axiom of Extensionality from their logic, it follows that $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$.

In contrast, standard textbook proofs [72, 95] for the same proposition require proving two separate cases—$A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$ and $(A \cup B) \cap (A \cup C) \subseteq A \cup (B \cap C)$. The equational logic removes the need to prove these cases separately. Furthermore, Gries and Schneider argue that textbook narrative proofs are lengthened with unnecessary prose that obscures the proof strategies a student should learn—strategies that are made explicit with their formal logical system.

Gries and Schneider developed a textbook [64] for teaching discrete mathematics—the topics of set theory, graph theory, number theory, combinatorics, and the analysis of programs with Hoare formalizations, are all developed in this textbook, using their formal logical system. Using this textbook, Cornell University's Computer Science Department offered a discrete mathematics course that taught proofs using the system. Of the 70 students who took this course during the Spring 1993 semester at Cornell, 65 wrote "overwhelmingly positive" comments about it [65].

More recently, Wolfram Kahl, who also teaches courses using Gries and Schneider's formal logical system, developed an automated proof verifier for it called CalcCheck [83]. Kahl's reason for creating CalcCheck was to give students immediate feedback on the proofs they construct, which is also a central motivation for the development of Complexity Tutor. However, one potential advantage of Complexity Tutor over the approach of using Gries and Schneider's formal logical system with an automated proof verifier like CalcCheck, is that Complexity Tutor does not require instructors to radically change how they teach their courses. Whether or not using a formal logical system like Gries and Schneider's is a superior way to teach theoretical computer science topics than using narrative proofs, it is certainly not the norm. Complexity Tutor was designed to be useful to students even when instructors are not willing to change the way they present the material they are teaching.

CHAPTER 3

ARCHITECTURE AND SYSTEM DESIGN CHOICES

Complexity Tutor currently has two modules—the Theorem Proving Environment and the Algorithm Environment. This chapter introduces both of these modules and explains the motivation of their design.

3.1 Theorem Proving Environment

The Theorem Proving Environment recasts the task of theorem proving as a kind of puzzle, which can be solved using a simple drag-and-drop interface. A collection of assumptions and assertions are given as "puzzle pieces" to be used to construct a proof. An assumption is a statement considered valid without need of further substantiation, such as an axiom. An assertion is a statement that needs substantiation before it can be considered valid. Assertions may be validated from assumptions and other assertions.

The interface (Figure 3.1) consists of a Proof Space, an Assumption Box and an Assertions Box. Interactions with the interface can be performed entirely with a computer mouse or similar pointing device. Alternatively, the interactions could be adapted in a straight-forward manner to a touch screen interface like a tablet.

3.1.1 Proof Space

The Proof Space is a canvas filled with statements (assumptions and assertions) and arrows connecting them to each other. Next to each statement is a status indicator, represented by a dot. There are three possible statuses for each statement—

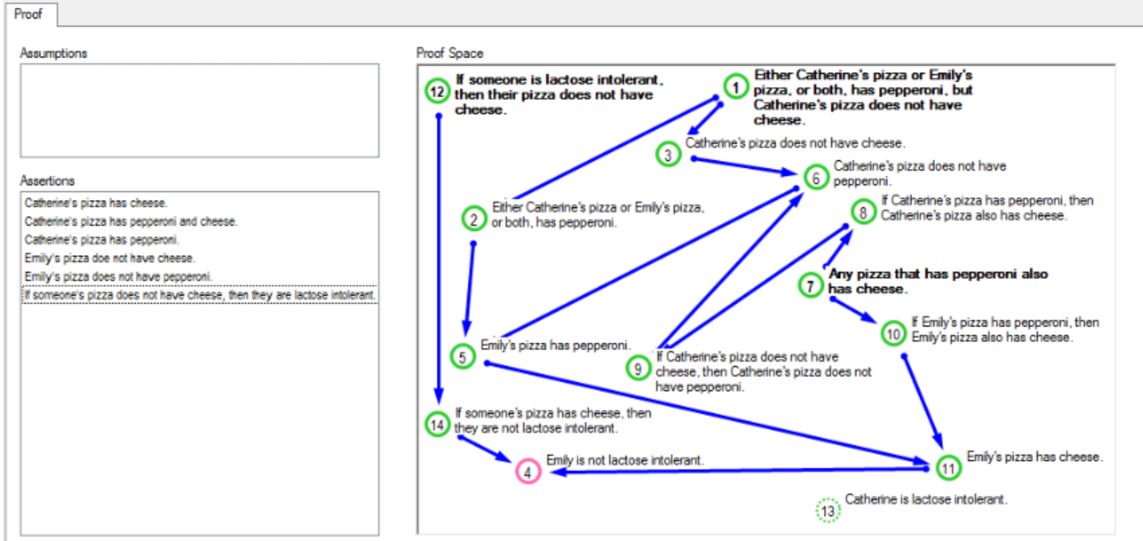


Figure 3.1: Screenshot of completed Pizza Proof Problem in the Theorem Proving Environment. The Assumptions Box is on the top left, the Assertions Box is on the bottom left, and the Proof Space is on the right.

justified, unjustified and erroneous—indicated respectively by a complete dot, incomplete dot or hashed dot, as illustrated in Figure 3.2. The justified status indicates that a statement has already been proven or that it does not need to be proven because it is an assumption. The unjustified status indicates that a statement may or may not be provable but has not yet been proven. The erroneous status indicates that a statement is known to be incorrect or nonsensical. Each dot is also labeled with a number, which is a unique identifier for its associated statement.

An arrow pointing from a given statement A to a given statement B indicates that statement A is being used to partially or fully justify statement B.

The visual arrangement of the Proof Space can be altered by dragging statements to different locations in the Proof Space. If a statement is dragged past the right or bottom borders of the Proof Space, then scroll bars will automatically appear, allowing the Proof Space to be scrolled.

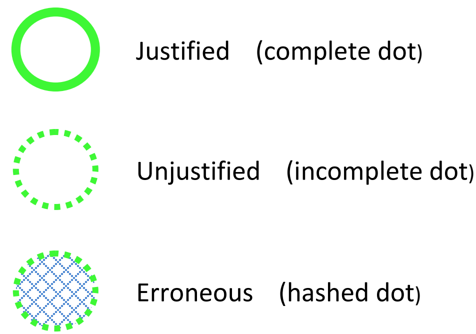


Figure 3.2: Status indicators used in the Theorem Proving Environment

3.1.2 Assumptions Box

The Assumptions Box contains a list of assumptions that the Theorem Proving Environment will allow to be used for the current proof problem that is being working on. This would include assumptions that pertain to the specific problem, and may also include assumptions derived from facts that have been previously taught, i.e., definitions, theorems and lemmas from the course textbook, or even claims that have been proven in previous exercises.

Assumptions chosen from the Assumption Box can be dragged onto the Proof Space, where they will immediately receive a justified status indicator.

3.1.3 Assertions Box

The Assertions Box contains a list of assertions that the Theorem Proving Environment will allow to be used for the current proof problem that is being worked on. One of the assertions corresponds to the goal of the problem, i.e., the statement that needs to be proved to solve the problem. Of the remaining assertions, some may be useful in the endeavor of proving the goal assertion, once justified. Others can be justified, but won't be useful for proving the goal assertion. Finally, there are some assertions that cannot be justified.

Assertions chosen from the Assertions Box can be dragged onto the Proof Space, where they will immediately receive either an unjustified status indicator or an erroneous status indicator.

For a given problem, there may be a very large pool of possible assertions that can be used—potentially hundreds. The Theorem Proving Environment can be configured so that initially, the Assertions Box only reveals a limited selection of these possible assertions. As progress is made by the student over time, more assertions get unlocked and added to the Assertions Box—i.e., using specific assumptions and assertions in the Proof Space triggers the unlocking of other assertions. The purpose of this feature is to keep students from being overwhelmed with having to consider too many assertions at once.

3.1.4 Demonstration of solving a simple proof problem

To illustrate how a student would use the Theorem Proving Environment, consider the following problem:

> “Emily and Catherine each have their own pizza. Using the given assumptions, prove that Emily is not lactose intolerant.”

These are the assumptions for the problem:

1. ”Either Catherine’s pizza or Emily’s pizza, or both, has pepperoni, but Catherine’s pizza does not have cheese.”

2. ”Any pizza that has pepperoni also has cheese.”

3. ”If someone is lactose intolerant, then their pizza does not have cheese.”

Figure 3.1 is a screenshot of the completed proof for this problem in the Theorem Proving Environment.

3.1.4.1 Choosing assumptions and assertions

After studying the three assumptions and the assertions, in Figure 3.3, suppose the student decides to drag the assumption "Either Catherine's pizza or Emily's pizza, or both, has pepperoni, but Catherine's pizza does not have cheese." into the Proof Space. Notice that when this assumption is moved into the Proof Space, the text of the assumption appears with a dot "1" adjacent to the text. Also notice the text of the assumption is displayed in bold print to distinguish it from assertions which are displayed in regular print.

The student then determines what should follow this assumption. The student peruses the list of assertions, in Figure 3.3, decides the assertions "Either Catherine's pizza or Emily's pizza, or both, has pepperoni." and "Catherine's pizza does not have cheese." could follow the assumption in the Proof Space, and drags these two assertions into the Proof Space. The text of the two assertions appear with partial dots "2" and "3", indicating that the associated assertions need substantiation to be justified. When these assertions are justified, the partial dots will change to complete dots.

3.1.4.2 Justifying assertions

To see if the belief that the two assertions follow the assumption was correct, the student left clicks dot "1" (assumption) followed by a left click of partial dot "2" (assertion). An arrow will appear to connect the dots with the arrow pointing at dot "2" as shown in Figure 3.4. Also notice that the partial dot "2" has changed to a complete dot "2" indicating the assertion was justified by the connection with dot "1". The student repeats this process to connect dots "1" and "3". As shown in Figure 3.4, an arrow connects the two dots and the partial dot "3" is changed to a complete dot indicating the assertion associated with dot "3" was substantiated and justified by the connection.

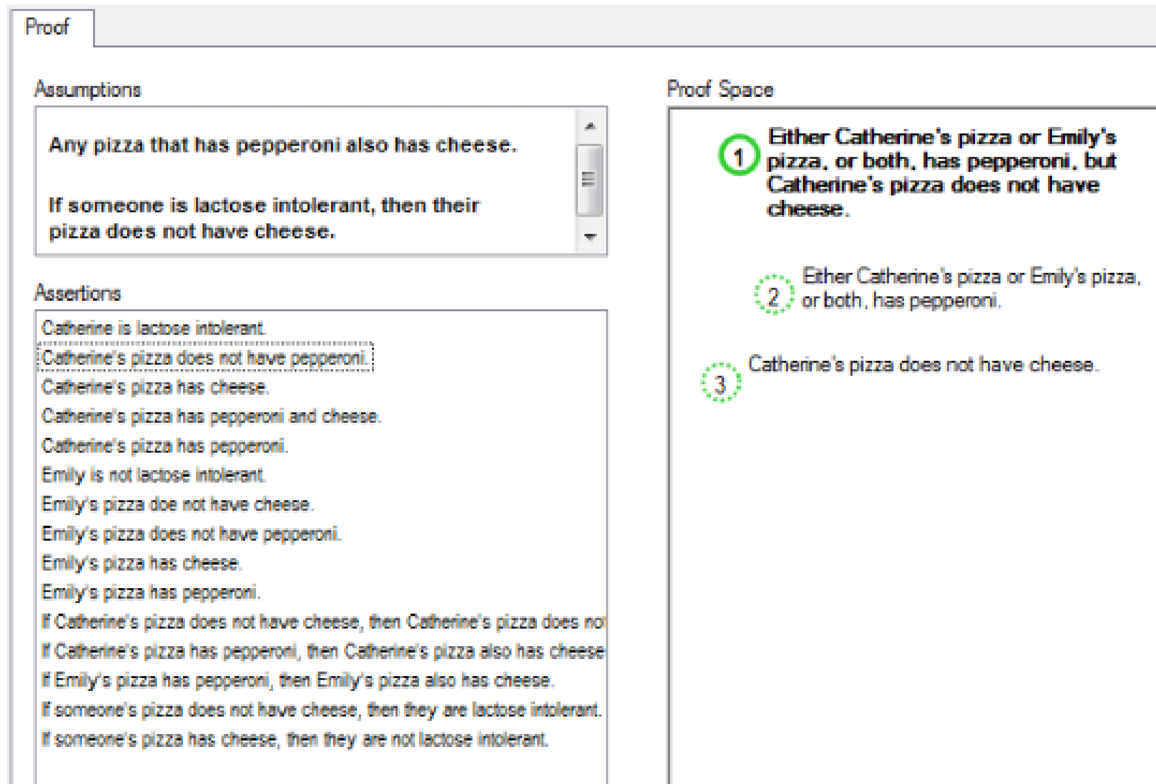


Figure 3.3: Moving assumptions and assertions into the Proof Space, assumptions are indicated by complete dots and assertions are indicated by partial dots.

Notice in Figure 3.4 that dots “2” and “3” have moved from their original locations shown in Figure 3.3. The Theorem Proving Environment permits the student to drag the dots to different locations in the Proof Space to arrange the proof spatially in the way that makes the most sense to him/her.

Also notice the text associated with dot “3” is not shown in Figure 3.4. The Theorem Proving Environment allows the student to make the text associated with a dot disappear and reappear by double clicking the dot. This feature allows the student to hide the text of dots they are not considering at the moment. This can reduce cognitive load and promote better focus and efficiency for the student to solve the part of the problem they are currently working on [38]. Additionally, when students hover their mouse pointer over any dot, it will intermittently display the associated assertion, if it has been hidden.

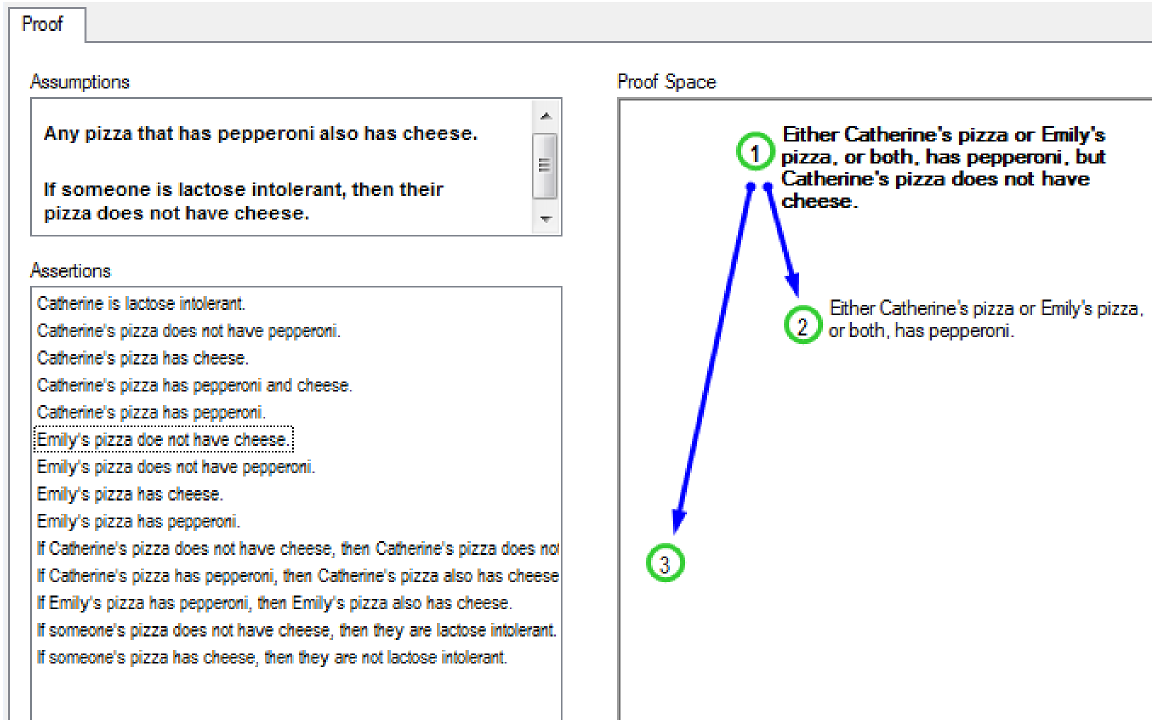


Figure 3.4: Connecting the dots, validated assertions become complete dots, toggling text and relocating dots.

3.1.4.3 Completing the proof

The student continues selecting assertions and assumptions, moving them into the Proof Space, and connecting them until the proof is completed. The proof is completed when the goal assertion "Emily is not lactose intolerant" is justified (its partial dot has been converted into a complete dot) and all assertions leading to the goal assertion have also been justified. Notice that Figure 3.1 shows the goal assertion dot "4" and all of the other dots connected to it are complete dots. Dot "13" remains unjustified, but that is because that assertion was never needed for the proof. Complexity Tutor will also display a message on the computer screen that the proof has been successfully completed.

Notice the remaining assertions in the Assertions Box even after the proof is completed in Figure 3.1. These assertions are *distractors*, unnecessary to complete the proof. Some of the distractors are valid assertions that can actually be justified in

the proof space, but will lead down paths that do not connect with the goal. Others are "bugs" that get denoted with hashed dots when dragged into the proof space. The latter could be used by instructors to correct common misunderstandings that students might have.

3.1.5 Comparison to standard proof writing exercises

Typically, in a theoretical computer science or math course, students are assigned problems where the task is to write a proof from scratch. This task has notable differences from the proof construction puzzle task presented by the Theorem Proving Environment, demonstrated in Section 3.1.4. A key difference is that unlike normal proof writing, the Theorem Proving Environment presents students with a list of possible assertions to be used in constructing the proof.

However, providing the student with pre-formed assertions useful to the proof should reduce extraneous cognitive load. Cognitive load theory researchers recommend substituting worked examples and completion problems in place of conventional problem solving tasks. A completion problem is one where a partial solution is provided to the student with missing pieces that the student must determine. The recommendation is based on the theory that the reduced extraneous cognitive load of worked examples and completion problems leads to more effective learning than having the student do the equivalent conventional problem solving task [155]. Support for this theory has been found by education researchers in a number of different subjects, including algebra [43, 154], geometry [121, 139], statistics [120], and programming [157, 160, 161].

Computer science education researchers have looked at Parsons Problems [122] as an example of a type of completion problem that could be used to reduce the cognitive load of learning programming [51]. Parsons Problems are puzzles, where students are given blocks of code that they need to rearrange to create a program

with desired functionality. This is analogous to a proof puzzle, where the students would be given assertions that they need to rearrange to form a proof. A study found a strong correlation between the ability to solve Parsons Problems and the ability to correctly write code from scratch, and concluded that these tasks likely require the same skills [46]. The authors of that study in fact suggested that learning to write mathematical proofs might benefit from an approach similar to Parsons Problems.

3.1.5.1 Proofs as graphs

Proofs constructed in the Theorem Proving Environment are not expressed in the narrative structure that is the norm for mathematical discourse. Rather, they are expressed visually as inference graphs, which explicitly show the implication relations between different assertions. Yet, there are reasons to believe that it is pedagogically superior to train students to construct proofs graphically.

When an expert is constructing a proof, cognitively it is first represented as a graph of inferences, and then subsequently that graph is mapped onto a narrative argument. For students who are learning to construct and understand proofs, this additional step also adds a layer of extraneous cognitive load.

Furthermore, when a student is only taught to write proofs in the flow of a narrative, they are more likely to view proof construction as a linear process, where by a series of deductive rules are applied in sequence starting from the assumptions. In comparison, the Theorem Proving Environment encourages students to think non-linearly about proofs, and move away from thinking that the steps of a proof need to be produced in the order they appear in narrative. With this interface, it is just as easy for a student to work forward from the initial assumptions in their proof as it is to work backward from the goals. Bi-directional search is a proof strategy which alternates between working forward from assumptions and backward from goals—it

has been proposed as a cognitive model for how skilled geometry students produce proofs [6].

The student can even start out by trying to prove a lemma that might at first seem unrelated to the initial assumptions and goals, but for which the student thinks it might be helpful later. If the student gets stuck at any point in the part of the proof they're working on, they can switch to working on another part of the proof. This means that the student is less likely to give up out of frustration. The freedom of exploration encouraged by the Theorem Proving Environment may also lend itself to the advantages witnessed in the *goal-free effect*, a principle from cognitive load theory that demonstrates learners are more effective when they are deriving knowledge from a problem without concern for attaining a particular goal [155].

Others have seen the benefit of representing proofs as inference graphs. Two intelligent tutoring systems for geometry proofs described in Section 2.4.3, GPT [5] and ANGLE [89], use an interface similar to the Theorem Proving Environment to construct proof graphs. The Deep Thought [112] system described in Section 2.4.6 also uses a graphical representation of proofs.

In a study of geometry students in Taiwan, students used a geometry software called MR Geo, where they could interact with multiple representations of geometry proofs, including both a formal proof representation similar to the two-column format often taught in high school, and a graphical representation where the proofs were visualized as trees. The students in the study were divided into three groups based on a pre-test—high-achievement, medium-achievement and low-achievement. The study found that while the high-achievement and medium-achievement groups found equal comfort with proof tree representation and the formal proof representation, the low-achievement group strongly preferred the proof tree over the formal proof [168]. This is evidence that the students who struggle the most are also most likely to benefit from a proof construction framework that expresses proofs graphically.

Cong-Cong Xing, a computer science educator at Nicholls State University in Louisiana, used a similar graphical proof format to teach set theory in a discrete math course [171]. Prior to the introduction of graphical proofs in his course, his experience was that his students failed to understand narrative set theory proofs, due to the fact that his students had limited mathematical ability and limited background with English. After the introduction of the graphical format, he reported that he got a lot of positive feedback from students and that learning outcomes were improved [171].

Finally, there is also a technical motivation for the design choice of using graphs to represent proofs in the Theorem Proving Environment. Representing proofs as graphs makes the inferences in the proof very explicit, not only to the learner but also to the computer that is tutoring the learner. A significant problem with using narrative discourse structure to represent proofs is that natural languages permit a lot of ambiguities. A single sentence in English can correspond to many different valid syntactic parse trees, each with completely different semantic interpretations. On top of that, when you look at larger pieces of discourse such as a narrative proof, anaphora resolution becomes a real challenge. Anaphora resolution is the problem of matching references of discourse elements with the earlier discourse elements they refer to. For instance, if the student uses the pronoun "it" in a proof, it might be very ambiguous what "it" refers to.

Writing a proof that is unambiguous is difficult when there are many things in the proof that may need to be referenced – the analogy to anaphora resolution would be having a programming language with only a fixed number of local variables you can refer to at any time. Exclusively using the pronouns of English or any other natural language to describe anything sufficiently complex in an unambiguous way is not much different than writing a complex computer program in assembly language that only uses a limited number of hardware registers, and swapping back and forth between those registers to access everything you need to reference.

Anaphora resolution has turned out to be a very difficult problem for computational linguists to address [110], and so the state-of-the-art in natural language processing (NLP) is still far from being able to fully process even technical discourse like mathematical proofs [173], where a well-trained writer will try to avoid ambiguities. Of course, students tend to not have much experience in mathematical writing, and as a teaching assistant, the author of this dissertation found that their proofs often have so many anaphora ambiguities that even when being read by a human who has complete contextual knowledge of what they are trying to argue, it can at times still be hard to discern what they are trying to say. It is therefore fruitless to even think about using NLP to have a computer process student discourse of proofs, because NLP cannot be expected to ever be better than human understanding of natural language, and often students write proofs that are difficult even for a human domain expert to understand.

One alternative to natural language that was briefly considered for Complexity Tutor was having students specify their proofs in a *controlled natural language* (see [58,101,123,137,138,172] for examples). A controlled natural language is a subset of a natural language (such as English), where every sentence you can construct has one unique and unambiguous semantic interpretation. However, this idea was rejected, not only because the learning curve of students learning such a language might be rather steep, but because of the technical challenges in developing such a language.

It is the belief of the author that while the Theorem Proving Environment has students constructing proofs in a format that is very different from how they will see those same proofs presented in their textbooks and lectures, this does not hinder either the pedagogical goal of helping a student produce and comprehend proofs, or the goal of helping them understand the course material. Instead, it should actually benefit those pedagogical goals, since it removes the writing barrier.

3.1.6 Requirements for domain knowledge representation

For each given problem, the domain knowledge used by the Theorem Proving Environment to help the student with their proof can be formally represented as a *directed hypergraph*.

The *vertices* of this hypergraph represent all possible assertions and assumptions that can ever be used in proofs that can be constructed in the Theorem Proving Environment for the given problem. One vertex will be the *goal vertex*, corresponding to the assertion that must be proved for the student to have completed the problem. Some assertions are labeled as being erroneous. An erroneous assertion may infer non-erroneous assertions, but it may not be inferred by non-erroneous assertions.

The *hyperarcs* of the hypergraph represent inference relations. Suppose there are assumptions and assertions represented by vertices A, B, C, D, E, and F. The domain expert might want to represent that the assertion corresponding to A can be justified by the assumptions and assertions corresponding to B, C, and D, and additionally this same assertion can alternatively be justified by assumptions and assertions corresponding to D,E, and F. This would be represented by two distinct hyperarcs, $\{A\} \rightarrow \{B, C, D\}$ and $\{A\} \rightarrow \{D, E, F\}$. In general, a single hyperarc represents that a given assertion can be justified by a set of assumptions and assertions.

It should be noted that the current implementation of the Theorem Proving Environment uses a more restricted representation of knowledge. This restricted representation does not permit there to be more than one way to justify any assertion. So for instance, if $\{A\} \rightarrow \{B, C, D\}$ was a hyperarc, then there would be no other hyperarcs that also contain $\{A\}$ as the *source* vertex. This restricted representation can also be modeled by a simple *directed graph*, where if the assertion corresponding to A can be justified by the assumptions and assertions corresponding to B, C, and D, then the graph contains arcs $A \rightarrow B$, $A \rightarrow C$ and $A \rightarrow D$.

3.1.7 New features added to the Theorem Proving Environment after preliminary experimental testing

Two new major features were added to improve the Theorem Proving Environment after preliminary experiments evaluating Complexity Tutor had been conducted during the Fall 2016 and Spring 2017 semesters. This section describes both of these features, which were designed to increase the pedagogical utility of Complexity Tutor and to decrease extraneous cognitive load for its users.

3.1.7.1 Visual hint mechanism for coarse-grained inferences

Recall the notion of *proof granularity* discussed in Section 2.1.4. While it's easy for the domain expert or whoever authors problems for Complexity Tutor to choose the level of proof granularity that will be required for a given problem, the granularity level must still be fixed, and currently the Theorem Proving Environment rigidly requires all students to construct proofs at that fixed level of granularity.

Future work will look at adapting proof granularity expectations to the needs of individual students, by learning a student model based on a student's prior performance (Section 6.6). However, in the mean time, a mechanism has been designed to help students when they attempt coarse-grained inferences in the Theorem Proving Environment.

Early versions of Complexity Tutor that were tested gave no feedback when a student attempted to produce a coarse-grained inference. This is not ideal. Consider that the absence of feedback is in itself a form of feedback. The student is being told that the inference is not acceptable, because no arrow is produced, but they don't know whether the inference is not acceptable because it is completely incorrect or because it has the wrong level of granularity. The student may get confused and assume that their inference is completely invalid, when in actuality it is just too coarse-grained.

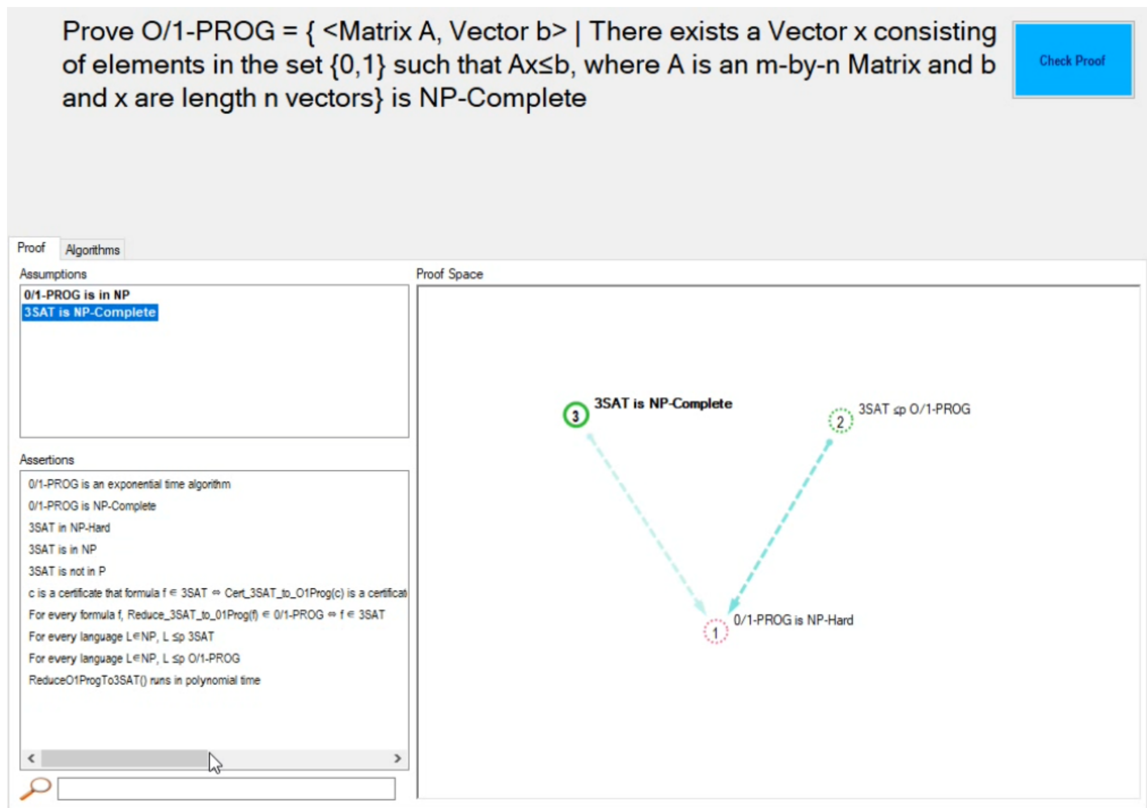


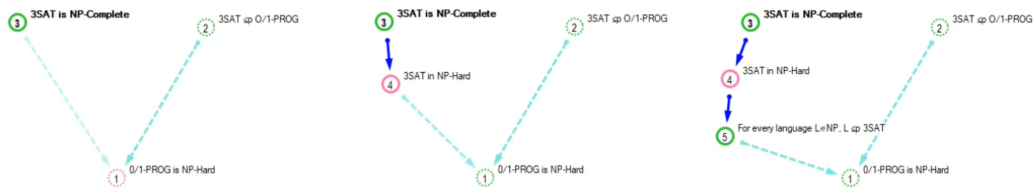
Figure 3.5: Screenshot of a student producing a low-grained proof sketch in the Theorem Proving Environment, to show that 0/1-PROG is NP-Hard.

In comparison, with hand-written proofs, the student gets no immediate feedback about any statement in their proof, and it is later the responsibility of a human grader to decide if individual proof steps have an acceptable level of granularity. Generally, if a given proof step is only slightly more coarse-grained than desired, the grader may still give full credit for it, and they may give partial credit for other proof steps that are significantly too coarse-grained. In the latter case, the human grader will likely also give some feedback to the student to indicate that the inference is too coarse-grained.

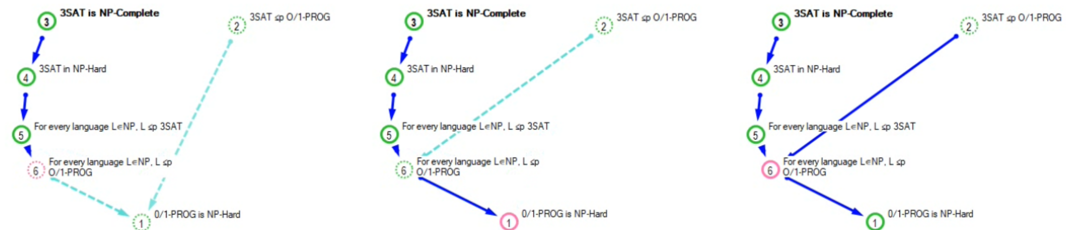
Figure 3.5 illustrates the new mechanism in the Theorem Proving Environment for giving feedback for coarse-grained inferences. The student attempts to show that the statements “3SAT is NP-Complete” and “3SAT $\leq_p$ 0/1-PROG” imply the state-

ment “0/1-PROG is NP-Hard”. They do so by clicking dot “3” and then dot “1”, and likewise clicking dot “2” and then dot “1”. However, instead of normal arrows being produced, *hint-lines* are produced, which indicate this to be a coarse-grained inference. The hint-lines are visually represented as arrows with dashed lines.

Notice that in Figure 3.5, the hint-line between dots “3” and “1” is slightly more faded than the hint-line between dots “2” and “1”. The rule for creating a hint-line is that there must be a path of chained inferences leading from one statement to another, however the paths between “3” and “1” and between “2” and “1” are of different lengths. The longer the path is, the more faded and transparent the hint-line becomes. When the path becomes too long, the hint-line becomes invisible, and beyond that it is non-existent. Thus, hint-lines visually indicate the existence of a coarse-grained inference as well as the level of its granularity.



(a) Initial proof sketch (b) First step of refinement (c) Second step of refinement



(d) Third step of refinement (e) Forth step of refinement (f) Fifth step of refinement

Figure 3.6: Hint-lines help to refine a proof sketch into a finer-grained proof.

Furthermore, the hint-lines used in Figure 3.5 idiomatically form a proof sketch. Students can now start with a proof sketch like in Figure 3.5, and refine it in the

Theorem Proving Environment into a fine-grained proof, which is illustrated in Figure 3.6. This process may improve a student's intuition for theorem proving, by showing them how to start with a general idea for a proof and refine it into a detailed proof.

In this process, hint-lines are automatically updated in the Proof Space as new connections are made. Figure 3.6a shows the initial proof sketch with a hint-line leading from "3SAT is NP-Complete" to "0/1-PROG is NP-Hard", and a hint-line also leading from "3SAT $\leq_p$ 0/1-PROG" to "0/1-PROG is NP-Hard". In Figure 3.6b, the student uses the definition of NP-Completeness to infer the assertion "3SAT is NP-Hard" from "3SAT is NP-Complete". The hint-line that was previously leading from "3SAT is NP-Complete" to "0/1-PROG is NP-Hard" is replaced by a brighter hint-line leading from "3SAT is NP-Hard" to "0/1-PROG is NP-Hard". This is because "3SAT is NP-Hard" is the next step on the path from "3SAT is NP-Complete" to "0/1-PROG is NP-Hard". Next, in Figure 3.6c, the student realizes that they should expand the definition of NP-Hardness. So they have shown that "3SAT is NP-Complete" implies "3SAT is NP-Hard", and they have shown that this in turn implies "For every language $L \in$ NP, $L \leq_p$ 3SAT". The hint-line now leads from that last assertion in this chain of inferences to "0/1-PROG is NP-Hard". Similarly, in Figure 3.6d, the assertion "For every language $L \in$ NP, $L \leq_p$ 0/1-PROG" is added to the chain of inferences, and the hint-line is then updated to lead from this assertion to "0/1-PROG is NP-Hard". Note that "For every language $L \in$ NP, $L \leq_p$ 0/1-PROG" directly implies "0/1-PROG is NP-Hard", however the student still needs to show that they know this by directly clicking on the two assertions to produce a connection, which they do in Figure 3.6e. But this then updates the hint-line that stemmed from "3SAT $\leq_p$ 0/1-PROG", so that it now leads to "For every language $L \in$ NP, $L \leq_p$ 0/1-PROG". In Figure 3.6f, the student removes this final hint-line by making the connection from "3SAT $\leq_p$ 0/1-PROG" to "For every language $L \in$ NP, $L \leq_p$ 0/1-PROG".

What would happen if the student made these inferences in a different order than shown in Figure 3.6? Suppose that after producing the initial proof sketch, the student then connected “3SAT is NP-Hard” to its definition, “For every language $L \in$ NP, $L \leq_p$ 3SAT”. That latter action would not update the hint-lines, since it's not clear that the student realizes that this inference they just made has anything to do with refining the coarse-grained proof that “0/1-PROG is NP-Hard” follows from “3-SAT is NP-Complete”. However, if the next action that the student made was to connect “3-SAT is NP-Complete” to “3-SAT is NP-Hard”, then the hint-line would get updated, resulting in what is shown in Figure 3.6c.

Before describing the actual procedure that updates the hint-lines, it is important to point out that for any two statements A and B, there could in theory be multiple paths from A to B in a given proof, since proofs can be directed acyclic graphs. So in some cases, it will be ambiguous which path a given hint-line should correspond to.

In the current implementation of hint-lines, each hint-line corresponds to a specific path at the time it is generated, and that specific path is chosen to be a shortest path when there are multiple possibilities. So, assume that a given hint-line from statement A_1 to statement A_n corresponds to a path of statements $A_1, A_2, \ldots, A_{n-1}, A_n$.

Then the procedure to update that hint-line is as follows. Let j be the maximal value such that $1 \leq j < n$ and for every edge in the path $A_1, \ldots, A_j$, an arrow has been produced in the Proof Space. Let k be the minimal value such that $1 < j \leq n$ and for every edge in the path $A_k, \ldots, A_n$, an arrow has been produced in the Proof Space. If $j \geq k$ then remove the hint-line entirely, since it would no longer be needed. Otherwise, replace the hint-line by one from statement A_j to statement A_k with corresponding path $A_j, \ldots, A_k$.

What if the path that was initially chosen for a given hint-line does not correspond to the coarse-grained inference the student had in mind? This could potentially lead to some confusion, although it is doubtful whether this hypothetical scenario would

even occur. In fact, the notion of what an inference is implies that it should be conceptually unique in a proof, regardless of its level of granularity. Therefore, one could argue that in most circumstances, a coarse-grained inference from statement A to statement B should only be considered if there is a unique path from A to B. In the rare circumstances where that is not the case, the shortest path heuristic should usually pick the coarse-grained inference that is most appropriate.

Finally, it should be mentioned that there is a limit placed upon the number of hint-lines that will be displayed simultaneously in the Theorem Proving Environment. Currently, this limit is set to be 10 hint-lines. Once that limit has been reached, no new hint-lines will be produced until the student removes some of the existing hint-lines by refining their proof through the process shown in Figure 3.6.

3.1.7.2 Search Box for finding assertions

While listing all possible assertions was argued in Section 3.1.5 to be a benefit for students learning to construct proofs, since it removes the cognitive load of students having to figure out how to correctly express the assertions needed for a proof, extraneous cognitive load also increases as the list of possible assertions grows in size. When the Assertions Box has close to a hundred possible assertions in it, sifting through those assertions to find the ones that are relevant to the proof step being worked on is like searching for a needle in a haystack. Indeed, students using the Theorem Proving Environment were frequently observed scrolling back and forth through the Assertions Box multiple times before selecting assertions.

Search is a natural solution for mitigating this problem, which users should already be acquainted with since search engines like Google are ubiquitous in society. Consequently, a simple search engine feature was integrated with the existing Assertions Box interface. The user is now presented with a Search Box underneath the Assertions Box—as soon as the user begins typing only a few characters into the

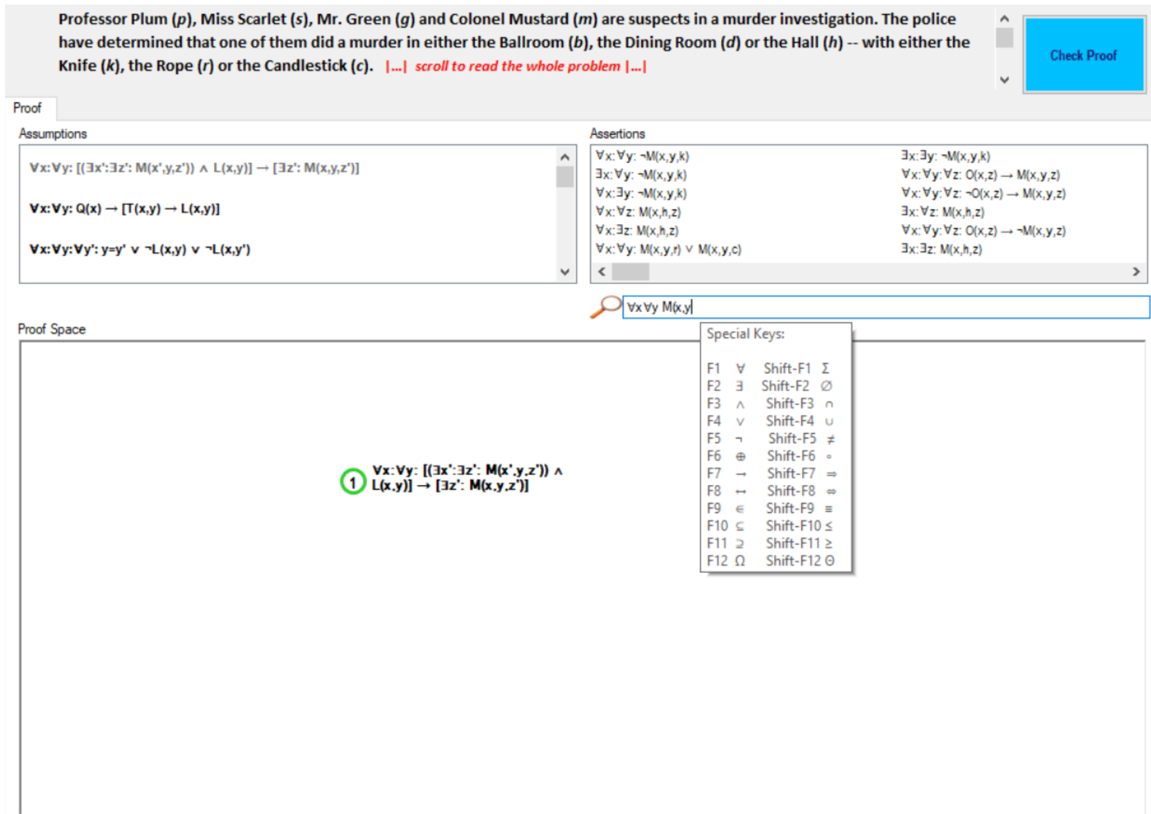


Figure 3.7: Screenshot of a symbolic formula being entered in the Search Box. Assertions in the Assertions Box are ordered by their relevance to what has been typed.

Search Box, the assertions in the Assertions Box are dynamically reordered on the fly in terms of their relevance to what the user is typing at that moment. Later, the user can choose to reorder the Assertions Box alphabetically for manual perusing.

The users can also type mathematical symbols not found on their keyboard into the Search Box using simple key sequences, which are referenced in a tooltip that is displayed to the user when they hover the mouse pointer over the Search Box. Figure 3.7 shows this.

Under the hood of the Search Box is a *search algorithm*, which takes the search *query* that the user types and produces an ordered *ranking* of the assertions. There are different popular strategies for designing search algorithms, but the search algorithm that was ultimately chosen was designed with the following considerations:

1. Users should be able to express assertions in their own words, as they would if they were writing a proof with pen and paper, and the corresponding assertion in the Assertions Box (if one exists) should be highly ranked. For instance, if the user were to type "Emily or Catherine has a pizza with pepperoni" as their query, then the assertion "Either Catherine's pizza or Emily's pizza, or both, has pepperoni" should be highly ranked.

2. The search algorithm should produce good search results if a user types fragments of an assertion as a query, rather than a full assertion. For instance, the user may want to find all assertions that reference the entity "Emily" or all assertions that refer to someone being "lactose intolerant". Thus, the user is not required to express complete assertions to make use of the search box. Also, if a user stumbles upon an assertion in the Assertions Box at some point, and later realizes it might be useful, they can quickly find it without needing to fully recall what the assertion precisely stated.

3. The search algorithm should be reasonably resilient to spelling and typographical errors. If the user types "cathrine" then assertions that contain "Catherine" should be highly ranked.

4. The user may type symbolic expressions or formulas as part of their query, and the search algorithm should return useful results even if the symbolic expression that the user typed does not exactly appear in any of the assertions in the Assertions Box.

5. It would be nice if the search algorithm is able to return the assertion that the user is looking for before they have even finished typing their query.

While the first consideration might seem to imply the user's query would need to be semantically parsed in order to find the semantically equivalent assertion in

the Assertions Box, that is actually not going to be necessary most of the time. Notice that assertions that are semantically similar tend to share syntactically similar words. The assertion "Emily or Catherine has a pizza with pepperoni" shares 6 stem words in common with the semantically similar assertion "Either Catherine's pizza or Emily's pizza, or both, has pepperoni". Furthermore, there are not likely to be many assertions in the Assertions Box that use those 6 stem words.

Therefore, a *bag-of-words model*, which ranks assertions according to how many stem words are matched with the query, would suffice for addressing the first two considerations. However, such a model would not address the remaining considerations.

To address all five considerations, a *character-level n-gram model* was used instead to create a character-level measure of string similarity between query strings and the assertions in the Assertions Box. Specifically, strings are modeled in terms of their *character bigrams* and *character unigrams.* A character bigram is a pair of characters appearing in sequence in a string, and a character unigram is simply a single character appearing in a string. Intuitively, two strings x and y can be thought of as similar, if most bigrams and unigrams in x map to corresponding bigrams and unigrams in y and vice-versa.

This can be formalized by representing each string as a *multiset* containing its character bigrams and character unigrams. For instance, the string "pepperoni pizza" would be represented by the multiset {'pe', 'ep', 'pp', 'pe', 'er', 'ro', 'on', 'ni', 'i ', ' p', 'pi', 'iz', 'zz', 'za', 'p', 'e', 'p', 'p', 'e', 'r', 'o', 'n', 'i', ' ', 'p', 'i', 'z', 'z', 'a'}. Notice the multiplicities, such as the bigram 'pe' occurring twice and the unigram 'p' occurring four times.

Then, for any two strings x and y, with corresponding multisets X and Y, the cardinality of $X \cap Y$ gives a measure of how similar x and y are. Using the following formula, this similarity measure can be normalized so that any two strings have a similarity score that ranges from 0 to 1:

$$\text{similarity}(x, y) = \frac{2|X \cap Y|}{|X + Y|} = \frac{2|X \cap Y|}{(2|x| - 1) + (2|y| - 1)} = \frac{|X \cap Y|}{|x| + |y| - 1}$$

Given a query string q, the search algorithm computes similarity(q, a) for every assertion string a, and ranks them according to their scores.

3.2 Algorithm Environment

A distinguishing characteristic of many proofs in theoretical computer science is that they often reference algorithms. When a student is asked to construct such a proof, they will implicitly also have to construct an algorithm, and the correctness of the proof will depend on the correctness of the algorithm.

In normal pen-and-paper proofs, students will describe these algorithms using pseudocode. The Algorithm Environment simulates this process by having students write actual Python code in place of pseudocode, which can be computationally interpreted and evaluated for correctness. In principle, this could be done with any programming language but Python was chosen for its succinctness and its similarity to pseudocode. Guido van Rossum, the creator of the language, considers its resemblance to "executable pseudocode" to be one of its main strengths [162].

Here are two justifications for having students write algorithms in a real programming language like Python rather than pseudocode:

1. Actual code is always concrete and unambiguous in its interpretation. Good pseudocode should also be unambiguous, however students do not always write good pseudocode. Often times, the pseudocode that students write is ambiguous in its specification, and it may omit details that the student thinks are unimportant to the correctness of the algorithm but actually are important. Forcing a student to actually write code for an algorithm that a computer can understand forces them to show they fully understand all the details of the algorithm.

2. A computer can interpret a student's algorithms when it is written with real code, permitting an intelligent tutoring system to give feedback and guidance to the student about the algorithm. In contrast, pseudocode is only ever intended for human consumption. A programming language like Python has many of the same expressive benefits of pseudocode, but also has the added benefit of being machine interpretable.

3.2.1 Using the Algorithm Environment to tutor NP-completeness reductions

Complexity Tutor was originally designed to help students understand NP-completeness, a topic that computer science students frequently struggle with.

There are two common ways that students struggle with this topic. First, they may have conceptual misunderstandings about the topic. For instance, the author's study for this dissertation found a number of students who appeared to not understand that "NP" and "NP-complete" are different concepts—the Theorem Proving Environment should help correct these misunderstanding while students are constructing their proofs. Second, even if students correctly understand the concepts, they may struggle to produce a correct *NP-completeness reduction*, which is a type of algorithm. The Algorithm Environment is designed to help students with this latter problem.

The Algorithm Environment integrates directly with the Theorem Proving Environment. To illustrate, consider the problem presented to students in Figure 3.8, which defines two formal languages, PARTITION and BIN-PACKING. The problem asks students to prove that BIN-PACKING is NP-Complete, given that BIN-PACKING is in NP and that PARTITION is NP-Complete.

Figure 3.8 shows a partially constructed proof in the Theorem Proving Environment for this problem. Two assertions in the proof remain unjustified:

Figure 3.8: Screenshot of partially completed NP-completeness proof in the Theorem Proving Environment. The problem asks students to prove that BIN-PACKING is NP-complete. Assertions "8" and "9" remain unjustified.

- Assertion 8: "c is a certificate that set $X \in$ PARTITION $\iff$ `Cert_Partition_to_BinPacking(`c`)` is a certificate that `Reduce_Partition_to_BinPacking(`X`)` $\in$ BIN-PACKING"

- Assertion 9: "`Reduce_Partition_to_BinPacking()` runs in polynomial time"

Neither of these assertions can be justified by the remaining assumptions or assertions given in the Theorem Proving Environment. The assertions refer to two Python functions, `Reduce_Partition_to_BinPacking` and `Cert_Partition_to_BinPacking`, which students will need to write code for. The two assertions together can be thought of as specifying requirements for the code that the student

will write. Once the student has finished writing code that meets these requirements, both assertions will be considered justified.

The first assertion specifies that the student must construct a correct reduction from PARTITION to BIN-PACKING. `Reduce_Partition_to_BinPacking` must convert instances of PARTITION to instances of BIN-PACKING. `Cert_Partition_to_BinPacking` must convert certificates of PARTITION to certificates of BIN-PACKING. When a student has written correct code for both of these functions, they have constructed what is known as a *Levin reduction* (see Section 3.2.2).

The second assertion specifies that `Reduce_Partition_to_BinPacking` must have polynomial running-time.

To write code for `Reduce_Partition_to_BinPacking` and `Cert_Partition_to_BinPacking`, students will select the "Algorithms" tab, which reveals the Algorithm Environment (Figure 3.9). Here, students will write Python code using a simple Python editor that provides syntax highlighting, line numbering, automatic indentation, and parenthesis matching. Initially, students are given function stubs for `Reduce_Partition_to_BinPacking` and `Cert_Partition_to_BinPacking`, along with comments that explain the input and output conditions for these functions.

There is also a status indicator in Figure 3.9, which shows a question mark to indicate that the code in the editor has not yet been verified. After students write some code in the editor, they can click the "Check Algorithms" button to have the Algorithm Environment analyze their code.

While the code is being analyzed, students will not be able to modify it in the editor. The Algorithm Environment will first check to make sure that the code is valid Python code, just as a normal Python interpreter would. It will then perform additional checks on the code, to determine if the code represents a correct reduction

Assume PARTITION is NP-Complete and BIN-PACKING is in NP. Prove that BIN-PACKING is NP-Complete.

PARTITION
INSTANCE: A set X of positive integers.
PROBLEM: Is there a subset $S \subseteq X$ such that: $\sum_{x \in S} x = \sum_{x \in X/S} x$ (i.e. X be partitioned into two sets with equal sums)

BIN-PACKING
INSTANCE: A set of reals Y from the domain [0, 1) and a positive integer K.
PROBLEM: Is there a partition of Y into $Y_1,\ldots,Y_K$ such that for every Y_i: $\sum_{y \in Y_i} y \leq 1$

Check Proof

Proof | Algorithms

```
1   ## HINT: If you have an integer value x, use float(x) to convert
2   ##       it to a float.
3
4   # Input: X - list of positive integers, e.g. X=[1,2,3]
5   # Output: (Y,K) - Y is a list of floats, K is positive integer
6   def Reduce_Partition_to_BinPacking(X):
7
8       return (Y,K)
9
10  # Input: Certificate c = (part1,part2) where part1 and part2 are
11  #        two lists of positive integers with equal sum
12  #        e.g. c = ([15,9,7],[14,3,6,8])
13  # Output: Certificate c2 is a list of lists (aka bins) of floats
14  #         where each list sums to no more than 1
15  #         e.g. c2 = [[0.2,0.7],[0.1,0.4,0.2,0.3],...,[0.9,0.004]]
16  def Cert_Partition_to_BinPacking(c):
17
18      return c2
19
```

Check Algorithms

Errors:

STATUS:

Figure 3.9: Screenshot of the Algorithms Environment. Students are to write Python code for `Reduce_Partition_to_BinPacking` and `Cert_Partition_to_BinPacking`, to construct a reduction from PARTITION to BIN-PACKING.

from PARTITION to BIN-PACKING. Refer to Section 3.2.3 for technical details on how this works.

If the Algorithms Environment determines that the code does not represent a correct reduction, then the status indicator changes to display a “no symbol” and feedback is given to the student under “Errors”, as shown in Figure 3.10. In this particular example, the feedback tells the student that their reduction cannot be correct because their code for `Reduce_Partition_to_BinPacking` does not meet the specified output conditions. The student is also given a specific input for PARTITION where the reduction fails. This serves as a hint to the student about how they should proceed to debug their algorithm.

Assume PARTITION is NP-Complete and BIN-PACKING is in NP. Prove that BIN-PACKING is NP-Complete.

PARTITION
INSTANCE: A set X of positive integers.
PROBLEM: Is there a subset $S \subseteq X$ such that:
$\sum_{x \in S} x = \sum_{x \in X/S} x$ (i.e. X be partitioned into two sets with equal sums)

BIN-PACKING
INSTANCE: A set of reals Y from the domain [0, 1) and a positive integer K.
PROBLEM: Is there a partition of Y into $Y_1,\ldots,Y_K$ such that for every Y_i:
$\sum_{y \in Y_i} y \le 1$

Check Proof

Proof | Algorithms

```
1  ## HINT: If you have an integer value x, use float(x) to convert
2  ##       it to a float.
3
4  # Input: X - list of positive integers, e.g. X=[1,2,3]
5  # Output: (Y,K) - Y is a list of floats, K is positive integer
6  def Reduce_Partition_to_BinPacking(X):
7      K = 2
8      Y = []
9      for x in X:
10         Y.append(float(x)*0.5)
11     return (Y,K)
12
13 # Input: Certificate c = (part1,part2) where part1 and part2 are
14 #        two lists of positive integers with equal sum
15 #        e.g. c = ([15,9,7],[14,3,6,8])
16 # Output: Certificate c2 is a list of lists (aka bins) of floats
17 #         where each list sums to no more than 1
18 #         e.g. c2 = [[0.2,0.7],[0.1,0.4,0.2,0.3],...,[0.9,0.004]]
19 def Cert_Partition_to_BinPacking(c):
20     c2 = c
21     return c2
22
```

Check Algorithms

Errors:
Output of Reduce_Partition_to_BinPacking is not in the right format. The first element of your output should be a list of floats, each between 0 and 1. When given input of [2, 2, 3, 5, 6] the output of your Reduce_Partition_to_BinPacking function was: ([1.0, 1.0, 1.5, 2.5, 3.0], 2). And 1.5 is not between 0 and 1.

STATUS:

Figure 3.10: An incorrect reduction. The student is given feedback about `Reduce_Partition_to_BinPacking` producing incorrect output.

In general, the feedback given to students range from notifications of syntax errors to explanations of why their reductions are incorrect. After receiving the feedback, students can attempt to fix their code in the editor and click the "Check Algorithms" button again. Then, if the Algorithms Environment determines that the code now represents a correct reduction from PARTITION to BIN-PACKING, the status indicator changes to display a check mark, as shown in Figure 3.11. Finally, when students click the "Proof" tab to return to the Theorem Proving Environment, they will see that the two previously unjustified assertions are now justified, and the proof is complete. This is shown in Figure 3.12.

Assume PARTITION is NP-Complete and BIN-PACKING is in NP. Prove that BIN-PACKING is NP-Complete.

PARTITION
INSTANCE: A set X of positive integers.
PROBLEM: Is there a subset $S \subseteq X$ such that: $\sum_{x \in S} x = \sum_{x \in X/S} x$ (i.e. X be partitioned into two sets with equal sums)

BIN-PACKING
INSTANCE: A set of reals Y from the domain [0, 1) and a positive integer K.
PROBLEM: Is there a partition of Y into $Y_1, \ldots, Y_K$ such that for every Y_i: $\sum_{y \in Y_i} y \leq 1$

Proof Complete

Proof | Algorithms

```
3
4   # Input: X - list of positive integers, e.g. X=[1,2,3]
5   # Output: (Y,K) - Y is a list of floats, K is positive integer
6   def Reduce_Partition_to_BinPacking(X):
7       K = 2
8       Y = []
9       for x in X:
10          Y.append(float(2*x)/sum(X))
11      return (Y,K)
12
13  # Input: Certificate c = (part1,part2) where part1 and part2 are
14  #        two lists of positive integers with equal sum
15  #        e.g. c = ([15,9,7],[14,3,6,8])
16  # Output: Certificate c2 is a list of lists (aka bins) of floats
17  #         where each list sums to no more than 1
18  #         e.g. c2 = [[0.2,0.7],[0.1,0.4,0.2,0.3],...,[0.9,0.004]]
19  def Cert_Partition_to_BinPacking(c):
20      bin1 = []
21      bin2 = []
22      for x in c[0]:
23          bin1.append(float(x)/sum(c[0]))
24      for x in c[1]:
25          bin2.append(float(x)/sum(c[1]))
26      c2 = [bin1,bin2]
27      return c2
```

Check Algorithms

Errors:

STATUS: ✓

Figure 3.11: A correct reduction from PARTITION to BIN-PACKING.

3.2.2 Using Levin reductions rather than Karp or Cook reductions

The notion of NP-completeness was introduced in Stephen Cook's 1971 seminal paper [42], and then expanded on a year later by Richard Karp, who showed that 21 classic combinatorial problems were NP-complete [84]. Independently of Cook and Karp, Leonid Levin also discovered the concept of NP-completeness around the same time, while working in the Soviet Union [96].

Interestingly, these three progenitors used different types of reductions in their original formulations of the concept of NP-completeness. As such, complexity theorists have named the three types of reductions after them.

Given X and Y as languages in NP, consider the following definitions.

Figure 3.12: Screenshot of completed NP-completeness proof in the Theorem Proving Environment. Assertions "8" and "9" are now justified.

Cook reduction

X is *Cook-reducible* in polynomial time to Y if there is a polynomial time algorithm that can decide X using an oracle for deciding Y.

Karp reduction

X is *Karp-reducible* in polynomial time to Y if there is a polynomial time algorithm that computes a function f such that x is in X if and only if $f(x)$ is in Y.

Levin reduction

X is *Levin-reducible* in polynomial time to Y if:

1. there is a polynomial time algorithm that computes a function f such that x is in X if and only if $f(x)$ is in Y.
2. there is a polynomial time algorithm that computes a function g such that c is a *certificate* that x is in X if and only if $g(c)$ is a *certificate* that $f(x)$ is in Y.[1]

Notice that if X is *Levin-reducible* to Y, then X is *Karp-reducible* to Y, which in turn implies that X is *Cook-reducible* to Y. Hence, Cook reductions are the most general notion of a reduction, while Levin reductions are the most restrictive.

However, Karp reductions have the most prevalent usage in complexity theory literature. They are preferred over Cook reductions for their closure properties—the popular complexity theory separation conjecture, NP $\neq$ co-NP, does not make sense under Cook reductions.

Karp reductions are also most commonly taught to students studying the subject of NP-completeness for the first time, usually without reference to the other kinds of reductions. Two popular undergraduate textbooks that teach NP-completeness [44, 146] make no reference to the alternative reduction types.

In comparison, Levin reductions seem to be far less popularized than either Karp reductions or Cook reductions. Even a comprehensive graduate-level textbook on computational complexity [15] makes only two scant references to Levin reductions.

So why use Levin reductions rather than Karp reductions? Sanjeev Arora wrote a paper [14], which discusses Levin reductions in some detail. He states the following proposition:

---

[1] *Certificates* are formally defined in terms of *verifier* algorithms. Thus, the definition of *Levin-reducible* implicitly assumes that fixed *verifiers* have been chosen for X and Y. However, for most NP-complete languages, intuitively obvious *certificates* exist that can be described without explicit reference to a corresponding *verifier*

> All known NP-completeness reductions are either Levin reductions or easily modified to be Levin reductions.

This proposition is backed by the fact if you look through the literature of NP-Completeness results, the proofs of correctness for the Karp reductions usually implicitly contain a certificate reduction as justification. In other words, they are Levin reductions in disguise!

In the paper, Arora mentions that Alexander Razborov conjectures that this is because all of these reductions are provable in Samuel Buss's proof theory, S_2^1. If a Karp reduction is provable in S_2^1 then there is a witnessing theorem that proves the existence of a corresponding Levin reduction. Arora furthermore mentions that one way to break a Karp reduction away from being Levin is to use a cryptographic one-way permutation function to break the Levinness, but he knows of "no useful reduction that is not Levin".

So that should be a strong motivation for why it is acceptable to teach Levin reductions in place of Karp reductions (which are not even as general as the less-restricted Cook reductions), besides the fact that the student would be expected to be able to justify the correctness of their Karp reduction anyway.

In a handwritten assignment, the student would write that justification about the correctness of the reduction in the body of their proof. On the other hand, Complexity Tutor takes the view that the certificate part of a Levin reduction encapsulates the bulk of that justification and therefore can substitute for it.

Pedagogically, the author of this dissertation also thinks it is a good idea to distinguish language-to-language Karp reductions from certificate-to-certificate reductions as explicitly as possible, since he has noticed a common confusion among students when they are writing reductions is that they confuse instances of a language with certificates for those instances. Thus, teaching Levin reductions may help in this regard.

Furthermore, the structure of Levin reductions lends an obvious way to scaffold the difficulty of problems given to students. Instead of expecting students to give both the language reduction and the certificate reduction at the same time, we can break this up into separate problems. Sometimes, the student could be given a certificate reduction and asked to find a matching language reduction. Other times a student could be given the language reduction and ask them to provide the corresponding certificate reduction. In either case, the student is provided with a strong hint to the other reduction, which doesn't exist with Karp reductions.

3.2.2.1 Technical motivation for using Levin reductions

A significant reason for choosing to have students write code for Levin reductions in the Algorithm Environment stems from the requirement that the reductions need to be automatically verified. There is no known tractable way to automatically verify the correctness of Karp reductions or Cook reductions, even heuristically.

Consider the idea of using a set of test cases to heuristically verify the correctness of an algorithm. This is common practice in both industry and academia. And this is easy if you can uniquely determine the output that the algorithm should produce for any arbitrary input. For instance, if you are trying to verify the correctness of a sorting algorithm, then given $(8, 5, 2, 4, 2)$ as input, you know the output should always be $(2, 2, 4, 5, 8)$. For any given input to a sorting algorithm, there is only one possible correct output.

However, this is not a true for Karp reductions. For any given input to a Karp reduction algorithm, there are an infinite number of valid outputs that could be produced. Suppose you have a potential Karp reduction from NP-complete language X to NP-complete language Y. If you give an instance of X as input, then valid outputs could be any instance of Y. But to decide if the output belongs to Y is

intractable, since Y is NP-complete. Therefore, there's no easy way to verify if a Karp reduction is correct for even a single test case.

Verifying Cook reductions is no easier, since Cook reductions assume the availability of an efficient oracle for an NP-complete problem, and no such oracle can be implemented. Hence, it is not clear even how to computationally run a Cook reduction algorithm.

Now, consider a Levin reduction from an NP-complete language X to an NP-complete language Y. As specified by the definition, the Levin reduction is supposed to give a function f to transform instances of X to instances of Y, and a function g to transform certificates for X to certificates for Y. Since Y is in NP, there must also be a polynomial time verifier function v where $v(y, c) = 1$ if and only if c is a certificate that $y \in Y$.

Consider a test case (x_1, c_1), where $x_1 \in X$ and c_1 is a certificate that $x_1 \in X$. If $v(f(x_1), g(c_1)) = 1$, then $f(x_1) \in Y$ and $g(c_1)$ is a certificate that $f(x_1) \in Y$. Hence, according to the conditions for Levin reductions, both $f(x_1)$ and $g(c_1)$ give correct outputs.

On the other hand, consider a test case (x_2, c_2), where $x_2 \notin X$ and c_2 is not a certificate that $x_2 \in X$. If $v(f(x_2), g(c_2)) \neq 1$, then $g(c_2)$ is not a certificate that $f(x_2) \in Y$. In this case, $g(c_2)$ would give correct output according to the second condition for Levin reductions. The output for $f(x_2)$ may or may not be correct.

Nevertheless, it is clear that for any test case (x, c), computing $v(f(x), g(c))$ determines whether f and g satisfy the second condition for Levin reductions. Also, $v(f(x), g(c))$ can be computed efficiently, assuming that both f and g have efficient algorithms. It seems unlikely to produce an f and g that would satisfy the second condition for Levin reductions, but not the first condition, so this should be an effective way to verify Levin reductions.

3.2.3 How the Algorithm Environment works under the hood

The Algorithm Environment uses hidden test cases to automatically evaluate whether the functionality of a student's algorithm is correct. This method of evaluation is often referred to as *black-box testing*, since the algorithm is treated as a "black box" and is evaluated based on its behavior rather than its internal structure.

Similar automated evaluation methods have long been used in non-theoretical computer science courses to grade programming assignments, with one of the earliest examples of an automated grader being documented in 1960 [74]. The reader may refer to [49] for an overview of the history of automated evaluation of programming assignments.

Programming competitions, such as the ACM International Collegiate Programming Contest, also use black-box testing to automatically provide feedback to contestants [57], as do programming challenge websites like TopCoder and HackerRank. In particular, these competitions and the challenge websites use black-box testing to evaluate one's ability to design and implement correct algorithms. This is a very similar intent to that of the Algorithm Environment's usage of black-box testing, which also seeks to evaluate the ability to design correct algorithms.

A contribution of this dissertation is the development of a novel way to use black-box testing to automatically evaluate the correctness of NP-completeness reductions.

Suppose a student is tasked with showing that NP-complete language X reduces to NP-complete language Y in polynomial time. In the Algorithm Environment, the student must implement two Python functions that represent a Levin reduction. The first, `Reduce_X_to_Y`, is required to convert instances of X to instances of Y. The second, `Cert_X_to_Y`, is required to convert certificates of X to certificates of Y. Both functions need to run in polynomial time.

To automatically evaluate the correctness of the student's code, a problem specific evaluation module is needed, which contains Python implementations of two polynomial time verifier functions, `Verify_X` and `Verify_Y`, where:

- `Verify_X(`x`,`c`)` is `true` $\iff$ c is a certificate that verifies that $x \in X$.
- `Verify_Y(`y`,`c`)` is `true` $\iff$ c is a certificate that verifies that $y \in Y$.

The problem specific evaluation module also contains a set of test cases. Each test case is of form (x, c) and is classified as being either positive or negative. If the test case is positive then $x \in X$ and c is a certificate that $x \in X$. If the test case is negative then $x \notin X$ and c is an arbitrary valid input for `Cert_X_to_Y`.

The Algorithm Environment will use the following process to evaluate the student's code. This process terminates as soon as the code is determined to be incorrect or if all test cases have been exhausted:

1. A Python interpreter is instantiated in a new system process, along with the student's Python code and the problem specific evaluation module. Running the Python interpreter in a separate process from COMPLEXITY TUTOR gives reasonable assurance that the student's code will not be able to crash COMPLEXITY TUTOR when it is run in the interpreter.

2. The Python interpreter checks the student's code to make sure it is syntactically valid. If it is not syntactically valid, then the code is marked as incorrect and feedback of the syntax error is returned to the student along with the line number where the syntax error occurred.

3. For each test case (x, c) in the problem specific evaluation module, the Python expression `Verify_Y(Reduce_X_to_Y(`x`),Cert_X_to_Y(`c`)) == Verify_X(`x`,`c`)` is evaluated. If this expression evaluates to `true` then the test case passes, as justified by Section 3.2.2.1. If this expression evaluates to `false` then

the code is marked as incorrect, and the student is given feedback notifying them that the reduction is not correct. Notice that the evaluation of this expression may also trigger runtime exceptions, if `Reduce_X_to_Y` or `Cert_X_to_Y` are not properly defined in the student's code or if the student's code contains bugs. For instance, a common bug that triggers an exception is if the student references a variable before assigning a value to it. As soon as an exception is triggered, the code gets marked as incorrect and feedback of the exception along with the line number of the code that triggered it is returned to the student.

4. If all test cases have passed, then the student's code is marked as correct.

Finally, a timer is placed on the whole evaluation process, and if the timer runs out, then the code is automatically marked as incorrect. This is necessary for three reasons.

First, the student may have accidentally introduced an infinite loop into their code, which will prevent the evaluation process from ever terminating. Since the *halting problem* is *undecidable* [146], there is no way to pre-emptively identify this scenario—the best that can be done is to give the evaluation process a reasonable time limit to finish and give up if this reasonable time limit is exceeded.

Second, the student will only have so much patience to wait for their code to be evaluated. If the evaluation takes too long, then COMPLEXITY TUTOR is not fulfilling its promise to give the student "immediate feedback". Furthermore, while there may be some pedagogical value to giving students feedback on the correctness of slow algorithms, it is unlikely that many students would deliberately implement an inefficient algorithm. More than likely, the inefficiency of the algorithm was unintentional, often the result of a bug. In that case, the student would want to be notified as soon as possible that their algorithm is inefficient.

Third, the timer provides a way to assess if `Reduce_X_to_Y` and `Cert_X_to_Y` are likely to both run in polynomial time, one of the conditions required for the NP-

completeness reduction to be correct. Some of the test cases are designed to generate inputs to `Reduce_X_to_Y` and `Cert_X_to_Y` that are large enough in size that if the functions are implemented with non-polynomial time algorithms, then the timer should run out before the functions have terminated.

This main downside of this approach to testing if an algorithm runs in polynomial time is that there can easily be false negatives, i.e., inefficient polynomial time algorithms that trip up the timer. However, this should not be much of a problem for evaluating NP-completeness reductions, because usually the most obvious correct reductions are very efficient polynomial time algorithms, often running in linear time. So it seems unlikely that students would naturally stumble upon inefficient polynomial time reductions before considering the efficient ones.

3.2.3.1 Limitations of the black-box testing framework and alternatives

Black-box testing is not a foolproof method of automatically evaluating the correctness of algorithms. In general, it is possible to engineer incorrect algorithms that with high probability will pass all the hidden test cases.

A simple example is mentioned in [57]. Consider the task of determining if two strings are identical. An incorrect algorithm that still works most of the time would be to hash both strings and compare the hash values. For any two random strings, with very high probability, if they have the same hash value then the strings will be identical.

Thus, black-box testing may not be suitable for algorithmic problems where it is easier to come up with an algorithm that gives the correct answer with very high probability than an algorithm that always gives the correct answer.

However, the widespread use of black-box testing in programming competitions gives some faith that it is still a suitable evaluation method for a large number of undergraduate-level algorithms problems. After all, if there was a competitive advan-

tage to trying to engineer algorithms that trick the evaluator, then this tactic would be common in programming competitions.

Furthermore, in a pedagogical setting where a student is trying to learn rather than win a competition, it is even less likely that a student would accidentally engineer an algorithm that tricks the evaluator.

The bigger issue with the practicality of using black-box testing as a tool for tutoring theoretical computer science topics is figuring out how to assess the running time of an algorithm. The crude method of efficiency determination that the Algorithm Environment currently uses to evaluate NP-completeness reductions, may be sufficient when trying to distinguish between a very efficient polynomial time algorithm and a non-polynomial time algorithm, but it would not be suitable for an algorithms course where finer-grained asymptotic running time analysis is desired.

The average-case running time of an algorithm could be empirically estimated by sampling the execution time for many different sizes of random input to get a set of points that can be fit to a curve.

However, this still does not give the worst-case running time, which is usually the main consideration in theoretical computer science. An area of research that might hold some promise is *automated complexity analysis*, which has developed analytical tools for determining worst-case asymptotic bounds on the running time of certain classes of algorithms. See [4, 25, 36, 55, 61, 99, 145] for examples of this research.

3.3 Authoring new problems

COMPLEXITY TUTOR was designed so that domain experts could easily create new problems programmatically by writing simple Python scripts to interact with the underlying engine. The following example code demonstrates how to create a new problem for the well-known syllogism, *"All men are mortal. Socrates is a man. Therefore, Socrates is mortal."*

```
from CCTutorEngine import *
ProblemDescription.Text = "Prove Socrates is mortal."
ax1 = ProofItem("All men are mortal.")
ax1.isAssumption = True
ax1.show()
ax2 = ProofItem("Socrates is a man.")
ax2.isAssumption = True
ax2.show()
goal = ProofItem("Socrates is mortal.")
goal.Requirements.Add(ax1)
goal.Requirements.Add(ax2)
goal.show()
```

The first line tells Python to import the classes used by COMPLEXITY TUTOR.

In the second line, `ProblemDescription` refers to an internal object representing a description of the problem to be presented to the student. A problem author can either supply a simple text description directly, using the `Text` property (as illustrated above), or they can use a word processor to create a **Rich Text Format (RTF)** file and supply that as the description with the `LoadFile` method of `ProblemDescription`. The latter permits the author to have more precise control over formatting of the problem description, to include variations in font, text color and highlighting, and to include embedded figures.

Assumptions and assertions are both internally represented in the Theorem Proving Environment as `ProofItem` objects. In the example above, `ax1` and `ax2` are `ProofItem` variables defined to represent the assumptions "Socrates is a man" and

"Socrates is mortal". The `isAssumption` property is set to `True` for both to indicate that they are assumptions rather than assertions.

The `goal` variable, which is also a `ProofItem` object, always represents the unique assertion that a student must prove to complete the problem. In this case, it is the assertion "Socrates is mortal".

Every assertion `ProofItem` object has a `Requirements` object, which is a list of other `ProofItem` objects (assumptions and assertions) that the student should use to justify the assertion. In the example, the `Add` method has been used to add the assumptions corresponding to `ax1` and `ax2` to the `Requirements` object of the `goal`.

The `show` method tells the Theorem Proving Environment to display the assumption or assertion corresponding to a `ProofItem` object immediately when the student starts working on the problem. By default, assumptions and assertions are not immediately revealed. As explained in Section 3.1.3, assertions can be unlocked as the student progresses. Every `ProofItem` object has an `OpensUp` object to implement this functionality. An `OpensUp` object is a list of `ProofItem` objects, representing assumptions and assertions that will be unlocked by a given assertion.

If the line `goal.show()` were replaced by `ax1.OpensUp.Add(goal)` in the above example, then "Socrates is mortal" would not be revealed in the Assertions Box until the student had moved "All men are mortal" to the Proof Space.

`ProofItem` objects also have an `isWrong` property, which when set to `True` indicates an erroneous assertion. For instance, the author might add the erroneous assertion, "Socrates is dead", which cannot be inferred from any of the given assumptions.

Constructing problems that make use of the Algorithms Environment is also done with Python. See the Appendix for examples.

When a problem is ready to be released to students, the author converts the Python source code to Complexity Tutor's encrypted file format, **Complexity Tutor Problem (CTP)**, which prevents students from using the source code to reverse engineer a solution to the problem.

3.4 Developing a complete intelligent tutoring system

The author's vision for Complexity Tutor is that it will one day become an intelligent tutoring system that adapts to individual student needs when learning theoretical computer science topics. The work of this dissertation represents progress towards that vision. Chapter 6 will give a roadmap for future work.

The following describes the protocol of interaction between Complexity Tutor and the student user when Complexity Tutor is used as an intelligent tutoring system:

1. A problem is selected for the user. This could either be a problem from a fixed set chosen by the instructor (similar to a traditional homework assignment) or it could be a problem that is adaptively chosen based upon the likelihood that the student will be able to solve it, and whether it will increase their proficiency in some area. The latter involves using machine learning to build an elaborate *student model* (Section 6.6).

2. If the problem requires the student to construct a proof, the Theorem Proving Environment is loaded (Section 3.1). If the problem also requires the student to specify an algorithm as part of their proof, then the Algorithm Environment is loaded (Section 3.2).

3. The individual problem can be customized in several different ways, to permit scaffolding and to help a student with a specific task. First, the assumptions given in the Assumptions Box (Section 3.1.2) can be varied to alter the subgoals

that need to be completed. For instance, if a student is asked to prove that a language is NP-Complete, one version of the problem might allow them to assume the language is in NP, if it is desired for them to focus on the subgoal of doing the reduction rather than the subgoal of showing that the language is verifiable. Second, the number of assertions given in the Assertions Box (Section 3.1.3) can be adjusted. The more possible assertions that are given, the more flexibility students have in things they can attempt in their proof. However, increasing the number of assertions also makes the problem more challenging, since it increases the proof search space. Third, the student can be initially given a partial proof in the Proof Space, showing them the general outline of the proof or what proof schema (Section 1.2.4) to use. Similarly, partial code can be given to the student in the Algorithm Environment. Finally, the granularity (Section 2.1.4) of inferences that the student is allowed to make can be adjusted. All of these parameters for problem customization can be either manually tweaked by the instructor, or automatically tailored to a particular student by learning a student model (Section 6.6).

4. The student completes their proof in the Theorem Proving Environment, while in the background, the tutor tracks errors that are made and updates the student model accordingly (Section 6.6). A proof step (inference) is prohibited if it is logically unsound or deemed too coarse. The proof is verified according to the problem's hypergraph (Section 3.1.6) with any required algorithmic reductions being verified with test cases (Section 3.2.3).

5. When the student has developed a complete proof without errors, they may move on to the next problem. If they are unable to finish a correct proof, they are also allowed to pass on to a new problem, but the student model stores the

fact that they could not solve this problem and gives it to them again later after they have had more practice (Section 6.6).

CHAPTER 4

DESIGN OF EXPERIMENTS

A multi-semester study was conducted to evaluate the efficacy of Complexity Tutor. This study was designed to adhere to the strict guidelines of the Institutional Review Board at the University of Massachusetts Amherst. Participation in the study was completely voluntary, and extensive effort was taken to protect the privacy of those who participated.

Complexity Tutor was evaluated in two different undergraduate computer science classes. During the Fall 2016 and Spring 2018 semesters, Complexity Tutor was evaluated in an algorithms class, for tutoring students in the topic of NP-completeness. During the Spring 2017 and Fall 2017 semesters, Complexity Tutor was evaluated in a discrete math class, for tutoring students in first-order prepositional logic proofs.

Table 4.1 shows a break-down of how many students volunteered each semester to participate in the study, how many of those volunteers actually completed the experiments of the study, and how many were ultimately included for research analysis.

4.1 General study procedures

The same general protocol was followed each semester, which is detailed here.

4.1.1 Soliciting volunteers

A brief announcement, informing students of the study, would be made at the beginning of a class lecture. During the announcement, each student in the lecture

Table 4.1: Participation in study.

	NP-Completeness Experiments				Prepositional Logic Experiments				Total
	Fall 2016		Spring 2018		Spring 2017		Fall 2017		
	Experimental	Control	Experimental	Control	Experimental	Control	Experimental	Control	
Initial volunteers for study	28	27	34	33	57	58	42	42	321[a]
Volunteers who completed experiments	20	19	24	18	38	39	20	23	201[a]
Drop-out rate	29%	30%	29%	45%	33%	33%	52%	45%	37%
Subjects evaluated	25[b]	22[b]	24	18	32[c]	36[c]	20	22[c]	199[a]

[a] It is possible that some of the same students may have participated during multiple semesters. Since the sets of participants for each semester are not known to be disjoint, the totals may not accurately reflect the exact number of people who participated.
[b] In Fall 2016, some subjects were permitted to join the study after experiments had already begun, and are not counted among the initial volunteers.
[c] Some subjects were excluded from analysis due to missing or corrupted data.

would also be provided with two copies of an informed consent form. The announcement and consent form provided general details about the study and information about the rights of participants and non-participants in the study. The students were instructed to read the consent form and to keep one copy of the form for reference. If they decided they wanted to participate in the study, they were to return the second copy of the consent form by a specified deadline, signed and providing their name and email address.

A script for the announcement and the text of the consent form are provided in Appendix A. To reduce the chance of biasing potential volunteers, students were not told what kind of problems would be given in the experiments or even the exact nature of the software that would be used. For instance, the consent form merely stated that the study's purpose was to evaluate "a novel computerized self-tutoring system that will provide immediate feedback for students to rectify learning problems, and will assist students to learn theoretical topics at their personal pace of learning." Additionally, the form suggested that a potential benefit of the study was that participants might gain more exposure and practice with learning material relevant to their course, but it did not say what that specific material would be.

Students would also be informed that if they participated, then their instructor would give them up to 5 extra credit points that would go toward their final exam grade. In all semesters except Fall 2016, the policy for rewarding extra credit was that any student who participated would receive a minimum of 3 extra credit points, and the remaining 2 points would be awarded based on individual performance in the study. In Fall 2016, the first semester of the study, every student who participated received 5 extra credit points regardless of their performance in the study. This led to some subjects that semester not putting much effort into their participation, so the policy was changed for subsequent semesters.

The instructor would also offer an alternative extra credit assignment for students who did not want to participate in the study, so that no student would feel compelled to participate just to earn the extra credit.

Once the deadline for returning the consent forms had passed, the subjects who had volunteered at that point were evenly split into two groups. One group was the Experimental group, which used Complexity Tutor to solve a set of practice problems related to a particular topic taught in the course. The other group was a control group, which was given the same practice problems to solve but without the assistance of Complexity Tutor. The control group was asked to write or type their solutions to the problems similar to how they would do for a normal homework assignment.

Subjects could drop out of the study at any time, for any reason, and did not need to inform anyone to do so. As such, despite subjects initially being evenly split between experimental and control groups, the two groups would generally end up being slightly unequal in size since not everyone who was initially chosen for a given group would end up completing the experiments. Nevertheless, this didn't seem to affect the distribution of the two groups very much.

4.1.2 Logistics of the experiments

For logistical convenience, the experiments were designed so that subjects who participated could do so completely remotely. On the start date of an experiment, each subject would receive an email with detailed instructions to follow to complete the experiment. The experimental and control groups received different emails. In the emails, subjects were given deadlines to complete specific tasks. Examples of these emails are found in Appendix A. Subjects could also communicate with the principal investigator by email if any questions came up during the study.

Extreme care was taken to ensure that all data associated with subjects in the study was anonymized, in order to protect privacy. This was accomplished by giving every subject a random identifier, referred to as a Participant ID. Subjects were instructed to go to a website that would automatically generate a random Participant ID for them. They were instructed that it was very important to write this Participant ID down, since it would be the only way to keep track of them during the study, or for them to receive extra credit. The study was designed so that not even the principal investigator would be able to link the research data with the names of actual students who participated.

For instance, when subjects submitted data from the experiments, they did so by anonymously uploading it to a server with only their Participant ID attached. At the end of the semester, a list of Participant IDs and the extra credit each earned would be sent to the course instructor. Subjects who wanted to receive this extra credit would then need to reveal their Participant ID to the instructor only. Thus, the instructor would potentially know who in their class had participated in the study and how much extra credit they had earned in doing so but would have no direct access to the research data. However, if a subject decided for some reason that they did not want to claim the extra credit, for instance if they decided to drop out after submitting data, then their data would be completely anonymous to everyone.

In some semesters, data was collected on how subjects performed on actual class exams. This data was provided by the instructors, who could link the exam data to the subjects' Participant IDs after they had claimed the extra credit.

A small number of subjects were confused by the anonymization process. Some lost their Participant ID despite being instructed multiple times to write it down and maintain it. One subject even had the false impression that they could go to the original website that generated the Participant ID to retrieve it. Fortunately, if a

subject had forgotten their Participant ID but not yet submitted any data, this could easily be rectified by the subject acquiring a new Participant ID.

More problematic would be if a subject accidentally uploaded their data without a Participant ID. To discourage this scenario, subjects were requested to put their Participant ID in two places. First, subjects were instructed to put their Participant ID in the file names of data they submitted. Then when they actually uploaded the files, the upload form again asked them to supply their Participant ID in the upload form itself. The instructions in the email also emphasized many times the importance of the Participant ID. Nevertheless, there still ended up being some data that was uploaded without an associated Participant ID. In most cases, the subject later realized that they had forgotten to include their Participant ID in the upload, and sent a second upload that included the Participant ID. It was possible to determine when this happened by comparing file checksums of submissions.

4.1.3 Procedures for the experimental group

Subjects in the experimental group used their own personal computers to run Complexity Tutor. They were first instructed to verify that their computer would meet the minimum requirements needed to run Complexity Tutor—they needed the Windows operating system and a screen resolution of at least 1024×768.

Complexity Tutor would run on any version of Windows that supports the .NET Framework. This is most versions of Windows, and the oldest version of Windows that was tested and confirmed to work with the software is Windows 2000. However, for versions of Windows older than Windows 8, subjects would need to download and install a newer version of the .NET Framework, which they could freely obtain from Microsoft. In most cases, for modern versions of Windows, Complexity Tutor runs out of the box without any special installation procedure.

Subjects who did not have Windows on their computer could either borrow a Windows laptop from the university, or they could obtain the Windows operating system freely to run on their own computer via Microsoft Imagine, a program that provides free copies of Microsoft software to computer science students. Some subjects may not have bothered to jump through these extra hoops and decided to drop out of the study instead. However, there is no indication that participation in the study was significantly affected by subjects not having a Windows computer readily available.

One subject, upon learning that the study required Windows, asked if they could be switched to the control group but this was forbidden, since it would have potentially skewed the data for the control group. Anyone in the study who decided they wanted to stop participating because of the system requirements had the option of requesting the alternative extra credit assignment instead.

The screen resolution was required to be a minimum of 1024×768, because the Complexity Tutor interface would not fit on a lower resolution screen. According to W3Schools, a website that tracks screen resolutions of its visitors, only 1% of visitors in January 2014 had screen resolutions less than 1024×768 and 93% of visitors had a screen with greater resolution than that [165]. Thus, 1024×768 is a very conservative resolution to target. The W3Schools website shows a slight increase in the number of visitors with low screen resolutions in recent years, but this is undoubtedly attributed to visitors who are using mobile devices. Many subjects in the study actually complained about the limited amount of screen space used by Complexity Tutor and requested the ability to increase the size of the interface. While such a feature would be easy to implement, it was purposely excluded, to prevent subjects with large monitors and high screen resolutions from having an unfair advantage over everyone else in the study.

Subjects were asked to watch online tutorials, showing them how to solve sample problems in Complexity Tutor. After watching the tutorials, they could practice

with the same problems shown in the tutorials in Complexity Tutor, if they wanted to.

Next, subjects were given a set of problems to attempt to solve using Complexity Tutor. While the software was running on the subjects' computers, the subjects' interactions with the interface were automatically recorded in video files. Subjects did not have to complete the problems in a single sitting, and often made multiple attempts at the same problem, which were recorded as separate video files every time Complexity Tutor was restarted.

When subjects had completed the problems or given up, they were instructed to construct a ZIP file containing all the video files that had been collected. This ZIP file was renamed to include the subject's Participant ID and then uploaded anonymously to a secure file storage facility provided by Box. The webpage that subjects used to upload their files was password protected with a password only available to those who participated in the study.

After uploading their data, subjects were requested to fill out an online questionnaire to give qualitative feedback about their experience using Complexity Tutor.

4.1.4 Procedures for the control group

The control group was given the same problems to work on as the experimental group, but without the aid of Complexity Tutor. Subjects in the control group were instructed to treat the problem set as they would a normal homework assignment. Subjects were also asked to note approximately how much time they spent on each problem.

They were then to create a PDF from their solutions, either by scanning their hand written copy or typing it up. The PDF would be uploaded as their data for the experiment, using the same upload procedures used for the experimental group.

Subjects were advised not to write their name or any other identifiable information, besides their Participant ID on the submissions.

The control group's submissions were manually graded—scores and feedback were assigned for every problem, as would be done if it were a normal homework assignment. Dispostable, an anonymous email drop service, was used to deliver the scores and feedback. Subjects could login to the Dispostable email account using just their Participant ID and were advised to check it every few days after the submission deadline until they had received the feedback, since Dispostable automatically deletes messages over time. The subjects could also send reply email messages from the Dispostable account, to ask questions about the feedback. Thus, the process of how students normally receive feedback on their homework assignments was simulated as closely as possible, while maintaining the anonymity of the study participants.

4.1.5 Evaluation of submissions and follow-up

In all semesters other than Fall 2016, it was necessary to grade the submissions to assign extra credit for participation in the study. Each problem given to students in the experiment was worth the same amount, regardless of its perceived difficulty. Since there was only a maximum of 5 extra credit points allowed, and subjects received 3 points just for completing the experiment, each successfully completed problem was worth at most $2/k$ points, where k is the number of problems given.

The control group received the grades that had previously been given to them in feedback. The experimental group was graded based on what was observed in the videos, with variable scores assigned for incomplete proofs based on how many connections were missing in the Proof Environment.

In some semesters, subjects were also asked to complete a follow-up quiz at the end of the experiment. It was hoped that these quizzes could serve as a posttest for the experiment.

4.2 NP-completeness tutoring experiments

Experiments using Complexity Tutor to help students learn the topic of NP-completeness took place in an algorithms course during the Fall 2016 and Spring 2018 semesters. During the Fall 2016 semester, there were two instructors who co-taught the course. Only one of the instructors from the Fall 2016 semester was involved in teaching the course during the Spring 2018 semester. Nevertheless, it is presumed that that presentation of material was similar during both semesters, especially since the same textbook [87] was used both semesters.

In both semesters, NP-completeness was not covered by the instructor until near the very end of the course, and the experiment was started after the instructor had given lectures on the topic. Unfortunately, this meant that students participating in the study had to do so while under the pressure of other important responsibilities, such as final exams and end-of-semester projects.

The method of splitting study volunteers into the experimental and control groups differed between the two semesters. During the Fall 2016 semester, one of the instructors split the volunteers into matched groups based on prior performance in the course. During the Spring 2018 semester, volunteers were assigned to experimental or control at random with equal probability. The reason for the change in procedure was because it was realized that NP-completeness was a very different topic than earlier topics in the course, and so prior performance in the course may not be so relevant. Furthermore, there would likely be confounding variables that would more directly affect a student's ease at learning NP-completeness, such as what they had learned in other classes. As such, purely random assignment was the best way to deal with confounding variables. It was also logistically simpler.

As noted earlier in this chapter, the policy for rewarding extra credit for participation also differed between the two semesters. During the Fall 2016 semester, students would receive 5 points of extra credit on their final exam, so long as they were deemed

to have fully completed the experiment. During the Spring 2018 semester, students who completed the experiment received 3-5 points of extra credit on their final exam, based on their performance in the study. The change in incentive structure may have affected the results of the experiments, as subjects during the Spring 2018 semester may have put more effort into their participation. This is a variable to consider in analysis.

Subjects were given three problems to work on during the Fall 2016 semester. The first problem was a purely conceptual problem that could be solved entirely in the Theorem Proving Environment (Section 3.1) using definitions from the theory of NP-completeness. The other two problems required the student to complete an NP-Completeness reduction using the Algorithm Environment (Section 3.2), in addition to laying out the structure of a general NP-completeness proof in the Theorem Proving Environment.

During the Spring 2018, the same problems as Fall 2016 were given, but three additional conceptual problems were added. Thus, during Spring 2018, there were four conceptual problems that could be solved in the Theorem Proving Environment without requiring a reduction in the Algorithm Environment. Analysis of results from Fall 2016 had indicated that there was a huge difference in performance in favor of the experimental group over the control group on the one purely conceptual problem given that semester, and so it was desired to get more data on that phenomenon. Refer to Appendix B to see the problems given.

Every subject during both semesters was asked to watch a 20-minute "Crash Course in Python" training video. The purpose of this was two-fold. First, the video was designed to provide subjects in the experimental group with the basic knowledge of Python needed to use the Algorithm Environment to do the reduction problem. Second, the video was used as a litmus test to judge student familiarity with Python, since that could be a variable affecting performance in the study. Subjects were

given a multiple-choice survey question to respond to after watching the video, which assessed how familiar they were with the concepts taught in the video.

An important difference between the experimental and control groups was that the experimental group was required to do Levin reductions for the two reduction problems, whereas the control group was neither required nor expected to do Levin reductions. Unfortunately, Levin reductions were never taught to students in the class, and even worse the students' textbook muddled the distinction between Cook and Karp reductions. Since the Algorithm Environment requires Levin reductions for technical reasons, subjects in the experimental Group were asked to watch a short tutorial video, which introduced the concept of a Levin reduction and showed how to do one in the Algorithm Environment. Of the two reduction problems, the 0/1-PROG Reduction Problem gave subjects in the experimental group the certificate part of a Levin-reduction as a hint. The other reduction problem, the BIN-PACKING Reduction Problem required students in the experimental group to produce both parts of a Levin reduction.

More specific details of the two semesters are listed below.

4.2.1 Fall 2016

The initial announcement of the study was made to the class on November 3 and the experiment officially began a month later, on December 6. Initially, only 55 students volunteered, of which 28 were put in the experimental group and 27 were put in the control group. From these initial groups, 20 of the volunteers placed in the experimental group completed the experiment, and 19 volunteers placed in the control group completed the experiment. However, some additional students were allowed to join the study late, after the initial deadline to volunteer had long passed and the experiment had already begun. There were 6 students who joined the study after December 12, of which 3 were placed into the experimental group and 3 were

placed in the control group based on the normal matching criteria. These 6 subjects received 'L' suffixes on their Participant ID to indicate that they had joined the study late. There were an additional 3 students who joined the study even later, and these 3 subjects were all placed in the experimental group and were given 'W' suffixes on their Participant ID to indicate that they both joined the study late and that they had been added without respect to the normal matching criteria. In total, there ended up being 25 subjects in the experimental group and 22 subjects in the control group.

The reason for permitting students to join the study late was because it was feared that the initially low turn-out would not give enough data to draw statistically valid conclusions. The reason that some students wanted to join the study late is unknown. One possibility is that they were unaware of the study until other classmates told them about it—maybe they had been absent on the day that the initial announcement was made. Another possibility is that their perspective on the value of participating in the study changed by the time the experiment began.

Due to the fact that subjects joined the study at different times, and also due to the fact that the experiment was started near the end of the semester, subjects were not given a concrete deadline of when they were expected to complete the experiment. It was hoped that this flexibility would encourage more to complete the experiment, but in retrospect it probably also encouraged subjects to procrastinate. Subjects who completed the experiment by December 28 were also still eligible for the extra credit bounty, even though this was after the semester had officially ended. The final exam for the class took place on December 19. In the experimental group, only 8 subjects completed their participation before the day of the final exam. In the control group, only 9 subjects completed their participation before the day of the final exam.

Thus, while it was initially hoped that parts of the final exam could be used as a posttest for subjects participating in the study, not enough subjects completed the experiment before taking the exam to draw statistically valid conclusions. Exam

scores for one question related to NP-completeness were initially looked at, but that question was deemed to not be particularly relevant to the study—40% of the points on that question were testing a student's conceptual understanding of vertex covers and dominating sets, 40% of the points were assigned to proving a specific reduction correct, and only 20% of the points involved completing an NP-completeness proof. In other words, 80% of the points from that question were assigned for completing tasks that the experiment would not have likely prepared subjects for.

4.2.2 Spring 2018

A newer version of Complexity Tutor with the additional features mentioned in Section 3.1.7 was used for this semester. The BIN-PACKING Reduction Problem was also modified so that subjects in the experimental group would receive scaffolded hints rather than the very limited feedback given during the Fall 2016 semester. This is explained in Section 5.2.2. The 0/1-PROG Reduction Problem was not modified. Also, three new conceptual problems were given, as mentioned above.

There was also a slight deviation from the normal protocol for the control group this semester. The control group did not receive feedback on their submissions like in the experiments from the other semesters.

The initial announcement of the study was made to the class on February 26 and the experiment officially began almost two months later, on April 19. There were 67 volunteers, of which 34 were put in the experimental group and 33 were put in the control group. From these initial groups, 24 of the volunteers placed in the experimental group completed the experiment, and 18 volunteers placed in the control group completed the experiment. There was also one additional volunteer from the experimental group not counted because the data they submitted was for problems

from the prepositional logic experiment instead of the NP-completeness experiment.[1] Notice that there was a slightly higher drop-out rate from the control group than the experimental group during this semester. Unlike Fall 2016, students were not allowed to join the study late after the experiment had commenced.

During this semester, the experiment was divided into three distinct phases. Directions for the first phase were emailed on April 19. This phase consisted of preliminary preparations for the experiment—subjects acquiring their Participant ID and watching the tutorial videos (if they were in the experimental group). Directions for the second phase were emailed a few days later on April 23. The second phase was where the subjects were actually given the problems to work on. Subjects were given a deadline of May 1 to complete this second phase and submit their data. The third phase involved subjects taking a short follow-up quiz. Directions for this third phase were emailed on May 2. The directions also explicitly stated that the quiz should not be looked at until the second phase had been completed, since some subjects completed the second phase past the May 1 deadline.

The reason for explicitly breaking the experiment into these three phases was because it was hoped that it would encourage subjects to pace themselves through the experiment, and to carefully read and follow the directions. There had been numerous problems with subjects not correctly following directions in previous semesters, when the directions for the whole experiment had been sent in a single email. There was no equivalent of the third phase in the Fall 2016 experiment. In that semester, it had been hoped that questions from the final exam could be used as a posttest for the experiment, but there were no suitable questions on that final exam. For the Spring

[1] It is not known whether this was accidental or intentional. The prepositional logic experiments were done during alternate semesters from the NP-completeness experiments. It is therefore likely that this particular subject had been enrolled in the discrete math class in a previous semester and volunteered for the study then as well.

2018 semester, the choice was made to give subjects a quiz that would hopefully serve as a better posttest for measuring what students had learned during the experiment.

The posttest quiz is found in Appendix E. Designing it was a challenge. On one hand, the quiz needed to properly assess what subjects had learned about NP-completeness, based on what they could have possibly learned from the six problems given in the experiment. On the other hand, the quiz needed to be designed so that it could be completed quickly, since it was deemed that subjects would not have much time available. So while ideally, subjects would have been given an hour long test, the quiz was designed to be completed in 10-30 minutes. Note also that subjects were given the quiz on May 2, which was two days before their final exam in the algorithms class on May 4.

There were five problems on the quiz. Four of the problems did not require subjects to construct a proof but rather to evaluate the correctness of a potential proof. It is not known how well this type of problem correlates with the proof construction problems given during the experiment, but it was chosen because reading a potential proof and looking for its flaws should take considerably less time than constructing a proof from scratch. The fifth problem was a proof construction problem similar to the ones given for practice in the experiment. In fact, this fifth problem was nearly identical to Conceptual Problem 1 (Appendix B), requiring the same argument structure.

One constraint of the quiz was that for logistical reasons, it could not be proctored. Subjects were responsible for administering the quiz themselves at their own convenience. As such, it was explicitly stated on the quiz that while it had to be submitted to receive extra credit for participation in the study, the quiz would have no impact on how much extra credit was received or for any other part of their grade in the course. This was to remove any incentive that subjects might have of cheating. But the downside of not giving a direct incentive for doing well on the quiz is that subjects might put little effort into it. This was another reason that the quiz was

designed so that it could be completed quickly, since subjects would be more tempted to not spend the full amount of time on a long quiz, knowing it would not impact their grade at all.

A potential indirect incentive that subjects might have perceived for putting effort into the quiz is that it could help them prepare for their final exam. However, the majority of subjects submitted their quiz after they had already taken their final exam. Only 18 subjects submitted the quiz prior to the day of the final exam, of which 11 were from the experimental group and 7 were from the control group. Of the remaining subjects, 18 submitted the quiz the day after the final exam, on May 5. A reminder email was sent on May 7 to remind the remaining subjects to submit their quiz, and almost everyone submitted their quiz after that reminder. There was only one subject, who was from the experiment group, who never submitted the quiz.

The fact that almost everyone submitted the quiz stands in stark contrast to the experiment that was done in the Fall 2017 discrete mathematics class. In that other experiment, it was also attempted to use a quiz as a posttest but 82% of subjects did not even submit it. What might explain the difference in outcome? Subjects in Spring 2018 were continuously reminded that there were three phases of the experiment that needed to be completed, and so they were likely anticipating this third phase of the experiment. In comparison, the Fall 2017 experiment was not explicitly broken up into phases. Subjects in Fall 2017 were told in the beginning that they would be asked to complete a follow-up quiz, but they were not given as many reminders.

Timing may be another factor. For the Spring 2018 experiment, the quiz was delivered to subjects the day after the deadline for submitting their data for the practice problems. However, Fall 2017 subjects had to wait two weeks from the initial deadline for submitting data before they were given the quiz. The main reason for this delay was grading and giving feedback to the control group subjects. Since feedback was not given to control subjects in Spring 2018, this delay was not necessary

and the third phase of the experiment could commence as soon as subjects completed the second phase.

4.3 Prepositional logic proof tutoring experiments

Experiments using Complexity Tutor to help students learn prepositional logic proofs took place in a discrete math course during the Spring 2017 and Fall 2017 semesters. During both semesters, the same instructor taught the course, so there was presumably consistency in how the course material was presented.

In both semesters, prepositional logic proofs were taught to students fairly early in the course, and students were evaluated on their knowledge of this topic during the first midterm exam, before the experiment started. Volunteers were split into two matched groups, experimental and control, based on performance on relevant problems from this first midterm exam.

Notice that the timeframe of intervention differs significantly between the NP-completeness experiments and the prepositional logic experiments. In the case of NP-completeness, subjects had barely been exposed to a new concept and had little time to practice working on problems involving the concept or get feedback on their understanding of the concept. However, in the case of prepositional logic, subjects had a significant amount of time to practice homework problems, seek clarifications from their instructor, prepare for an exam that assessed their understanding of the topic, and seek additional clarifications after the exam—all before the experiment began.

During both semesters, subjects were given the same three problems to work on in the experiment. The Pizza Problem, deemed to be the easiest, was a simple logic problem where all assumptions and assertions were given to subjects in plain English. The Muddy Dog Problem was also given in plain English, but was deemed to be harder than the Pizza Problem, since there were significantly more proof steps involved.

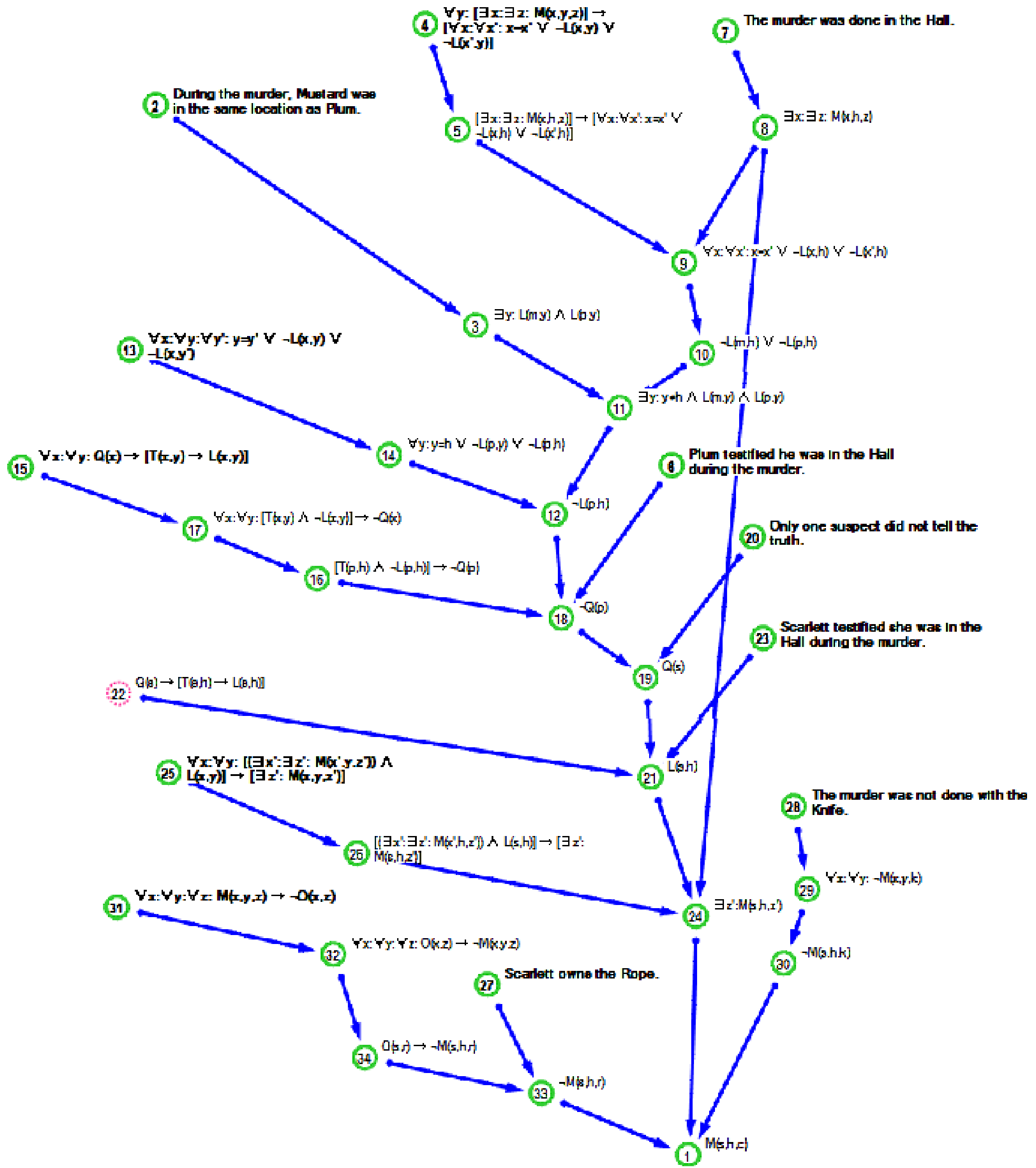


Figure 4.1: Proof graph of Murder Mystery Problem in Complexity Tutor.

The Murder Mystery Problem was deemed to be the hardest, and used a mixture of symbolic logic and English to describe the problem's assumptions, requiring subjects to translate between them. It also had a fairly involved proof with the most proof

steps and a large number of assumptions to deal with. Figure 4.1 shows the solution to this problem in Complexity Tutor. All three problems are given in Appendix C. An important point to make is that the author was careful to design all three problems to require the same level of granularity (Section 2.1.4) for solutions in Complexity Tutor, so it is fair to compare difficulty based on the number of proof steps required.

Exam data was also collected from the instructor for the subjects in the study. Scores on specific questions on the midterm and final exam were analyzed. These specific questions can be found in Appendix D. The intent was to use the relevant questions from the midterm exams as pretests, and the relevant questions from the final exams as posttests. Performance improvement between the midterm and final was also analyzed.

More specific details of the two semesters are listed below.

4.3.1 Spring 2017

In this semester, two separate announcements were made informing the class about the study. The first announcement was made on March 1. But at that time, the informed consent forms were not ready for distribution since the Institutional Review Board was still in the process of approving changes to the protocol to permit extra credit to be based on performance in the study. Thus, a second announcement was made on April 3, which is when the consent forms were given to students.

After the announcements, there were 115 students who volunteered for the study, more than in any other semester. It is possible that the additional announcement led to more students being aware of the study. It would have also given students more time to think about whether they wanted to participate. The 115 volunteers were split into matched groups of experimental and control, based on a composite of their scores on Questions 1–3 from their first midterm exam.

The experiment commenced on April 21 and subjects were given a deadline of May 3 to complete it. For each subject who completed the experiment, their scores on two midterm exam questions and two parts of a final exam question were collected for analysis. These exam questions are in Appendix D.

For the midterm, the scores of Question 1 and Question 3 were collected. Question 1 asked students to translate several symbolic logic assertions to and from English. Question 3 asked students to construct a proof using the statements from Question 1. It is presumed that the score for Question 3 would be more relevant to the study than Question 1, since Question 3 unlike Question 1 involves proof construction. However, since Question 3 depends on Question 1, and since Question 1 might still be somewhat useful, both scores were collected. The scores for Question 2 were also obtained but not used in analysis, since that problem permitted students to give a truth table as an answer rather than a normal proof.

For the final, Question 2 was looked at, which essentially combined two types of questions from the midterm. The first part of Question 2 was a translation problem like Question 1 from the midterm. The second part of Question 2 was a proof construction problem similar to Question 3 from the midterm. Thankfully, it was possible to obtain the individual subscores for Question 2, to separate the translation problem from the proof construction problem.

Of the initial volunteers, 38 from the experimental group and 39 from the control group completed the experiment. However, two subjects from the experimental group were excluded from analysis in the study, because the data they submitted was thought to be corrupted. An additional four subjects from the experimental group and three from the control group were also excluded because their exam scores could not be obtained—these subjects may not have claimed extra credit for their participation, because the instructor did not have a record of them. Thus, the final total of

subjects who were analyzed in the study was 32 in the experimental group and 36 in the control group.

4.3.2 Fall 2017

A newer version of Complexity Tutor with the additional features mentioned in Section 3.1.7 was used for this semester. One of the primary motivations for the Fall 2017 experiment was to evaluate if these new features were useful to students, and to look at how students used them. The problem files given to subjects in the experimental group were also slightly modified from the versions used in Spring 2017.

The versions of the Muddy Dog Problem and Murder Mystery Problem used in Spring 2017 did not show all the available assertions initially. Instead, many assertions were triggered to be unlocked when certain progress was made in the Proof Space. One reason for this was to not overwhelm students by showing them too many assertions at once. However, with the new search feature, it was hypothesized that this would help students find the assertions they were looking for even if there were many in the Assertions Box. It was also hypothesized that the unlocking of assertions might be confusing to some students.

The versions of the Muddy Dog Problem and Murder Mystery Problem used in Fall 2017 displayed all available assertions initially without requiring any to be unlocked. The Pizza Problem was not modified, because this problem never required any assertions to be unlocked, since it does not even use many assertions.

The initial announcement for the study was made in mid-September, although the exact date was not recorded. There were only 84 students who volunteered for the study, significantly less than the previous semester. It is unknown why there were so many fewer volunteers than Spring 2017. Perhaps the early announcement had an adverse effect, or perhaps it was simply the multiple announcements in Spring 2017 that caused that semester to have a higher than usual turnout of volunteers. The 84

volunteers were split into matched groups of experimental and control, based on their score for Question 2 from the first midterm exam.

Originally, students were told that the experiment would begin on October 20, but the start date had to be delayed. On October 30, subjects who had volunteered to participate were notified by email that the experiment would begin on November 9, after their second midterm, so they did not need to worry about studying for the second midterm and doing the experiment at the same time.

The experiment was officially started on November 9, and subjects were given an initial deadline of November 30 to complete it. Notice that this was more time than given in the Spring 2017 semester, but the reason for giving a more generous deadline was that it was hoped it would decrease the drop-out rate, since there was a smaller pool of volunteers to begin with. Unfortunately, giving more time ended up having the opposite effect, as not many submissions were received by the initial November 30 deadline. On December 1, subjects were surveyed to ask if they would like an extension for completing the experiment. Many responded that they would like the extension, so an extension was granted until December 9. The fact that so many subjects needed an extension, despite being initially given a more generous deadline than the Spring 2017 subjects were given, is evidence that giving more generous deadlines actually increases the chances of procrastination.

For each subject who completed the experiment, their scores on Question 2 from the midterm exam and Question 1 from the final exam were collected for analysis. These exam questions are in Appendix D. Question 2 on the midterm asked for three proof constructions, although students were permitted to substitute truth tables for proofs when only propositional logic was required. Question 1 on the final had two parts, a logic translation part and a proof construction part, similar to the Spring 2017 semester.

Unfortunately, unlike in Spring 2017, it was not possible to obtain the individual subscores for the proof construction part separated from the logic translation part.

Of the initial volunteers, 20 from the experimental group and 23 from the control group completed the experiment. There was one anonymous experimental group submission that is not counted because the submission did not have a Participant ID. One of the subjects from the control group was also later excluded from analysis because their exam scores could not be obtained. Thus, the final total of subjects who were analyzed in the study was 20 in the experimental group and 22 in the control group. Note that two subjects in the experiment group made no attempt on Murder Mystery Problem, so technically they did not complete the study correctly since the directions specifically said that subjects should attempt to solve all problems. However, these two subjects are still included in the analysis.

There was also an additional quiz that was given to subjects this semester as a follow-up to the experiment. Initial analysis from the Spring 2017 semester had indicated that performance on the exam questions analyzed did not correlate with performance on the problems subjects worked on for the experiment, so it was speculated that the exam problems may not even be a suitable posttest for the study. Thus, a separate quiz that might be a better posttest for the study was developed. See Appendix E. Due to logistical constraints, the quiz could not be proctored. Subjects were responsible for administering the quiz themselves at their own convenience, and were told that the quiz should take less than 30 minutes to complete, although they could spend as much time as they want. They were also instructed to complete the quiz in a single sitting.

Unfortunately, very few quiz submissions were received. A total of 4 quizzes were received from the experimental group, and a total of 4 quizzes were received from the control group—a grand total of 8 submissions. Of the 8 submissions, everyone essentially got the single question on the quiz correct and scored 95% or higher, so

there is almost nothing that can be inferred from the quiz submissions. It is not known why there were so few submissions. The quiz was sent to subjects on December 16, two weeks after the initial deadline for completing the experiment, and less than a week after the extended deadline. The main reason for the brief delay was the time it took to give feedback to the control group on their submissions from the experiment. A reminder email about the quiz was sent on December 26, but unfortunately that did not significantly help.

4.4 Limitations of the study

There were a number of limitations in how this study was designed that compromised some of the research objectives, but were necessary to meet the requirements of the Institutional Review Board and the instructors who graciously allowed the study to take place in their classes. One of the biggest limitations is that there was no control over how instructors presented material. For instance, the instructors who taught NP-completeness did not specifically cover anything about Levin reductions. Performance on that part of the study would presumably look different if students had received more exposure to Levin reductions.

The fact that subjects had to specifically opt-in to participating in the study rather than opt-out meant that large percentages of students either did not volunteer for the study or later dropped out, which in some semesters led to getting less data than was desirable. There would also likely be some bias in who chose to participate and stay in the study. As mentioned in Chapter 5, it would be nice to also have a third comparison group that did not do any practice problems but took the same pretests and posttests, but that would require a higher participation rate to afford splitting subjects into three groups.

The fact that the experiments were framed as being optional extra credit activities for external research rather than an integral part of the classroom learning also likely

affected the results of this study. It cannot be guaranteed that subjects treated the practice problems in the same way they would treat a bona fide class assignment, even though they were instructed to do so. When there was a conflict between what they were to be learning from participating in the experiments and what they thought they already understood from official classroom activities such as instructors' lectures, presumably they might discount the new information learned in the experiment, being less trusting of its relevance to the class, since it came from an outside source.

The pretest and posttest conditions were not ideal either. The problem with using actual exam questions from the discrete math course as pretests and posttests for the prepositional logic experiments is that those exams also assessed material that was not part of the study. Even though only scores from the questions relevant to prepositional logic proving were used, the other questions could still have affected performance on those questions.

Consider that not all students will do the questions on the exam in the same order, and on a timed exam that makes a big difference. One test taking strategy is to do the easiest questions first, and another strategy is to do the hardest questions first. A student who feels more confident in their abilities may be more inclined to choose the latter strategy, but sometimes that confidence is overestimated. The author can recall numerous occasions where he ran out of time to finish an easy question because he spent too much time on a hard question.

In the case of the posttest quiz given to subjects in the NP-completeness experiments, that was not an issue. However, that quiz was not proctored, because there was no logistically feasible way to ensure that all subjects doing the experiment would be available at the same time for a quiz proctored outside of class. Since the quiz was not proctored, there was the additional measure of telling subjects that the quiz would not affect their grade, to discourage cheating. However, this likely also led to subjects not putting their best effort into the quiz.

Time was another limiting factor in the study. The protocol approved for the study capped the expected amount of time that participants would be required to spend on research activities to 15 hours. Realistically, even expecting subjects to spend that much time was barely reasonable since they were only receiving a small amount of extra credit as a reward for their participation.

None of these concerns should take away from the importance of this study taking place. When a new pedagogical method is introduced in an educational field where it has never been tested before, there will understandably be some caution to testing it on actual students, which will limit the scope in which it can be evaluated in initial studies.

Research progress must therefore be gradual. As will be shown in Chapter 5, not all research questions are answered conclusively based on the results. Some findings are speculative and some results are inconclusive, which is to be expected, given the parameters of the study. However, it is the hope of the author that the evidence presented will still be enough to support future studies that evaluate the efficacy of Complexity Tutor more in-depth without the limitations of this study.

In the future, the author would like to evaluate Complexity Tutor when it is used as an integral part of classroom learning, as was done with many of the systems described in Chapter 2.

CHAPTER 5

EXPERIMENTAL RESULTS AND ANALYSIS

This chapter presents what was learned from the study described in Chapter 4. The results are divided into three sections.

In Section 5.1, the Theorem Proving Environment is evaluated, looking at how subjects in the experimental groups performed on problems that only involved using the Theorem Proving Environment. The results are significantly different for the topics of NP-completeness and prepositional logic, and it is conjectured that this is explained by subjects' prior conceptual understanding of the material. Some findings from this section were originally presented in a SIGCSE conference publication [103].

Next, Section 5.2 looks at how subjects performed in the experiments where they used the Algorithm Environment to do NP-completeness reductions. Initial results for the Fall 2016 semester were disappointing, since no subjects from that semester were able to solve either of the reduction problems (Appendix B) using the Algorithm Environment. However, there was remarkable improvement in Spring 2018, which is likely due in part to extra scaffolding of hints for one problem.

Finally, Section 5.3 presents the feedback that subjects in the study gave about Complexity Tutor, which was largely positive—a strong majority of them would recommend the system to peers who are also learning proof construction.

5.1 Evaluation of the Theorem Proving Environment

The video recordings submitted by subjects in the experimental groups were analyzed to determine if they encountered difficulties in using the Theorem Proving

Environment, and if they were able to solve the proof problems with the tutoring system. No significant difficulties in figuring out how to use the Theorem Proving Environment were observed in the videos. However, a number of subjects mentioned on their questionnaire that they would have liked more screen space for their proof.

Summary statistics are presented in Tables 5.2 and 5.1, comparing performance of the experimental groups and control groups for the prepositional logic problems and for the NP-completeness problems. As can be seen from these tables, for most of the problems given in the study solely involving the Theorem Proving Environment, subjects in the experimental groups were able to successfully construct proofs using the system.

5.1.1 Theories to explain differences between the results for the logic and NP-completeness experiments

Notice the stark contrast between Tables 5.1 and 5.2. For the logic problems given to subjects in the discrete math classes, the statistics are very similar for both the experimental and control groups. However, for the conceptual problems on the theory of NP-completeness given to the subjects in the algorithms classes, the experimental group performed significantly better than the control group.

What explains why the statistics are so similar for the experimental and control groups in the prepositional logic experiments, and yet so far apart for the NP-completeness experiments? To even attempt to answer that question, a theory is needed to ascertain what will predict a student's ability to produce a correct proof for a given proof problem. The author suggests that there are two variables that come into play in the normal setting that the control group experienced (i.e., without the intervention of Complexity Tutor):

1. Conceptual misunderstandings of the student may lead them to make incorrect inferences, thus producing an incorrect proof. In the nomenclature used by

Table 5.1: Summary statistics for the prepositional logic problems. The *Made Attempt* row indicates the percentage of subjects who attempted each problem. For the experimental group, a subject was only considered to have attempted a problem if they made at least one connection in the Theorem Proving Environment. The *Mean* and *Median* rows indicate average scores over the entire sample, where subjects who did not attempt a problem were factored into the average with a 0% score. The *Perfect* row indicates the percentage of subjects who got a perfect score amongst those who attempted the problem.

	Pizza				**Muddy Dog**				**Murder Mystery**			
	Spring 2017		Fall 2017		Spring 2017		Fall 2017		Spring 2017		Fall 2017	
	Experimental	Control	Experimental	Control	Experimental	Control	Experimental	Control	Experimental	Control	Experimental	Control
Sample Size	32	36	20	22	32	36	20	22	32	36	20	22
Made Attempt	97%	100%	100%	100%	91%	97%	95%	100%	78%	81%	75%	95%
Mean	95%	96%	90%	90%	79%	82%	79%	75%	49%	46%	44%	47%
Median	100%	100%	100%	100%	95%	100%	93%	80%	50%	50%	25%	38%
Perfect	90%	92%	80%	77%	55%	60%	47%	36%	44%	31%	40%	19%

Table 5.2: Summary statistics for the conceptual NP-completeness problems. The *Made Attempt* row indicates the percentage of subjects who attempted each problem. For the experimental group, a subject was only considered to have attempted a problem if they made at least one connection in the Theorem Proving Environment. The *Mean* and *Median* rows indicate average scores over the entire sample, where subjects who did not attempt a problem were factored into the average with a 0% score. The *Perfect* row indicates the percentage of subjects who got a perfect score amongst those who attempted the problem.

	Conceptual Problem 1				**Conceptual Problem 2**		**Conceptual Problem 3**		**Conceptual Problem 4**	
	Fall 2016		Spring 2018		Spring 2018		Spring 2018		Spring 2018	
	Experimental	Control	Experimental	Control	Experimental	Control	Experimental	Control	Experimental	Control
Sample Size	25	22	24	18	24	18	24	18	24	18
Made Attempt	100%	100%	100%	100%	100%	100%	100%	100%	96%	94%
Mean	94%	59%	98%	47%	96%	38%	100%	34%	92%	42%
Median	100%	55%	100%	40%	100%	30%	100%	35%	100%	40%
Perfect	88%	27%	96%	11%	92%	0%	100%	0%	91%	6%

intelligent tutoring system researchers, these misconceptions that students may have are often referred to as "bugs" [169].

2. There is an innate complexity to the solutions of proof problems, which will be informally referred to as "proof complexity".[1] As the complexity of the proof that must be constructed to solve a given proof problem increases, the likelihood that a student of a given ability level will succeed decreases and the chances that they will make mistakes due to the burdens of cognitive load increases.

For the NP-completeness experiments, there is strong evidence that most students in the control group lacked the prerequisite conceptual understanding of the material to succeed. Thus, the first variable mentioned above was likely mainly responsible for predicting the performance of students in the control group. For instance, a very common misconception identified in subjects from the control group is that subjects would conflate the concepts of NP and NP-Complete. For Conceptual Problem 1, many subjects tried to argue that PATH was NP-Complete because it can be verified in polynomial time. This indicates that PATH is in NP but not that it is NP-Complete.

On the other hand, there were no similar patterns of conceptual misunderstandings that were identified among the subjects in the prepositional logic experiments. If subjects had significant conceptual misunderstandings, they most likely would not have done well even on the Pizza Problem, and almost everyone did well on that problem. Consider that subjects had significant opportunities to correct any misconceptions they might have had before the experiments even began, since the experiments were started after the subjects had already taken a midterm exam on the material covered.

[1] There is also a formal notion of *proof complexity* studied by theoreticians, such as used in this paper [26] on the proof complexity of resolution proofs. That is not what is being referred to here, although there may be some overlap between the informal pedagogical notions discussed here and the formal notion.

Thus, the author conjectures that the second "proof complexity" variable above was the main determining factor for how subjects statistically performed on the prepositional logic problems given in the experiments. Note that there are a number of different ways one might consider measuring this "proof complexity" variable. It could be measured simply by the total number of proof steps in the proof, or it could be measured by how far away the furthest assumption from the goal in the proof graph is, or it could be measured by the branching factor of the proof graph. It could even be measured in terms of the complexity of assertions used in the proof, which itself might be measured in terms of parse trees from the grammar rules expressing those assertions.

By any of these measures, you can produce an ordering on a set of proof problems, according to the complexity of the simplest proof that will solve the problem. The three logic problems from the study have the same ordering of complexity using first-order logic inferences under all the measures mentioned above—the proof required for the Pizza Problem is simplest in complexity, the proof required for the Muddy Dog Problem is more complex than the Pizza Problem, and the proof required for the Murder Mystery Problem is most complex of all.

As would be expected, the performance of subjects in the study on those problems decreases in that order as well, as shown in Table 5.1. So assuming that a "proof complexity" variable explains the performance of the control group subjects in the prepositional logic experiments, it is interesting that the experimental groups performed very similar to the control, despite the fact they were constructing their proofs in the Theorem Proving Environment rather than with pen and paper. This can be considered a feature rather than a flaw of the Theorem Proving Environment, if it were to mean that students solving the proof puzzles provided by the Theorem Proving Environment are receiving the same intellectual benefit as they would receive doing the proofs with pen and paper.

After all, one of the main reasons that students are assigned to practice constructing proofs in the first place is because it is believed that proper practice will improve one's ability to solve proof problems that require complex solutions, in the same way that physical exercise can improve one's physical strength and endurance. Otherwise, if there were no benefit to practice, why do it?

Even if practicing proof construction in the Theorem Proving Environment only provided the exact same benefits for students without misconceptions as practicing proofs on paper and having them graded by an instructor, the Theorem Proving Environment still has the advantage of not requiring all that grading work to be done. Students can receive more practice than they would otherwise, because the amount of practice they can do is not bandwidth limited by the amount of papers their instructor can grade.

Of course, the study would have needed to be designed differently to best evaluate this. There would need to be two comparison groups instead of just one control, so subjects would need to be split into three groups. One group would use the Theorem Proving Environment to do practice problems, another group would do the same problems with pen and paper, and the final group would not do the practice problems at all. Such an experimental design could be used to measure whether there are comparable benefits to practicing proof construction in the Theorem Proving Environment as there are to practicing the normal way. The study would also likely need to run longer to be able to measure the benefits of practice over time.

At this point, it is merely a conjecture that practice in the Theorem Proving Environment provides similar or better benefits to traditional proof practice when students have no significant conceptual difficulties. There are, however, already two pieces of evidence from this study that support the conjecture:

1. If two activities provide similar intellectual challenge, one would expect to see similar performance statistics on those activities, which is what is observed in

Table 5.1, with the descriptive statistics being similar for both the experimental and control groups on all three logic problems.

2. A positive association was found between subjects successfully constructing proofs in the Theorem Proving Environment and improving their exam scores in the Spring 2017 experiment. The details of the analysis determining this association are in Section 5.1.3. Interestingly, the association was absent from the control group, suggesting that using the Theorem Proving Environment to practice proof construction may in itself provide benefit for some students that traditional proof construction practice will not, even when the traditional proof construction practice is done correctly. However, more data is needed to establish statistical rigor for the association that was found in the experimental group, and also one must be hesitant in assuming that the association is a causal relationship, until other possibilities can be ruled out.

5.1.2 For the NP-completeness experiments, did the Theorem Proving Environment correct students' misconceptions?

At first sight, Table 5.2 would appear to be very good news for the Theorem Proving Environment. The experimental group performed tremendously better than the control group on all four conceptual problems related to NP-completeness. As explained in the previous section, it is believed that the subjects in the control group had significant conceptual misunderstandings leading them to make incorrect inferences and ultimately perform very poorly on these problems.

On the other hand, it is believed that subjects in the experimental group performed much better on the same problems, because the Theorem Proving Environment prevented the subjects from making incorrect inferences and because the proofs that subjects had to construct had low "complexity" (in the sense described in the

Table 5.3: Results of Spring 2018 posttest quiz, questions 1–4. The "Partially Correct" row indicates that a subject correctly identified some flaws in a proof, but did not identify all of the flaws and/or did not give good justifications for the flaws they did identify. Other rows are self-explanatory.

	Question 1		Question 2		Question 3		Question 4	
	Experimental	Control	Experimental	Control	Experimental	Control	Experimental	Control
Sample Size	23	18	23	18	23	18	23	18
Correct	4%	22%	87%	78%	17%	0%	9%	6%
Partially Correct	52%	61%	0%	0%	9%	6%	22%	6%
Incorrect	43%	17%	13%	22%	74%	94%	70%	89%

previous section). But does this mean that the experimental group also had better learning outcomes?

After the Fall 2016 experiment revealed that experimental subjects performed much better than control subjects on Conceptual Problem 1, it was hypothesized that by showing subjects how to make correct inferences in their proofs, the Theorem Proving Environment was correcting their conceptual misunderstandings.

Unfortunately, the posttest quiz given in Spring 2018 would appear to disprove this hypothesis, at least under the conditions of the study. The results of this posttest quiz are shown in Tables 5.3 and 5.4, and the questions that were given on the quiz are in Appendix E.

Before delving into theories of why the experimental group did not perform well on the posttest quiz, it is important to remind the reader that the quiz was administered under conditions that were less than ideal, and which might have caused the results to

Table 5.4: Results of Spring 2018 posttest quiz, question 5. The "Correct" row indicates that the subject's proof was reasonably well-written and had no observable flaws. The "Conceptually Incorrect" row indicates that the subject's proof demonstrated that the subject had a major conceptual misunderstanding. The "Undetermined/Other" column is all subjects who did not fit the other categories, including subject 94521523, who said they did not have time to finish the quiz.

	Question 5	
	Experimental	Control
Sample Size	23	18
Correct	22%	44%
Conceptually Incorrect	52%	50%
Undetermined/Other	26%	6%

be worse than they would be in an ideal setting. The quiz was given to subjects when they were preparing for final exams, which presumably distracted their focus from the quiz. To entice subjects to take the quiz, they were told that it could be completed in very little time. Unfortunately, this probably resulted in subjects rushing through the quiz, not giving it an appropriate amount of time, and not even carefully reading the quiz questions. This is evidenced by the fact that some subjects' responses to questions on the quiz completely ignore what was written in the question prompts.

Furthermore, subject 35917331, who was one of the only two subjects from the experimental group to get all the practice problems correct (including the reduction problems mentioned in Section 5.2.3), never even submitted the quiz. And subject 94521523, who was the other subject from the experimental group that got all practice problems correct, did not finish the quiz and wrote this for the last question:

> "The promised 30 minutes have elapsed, so I give up now because that's all the time I budgeted; I have a meeting soon (and am supposed to do this only in one sitting)"

That subject, unlike most others, did well on the first four questions, so there is a good chance they would have done well on the whole quiz if they had allotted time correctly.

The first four questions on the quiz asked the subjects to analyze four different proposed proofs and point out flaws if any existed, or to state that a proof was correct if there were no flaws. The proof contained in the second question was the only one that was correct. The reason why subjects did substantially better on this question than the others, was because they had a very noticeable bias towards marking the proofs as correct whether they were or not. All of the remaining proofs in first, third and fourth questions had major conceptual flaws such as the "bug" of considering NP and NP-Complete to be the same class.

Note that performance on these first four questions could particularly be adversely affected if a subject had been rushing through the quiz or was sleep deprived, since either of those conditions could prevent them from spotting flaws that they would otherwise recognize.

However, the results for the fifth question, shown in Table 5.4, give the strongest evidence that practice with the Theorem Proving Environment sadly failed to correct misconceptions about NP-completeness. This fifth question is nearly identical to Conceptual Question 1 in the experiment. The only difference is that the language PRIMES is substituted for PATH in the problem description, but a subject from the experimental group could literally take the proof they had constructed in the Theorem Proving Environment for Conceptual Problem 1, substitute the word PRIMES for PATH, and have a perfectly correct proof!

But obviously, all the subjects who got this quiz question incorrect did not do that. There are three possible reasons to consider:

1. Subjects might have managed to construct the proof for Conceptual Problem 1 in the Theorem Proving Environment, but had so little understanding of it that they would not be able to recognize when a very similar problem could be proven with the same argument.

2. Subjects understood how to construct the proof expected for Conceptual Problem 1 while they were working on it, but they later lost this knowledge because it was not reinforced in any way.

3. Regardless of whether the subjects understood and retained knowledge of the proof they constructed for Conceptual Problem 1, they never learned that other "buggy" proofs they had in mind for that problem were in fact incorrect.

It is unknown whether the first or second possibility is true, but the evidence from fifth quiz question suggests that the third possibility is definitely true. The best that can be hoped for is that only the third possibility is true.

What was the most common "bug" that subjects displayed in their attempted proofs of this fifth quiz question? The question asks subjects to assume that P=NP, a proposition that most knowledgeable computer scientists would consider to be absurd but that is still not known to be false with absolute certainty. From this absurd assumption, the subject is asked to prove something that would never be true without the absurd assumption, i.e., that PRIMES is NP-Complete. However, what many subjects attempted to do was to give a polynomial time reduction from a known NP-Complete problem to PRIMES that does not depend on P and NP being equal.

The "bug" then is that the subject assumes that whenever they are asked to prove a language NP-Complete, they must do so with a general reduction that works irrespective of the assumptions they are given. The reasons subjects had this "bug" is likely because the instructor never taught them an example where the correctness of a reduction hinged on a problem-specific assumption. Incidentally, Kurt VanLehn's *Repair Theory* [163] predicts this very outcome—that "bugs" are introduced when there is a bias in the examples students are taught. Furthermore, the subjects in the study lacked the conceptual understanding to realize that if PRIMES was in P, as stated in the problem, it would be hopeless for them to try to produce a general reduction from PRIMES to an NP-Complete problem.

Notably, not a single subject even complained about being asked to prove a polynomial-time language to be NP-Complete, so they even lacked a rule telling them that this is generally not a good idea to try.

Sadly, it seems evident that many students in this class did not understand why they were studying NP-Completeness in the first place.

Another observation to make is that all subjects in the experimental group wrote coarse-grained proofs for the fifth quiz question, despite the practice problems given to them in the Theorem Proving Environment training them to do fine-grained proofs. This also made assessment of the subjects' proofs more difficult, because in a number of cases, the subjects' proofs did not give enough information to portray what they understood.

To illustrate this dilemma, consider the proofs of subjects 33853006 and 78099281.

Here is subject 33853006's proof:

> "We know that for a problem to be NP-Complete, it needs to definitely be in NP. Therefore, NP-Complete is a subset of NP. Then, since we are given the assumption that P=NP, that means that NP-Complete is ALSO a subset of P. This means that all NP-Complete problems are solvable in polynomial time, so NP-Complete = P, and since PRIMES is in P, PRIMES is NP-Complete as well."

This proof is at a level of granularity where you can tell there is an incorrect inference, i.e., "This means that all NP-Complete problems are solvable in polynomial time, so NP-Complete = P". For the inference to be correct, they should have instead written, "This means that all NP-Complete problems are solvable in polynomial time, so NP-Complete $\subset$ P". They have actually not shown anywhere in their proof that P is contained in NP-Complete. Even though the proof is not correct for this reason, this mistake may not be conceptual in nature or a mistake they would make consistently, but rather a careless oversight on their part. The rest of the logic in the proof holds together. So this subject's proof was classified in Table 5.4 as "Other/Undetermined" because while not correct it does not demonstrate that the subject has a major conceptual misunderstanding.

On the other hand, subject 78099281 wrote for their proof:

> "If P=NP then every problem in NP can be solved in polynomial time, including those problems in NP-Complete (which are a subset of NP). Thus P=NP=NP-Complete. Since PRIMES is in P, by this equality it is also in NP and NP-Complete."

The line "Thus P=NP=NP-Complete" is such a coarse-grained proof step that one cannot determine if the subject would have made the same mistake as subject 33853006 if forced to explain that step at a finer granularity or not. Note that the inference is still incorrect, because the empty language ($\emptyset$) and its complement (Σ^*) belong to NP but would never be considered NP-Complete under Karp reductions.

In general, subjects were given the benefit of doubt if there was no evidence of an incorrect inference, and so coarser-grained proofs were often classified as "Correct". Keep in mind that if the quizzes were to be classified by the granularity expectations used for the problems given in the Theorem Proving Environment, they would all be considered incorrect.

So one should consider the percentage in the "Correct" row of Table 5.4 to be an upper-bound on the percentage of subjects who actually had a completely correct understanding of the proof, and the percentage in the "Conceptually Incorrect" row to be a lower-bound on the percentage of subjects who had conceptual misunderstandings.

Regardless, the results from the posttest quiz do not paint a flattering picture of the Theorem Proving Environment's current ability to correct misconceptions. That said, it is possible that if the subjects had more time to practice more problems in the Theorem Proving Environment, and if it had been more of an integral part of their learning experience, the outcomes might have been more positive. One must consider that the subjects may have given more weight to the misconceptions they had learned from their instructor's teaching than what COMPLEXITY TUTOR tried to teach them, because COMPLEXITY TUTOR was never officially part of the class.

5.1.3 Analysis of the relationship between performance on the practice problems and exam improvement in the logic experiments

In the prepositional logic experiments, scores from relevant exam questions were analyzed to determine what effect the experiment had on student exam performance. It was observed that some subjects who did very well on the relevant exam questions put little effort into the practice problems given in the experiments—most likely, they did not feel they really needed the extra credit provided by the study. Thus, only comparing scores from the final exam to how subjects performed on the practice problems would be misleading, because some subjects received very low scores for the practice problems simply because they did not give much effort to their participation in the experiment.

As such, it is more meaningful to look at the improvement of subjects between their midterm and final exams, and whether that is explained by the experiment or not. An exam improvement score was calculated for Spring 2017 as the difference between the scores of Question 2b from the final exam and Question 3 from the midterm exam of that semester. Both of these questions are relevant to the study since they required subjects to produce a similar kind of prepositional logic proof. Note that this improvement score can be positive or negative. It is positive if the subject's final score was higher than their midterm, and it is negative if the subject's final score was lower than their midterm.

The first observation to make, from Figure 5.1, is that overall, the experimental group did not have better exam improvement than the control group. However, the variance in improvement is greater for the control group than the experimental group. These observations are not very surprising, because there are many variables outside of the study that could influence exam improvement. One would not expect that performance on three practice problems alone would determine exam improvement, since there are many other ways that subjects may have tried to prepare for the

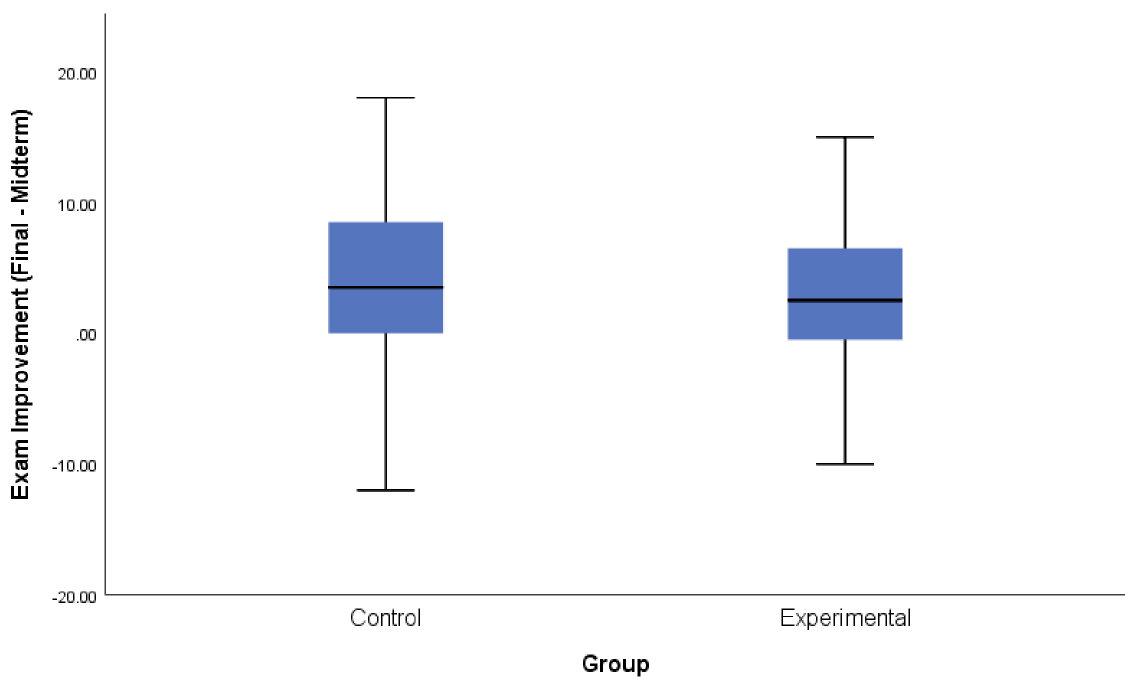


Figure 5.1: Boxplot of exam improvement by group from Spring 2017. The variance for the control group was 47.5 and the variance for the experimental group was 34.4.

final exam that are not part of the study. Obviously, there is a large combination of different variables that could contribute to exam improvement. For the control group, it may be that variables not controlled by the experiment were greater factors in determining exam improvement, thus explaining the higher variance in the exam improvement score for that group.

A Pearson's product-moment correlation was run to assess the relationship between subjects' combined performance on the three practice problems from the study (via extra credit scores) and their exam improvement scores. Note that the extra credit score is a continuous variable that considers fractional point values. Preliminary analyses showed that the exam improvement scores were normally distributed but the extra credit scores were slightly skewed, as assessed by visual inspection of Normal Q-Q Plots (Figures 5.2 and 5.3). Efforts to transform the data did not

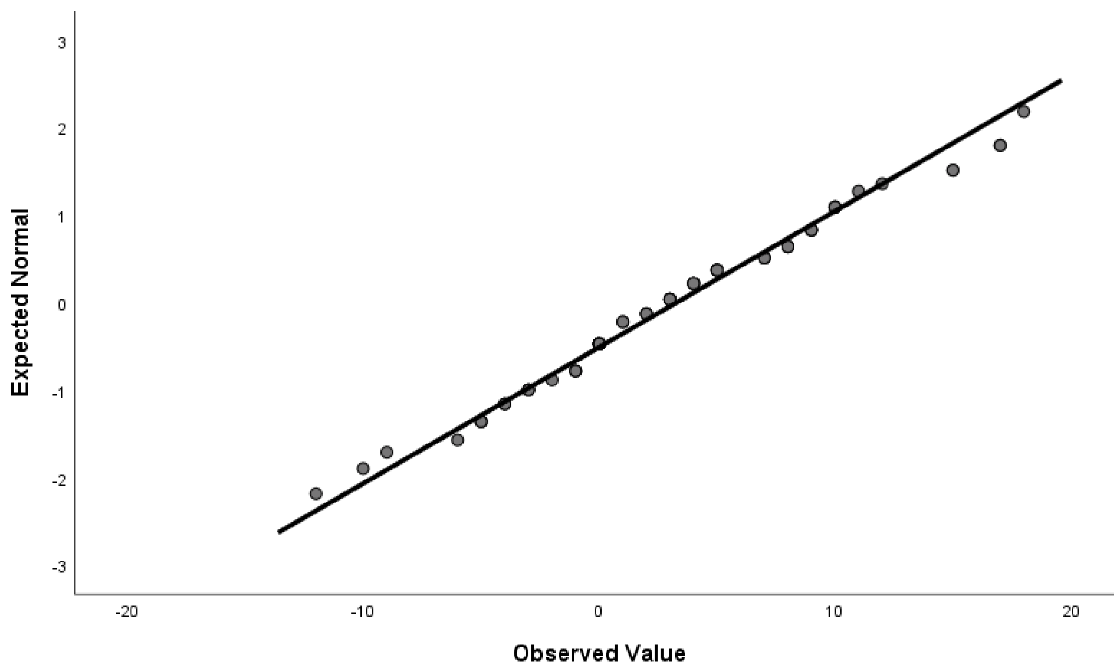


Figure 5.2: Normal Q-Q Plot of exam improvement scores from Spring 2017.

significantly improve the normal fit of the extra credit scores, so the data was left untransformed. Note that this raises the possibility that the statistical significance tests for Pearson's correlation will be invalid, since bivariate normality is a precondition for those tests. To compensate, an additional analysis was done in Section 5.1.3.1.

As shown in Figure 5.4, when considering all subjects from Spring 2017, there was no statistically significant correlation between their performance on the practice problems and their exam improvement. Amongst the subjects who got perfect scores on all three practice problems, there are subjects who had very positive exam improvement and subjects who did much worse on the final exam than the midterm exam. This indicates that the practice problems overall did not have a significant effect on exam improvement.

However, when you compare the experimental group (Figure 5.5) and the control group (Figure 5.6), differences do become evident. For the control group, there is

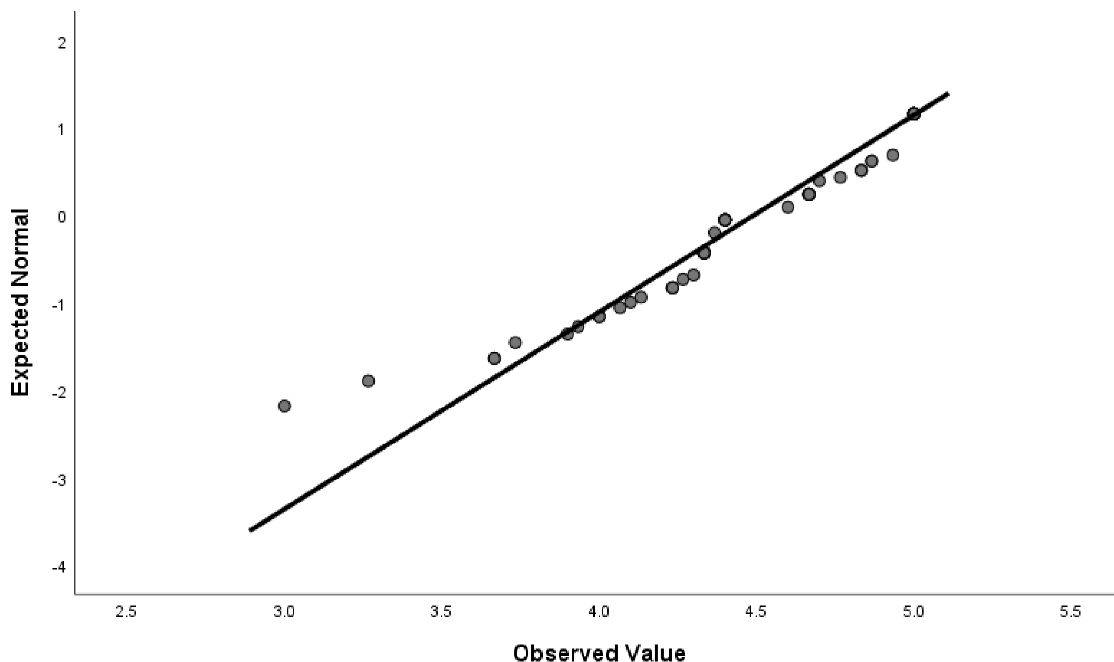


Figure 5.3: Normal Q-Q Plot of extra credit scores from Spring 2017.

still no statistically significant correlation between performance on the practice problems and exam improvement. However, for the experimental group, there is a mild positive correlation between these variables. This suggests that practice with proof construction in Complexity Tutor may have been more beneficial than practicing the problems with pen and paper.

It can be inferred that for subjects in the control group, the additional practice was not likely beneficial, since doing well on the practice problems did not correlate with improved exam performance. However, there may have been something beneficial in the way subjects constructed proofs in the Theorem Proving Environment that led those who used it to have improved exam performance. Perhaps, it was the fact that the Theorem Proving Environment forced subjects to construct proofs with fine granularity, not skipping logical inferences. Or perhaps, it was the fact that proofs were presented in a graphical layout that might have helped.

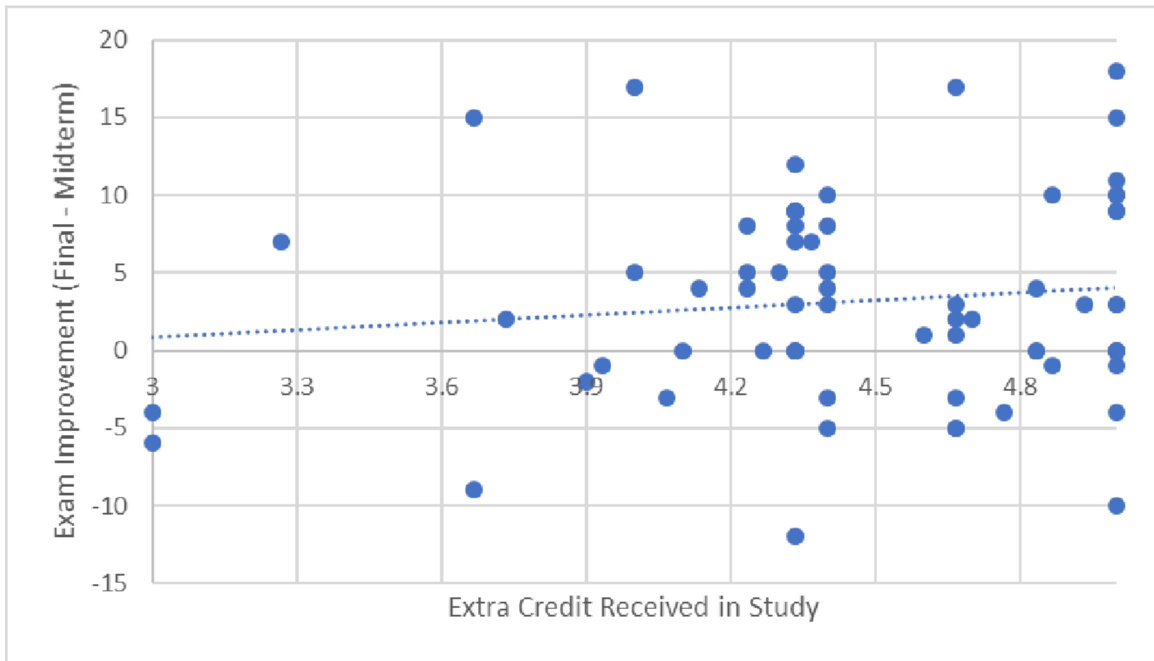


Figure 5.4: Scatterplot of all subjects from Spring 2017. Overall, there was no statistically significant correlation between extra credit earned in the study and exam improvement, $r(66) = .07$, $p = .566$.

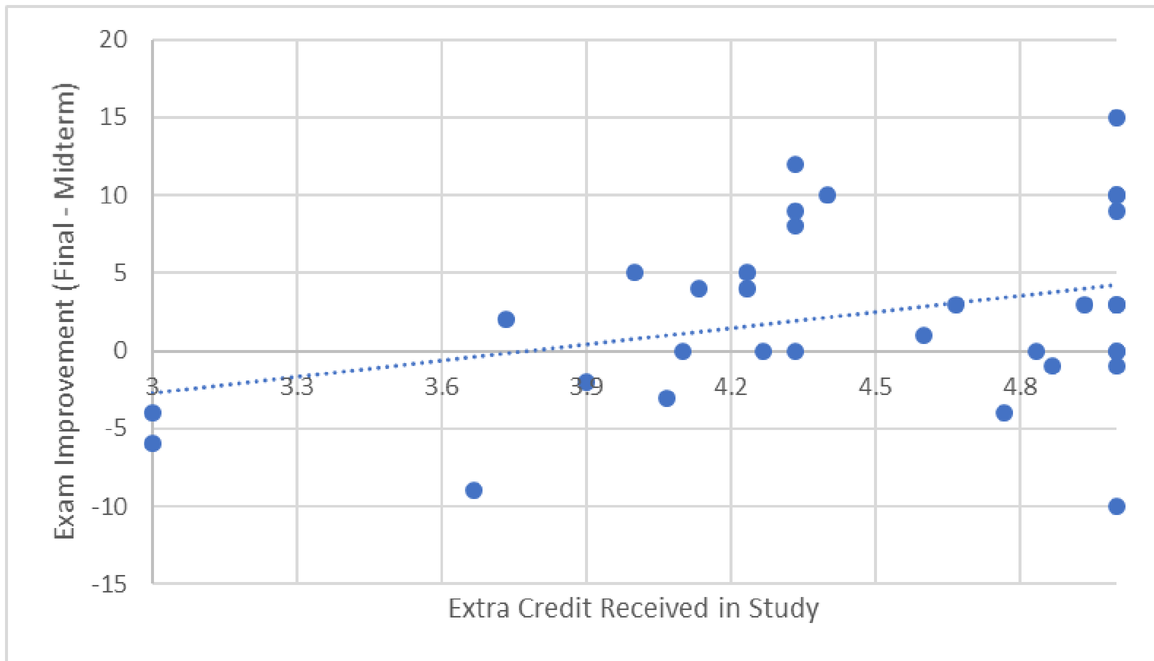


Figure 5.5: Scatterplot of experimental subjects from Spring 2017. For the experimental group, there was a mild positive correlation between extra credit earned in the study and exam improvement, $r(30) = .28$, $p = .120$.

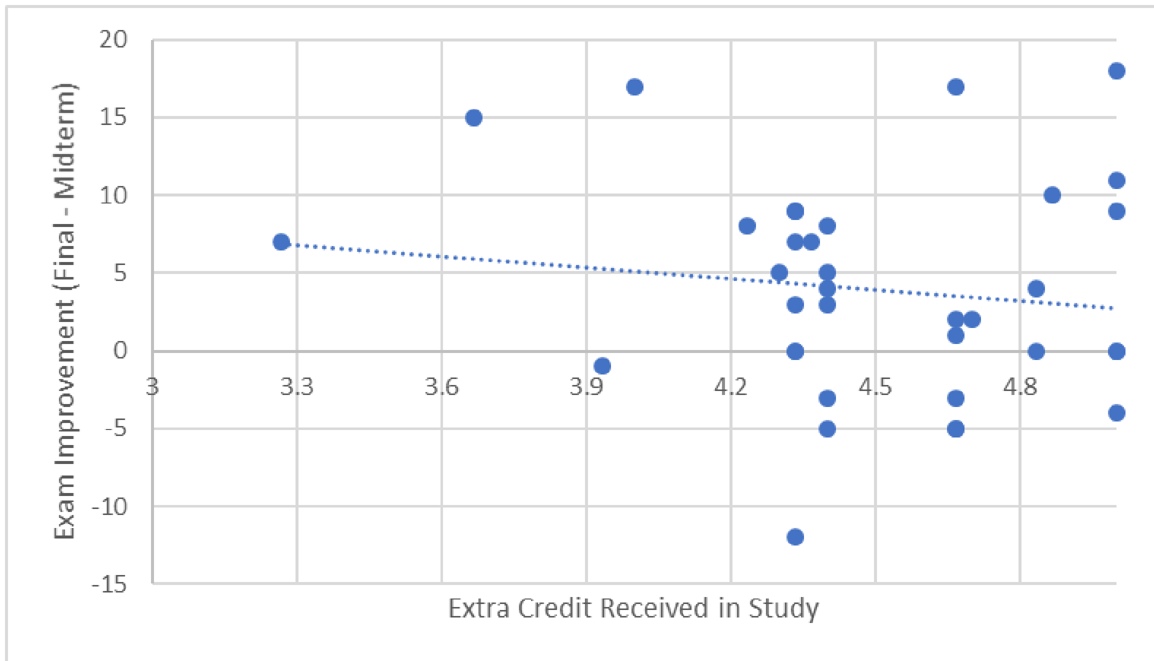


Figure 5.6: Scatterplot of control subjects from Spring 2017. For the control group, there was a very weak negative correlation between extra credit earned in the study and exam improvement that is not statistically significant, $r(34) = -.13$, $p = .440$.

Table 5.5: Pearson correlations between individual practice problems and exam improvement in Spring 2017.

	Exam Improvement	
	Experimental	Control
Sample Size	32	36
Pizza Problem	.29	-.02
Muddy Dog Problem	.37*	-.15
Murder Mystery Problem	.10	-.07

* = statistically significant at $p < .05$ level.

Further analysis was performed to determine if there was a relationship between performance on specific practice problems and the exam improvement score. Table 5.5 reveals that there was a statistically significant, moderate positive correlation for the experimental group between performance on the Muddy Dog Problem and exam improvement scores, $r(30) = .37$, $p = .036$, with performance on the Muddy Dog Problem explaining 14% of the variation in exam improvement scores for the experimental group. This single problem not only had the strongest correlation for the experimental group, but also the only statistically significant correlation in the entire analysis. This was not even close to true for the control group either.

Why would there be so much stronger correlation for the Muddy Dog Problem than any of the other problems? One interesting thing to note is Question 3 from the midterm exam, Question 2b from the final exam, and the Muddy Dog Problem are all problems involving dogs, so perhaps there is a *thematic effect* similar to what was witnessed in the *Wason selection task* (Section 2.3) that explains the performance differences.

Another possibility is that the proof required for the Muddy Dog Problem more closely matched the "complexity" of the proofs required for the exam questions. Note that the Muddy Dog Problem was originally created by the instructor who taught the course and who also created the exam questions. The Murder Mystery Problem, which had the lowest correlation with exam improvement, was significantly harder for subjects to solve than the other problems and required the most complex proof.

A hypothesis then is that perhaps, in terms of exam performance, students benefit the most from practicing problems which are close to the same difficulty as exam questions. Practicing problems much more difficult than the exam questions may be significantly less beneficial to exam performance, even if one is able to solve those harder problems.

It is important to remember that correlation is not causation. There could be an unknown third independent variable that predicts both the performance in the study and performance on the exam. For instance, maybe some of the subjects practiced some additional problems on their own that were not part of the study, and that additional practice led them to be both more successful at completing proofs in Complexity Tutor and getting a high score on the relevant exam questions. However, that would not explain why there is no correlation seen in the control group. As such, the hypothesis that practice with Complexity Tutor improved exam performance fits what is observed with the data better, even though that hypothesis is still not conclusively proven.

5.1.3.1 Second analysis with Goodman and Kruskal's γ

Pearson's correlation can only determine the strength of linear correlations, and the skew in the Spring 2017 extra credit scores adds some uncertainty to the reliability of its statistical interpretation. Furthermore, Pearson's correlation is very sensitive

to outliers, and there are a number of subjects in Figures 5.5, 5.6 and 5.4 who might be outliers—particularly the ones with very low exam improvement scores.

As such, Goodman and Kruskal's γ was also used as a secondary measure to determine the association between extra credit scores and exam improvement scores. Without assuming a linear relationship exists, Goodman and Kruskal's γ is good for assessing any monotonic relationship, which it calculates the strength of by tabulating *concordant* and *discordant* pairs of points in the data.

The results showed a weak, positive association between extra credit scores and exam improvement for the experimental group ($\hat{\gamma} = .193$, $p = .214$). There was also a weak, negative association between extra credit scores and exam improvement for the control group ($\hat{\gamma} = -.105$, $p = .481$). Neither association was statistically significant, although the one for the control group had a very high probability of the null hypothesis being true.

For the Muddy Dog Problem, the results showed a moderate, positive association between performance on that problem and exam improvement for the experimental group, which was almost statistically significant ($\hat{\gamma} = .305$, $p = .087$). There was also a weak, negative association between performance on the Muddy Dog Problem and exam improvement for the control group, which was not statistically significant ($\hat{\gamma} = .189$, $p = .269$).

In conclusion, a larger sample of data is necessary to confirm the statistical significance of the apparent positive association between practicing proof problems with Complexity Tutor and exam improvement.

5.1.3.2 Analysis of Fall 2017 results

For Fall 2017, there were unfortunately no statistically significant correlations to be found amongst either the experimental group or the control group, when a similar

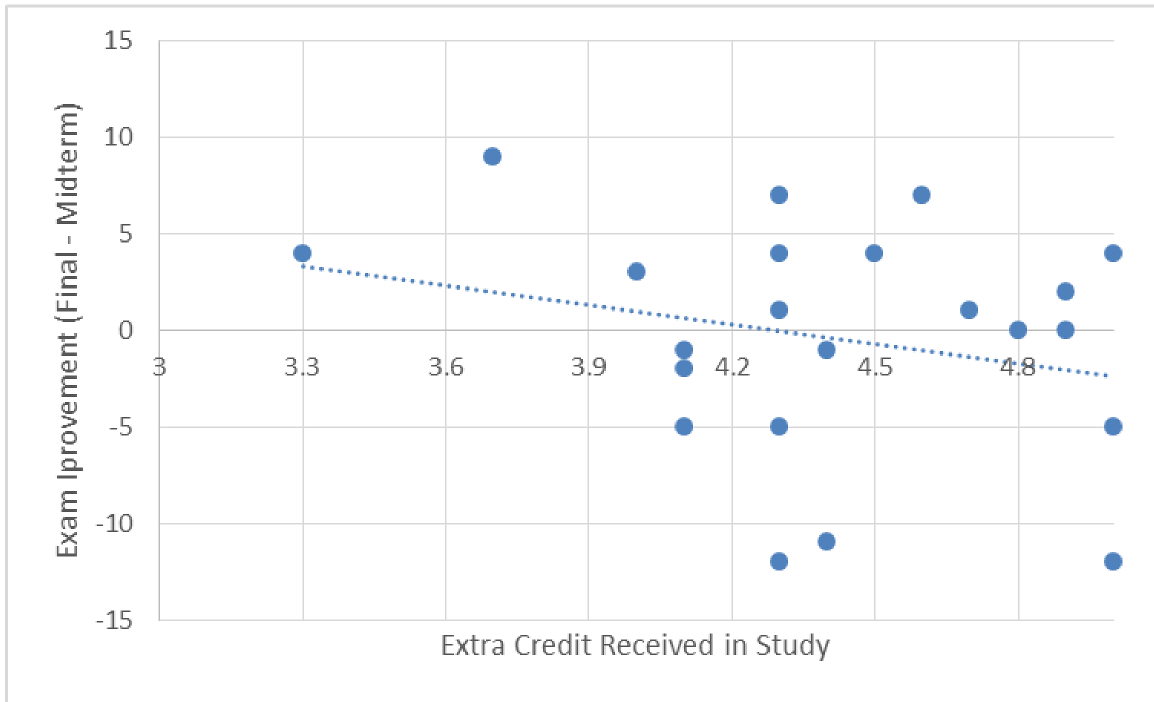


Figure 5.7: Scatterplot of control subjects from Fall 2017. For the control group, there was a weak negative correlation between extra credit earned in the study and exam improvement that is not statistically significant, $r(20) = -.25$, $p = .266$.

exam improvement score was compared to extra credit earned in the study. Refer to Figures 5.7 and 5.8.

Several factors must be considered before inferences can be drawn about why results differed for this semester from those of Spring 2017. First, the sample sizes were much smaller with only 20 subjects in the experimental group and 22 subjects in the control group.

Second, the questions used to calculate the exam improvement score for Fall 2017 give it a different and messier interpretation than for Spring 2017. For Spring 2017, each of the two questions used to calculate exam improvement only required the construction of exactly one prepositional logic proof. For Fall 2017, each question taken from the exam was actually multiple different questions combined together, and unfortunately a breakdown of subscores of each of these different question parts

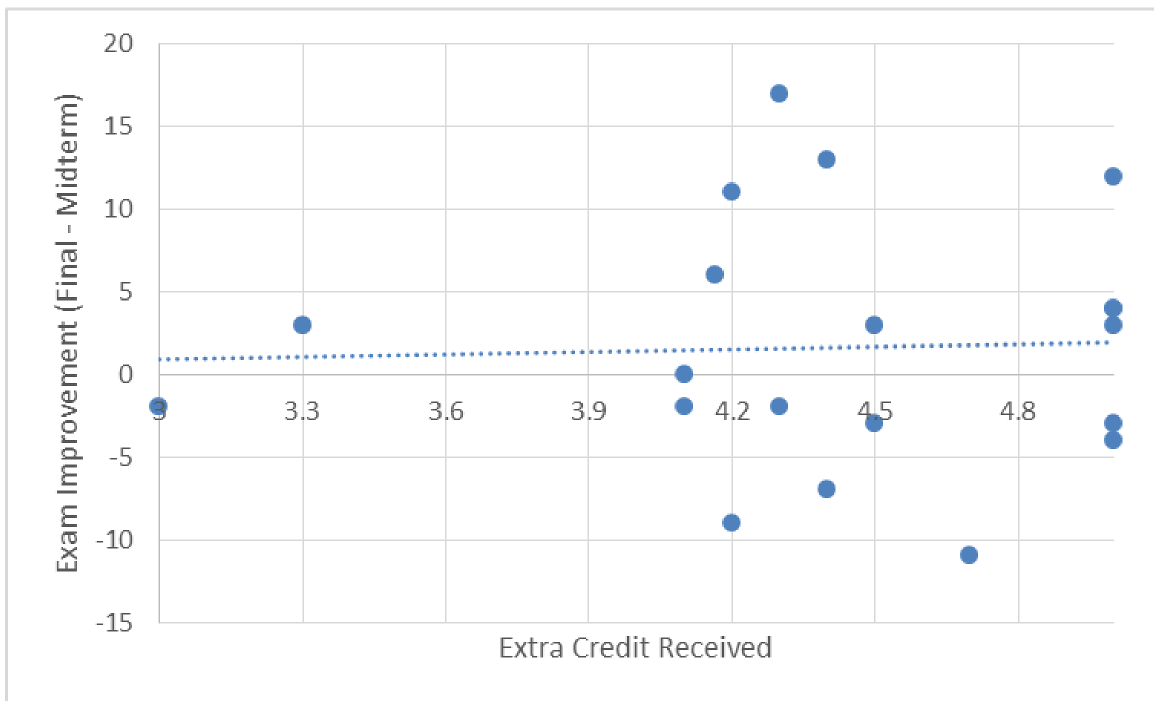


Figure 5.8: Scatterplot of experimental subjects from Fall 2017. For the experimental group, there was no statistically significant correlation between extra credit earned in the study and exam improvement, with the null hypothesis having a high probability of being true, $r(18) = .04$, $p = .880$.

was not available. Question 2 from the midterm exam required the construction of three different proofs, although the first proof could be substituted with a truth table. Question 1 from the final exam gave 10 out of 30 points for solving a translation problem, and the remainder of points for constructing two different proofs. Question 2 from the midterm exam was subtracted from Question 1 from the final exam to calculate the exam improvement score for Fall 2017.

Finally, a new version of Complexity Tutor with the features described in Section 3.1.7 was used for the first time in the Fall 2017 experiment. One hypothesis that must be considered is that one of the new features, most likely the hint line feature, might have been detrimental to helping students improve their ability at proof construction. Some subjects made comments to that effect, as explained in Section 5.3.4.

5.2 Evaluation of the Algorithm Environment

Table 5.6 presents summary statistics for how subjects in the study performed on the NP-completeness reduction problems using the Algorithm Environment. As can be seen, subjects in the experimental group had more trouble completing these problems than they did the conceptual problems only involving the Theorem Proving Environment, especially in Fall 2016. Of course, the control group did not do too well on the reduction problems either in that semester.

Overall poor performance for all subjects in the study in Fall 2016 may partially be explained by the fact that the incentive structure used for that semester rewarded all subjects the same amount of extra credit for their participation regardless of how much effort they put into the practice problems. However, the fact that none of the subjects in the experimental group were able to successfully produce a correct reduction in the Algorithm Environment that semester was disheartening.

Table 5.6: Summary statistics for the NP-completeness reduction problems. The *Made Attempt* row indicates the percentage of subjects who attempted each problem. For the experimental group, a subject was only considered to have attempted a problem if they made at least one connection in the Theorem Proving Environment or wrote some code in the Algorithm Environment. The *Mean* and *Median* rows indicate average scores over the entire sample, where subjects who did not attempt a problem were factored into the average with a 0% score. The *Perfect* row indicates the percentage of subjects who got a perfect score amongst those who attempted the problem.

	BIN-PACKING Reduction Problem				**O/1-PROG Reduction Problem**			
	Fall 2016		Spring 2018		Fall 2016		Spring 2018	
	Experimental	Control	Experimental	Control	Experimental	Control	Experimental	Control
Sample Size	25	22	24	18	25	22	24	18
Made Attempt	100%	100%	100%	89%	92%	82%	100%	78%
Mean	43%	52%	62%	52%	24%	32%	46%	32%
Median	50%	45%	50%	60%	20%	30%	30%	25%
Perfect	0%	18%	38%	13%	0%	0%	8%	7%

Hence, an extensive analysis of the videos of Fall 2016 subjects working on the reduction problems in the Algorithm Environment was performed, in order to pinpoint what may have caused hardship. The conclusions of that analysis follow.

5.2.1 Evaluation of interactions with the Algorithm Environment in the Fall 2016 experiment

There were 25 subjects in the experimental group in the Fall 2016 semester. Of these 25 subjects, 15 attempted to write Python code in the Algorithm Environment for the BIN-PACKING Reduction Problem. An additional 4 subjects looked at the Algorithm Environment but did not attempt to write code. And 6 subjects did not even bother to click the "Algorithms" tab in Complexity Tutor to look at the Algorithm Environment.

For the 0/1-PROG Reduction Problem, the results were even more lackluster. Only 5 subjects wrote code for 0/1-PROG Reduction Problem, and 9 subjects did not bother to even click the "Algorithms" tab. Of these 9 subjects, none had attempted to write code for the 0/1-PROG Reduction Problem either.

In all cases, no one managed to produce a correct *Levin reduction* in the Algorithm Environment. Since very few attempts were made to write code for the 0/1-PROG Reduction Problem, the focus for analysis was on looking at how subjects interacted with the Algorithm Environment when working on the BIN-PACKING Reduction Problem. Figure 3.9 from Chapter 3 gives a screenshot of how this problem is presented to subjects in the Algorithm Environment.

5.2.1.1 Did programming ability or prior familiarity with Python affect the results?

One hypothesis might be that the Algorithm Environment's requirement that subjects write Python code rather than pseudocode would explain why subjects in

the study were not able to use the Algorithm Environment to produce a correct reduction. As will be shown, this hypothesis is most likely not correct.

At the beginning of the experiment, subjects were instructed to watch a 20 minute "Crash Course in Python" tutorial video. This video briskly covered all the basics of the Python programming language that subjects would need to do the NP-completeness reductions in the Algorithm Environment.

The purpose of the video was two-fold. First, it was to give subjects the knowledge of Python needed to use the Algorithm Environment, since a large number of computer science students at the University of Massachusetts Amherst have never used Python before. Second, the video was used to benchmark subject familiarity with Python, to help determine if that was a variable affecting performance.

At the end of the experiment, subjects were given this survey question:

> **Which of the following best describes your reaction to the Crash Course in Basic Python tutorial video?**
>
> A. I've never programmed in Python before, and the tutorial was confusing.
>
> B. I've never programmed in Python before, but the tutorial was easy to understand.
>
> C. I already had some Python programming experience, but the tutorial taught me new stuff.
>
> D. I am a Python ninja, and there was nothing for me to learn from that tutorial video.

One subject did not respond at all to the survey. Of the remaining 24 subjects, the results of the survey are shown in Figure 5.9.

Interestingly, the subjects who had never used Python before but who found the tutorial easy to understand were the ones most likely to attempt to write Python

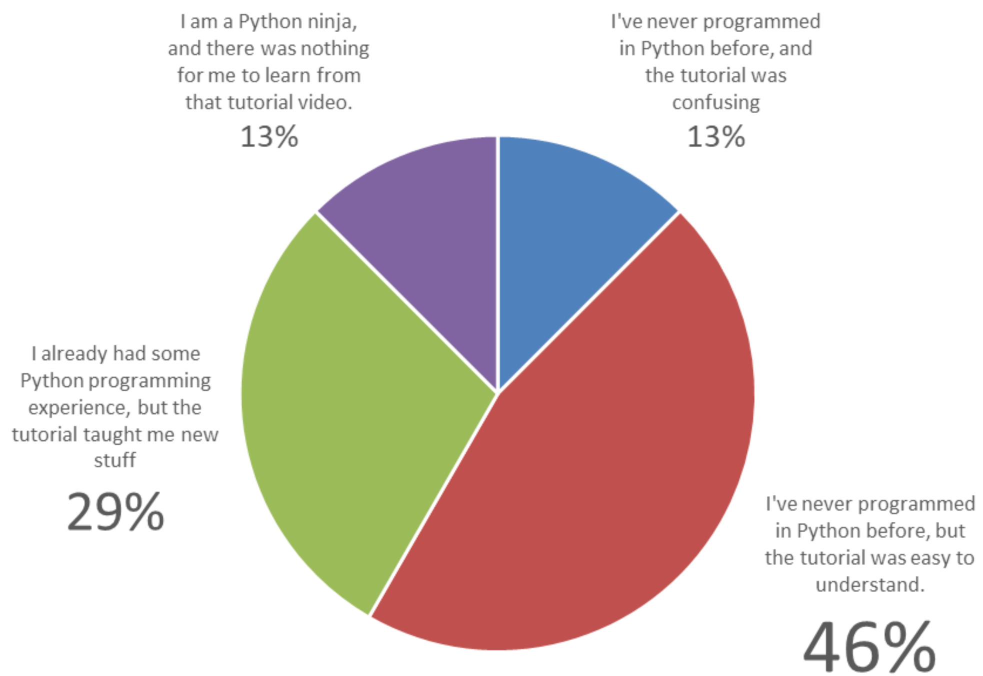


Figure 5.9: Survey results on familiarity with Python amongst Fall 2016 subjects from the experimental group.

Table 5.7: Did familiarity with Python affect a subject's willingness to attempt using the Algorithm Environment?

	Wrote code in Algorithm Environment	Looked at Algorithm Environment	Did not look at Algorithm Environment
I've never programmed in Python before, and the tutorial was confusing.	0	1	2
I've never programmed in Python before, but the tutorial was easy to understand.	8	2	1
I already had some Python programming experience, but the tutorial taught me new stuff.	4	0	3
I am a Python ninja, and there was nothing for me to learn from that tutorial video.	2	1	0

code in the Algorithm Environment, as shown in Table 5.7. This could perhaps be because these subjects were excited to have learned a new programming language and were eager to test out what they had learned.

The following is a list of all the *programming errors*[2] that were collectively made by the 15 subjects who attempted to write Python code for the BIN-PACKING Reduction Problem:

- Typographical errors in variable names or keywords
- Failure to close a matching parenthesis
- Referencing the wrong variable

[2]Errors that were deemed to be likely attributable to conceptual misunderstandings of the reduction problem are not counted in this list, even if they caused the Python interpreter to generate a runtime error. For instance, subject 43679835 received an "int is not iterable" error message because they were treating output variable Y as an integer in their code. This indicates that they most likely did not understand the problem constraints, requiring Y to be a list datatype.

- Using incorrect syntax to index into a list
- Omitting the colon after a `for`/`if` statement
- Writing "`for each`" instead of "`for`"
- Terminating a line with a semicolon
- Typing '\' instead of '/' for division
- Attempting to index into a list with no elements
- Applying `float()` to a list rather than to elements in a list (i.e., `float([x])` vs `[float(x)]`)
- Applying `list()` to a non-iterable element
- Attempting to use incorrect idiom to append elements to a list (e.g., `List = List + element`)

Each of these errors were made by at most two subjects, so none of the errors were particularly more common than the others. Notice that the first three errors listed above are general mistakes that any programmer could make, where as the remaining errors may specifically be caused by lack of comfort with the Python programming language.

Nevertheless, subjects made relatively few programming errors and wrote mostly correct Python code. Overall, each subject made an average of **1.13 programming errors**. The average number of programming errors made by the 8 subjects who had never programmed in Python before was only slightly higher—**1.38 programming errors**.

Furthermore, subjects were in most cases able to correct their errors very quickly, once the error was identified by the Algorithm Environment.[3] Often, subjects demonstrated the ability to correct simple syntactical programming errors within a matter of seconds. The pie chart in Figure 5.10 breaks down programming errors by how quickly they were resolved.

5.2.1.2 Were there any programming errors not identified by the Algorithm Environment?

All the errors mentioned in the previous section were obvious programming errors because they are either not valid Python or they triggered a runtime exception.

Sometimes, however, there may be less obvious programming errors. It is possible for someone to write fully valid Python code, but still intend the code to behave differently than how it actually does. This would still be considered a programming error, rather than a conceptual or algorithmic error, if understanding the programming language better would prevent the mistake from being made.

Such programming errors would not be identified by the Algorithm Environment directly, although the Algorithm Environment would indicate that the algorithm is not correct since it fails test cases. Without being able to read the mind of the programmer and knowing their intent, it is impossible to have absolute certainty of these *hidden programming errors*, but a few likely ones have been found that are not counted in the previous section.

[3] Note that when a programming error is said to be "identified by the Algorithm Environment", this means that the error directly triggers an error message indicating either a *syntax error*, *semantic error*, or *runtime error* has occurred in Python. As with most compiler error messages, the specific contents of the error message may be oblique and not always a correct characterization of the mistake that the programmer has made. Therefore, it is indeed impressive that subjects could quickly figure out the programming errors that produced these error messages. Note also that the specific version of Complexity Tutor used in Fall 2016 had a bug where the line numbers it specified for where errors occurred was off by 1, but this did not appear to hinder subjects much from quickly identifying their programming errors.

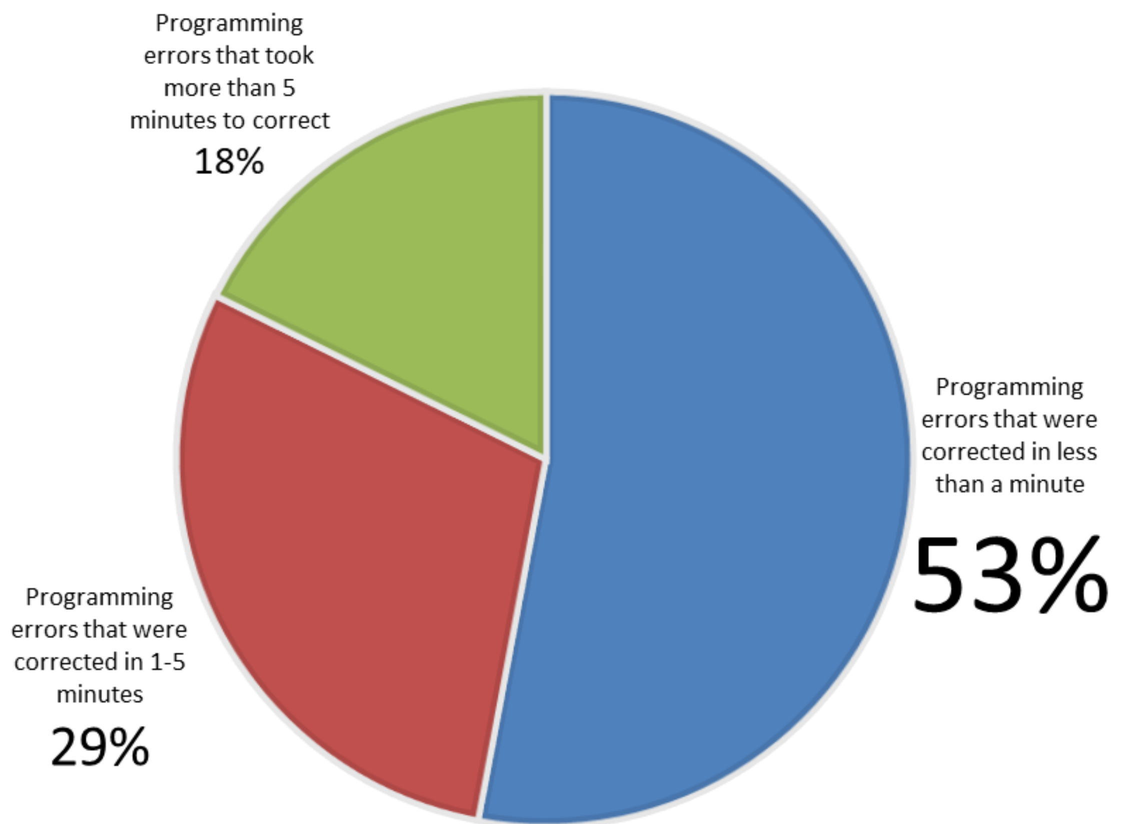


Figure 5.10: How long did it take for a programming error to be corrected once identified by the Algorithm Environment?

Consider subjects 197190801L and 13013435. These two subjects got far closer to producing a correct NP-completeness reduction than anyone else did for the BIN-PACKING Reduction Problem during the Fall 2016 semester. Both 197190801L and 13013435 had a nearly correct *Karp reduction*, even though their *certificate reductions* were not quite correct. However, one additional flaw in their code was that their reductions used integer arithmetic to do division, which would lead to serious rounding errors. This was not likely their intention, especially since they later converted the result of the integer division to a floating-point number. Most likely, they intended floating-point division, and either forgot or did not know that Python would assume integer division when both operands are integers.

Here is another example. Subjects 40470021 and 61907491, who had never programmed in Python before, both produced what appear to be very similar hidden programming errors. Since they were similar, the code that subject 40470021 produced will be used to illustrate:

```
1 def Reduce_Partition_to_BinPacking(X):
2     Y=list(X)
3     K=0
4     for x in X:
5         K=K+x
6         x=float(x/K)
7     return (Y,K)
```

On Line 6, the variable ‘x’ is reassigned to a new value and then promptly discarded before the next iteration of the loop. Thus, Line 6 in effect does nothing, so that is not likely what was intended. Presumably, subject 40470021 intended Line 6 to modify the list of elements in ‘X’ and did not realize that since those elements are immutable, a reassignment of the ‘x’ variable will never change them.

While hidden programming errors in the Algorithm Environment are a potential trap that students who write pseudocode on paper and have it graded by a human would not need to worry about, this phenomenon thankfully did not occur often. There were no other hidden programming errors identified beyond those of the four subjects mentioned above.

5.2.1.3 Summary of findings about substituting Python programming for writing pseudocode

It can be concluded from the analysis of programming errors of the subjects from Fall 2016 that requiring students to write executable Python code was not a considerable burden over writing pseudocode. Despite most subjects in the study not having background with Python, they were able to produce nearly flawless Python code. Simple programming errors that did occur were usually quickly corrected, once they were brought to the subject's attention. Hidden programming errors could potentially be a complication to overcome, but they did not seem to occur very often in the Fall 2016 study. Interestingly, the students who knew the least about Python were the most willing to engage in this experiment.

5.2.1.4 Why the Algorithm Environment did not help students successfully produce NP-completeness reductions in Fall 2016

If the requirement of programming was not the reason that Fall 2016 subjects struggled to successfully produce a correct reduction in the Algorithm Environment for the BIN-PACKING Reduction Problem, then what was?

There are two considerations. First, subjects in the experimental group were expected to produce Levin reductions, which they had not been previously taught about in their class, and for which they had only learned about from a short tutorial video. From watching subjects' interactions with the Algorithm Environment, it was

very clear that they were struggling the most with figuring out what the certificate reduction was supposed to be.

Also, the control group was not expected to produce Levin reductions, so the comparison between the two groups is not so fair. The four subjects from the control group who received a perfect score for the problem produced Karp reductions. There were also two subjects in the experimental group who produced conceptually correct Karp reductions, but are not counted in Table 5.6 amongst the "Perfect" row since they were expected to produce a correct Levin reduction and failed to do so. However, those two subjects received scores of 99.9% and 99.8% when partial credit was manually assigned, so statistically the averages of the experimental and control groups should still be comparable.

The second consideration is that the Algorithm Environment seemingly did not give sufficient guidance to help subjects overcome their confusion about the problem or Levin reductions in general. It is worth noting that there were two subjects, 786W and 55187920, who did not even seek guidance from the Algorithm Environment because neither of these subjects bothered to click the "Check Algorithms" button even once. Subject 786W attempted to write some code for the Karp reduction but did not write anything for the certificate reduction. Subject 55187920 did not even get that far—they gave up while they were in the middle of typing code for the Karp reduction.

So not counting these two subjects, in actuality there were only 13 subjects in Fall 2016 who attempted to use the Algorithm Environment to help them.

To describe the flow of interaction for these subjects, consider that when they first started working on this problem, they were presented with the following code[4]:

[4]The comments in this code have been slightly reformatted to fit neatly on this page. See Figure 3.9 from Chapter 3 for a screenshot of exactly what subjects are presented with for this problem.

```
1  def Reduce_Partition_to_BinPacking(X):
2     ## HINT: If you have an integer value x, use
3     ##       float(x) to convert it to a float.
4
5     # Input: X - list of positive integers,
6     #        e.g. X={1,2,3}
7     # Output: (Y,K) - Y is a list of floats,
8     #           K is positive integer
9     def Reduce_Partition_to_BinPacking(X):
10
11        return (Y,K)
12
13    # Input:  Certificate c = (part1,part2) where
14    #         part1 and part2 are two lists of
15    #         positive integers with equal sum
16    #         e.g. c = ([0.2,0.6],[0.4,0.4])
17    # Output: Certificate c2 is a list of list
18    #         (aka bins) of floats where each
19    #         list sums to no more than 1
20    #         e.g. c2 = [[0.2,07],
21    #           [0.1,0.4,0.2,03],...,[0.9,0.004]]
22    def Cert_Partition_to_BinPacking(c):
23
24       return c2
```

The starting code contains two empty function signatures and comments that attempt to explain the input and output conditions for the two required functions. When attempting to run this starting code in a Python interpreter, one would receive

errors because the output variables for both functions have not yet been defined. In fact, three subjects did try clicking the "Check Algorithms" button before writing any of their own code, just to see what would happen. This resulted in an error, explaining to them that the variable 'Y' on Line 11 is undefined, since 'Y' is the first undefined variable referenced in the code.

Whether subjects had received that error or not, all of them that wrote code would start with the `Reduce_Partition_to_BinPacking` function before approaching the more intimidating `Cert_Partition_to_BinPacking` function they had never learned about in their class. All but one of the 13 subjects would then click the "Check Algorithms" button before writing any code for the `Cert_Partition_to_BinPacking` function.

If the Python interpreter did not get stuck on any programming errors that the subject had generated while writing code for `Reduce_Partition_to_BinPacking`, it would at that point complain that the output variable 'c2' on Line 24 of `Cert_Partition_to_BinPacking` was still undefined. So subjects would receive an error message redirecting their attention to `Cert_Partition_to_BinPacking` even if their algorithm for `Reduce_Partition_to_BinPacking` was still incredibly flawed.

In fact, 8 of the 13 subjects were given this error message about `Cert_Partition_to_BinPacking`, even though the Karp reduction in `Reduce_Partition_to_BinPacking` remained conceptually incorrect. Presumably they might have at that point had the false hope that their Karp reduction was correct because the Algorithm Environment had given no complaints about it yet.

The certificate reduction was understandably more daunting than the Karp reduction, since subjects had little exposure to Levin reductions, but one would expect it would be even more daunting if the subject was trying to think of the certificate reduction without having a good grasp of the Karp reduction in mind. Thus, it is not surprising that three subjects gave up while trying to figure out what code to write

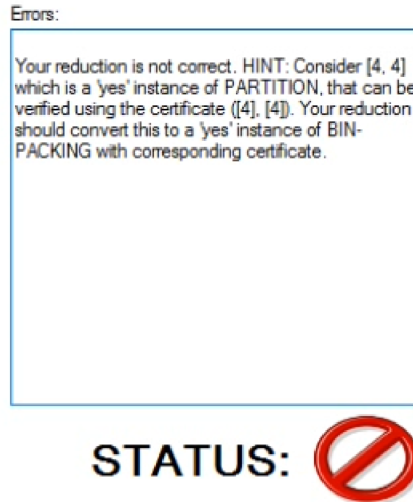


Figure 5.11: Error message in the Algorithm Environment indicating that a reduction for the BIN-PACKING Reduction Problem fails to pass the test case `[4,4]`.

for `Cert_Partition_to_BinPacking` without even reaching the point where they would be notified that that the code they had written for `Reduce_Partition_to_BinPacking` was also incorrect.

Of the subjects who persevered through their uncertainties, and managed to write valid Python code for both the `Reduce_Partition_to_BinPacking` and `Cert_Partition_to_BinPacking` functions, they would eventually be greeted with an error message like the one in Figure 5.11.

While this error message is more feedback than subjects in the control group would receive while working on the reduction problems, since they receive no immediate feedback, it is still a rather vague error message. The message says that there is something wrong with the reduction but it does not indicate whether there is a problem with `Reduce_Partition_to_BinPacking` or `Cert_Partition_to_BinPacking` or both. In almost every case, both functions were incorrect. The only hint given in the error message is a test case that the reduction failed on.

Four of the subjects gave up the first time they received an error message like the one in Figure 5.11. An important observation to make is that these four subjects had never programmed in Python before this experiment, according to their responses to the survey on Python familiarity. So even though they made very few programming errors, lack of confidence in their Python programming ability may have resulted in them giving up sooner than they would have if they were more confident.

In fact, subject 95873118, who was one of those four, typed a comment on the screen at the beginning of their video mentioning that they do not know Python. This subject went on to produce nearly flawless Python code, clearly underestimating their own abilities.

Consider that the error message in Figure 5.11 does not even specify what has been outputted by the code that the test case failed on. So the subject does not know if the reduction failed the test case because the algorithm was incorrect or because their code is not doing what they intended it to do. Subjects who are new to Python might understandably fear the latter.

In comparison, the two subjects who purported to be "Python ninjas" on the Python familiarity survey, did not give up immediately the first time they received the error message in Figure 5.11, but instead persevered despite receiving this same vague error message multiple times in row. In fact, subject 13013435 received the error message from Figure 5.11 a total of six times before finally throwing in the towel!

So what kind of errors were subjects making in their reductions that would trigger this frustrating message? Surprisingly, in many instances, subjects' reductions were so far from correct that they did not even fit the problem constraints, which had been specified in both the problem description and the comments of the code.

There are two possible reasons this may have occurred:

1. The subject did not understand the problem constraints, in spite of the descriptions given.

2. The subject understood the problem constraints but did not realize that their code was violating those constraints.

It is not possible to distinguish between these two possibilities with absolute certainty just by watching the subjects' interactions. However, in many cases it seems likely that if subjects understood the problem correctly, they would not have written the code that they did, because it should seem very obvious that the code would violate the problem constraints.

For instance, consider the code that subject 82638476, who purported to be a "Python ninja", had written for `Reduce_Partition_to_BinPacking` before giving up:

```
1 def Reduce_Partition_to_BinPacking(X):
2     Y=[]
3     for x in X:
4         Y.append([float(x)])
5     K = len(Y)
6     return (Y,K)
```

This code clearly does not meet the problem constraints given in the problem description, which state that `Y` should be a set of real numbers, where each number is in the range $[0, 1)$. Real numbers are representing by floating-point values in the Algorithm Environment. In Line 4, the subject seems to recognize that they should generate a floating-point value, but they fail to ensure that it is in the range $[0, 1)$. How is that possible when the problem constraints also specify that `X` should be a list of positive integers. Furthermore, they put each floating-point number in its own

list, and return a list of lists rather than a list of floating-point numbers. While the latter may be a simple bug that they did not intend, it seems unlikely that this "Python ninja" would assume that the `Y` value they were creating with this code was in the range $[0, 1)$. More likely, they were oblivious to this even being a problem requirement.

After recognizing that these kinds of mistakes were frequently being made by the Fall 2016 subjects, it was hypothesized that subjects might be more likely to succeed at producing a correct reduction in the Algorithm Environment if they were given basic advice about how their solutions failed seemingly obvious problem constraints before they ever received an error messages like the one in Figure 5.11.

5.2.2 Scaffolding of hints for the BIN-PACKING Reduction Problem

Based on observations made from the Fall 2016 experiment, the evaluation engine used to give feedback to students for their solutions to the BIN-PACKING Reduction Problem was modified to provide additional layers of scaffolding.

Refer to the description of the BIN-PACKING Reduction Problem in Appendix B and the screenshot shown in Figure 3.9 in Chapter 3. The process that the modified evaluation engine uses to evaluate students' solutions for this problem is now as follows:

1. The `Reduce_Partition_to_BinPacking` function is run on a sample input. At this point, the Python interpreter will catch runtime errors such as if any variables are referenced without being defined. If runtime errors occur, stop and report them. Otherwise, continue.

2. Check if the output produced by Step 1 is a tuple (Y, K) of two elements. If not, stop and report that the output of `Reduce_Partition_to_BinPacking` is not in the right format because it is supposed to be a tuple of two elements, and

display the incorrect output and the sample input that produced it. Otherwise, continue.

3. Check if the first element of the output of Step 1, problem variable Y, is a Python list. If not, stop and report that the output of `Reduce_Partition_to_BinPacking` is not in the right format because the first element of the output is not a list, and display the incorrect output and the sample input that produced it. Otherwise, continue.

4. Check each element in Y from the output of Step 1 to verify that they are all floating-point numbers in the range $[0, 1)$. For the first element that is not a floating-point or not in the range $[0, 1)$, stop and report that the output is not in the correct format because it contains elements that are either not floating-point or not in the range $[0, 1)$, and display the incorrect output and the sample input that produced it. Otherwise, continue.

5. Now, it is time to evaluate `Cert_Partition_to_BinPacking`. Run it on a sample input. The Python interpreter will catch runtime errors such as if any variables are referenced without being defined. If runtime errors occur, stop and report them. Otherwise, continue.

6. Check to make sure that the certificate output c produced by Step 5 is a Python list. If not, stop and report that the output of `Cert_Partition_to_BinPacking` is not in the right format because the output is not a list, and display the incorrect output and the sample input that produced it. Otherwise, continue.

7. Check to make sure that each element of the certificate output c produced by Step 5 is itself also a list. If not, stop and report that the output of `Cert_Partition_to_BinPacking` is not in the right format because it is supposed

to be a list of lists but contains non-list elements, and display the incorrect output and the sample input that produced it. Otherwise, continue.

8. Check to make sure that each sub-list of the output of Step 5 is a list of floating-point numbers. If not, report that the output is not in the right format because it is supposed to be a "a list of lists, where each sublist represents a 'bin' of floating point values", and display the incorrect output and the sample input that produced it. Otherwise, continue.

9. Finally, use a set of test cases to verify the complete Levin reduction, as described in Section 3.2.3. If a test case fails, stop and report it[5] *along with the specific output that the student's code produced for both the* `Reduce_Partition_to_BinPacking` *and* `Cert_Partition_to_BinPacking` *functions.*

The evaluation engine used in Fall 2016 jumped immediately to Step 9, and did not have the previous steps. Furthermore, it did not report the output of the students' code on the failed test cases. This left it up to students to trace their own algorithms, but students who were not confident in their knowledge of Python may have been intimidated by this prospect.

The general principle of the hint scaffolding is to report ways in which students' solutions violate basic problem constraints, starting with the violations that are most likely to be high-level misunderstandings of the problem structure before reporting violations of specific details that the student may have missed in the problem description, or that may have been caused inadvertently by a bug. Also, violations in the constraints of `Reduce_Partition_to_BinPacking` are reported before any advice is given about `Cert_Partition_to_BinPacking`. This would be especially

[5]Some test cases are withheld from being reported to prevent the student from writing code that is only tailored to specific test cases. However, it is not likely that a student would exhaust the list of reportable test cases before producing a correct reduction. In the unlikely event that this happens, the student receives a generic message just informing them that their reduction is incorrect.

important for subjects in the study, who likely had no exposure to Levin reductions prior to their participation. They should focus on understanding what is required of them for `Reduce_Partition_to_BinPacking`, which they are likely to be more familiar with since it is a normal Karp reduction, before worrying about the foreign `Cert_Partition_to_BinPacking`.

In Fall 2016, subjects were inadvertently encouraged by Complexity Tutor to focus on errors in `Cert_Partition_to_BinPacking` when they still had major conceptual misunderstandings about `Reduce_Partition_to_BinPacking`. With the new scaffolding, that is no longer as likely to happen.

5.2.3 Improvement in Spring 2018 results

For the Spring 2018 experiment, the hint scaffolding mentioned in Section 5.2.2 was used with the BIN-PACKING Reduction Problem. On the other hand, subjects were given the same version of the 0/1-PROG Reduction Problem that had been used in Fall 2016, with its limited feedback, so that it could be used as a baseline for comparison.

Table 5.6 shows significant improvements in the Spring 2018 experimental group's performance on the BIN-PACKING Reduction Problem over the Fall 2016 experimental group. There were 9 out of 24 subjects from the experimental group who successfully completed this reduction problem in Spring 2018 compared to 0 out of 25 in Fall 2016.

The experimental group in Spring 2018 also did favorably in comparison to the control group on this problem, where only 2 subjects received full credit for the problem. There were an additional 6 subjects in the control group who got partial credit of 70% on the BIN-PACKING Reduction Problem and also one who got 80%—those subjects were close to having a correct reduction. The remaining subjects in the control group were not even close to a correct solution.

Of course, the fact that the subjects in the experimental group had to do Levin reductions, but still out-performed the control group, is impressive.

Subjects in the experimental group still did relatively poorly on the 0/1-PROG Reduction Problem, as would be expected given that it had not been updated like the BIN-PACKING Reduction Problem to have scaffolded hints. However, even there, improved outcomes were noticeable in comparison to Fall 2016. Two subjects from the experimental group, 35917331 and 94521523, actually successfully completed the 0/1-PROG Reduction Problem. Subject 35917331 was able to complete this problem in two hours, and for subject 94521523, it took only one hour. Those two subjects also received perfect scores for all the practice problems given in the experiment.

It is conceivable that after these two subjects had successfully completed the BIN-PACKING Reduction Problem that their confidence and understanding of Levin reductions had increased to the point where the 0/1-PROG Reduction Problem was easier to conquer. It is also possible that they just happened to be exceptionally brilliant.

However, it is also worth noting that 7 of the 9 subjects who successfully completed the BIN-PACKING Reduction Problem also attempted the 0/1-PROG Reduction Problem, and all but one of those 7 subjects got reasonably close to a correct solution. This is why the averages for the experimental group were higher than the control group for the 0/1-PROG Reduction Problem. Only two subjects in the control group received a score above 50% on the 0/1-PROG Reduction Problem—one got the problem perfectly correct, and the other received a partial credit score of 70%. Everyone else in the control group did poorly on this problem.

Hence, all this would seem to indicate that the change in hint scaffolding for the BIN-PACKING Reduction Problem may have resulted in improved performance for the experimental group over Fall 2016.

Table 5.8: Did you find the Complexity Tutor tutoring system helpful in your learning how to construct proofs?

	Fall 2016	**Spring 2017**	**Fall 2017**	**Spring 2018**	**Total**
Responses	25	39	19	24	107
Yes	64%	36%	42%	50%	47%
Undecided	20%	23%	32%	42%	28%
No	16%	41%	26%	8%	25%

There are of course other variables that should be considered. Could a change in student population between those taking the algorithms course in Fall 2016 and those taking it in Spring 2018 explain the improvements made by the experimental group? Possibly, but not too likely, since the control group's performance was very similar between the two semesters.

A more likely scenario is that changing the extra credit incentive may have encouraged Spring 2018 subjects to put more effort into their participation. However, that alone probably does not explain the substantial jump in the number of subjects who could successfully complete the BIN-PACKING Reduction Problem. During the analysis of the Fall 2016 experiment, it became clear that many subjects analyzed did invest substantial time in attempting to use the Algorithm Environment to do the BIN-PACKING Reduction Problem but the feedback they were receiving was not helping them overcome their misunderstandings.

5.3 Feedback about Complexity Tutor

Tables 5.8, 5.9, 5.10, 5.11 and 5.12 show that a majority of subjects who used Complexity Tutor had favorable opinions of it, based on the results of the questionnaire that subjects were given.

Interestingly, for many of the questions, the most decisively favorable responses come from the semesters where the NP-completeness experiments were done, Fall 2016 and Spring 2018.

Table 5.9: Do you think the Complexity Tutor tutoring system trains you to be meticulous (careful about not skipping or overlooking obvious or evident assertions) when you construct proofs?

	Fall 2016	**Spring 2017**	**Fall 2017**	**Spring 2018**	**Total**
Responses	25	39	19	24	107
Yes	56%	54%	47%	71%	57%
Undecided	24%	13%	11%	13%	15%
No	20%	33%	42%	17%	28%

Table 5.10: Do you think the Complexity Tutor tutoring system helps you to develop the skills needed to construct proofs when you use the traditional pen and paper method for proof construction?

	Fall 2016	**Spring 2017**	**Fall 2017**	**Spring 2018**	**Total**
Responses	25	39	19	24	107
Yes	52%	44%	58%	63%	52%
Undecided	24%	18%	21%	17%	20%
No	24%	38%	21%	21%	28%

When asked if they would recommend Complexity Tutor to others learning proof construction, 83% of the subjects from Spring 2018 said they would, in spite of the fact that Section 5.1.2 gives strong evidence that the system did not correct their misconceptions about NP-completeness.

5.3.1 Would the questionnaire responses reflect the sentiment of students who did not participate in the study?

Someone playing devil's advocate might ask if the percentage of positive responses to Complexity Tutor might be inflated because a significant number of students who had less than favorable views of Complexity Tutor dropped out of the study?

While this cannot be completely disproven, there is reason to believe it is not the case. Table 4.1 from Chapter 4 shows that the drop out rate was nearly identical for both the experimental and control groups in every semester of the study, except for Spring 2018 where there was a slightly higher drop out rate in the control group.

Table 5.11: Do you want the Complexity Tutor tutoring system to be available in other courses with proof construction?

	Fall 2016	**Spring 2017**	**Fall 2017**	**Spring 2018**	**Total**
Responses	25	39	19	24	107
Yes	60%	68%	42%	63%	60%
Undecided	20%	8%	37%	38%	23%
No	20%	24%	21%	0%	17%

Table 5.12: Would you recommend the Complexity Tutor tutoring system to others learning proof construction?

	Fall 2016	**Spring 2017**	**Fall 2017**	**Spring 2018**	**Total**
Responses	25	39	19	24	107
Yes	74%	65%	70%	83%	72%
Undecided	0%	0%	0%	0%	0%
No	26%	35%	30%	17%	28%

But if dissatisfaction with Complexity Tutor had hypothetically led a large number of students to drop out from the study, then one would expect to see a higher drop out rate in the experimental group than in the control group.

Why? Well assume that there hypothetically was a large population of students who would drop out from the study because they did not like Complexity Tutor. However, it is reasonable to also assume that the variable that determines if someone would drop out from the study because they did not like Complexity Tutor and the variable that determines if someone would drop out from the control group are independent from each other, so that would imply there must also be sizable population that would not drop out from the control group, but would drop out from the experimental group because they do not like Complexity Tutor. That population should with equal probability end up in either the experimental or control group. But since they only affect the experimental group, the drop out rate from the experimental group should be higher than for the control group, assuming there is no variable that exclusively affects the control group in equal proportion.

5.3.2 What students liked about Complexity Tutor

Many subjects from the study expressed enjoyment about using Complexity Tutor—it was as Subject 92176666 (Spring 2017) expressed, a "fun way to learn".

Subject 48215832 (Spring 2017) wrote:

> "It is way more interesting than traditional homework."

Subject 34821464 (Spring 2018) wrote:

> "It was a fun practice over the dry material. I enjoyed it."

Subject 40739006 (Spring 2017) found the practice problems difficult but still enjoyable. "It is painful and fun," they wrote, explaining in another comment that the problems took a long time to complete in Complexity Tutor.

Subject 33853006 (Spring 2018) felt that Complexity Tutor helped them improve their understanding of NP-completeness, a topic they struggled with. "It seems really cool, and it helped me learn a hard topic I was confused about," they wrote and added in another comment:

> "It is impossible to complete the proof without thinking about every single last detail, which really helped me learn about NP-problems in general, as I had a really hard time understanding them before."

Subject 332756656L (Fall 2016) wrote:

> "I think if I had more practice using this application, it would be beneficial to my proof construction skills."

Subject 77839300 (Spring 2018) had this perspective:

> "I think that Complexity Tutor does give extra aid to students looking to see a different perspective for the problems they are working on in class or if they just want help."

Subject 83619768 (Spring 2017) wrote:

> "It will incredibly help students get past the initial exposure effect."

Subjects also gave positive comments about the graphical format. One subject, 37435222 (Spring 2017) compared it favorably to narrative proofs:

> "I enjoyed how it was all mapped out and not just sentences."

Many expressed appreciation for being able to visualize proofs in Complexity Tutor, like subject 68671124 (Fall 2016), who wrote:

> "It was helpful to visualize the proof in a tree-like form."

Subject 94521523 (Spring 2018) had this comment:

> "Thinking of proofs as graphs of assertions is very powerful and instructive."

Of course, not everyone was a fan of the graphical format. Subject 28824303 (Spring 2017) wrote:

> "I think the software is great, but I didn't feel that visualizing my proof steps as a tree helped me a lot. The leaf nodes of the (what I have most recently derived) tree are much more important to the proof."

However, that subject then explained what that they did like was the non-linearity of proof construction in Complexity Tutor:

> "One thing that is useful is being able to do the forward backward proof method. It was very straightforward to do backward proofs. Another thing that I thought was useful was being able to solve the problem in chunks. E.g., I could solve two different parts of a problem then use the 'backward' proof method to link the two parts together."

Subject 786W (Fall 2016) felt similarly:

> "It does break problems into subproblems which are easier to tackle."

Subject 95873118 (Fall 2016) liked that they were given the assertions they needed to use, instead of having to write them:

> "Having the pieces already helps to understand what parts are required whereas with a pen and paper you are on your own."

Many subjects also commented on how the platform forced them to be meticulous, such as subject 37351646 (Spring 2017) who wrote:

> "It was good at making me remember the smaller steps. You aren't allowed to jump, you really have to justify everything."

5.3.3 Constructive criticism from students

The vast majority of criticism from subjects in the study was concerning issues they had with the interface—mainly, not having enough screen space or not having a good way to manage visual clutter.

Subject 88507453 (Fall 2017) wrote:

> "Using Complexity Tutor was very interesting and useful. It was a little bit painful trying to cramp everything in such a small space. Also, not being able to remove the currently useless 'circles' from the screen was kind of inconvenient since it was unnecessarily taking up the already limited space."

Subject 93699577 (Fall 2017) wrote:

> "I think it is a good system overall, but I think the proof window (where you drag and drop all the assertions and stuff) can quickly get graphically

cluttered, and it can get tedious to manage the window with all the different texts and arrows in it. If it were somehow easier to organize/navigate in this proof box, then I think that would be a big improvement for the usability of Complexity Tutor."

Subject 44735408 (Fall 2017) suggested adding a zoom feature:

> "With larger proofs, scrolling became tiresome and perhaps adding a zoom feature rather than page scrolling would be very helpful."

Subject 32914951 (Spring 2017) had many thoughts about the interface:

> "I know I'm probably stating the obvious but there needs to be an option to remove assertions/premises from the proof space to avoid clutter. In addition I would suggest giving each assertion and/or premise its own text box or text bubble to further distinguish each one. There should be an option to label a statement as the conclusion. Like in photo editing software, there should be a select tool for selecting multiple statements and moving them. The proof space should scale with the width of the monitor when the window is maximized. It's rather awkward when everything is left justified.
>
> To some I would recommend it, particularly visual learners. The interface needs some tweaking to be useful, however, there was not enough freedom of choice on the part of the user. The advantage to the software is to lay out all the possibilities for the user to teach them which to choose and train how to read the notations. Need more space to read, should expand to full screen view and have boxes manually sizable by the user to help them view what they need to see with clarity. Good idea, with improvement could help many people in many disciplines."

A number of subjects also commented that they would like the ability to save their progress, such as Subject 40470021 (Fall 2016):

> "Saved progress would be great. And an easier way to undo actions."

Subject 32375112 (Spring 2018) wondered if the assertions being pre-written reduced their retention of what they learned:

> "The system is interesting but it's hard for me to say how much I retained really about the actual logic behind proving things since so many of the statements are largely pre-formulated—it was often more of just symbolic manipulation than actually understanding things like certifier functions."

Subject 16832623 (Fall 2016), when asked if the system helped them be meticulous, responded:

> "Yes, although it involved many extra steps to the proof that we did not cover in class which led to confusion."

5.3.4 Was the hint-line feature beneficial to students or not?

After the experiments from Fall 2016 and Spring 2017 had been completed, it was observed that some subjects wrote that they wished Complexity Tutor had provided more hints, when they got stuck.

Subject 49055538 (Fall 2016) wrote:

> "If there is a hint while I am stuck in a problem, it will be great. Otherwise I will just give up after many tries."

Subject 88229136 (Spring 2017) wrote:

> "If I got stuck and couldn't progress, well, too bad. You need to include a hint system or something."

However, after the hint-line feature from Section 3.1.7 was introduced, a number of subjects expressed concern that it was hindering their learning. Subject 94521523 (Spring 2018) wrote:

> "It is too easy to be guided by the hints and try even just one or two guesses, and have them immediately confirmed; I didn't feel forced to be careful."

Subject 35917331 (Spring 2018) wrote, "I would like to see an option to turn off hints." In another comment, they further elaborated:

> "The tutor gives you hints about how to go about the proof. Although I could see the hints being helpful if I were completely stuck, I often got hints without wanting any. This allowed me to do some of the proofs using the hints without really understanding what I was doing."

Both semesters where the hint-line feature was used had somewhat disappointing results. In Fall 2017, there was no association found between using Complexity Tutor and exam improvement, even though an association had been found the previous semester. In Spring 2018, subjects who used Complexity Tutor did not do well on a posttest evaluation. Is it possible that the hint-line feature is either partially or fully to blame for those results?

The above subjects' suggestions that the hint-line feature was helping them so much that they no longer had to think critically about the problem is a concern. While the hint-line feature may still be beneficial to some students, it may also need more tweaking to provide the right level of assistance. Developing a student model would also help ensure that hints are given to the students who will benefit from them the most.

5.4 Conclusion of study results

Many subjects enjoyed using COMPLEXITY TUTOR and saw much potential for it as a learning aid, although some had frustrations with the interface. The results from the discrete mathematics course experiments also show promise that constructing proofs in the Theorem Proving Environment may be a better way to practice developing skills in theorem proving than normal pen-and-paper construction, but a more comprehensive study is needed to give confirmation to this finding.

Programming in Python was not a significant hardship for students using the Algorithm Environment to do NP-completeness reductions, which indicates that it could potentially be a pedagogically suitable replacement for pseudocode in other types of theoretical problems that require algorithms. There was significant improvement on the NP-completeness reduction problems, once subjects were given hints to help them understand basic problem constraints. This led the experimental group for Spring 2018 to perform better than the control group did on the BIN-PACKING Reduction Problem, and it is conjectured that if the class was taught Levin reductions, the gap between the experimental group and the control group would grow even further. It is also conjectured that one of the main reasons that students do not succeed in abstract domains like NP-completeness is because they have trouble understanding the problems they are asked to solve.

A key consideration for experimental design that was highlighted by Section 5.1.3 is the importance of problem selection. One problem from the prepositional logic experiments correlated much more strongly with performance on the exam problems than any other. However, there is not yet a well-developed theory to predict for any two proof problems, how a student's ability to solve one will correlate with their ability to solve the other. Building a theory to explain this would be an interesting area for future research, and likely necessary to build a good intelligent tutoring

system. The author's ideas for "proof complexity" explained in Section 5.1.1 may be a starting-point for such a theory.

This study pointed out a potentially significant limitation of the Theorem Proving Environment, which is that it may not do enough to illuminate student misconceptions. However, to put this in perspective, it is doubtful than any of the systems or approaches covered in Chapter 2 would do any better than Complexity Tutor at resolving this problem.

Those systems assume that students will be trained to use a formal logical system, and that they will use it consistently throughout the course they are taking. To compare Complexity Tutor fairly against those systems, it would need to be used consistently in place of narrative proofs while teaching a course. Formal logical systems prevent the errors illustrated in Section 5.1.2 from occurring in the first place, but they cannot ensure that the student using them will not still have misconceptions about theoretical ideas somewhere in their mind.

In this regard, informal narrative has a pedagogical advantage over formal logical systems, because it allows students to express ideas close to as freely as their mind can conceive of them, illuminating misconceptions that a human grader can identify and attempt to correct. But even narrative has its limitations, as illustrated by Section 5.1.2, where there was ambiguity about what students understood based on what they wrote on the posttest quiz. Thus, human dialogue is even more effective at exposing misconceptions than narrative. Until machines become as intelligent as humans, no computerized system could ever completely replace Socrates or even Sigmund Freud, both who developed techniques of using human dialogue to reveal secrets buried in the mind.

Complexity Tutor was designed to bridge the gap between formal logical systems and the informal narrative arguments that have been used by mathematicians to train their pupils for millennia. Humans can be trained to reason in a formal log-

ical system, but there is evidence (Section 2.3) that it is not natural for them, which would explain why errors occur when humans reason in informal narrative that would not occur in a formal logical system.

Complexity Tutor gets closer to informal narrative than any other system that the author is aware of has, but there is still a gap.

Nevertheless, the author has ideas for ways to address Complexity Tutor's current shortcomings when it comes to identifying and correcting student misconceptions. Those ideas are presented in the next chapter on future work. Without further ado, please proceed to the next chapter.

CHAPTER 6

FUTURE WORK

The roadmap for COMPLEXITY TUTOR becoming an intelligent tutoring system that can adapt to the individual needs of students in their theoretical computer science classes is as follows. An accurate *student model* needs to be produced. A provisional one is suggested in Section 6.6. However, important questions came up from the findings of the empirical study in this dissertation, and the answers to these questions might alter the proposed student model:

- *In the practice of proof construction, what determines if successfully completing one set of problems will lead to successfully completing a different set of problems?*
- *Why did students retain "bugs" after using* COMPLEXITY TUTOR*?*
- *Is practice in* COMPLEXITY TUTOR *comparable to practicing proofs on pen and paper when the student does not have "bugs"?*
- *What factors determine if a student will successfully complete a proof?*

So, more experimental studies are needed to answer these questions, and that should be the first order of business. That will lead to refining and expanding the student model of Section 6.6 as appropriate.

Once a general student model is formed, the plan is to parameterize it using maching learning techniques, as was done for Deep Thought (Section 2.4.6). However, that will require a lot of data.

There are two dimensions to the required data. First, a large library of scaffolded problems that students can work on in a given domain is required. Second, data is required of a large number of students working on those problems over a lengthier period of time than was looked at in this dissertation's study.

The researchers behind Deep Thought had this data. Before they turned Deep Thought into an intelligent tutoring system, they already had a large library of scaffolded problems and many semesters of data of students working on these problems. However, it was not so difficult for them to produce the library of scaffolded problems to begin with, because developing new problems in formal logic is considerably less laborious than developing new problems for Complexity Tutor.

Hence, one of the priorities going forward will be to investigate methods for automatically or semi-automatically generating new problems for Complexity Tutor. This can happen in parallel to when the experimental studies are taking place to answer the questions mentioned at the beginning of this section. Having methods to automatically generate problems for Complexity Tutor would not only speed-up the development of a comprehensive scaffolded library of problems for one domain, but allow Complexity Tutor to be rapidly adapted to new domains.

Thus, researching the automation of problem generation is the most efficient way forward, and also of particular intellectual interest to the author.

Finally, along the way, Complexity Tutor should be updated with new features to address some of its shortcomings and make it more versatile. However, the roll-out of these new features in experimental studies should be slow so it is easy to make comparisons to previous studies and isolate variables of interest. One of the main lessons that the author learned from the work in the current study is that having too many variables change between experiments creates a lot of uncertainty when empirical results are analyzed.

The remainder of this chapter will address an assortment of ideas for improving and expanding upon Complexity Tutor, a provisional student model that might be used for intelligent tutoring, and ideas for the automated problem generation mentioned above.

6.1 Correcting student misconceptions

It was disappointing that so many subjects from the NP-completeness experiment in Spring 2018 remained confused about major concepts in NP-completeness, even after receiving practice with Complexity Tutor.

One possible hypothesis for why the Theorem Proving Environment may not have corrected the students' misconceptions is that the "immediate feedback" provided may have been too immediate, and as such, students did not have enough time to internally process their mistakes.

Furthermore, the feedback given to students has a positive bias—students are rewarded with *arrows* and *complete dots* for correct actions, but the negative feedback given is much more subtle, since the only negative feedback is the absence of a reward.

In consideration for this being the possible reason for why students did not have their misconceptions corrected, four new modes of interaction are proposed to explicitly draw students' attention to their misconceptions. These modes are *Delayed Feedback Mode*, *Debug Mode*, *Find-the-Bug Mode* and *Freestyle Mode*.

6.1.1 Delayed Feedback Mode

In Chapter 1, it was argued that computer science students should get immediate feedback on their theory problem sets like they get from their compilers for programming assignments. However, the Theorem Proving Environment, in its current form, actually delivers feedback that is more immediate than even a compiler would give.

After all, when students are writing code, they generally do not receive immediate feedback after every line of code they type. Instead, they finish their code and then give it to the compiler.

Delayed Feedback Mode would make the Theorem Proving Environment more like a compiler, where students would attempt to complete a construction of a proof before receiving any feedback from the compiler.

Students would still be given assumptions and assertions to use in their proof, but the status of assertions would not initially be displayed when dragged into the Proof Space. Furthermore, students would have the freedom to connect arrows from any assumption or assertion in the Proof Space to any other assumption or assertion.

Once the student believed that they had a correct proof, they could click a “Check Proof” button, which would evaluate their proof. If the student got the proof completely correct, they would be notified of such. Otherwise, the student would receive one of two kinds of feedback—Minimal Feedback or a Graphical Error Report. These two kinds of feedback are listed below.

6.1.1.1 Minimal Feedback

Minimal Feedback tells the student that their proof is incorrect and gives them some general descriptive and statistical information about what is wrong with the proof. Examples:

“Your proof has 5 arrows that are incorrect.”

“Your proof has 2 assertions that are incorrect.”

“The granularity of your proof is too low.”

However, Minimal Feedback does not inform the student of exactly what inferences are incorrect, but rather just gives a small hint, giving the student the opportunity to figure out the mistakes on their own.

Minimal Feedback may be appropriate for a student if they have put little effort into constructing their proof before clicking the "Check Proofs" button, and thus are nowhere near close to a complete proof.

Minimal Feedback may also be appropriate for a student who is very close to a correct proof, and will likely figure out the mistakes on their own if given the opportunity to do so.

6.1.1.2 Graphical Error Report

Mistakes in the student's proof space are identified and highlighted as follows:

- For all assertions A and B, if the student has connected an arrow from A to B but there is no path from A to B in the proof graph, then the arrow is part of an incorrect inference. The arrow's color is changed from blue to red to denote that it is erroneous.

- For all assertions A and B, if the student has connected an arrow from A to B and there is a path from A to B in the proof graph, but the length of the path is greater than some threshold (set to 1 by default), then the arrow is part of an inference that is too coarse in granularity. So replace it with a hint-line.

- For all assertions in the Proof Space, if an assertion is erroneous, give it a hashed dot. Then, for all arrows extending from it, change the color from blue to red to denote that they represent incorrect inferences.

- For all assertions in the Proof Space, if an assertion is correctly justified, then assign it a complete dot.

- For all remaining assertions in the Proof Space that have not been assigned a dot yet, assign a partial dot.

After receiving the Graphical Error Report, the student is placed in Debug Mode to correct their mistakes.

6.1.2 Debug Mode

In Debug Mode, a student corrects a Graphical Error Report. They first remove all of the incorrect inferences. From there, Debug Mode behaves the same as Delayed Feedback Mode, letting the student attempt to fix the proof. When they think they have corrected it, they can click, “Check Proof”. The feedback options that result from this are the same as for Delayed Feedback Mode.

Note that Graphical Error Reports do not have to be generated from a mistake the student made. Students can be given Graphical Error Reports of common “bugs” to correct (possibly mistakes that many other students made).

6.1.3 Find-the-Bug Mode

This mode can be thought of as the inverse of the Debug Mode. Instead of being given mistakes to correct in a Graphical Error Report, the student is given a proof attempt, and they must construct their own Graphical Error Report to identify all the mistakes.

6.1.4 Freestyle Mode

In Freestyle Mode, students are not given any assertions but write their own and connect them together with arrows. The purpose of this mode is to identify misconceptions students have that would otherwise not be identified, but could be identified in narrative proof.

Obviously, in this mode, Complexity Tutor cannot provide direct feedback, since its knowledge model does not cover arbitrary assertions. So an expert like the instructor would need to attempt to debug the proof. Once an expert does that, any “bugs” that were discovered could be automatically added to a “bug library” for use in the other modes.

Alternatively, instead of having an expert evaluate the Freestyle Mode proof attempts, those attempts could be crowd-sourced to other students to correct in Find-the-Bug Mode.

6.2 Developing a graphical interface that can represent a wide range of proof strategies

Frederic Fitch developed a notation [54], referred to as the *Fitch-style diagram* or *Fitch proof*, which has become popular in teaching formal logic. The main benefit of this notation is that it provides a way to express *subproofs* within a proof. This is important because common theorem proving strategies such as *proof by cases*, *proof by induction* and *proof by contradiction* all implicitly use the notion of a subproof. Fitch-style diagrams are used in some of the systems mentioned in Chapter 2, such as the EPGY Theorem Proving Environment (Section 2.4.2) and AProS (Section 2.4.4).

Pedagogically, Fitch-style diagrams are advantageous since they provide a framework for reducing a theorem proving goal to a hierarchy of subgoals. Structuring problems in terms of their subgoals as a general strategy has been demonstrated to help students adapt what they learn from a given problem solution to new problems [39]. This principle has already been shown successful in areas of computer science education [111].

Figure 6.1 shows an example of a Fitch-style diagram proving $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap B)$. The vertical bars in the Fitch-style diagram are referred to as *scope lines*, which along with indentation indicate where subproofs begin and end. Each subproof has its own *scope* with assumptions that only exist while the subproof is active.

In the graphical setting, a natural analogue for the scope lines is to draw boxes around subproofs. Figure 6.2 shows the Fitch-style proof from Figure 6.1 converted to a graphical format. It should be noted, for historical interest, that prior to the

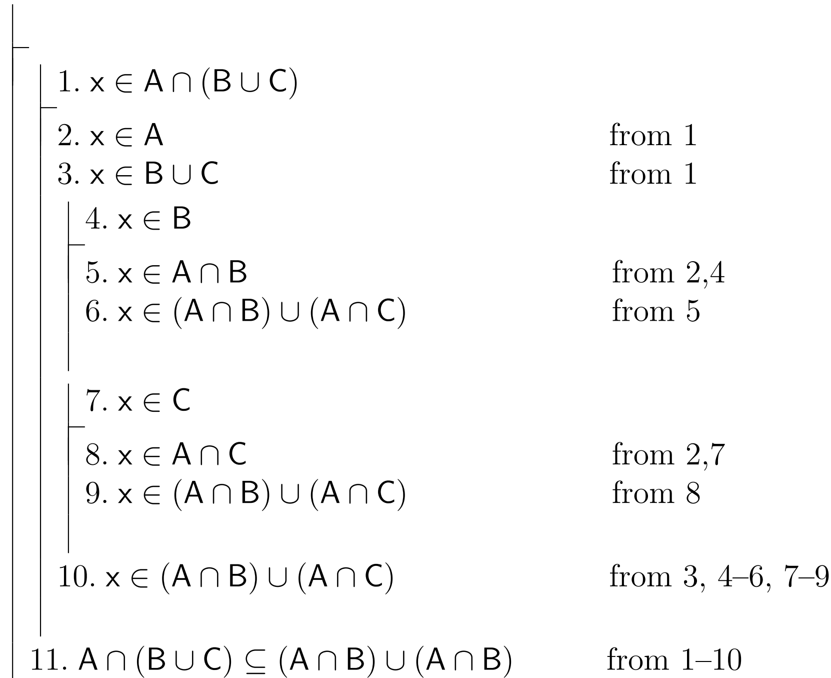


Figure 6.1: Fitch-style diagram.

popularization of Fitch-style diagrams for teaching formal logic, Stanisław Jaśkowski introduced [77] a similar convention of drawing boxes around subproofs. Stanisław Jaśkowski was also one of the main inventors of natural deduction logic. Fitch's notation became more popular than Jaśkowski's because it was easier to typeset with the technology of the time.

6.2.1 Interface challenges with using subproof boxes in the Proof Space

While the presentation shown in Figure 6.2 seems like a good solution for presenting Fitch-style proofs using the graphical idiom, this solution creates new interface design problems that must be resolved before being adapted to Complexity Tutor.

Consider that when a student is constructing a proof, they do not know what it will look like ahead of time. The biggest problem with introducing boxes for subproofs in the Proof Space, is that the size a box needs to be cannot be known by the student in advance. So either Complexity Tutor would have to decide how big the box

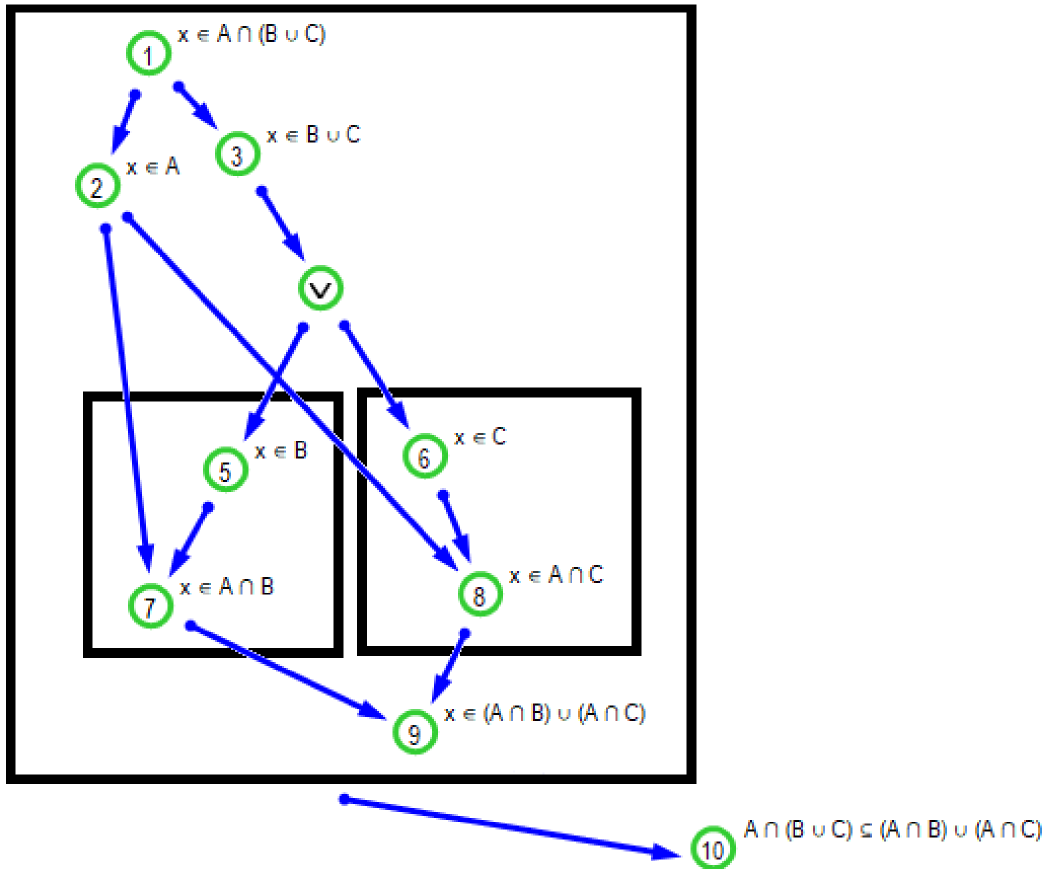


Figure 6.2: Graphical version of a Fitch-style diagram. Boxes indicate subproofs with their own scope. The dot with the disjunction symbol is used to indicate a split between the cases $x \in B$ and $x \in C$.

should be, or the student will have to be given some easy method for resizing the box on the demand.

If Complexity Tutor were to decide the size of the box in advance, based on knowledge of what is required for a particular subproof, then that perhaps unintentionally gives the student a hint about the subproof, since the student then knows it must fit within the box. The student's ability to explore different possibilities when constructing the subproof would also be limited, since they only have a fixed amount of physical space to work with.

On the other hand, if the box were to be resizable, the biggest problem would be re-arranging all the proof items outside of the box to make room for a larger box. If

the student were required to do this manually, it would be very tedious. Instead, it would be better if resizing the box automatically moved the other items in the Proof Space to make room for it. For instance, increasing the box 50 pixels in width might shift all items to the right of it by 50 pixels. The more complicated issue to deal with is what should be done if the box is reduced in size. For instance, a student might create a large box for their subproof and then later realize that they don't need all that space. When the box is reduced in size, the items around it cannot simply be automatically shifted inward, as that might cause them to overlap with each other. The alternative is to not automatically move any items outside of the box when the student reduces the box's size, but then the student will likely feel the need to move the items themselves to reclaim space, which again could be quite tedious.

This problem could be mitigated by discouraging students from creating boxes for their subproofs that are too large to begin with. For instance, perhaps, instead of giving the student the ability to resize the subproof box as they please, the box is only increased in size incrementally as the student adds connections inside the subproof.

Yet another idea would be to give the subproof boxes scroll bars like the Proof Space itself. While there is a certain conceptual elegance to this idea, since each "subproof space" would then behave much the same as the main Proof Space, the end result would be reminiscent of the *inline frame* (i.e., IFRAME) elements used in some websites from the 1990's and would be visually undesirable.

Beyond the issue of resizing subproof boxes, there must be no confusion over which proof items belong to the subproof and which do not. It should be assumed that any proof items that fully overlap the box belong to the subproof, and any proof items that are outside of the box do not. Since a proof item that partially overlaps the box and is partially outside of it would be ambigious, the interface must not permit this. Thus, the interface must be modified so that when proof items are being dragged in

the Proof Space, if they land at the border of a subproof box, they either "snap" to the inside of the box or to the outside.

On the other hand, if the student wants to move the subproof box itself in the Proof Space, this is even more problematic. The student should not be permitted to move the box in such a way that it would overlap with any other items in the Proof Space. This means that the student will either need to manually move other proof items to make space for the box in its new location, or there will need to be an intuitive method for automatically pushing the other items out of the way as the box is moved inside of the Proof Space.

6.2.2 Alternative possible subproof representations

It is clear that using boxes to represent subproofs will be encumbered by a number of interface issues. Here are two alternatives, which would be relatively easy to implement within Complexity Tutor's existing interface:

6.2.2.1 Labeling assertions to denote the subproof they belong to

Figure 6.3 shows an alternative to Figure 6.2, where instead of using boxes, each assertion is given a different type of label to classify the subproof it belongs to. Gerhard Gentzen used a similar idea for his natural deduction proof trees [60], where a unique identifier label was given to each node that represented a new assumption in the tree, while other nodes would use this same identifier to indicate when a particular assumption had been discharged, thus ending its scope. The downside of these types of schemes is that they don't make subproofs as visually obvious as the subproof boxes.

6.2.2.2 Using a separate Proof Space for each subproof

Complexity Tutor already uses a tabbed interface to switch between the Theorem Proving Environment and the Algorithm Environment. Each subproof could

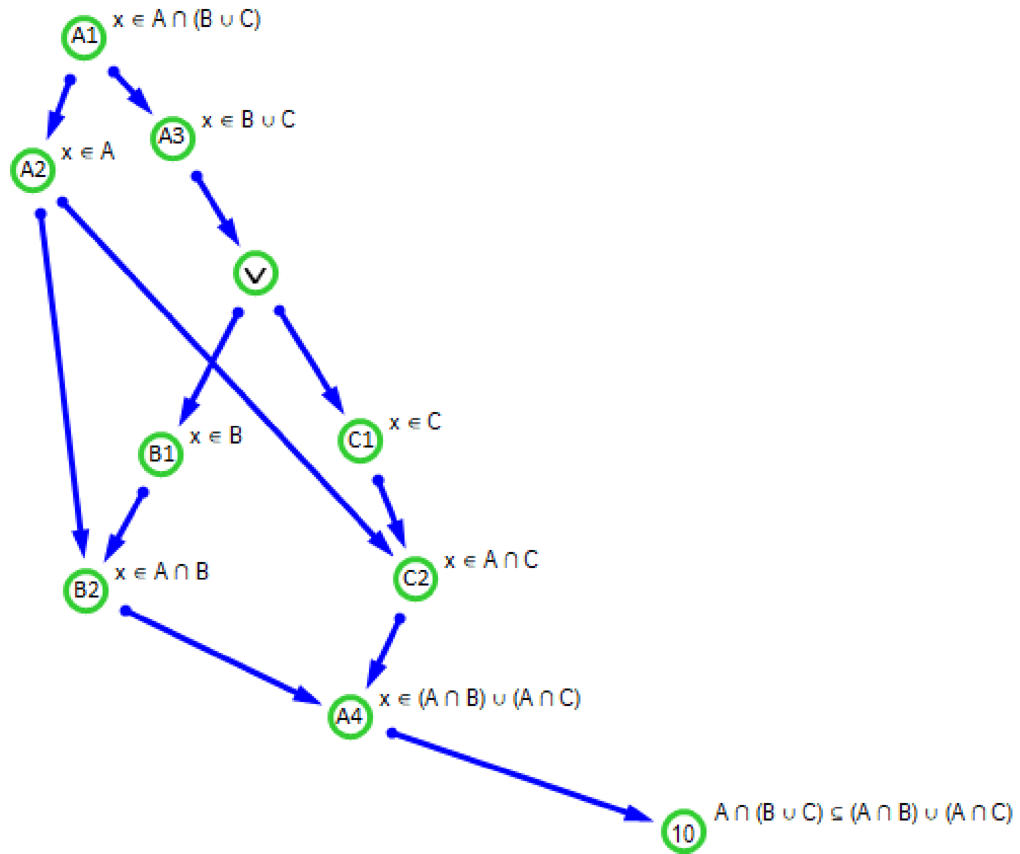


Figure 6.3: Alternative graphical representation of subproofs. The dots for each proof statement are labeled according to the subproof they belong to. In this example, there are three subproofs. Statements that belong to the outer subproof get 'A' labels. Statements that belong to the two inner subproofs get 'B' and 'C' labels respectively. The dot '10' is outside all the subproofs.

instantiate a new tab within COMPLEXITY TUTOR with its own separate Proof Space. The downside of this idea is that some subproofs, like the ones shown in Figure 6.2, are very simple and using a whole Proof Space to represent a very simple proof seems wasteful. Furthermore, it is beneficial to the student to be able to visualize the whole proof at once when completed, and this is not possible with this approach.

6.2.3 How proof strategies would be applied in the Theorem Proving Environment

Once an interface for subproofs has been implemented in the Theorem Proving Environment, integrating proof strategies that involve subproofs would be relatively straightforward. The proposed way to do this would be to have a context menu that appears when a student right-clicks a specific proof statement in the Proof Space. The context menu would show specific proof strategies that the student could apply to that specific proof statement.

For instance, in the example shown in Figure 6.2, a student would right-click on the "$x \in B \cup C$" assertion and be shown a context-menu with the option "Prove by cases". If they then selected this option, the node with the disjunction symbol would automatically be added to the Proof Space and connected from the "$x \in B \cup C$" assertion. Two new subproofs would be instantiated, one containing the assumption "$x \in B$" and the other containing the assumption "$x \in C$". When both subproofs had been completed, the student would then be able to justify "$x \in (A \cap B) \cup (A \cap C)$".

Note that the proof by cases strategy is applied to a premise ("$x \in B \cup C$") rather than its conclusion ("$x \in (A \cap B) \cup (A \cap C)$"). The strategy is open-ended because the student may not know exactly what they will conclude after breaking a particular assertion into cases. Most other proof strategies, such as proof by induction and proof by contradiction would be directly applied to a conclusion rather than a premise.

Some proof strategies may open up other modules in Complexity Tutor, such as the Algorithms Environment. Rather than having the Algorithms Environment immediately available at the beginning of a problem, it could be hidden until a "proof by algorithms" strategy is applied to an appropriate assertion.

This would also provide a general mechanism for incorporating new modules in Complexity Tutor. For instance, a computer algebra module which allows stu-

dents to manipulate equations, would be a useful addition to Complexity Tutor for doing proofs in subjects that involve equation manipulation.

6.3 Other general interface issues

In general, the more proof items that are in the Proof Space, the harder it becomes to arrange them in an orderly manner. There were comments (Section 5.3) about the Proof Space becoming cluttered, and it being cumbersome to move items around in the Proof Space.

One potential solution to this hindrance would be to use an automatic graph layout algorithm [23] to manage the Proof Space for the student. This would have the additional benefit of also solving all the layout problems mentioned in Section 6.2.1 concerning user-controlled layout of subproof boxes.

However, there are some potential downsides to consider with automatic layout. One downside is that if an algorithm is constantly updating the layout of the Proof Space, this might hinder a student's spatial memory—an assertion suddenly jumps from one location in the Proof Space to another, and then the user can no longer quickly find it. Therefore, automatic layout algorithms that are considered should have the property of making minimal modifications to an existing spatial arrangement when updates occur.

Also, the author hypothesizes that giving users the ability to spatially arrange items in the Proof Space as they see fit may help the process by which they cognitively structure information about the proof. In future studies, this hypothesis could be tested to see if students improve their learning more when the Proof Space has automatic layout or when they have the freedom of manual arrangement.

6.4 Inductive Example Construction Aid

The Inductive Example Construction Aid would be an additional module for helping students with theoretical computer science problems that involve *formal languages* as entities in the problem, such as NP-completeness reduction problems.

The analysis of the NP-completeness experiments (Section 5.2.1.4) shows that subjects likely were not able to succeed at producing a correct reduction because they did not understand the problem constraints. In other words, they did not have a good understanding of the languages of BIN-PACKING or PARTITION.

The Inductive Example Construction Aid is designed to help in such a situation, and is motivated by the pedagogical meta-skill introduced in Section 1.2.5 where students generate concrete examples of abstract languages. It uses Socratic dialog as the means to motivate students to develop this meta-skill.

For instance, in the example given in Section 1.2.5, where the student first tries to create an example of PARTITION, imagine this dialog between the student and the tutor taking place:

TUTOR: What is an example of PARTITION?

STUDENT: $\{5, 1\}$?

TUTOR: That's not correct. Please try again.

STUDENT: $\{1, 2, 2, 3\}$?

TUTOR: Yes, that $\{1, 2, 2, 3\}$ is an example of partition because if $X_1 = \{1, 3\}$ and $X_2 = \{2, 2\}$ then $X = \{1, 2, 2, 3\}$. Can you think of another example of PARTITION?

If however, the student did not figure out an example of PARTITION so quickly, the tutor might list some examples. An alternate dialog might take place:

TUTOR: What is an example of PARTITION?

STUDENT: $5 + 5$?

TUTOR: No, an example of PARTITION must be a set of non-negative integers.

Please try again.

STUDENT: $\{1, 0\}$?

TUTOR: That's not correct. Please try again.

STUDENT: $\{-1, 2\}$?

TUTOR: No, here are some examples of PARTITION: $\{1, 1\}$, $\{2, 2\}$, $\{1, 2, 3\}$, $\{2, 3, 5\}$, $\{1, 1, 2, 0\}$, $\{2, 4, 1, 5\}$. Can you give any other examples?

In the example, after the tutor is convinced the student has a good understanding of what PARTITION is, it will then ask the student to come up with examples of KNAPSACK.

Note that these dialogues also address the objective of Section 1.2.3, since showing examples is one way for the student to learn the notation used in the problem descriptions. A student who is not comfortable with the notation will not be able to produce examples, or they will produce some incorrect examples.

6.5 Hypothesis about problem description types

The author conjectures that when computer science students are given problems to solve in their theory classes, the way the problem is described will be a significant variable in determining if they are able to correctly solve it.

This conjecture comes from the arguments made in Sections 1.2.2 and 1.2.3, which together imply that for people who have little exposure to abstract mathematics, the initial barrier they face is mostly a language comprehension issue.

Further evidence to support the conjecture comes from the fact that subjects in the NP-completeness experiments seemed to struggle to understand the BIN-PACKING Reduction Problem. Hypothetically, if the problem had been described in a different way, they might have understood it better.

Future experiments will test if how students perform on a theoretical problem such as an NP-completeness reduction changes when the description type changes. If the test confirms the author's hypothesis, then a future research track would be to look at if language acquisition strategies can be effectively applied to the domain of theoretical computer science and mathematics. Three different problem description types will be considered:

1. *Formal set-builder notation*–This is a formal mathematical notation that is a very compact way to describe problems. It is used often in textbooks and sometimes homework assignments, but instructors almost never use it to teach their lecture materials. An example of this description type is the 0/1-PROG Reduction Problem from Appendix B.

2. *English vernacular*—This kind of description uses idioms that are common in computer science. Instructors usually use a similar type of description when they teach, probably because they think it makes most sense to explain the material in terms of these idioms. An example of this description type is the BIN-PACKING Reduction Problem from Appendix B.

3. *Abstract word problem* It is called "abstract" but in another sense one might consider it "concrete", because rather than give a general description of a mathematical problem, a specific concrete example is given. The reason then for calling it an "abstract word problem" is that the student is required to take a very concrete situation and abstract away the important details to put it in the context necessary to solve the problem, e.g., complexity theory. It is a test to see if the student really understands the fundamental underlying problem. An example of this description type is Question 4 from Appendix E.

6.6 Using machine learning to develop a student model

Once a sufficient amount of data on students' interactions with COMPLEXITY TUTOR is collected, machine learning can be used to build a student model, which can be used to customize the problem selection for each student. Additionally, having a well developed student model would allow COMPLEXITY TUTOR to adjust problem parameters and guidance offered to fit the individual needs of each student, giving them a personalized learning experience. This objective is similar to what Deep Thought (Section 2.4.6) accomplished for the simpler learning space of formal logic.

The student model has a large number of hidden and observed states. Machine learning is applied to update the hidden states from the observed states. The following is a non-exhaustive list of state variables that might be included in the student model:

1. For a *given problem*, what is the likelihood the student will be able to solve it correctly before giving up?

2. For a *given type of problem goal*, what level of proficiency does the student have at solving it?

3. For a *given problem description type* (from Section 6.5), what level of proficiency does the student have in understanding problems with that description type?

4. For a *given assumption* or *partial reduction algorithm available to the student*, what is the likelihood that given an arbitrary problem that is easier to solve using the assumption (as dictated by model proofs), the student will be able to recall and correctly apply the assumption?

5. For a *given subset of objects* in a *given problem*, what is the likelihood that seeing examples of each of the objects will significantly help the student solve the problem? This variable could predict when the Inductive Instruction Aid mentioned in Section 6.4 would be helpful.

6. For a *given inference concept* used implicitly in a *given step in a proof*, what is the likelihood that the student understands that this concept is being used even though the granularity is too low to force the student to explicitly make that inference?

The first five of these variables give us criteria for evaluating a student's overall ability, as well as giving us control variables for selecting what problem to give to the student next. Notice that variables 3–5 essentially quantify most of the meta-skills listed in Section 1.2, with the exception of applying proof schema, which should be somewhat correlated with variable 2. Variable 6, on the other hand, is used to adjust proof granularity for each student.

The most significant feedback that can be used to update these variables is whether or not the student solves a particular problem correctly without giving up. If the student does give up, then that means the previous assessment of variable 1 was wrong, and it should be updated to have a value of 0% likelihood, and machine learning can be used to propagate the effect of changing that variable to all the other variables.

If on the other hand, the student does correctly solve the problem, then increase the value of variable 1 for that problem.

There are of course other observations collected by the student model other than whether a student solves a problem or not.

For instance, one can look at how many cumulative errors the student made before they arrived at the correct solution, or how many they made before they gave up. If they made a lot of errors, that might explain that they were not that comfortable solving the problem.

But it might additionally explain that they were persistent in trying to solve the problem. It is probably important to treat students with different levels of persistence differently, especially when deciding what difficulty level the problem you give them

next should be. The amount of typing they did might also tell something about their persistence, or whether they chose to use the Inductive Example Construction Aid (Section 6.4).

Also, how did the number of errors change over time? Did the student start out making a lot of errors everywhere, or was there only one step in the proof they had significant trouble figuring out?

It is also important to identify specific errors that students make over and over again in their proofs throughout the tutoring session, because these might be "bugs" that need to be corrected. Thus, there is this important variable:

7. For a *given error that a student made*, what is the likelihood it is a "bug"?

Of course, "bugs" are not always observed, so it is a good idea to consider this hidden variable:

8. What is the likelihood that a student has a *given "bug"*?

And if the student could not solve a problem, what did they do correct? Did they state the correct goals? Did they use the right assumptions?

To be able to assess the likelihood of whether a student would be able to solve a particular problem, there are other variables that one might want to infer from:

9. Does the student know what goals to solve for *a given problem*?

10. For each of those *given goals*, is the student proficient in applying the associated proof schemas (Section 1.2.4)?

11. For each *entity in the problem* (e.g., a "language" entity for instance), has the student successfully solved other problems that require them to use that entity?

6.7 Automation of problem generation

Even though the Theorem Proving Environment in Complexity Tutor is a domain-independent framework for proof construction, it still requires a domain expert to manually author every single proof problem for a given domain.

It would be nice if there was a tool that could take a set of Complexity Tutor problems that have already been created for a given domain, and from that set of problems automatically generate new problems.

This task can be broken into two separate research challenges, each which is interesting in its own right:

Challenge 1

Given a formal language, can a new problem and its solution be generated?

Challenge 2

Given a set of problems, can the inference rules of a formal language that will produce it be generated? This is the inverse of the Challenge 1 problem.

Consider Challenge 2 first. Recall that EXCHECK from Section 2.4.1 had 700 context-free grammar rules to support a controlled natural language based formal logical system. The goal would be to automatically infer grammar rules like these from proof graphs, so there would be no need to manually construct a grammar with 700 rules. This is essentially a machine learning problem, referred to as *grammar induction*. Considering that grammar induction in other areas like programming languages [92,149] and natural languages sentences [69] have been successful, the author is optimistic that grammar induction on the significantly more structured domain of inference relations in Complexity Tutor proof graphs would not be overwhelming, and would produce positive results of well-specified inference rule grammars.

For Challenge 1, assume that a set of assumptions have already been specified in the formal language that was produced from Challenge 2. Then there are two

versions of Challenge 1, a hard version and an easy version. The hard version is when the goal has been pre-determined along with the assumptions. This is hard because it reduces to automated proof search in an arbitrary formal logical system with arbitrary inference rules.

Without having intelligent heuristics for searching for proofs in that particular proof space, one is unlikely to be able to produce a proof of the goal. The closest work that the author is aware of that comes anywhere close to addressing the hard version of Challenge 1 is a strategy for searching a restricted space of propositional calculus formulas with arbitrary inference rules [2]. That work uses a representation of formulas encoded by their truth table semantics to reduce the search space for problems with a limited number of variables.

That said, the easy version of Challenge 1 where the goal is not constrained seems to be much easier to accomplish. A random walk through the proof search space will result in an arbitrary goal that is generated and an arbitrary proof graph that is explored. This will produce new problems, although they may not be of desirable quality. To construct good quality problems, constraints must be placed on the random walk through the proof search space. For instance, inference rules that would produce assertions over a certain string length should probably be avoided. Furthermore, properties of other existing proof graphs for problems that are already considered to be good quality should be analyzed to construct a model that produces similar problems, using similar inference applications, etc.

6.8 Going beyond Levin reductions

When Complexity Tutor was first conceived, the intention was to build a tutoring system that would help students learn any topic from *complexity theory*, and that would have methods for automatically validating the correctness of any kind of algorithmic reduction students would be exposed to when learning about

computational complexity. This is after all where the name Complexity Tutor comes from. But this turned out to be a very ambitious goal.

The method for validating reductions that is described in Section 3.2.2.1 can only be applied to *polynomial-time reductions* between languages that belong to the complexity class NP. In fact, it only works specifically for *Levin reductions*. This provides a way to tutor NP-completeness, which is one of the most significant topics from computational complexity, and the one that is most commonly taught to undergraduate computer science students.

At the University of Massachusetts Amherst, students are required to take an algorithms course that covers the topic of NP-completeness, but they are not required to take courses that would teach them about other complexity classes.

So how could Complexity Tutor be adapted to helping students learn to do reductions involving other complexity classes? Restricting the programming model that students are allowed to use in the Algorithm Environment would potentially permit the representation of finer-grained reductions, such as *log-space reductions*, allowing Complexity Tutor to give problems involving smaller complexity classes than NP. However, automatically evaluating reductions involving larger complexity classes such as PSPACE or the class of all *computable languages* would require a very different approach, since it is not feasible to "run" those reductions.

Perhaps, at that point, it might make more sense to use a puzzle-like framework similar to Parsons Problems [122], in place of the existing framework of allowing students to freely write code for their reductions. Future research could compare using a Parsons Problem type framework to the existing Algorithm Environment to see which helps students more with learning reductions.

APPENDIX A

MATERIALS USED TO PRESENT AND DIRECT EMPIRICAL STUDY

A.1 Script announcing study

I am Mark McCartin-Lim, and I am a computer science Ph.D. student at the University of Massachussetts at Amherst.

As part of my dissertation research, I am developing a tutoring system that is designed to help students in theoretical computer science classes, such as the class you are currently taking.

I am conducting a research study to determine the usefulness and effectiveness of this system, and I need some students to participate in the study to test the system. I am inviting you to voluntarily participate in my research study. More information about the study and your rights as a participant or non-participant are described in the Informed Consent document that will be handed out. If you need still more information, you may obtain it from me via my email address found in the Informed Consent document.

As an additional incentive for you to participate in the study, Professor ____________ has kindly agreed to give an extra credit of up to 5 points toward the final exam of participants. He will also provide an alternative work assignment for non-participants wishing to earn extra credit.

Please take a copy as you leave the classroom if you want more information about the research study and your rights as a participant or non-participant. If you think you might want to participate in the study, please take two copies. If you decide to

participate, please return one signed copy at ________ on ________, and keep the other copy for your information or reference.

Are there any questions?

A.2 Text of Informed Consent Form

INFORMED CONSENT

for participation in the research study:

Novel Computerized Self Tutoring System for Proof Construction

Mark McCartin-Lim, Principal Investigator

Professor Beverly Woolf, Faculty Sponsor

Introduction and Purpose of the Research Study:

This document contains important information that prospective participants of the research study need to consider before they consent to participate.

The Principal Investigator (PI) has invented a novel computerized self-tutoring system that will provide immediate feedback for students to rectify learning problems, and will assist students to learn theoretical topics at their personal pace of learning. This UMass-Amherst Institutional Review Board approved research study will determine the usefulness of this tutoring system as an extra tool to help students in learning theoretical topics from computer science that involve proof construction.

Study Procedures:

The effectiveness of the computerized self-tutoring system for proof construction will be tested with student volunteers recruited from upper-year UMass Amherst computer science courses that involve theoretical content. The instructor of these courses will give extra credit of up to 5 points (minimum of 3 points for participating) that will go toward the final exam of students who volunteer to participate in the study. The instructor will also offer an alternative extra credit assignment to students

who want to earn extra credit but do not want to participate in the research study. Except for earning the extra credit, participating in this study will have no influence on a student's grade for the course.

The course instructor will select two groups from the participating students with random sampling stratified by grade distributions. One group will be the experimental group that will use the computerized self-tutoring system to solve practice problems similar to problems of existing homework assignments. The other group will be the control group, and will be given the same practice problems to solve as usually, by hand writing the problem solutions on paper and giving the solutions to the instructor for evaluation. The PI will not have access to identifiable data of the past performance of the participants, but will have access to aggregate non-identifiable data as permitted by FERPA. Any surplus participating students not needed for the experimental or control groups will earn their extra credit by completing the same work as the students in the control group, but they will not be a part of the research study.

For the experimental group students using the computerized self-tutoring system, they will either install software given to them by the PI on their personal computers, or they will be given access to use the software in a computer at a computing lab facility at UMass-Amherst. In addition to providing feedback to the student to correct mistakes, the software will provide data on the learning progress and performance proficiency of the student. This data will be made available without student identifiers to the PI.

These students will also be asked to anonymously complete a single page paper questionnaire to give their opinions and comments on the computerized self-tutoring system. Completing the questionnaire is optional; to complete the questionnaire, the participants will answer questions with check marks and brief written comments without any need to disclose their names or student identifiers. The objective of

the questionnaire is to obtain honest and unrestrained opinions, evaluations, and comments about the computerized self-tutoring system from the students who have used it.

After completing the practice problems, the students in both groups will be given a non-graded quiz, with questions similar to actual exam questions in the course, to assess their performance. The PI will analyze the performance data of these students without student identifiers to determine the efficacy of using the computerized self-tutoring system as a supplemental aid in learning proof construction.

Duration of Time Needed for Participation:

The participants will be given some practice problems typical of existing homework practice problems to solve, and time needed will be similar to that of a homework assignment. Since the proficiency for solving the problems is different for each student, an estimate of the time needed to solve the problems may range from 3 to 12 hours. The time needed for completing the questionnaire is about 10 minutes. The maximum time for completing the non-graded quiz is two hours.

Interactions of Participants with Principal Investigator:

There are no planned meetings for participants to attend with the PI during the study. The PI will communicate with the participants mainly by email when needed. For example, the PI will send emails to participants to provide information on obtaining and installing the needed software, etc. The participants may contact the PI by email when needed to obtain information on technical issues with the study.

Obtaining More Information About the Research Study:

Students needing more information about the research study should contact the PI, Mark McCartin-Lim markml@cs.umass.edu Phone: 413-842-6275

Obtaining More Information About the Student Protection:

Students needing more information about their rights as a participant in a research study may contact the University of Massachusetts Amherst Human Research Protection Office (HRPO) at (413) 545-3428 or humansubjects@ora.umass.edu.

Voluntary Participation and Right to Withdraw:

Participation in this study is voluntary, and there is no penalty for not participating. Participating or not participating in the research study will not impact a student's grade for the course. A participant may withdraw from the study without giving any reason. To withdraw, the participant will simply notify the PI of the decision to withdraw from the study.

Protection of Privacy and Personal Information of Students:

Data with student identifiers will have the student identifiers removed and replaced with a code. The PI will not have access to the key of the code used. The PI will not keep any data with student identifiers. Student privacy will be respected in this research study. Publications and public disclosures resulting from the study will not reveal student identifiers.

Possible Risks from Participation:

There are no known risks associated with this research study; however, a possible inconvenience may be the time it takes to participate in the study.

Potential Benefits from Participation:

Participants may not directly benefit from taking part in this study. However, participants of this research study will gain more exposure and practice with learning material that is relevant to the specific course they are currently enrolled in, and this may result in improved understanding of that material. Those who are chosen to use

the self- tutoring system will have the opportunity to use a learning method in this course not previously available to them. While it cannot be promised that this system will provide a more effective learning experience than the learning methods that the participant has previously been exposed to, existing positive results of tutoring systems in other domains make it hopeful that this tutoring system will also yield positive results. In addition to the above benefits, participation in this study will further research in effective ways to learn abstract topics in theoretical computer science and mathematics, helping future students taking those courses.

Consenting to Participate:

Before signing the consent to participate in the research study, students should take the necessary time to carefully read and comprehend this entire document.

A.3 Sample of directions emailed to experimental group (Fall 2017)

```
Hi there!

You are receiving this email, because you volunteered to
   participate in a research study on the affectiveness of
   using a tutoring system to teach theoretical computer
   science topics, such as what you are learning in CS 250.

If for some reason, you believe you have received this email
    by accident, please let me know.
```

You have been chosen to test a software tutoring system called Complexity Tutor. To participate, please CAREFULLY READ and follow the directions below:

Please complete STEPS 1-8 below by THURSDAY, NOVEMBER 30. By doing so, you will be eligible for up to 5 extra credit points on your final exam:

-- 3 points just for correctly following the instructions in this email
-- 2 points based on your performance on the three problems you will be asked to solve

NOTE that all URLs listed in this email are CASE-SENSITIVE, so please make sure to type them correctly. If you encounter any problems, send email to markml@cs.umass.edu .

STEP 1 - GET A PARTICIPANT ID

You will first need to get a Participant ID for the study. Throughout the study, you will be identified by this Participant ID and NOT your actual name. This protects your privacy and gives you a high level of anonymity. As

such, it is essential that you keep track of your Participant ID until you are finished with the study.

The anonymity also means that you are free to drop-out of the study at any time, and we won't know who has decided to do so.

However, if you want to receive the extra credit bonus you are entitled to for participating, you will need to reveal your Participant ID to the instructor to confirm your participation. However, the instructor will not keep this information, so there will be no permanent record linking you to your Participant ID after the course is over.

Please follow this link to generate a new Participant ID:
http://people.cs.umass.edu/~markml/participantID.php

VERY VERY IMPORTANT: Make sure you WRITE DOWN the Participant ID and DO NOT LOSE it. Every time you go to the above URL, it will generate a new ID!!!

STEP 2 - MAKE SURE YOU CAN RUN COMPLEXITY TUTOR

The minimal requirements for Complexity Tutor are a Windows PC with a screen resolution set to 1024 x 768 at the bare

minimum. If you are not sure if you have the right screen resolution, please check before you start the experiment.

You will also need to have the Microsoft .NET Framework Version 4 or higher installed. If you have Windows 8 or Windows 10, you almost certainly already have this installed, because it is installed by default with those operating systems. Even if you have Windows 7 or older, there is a decent chance you may already have it installed since it is frequently pushed to users by Windows Update.

However, if you do not have .NET Framework or your version is too old, you will need to install this version from Microsoft: https://www.microsoft.com/EN-US/DOWNLOAD/confirmation.aspx?id=17718

If you need to install .NET Framework, make sure to run the installer in Administrator Mode, by right-clicking the installer file and selecting "Run as administrator". You may need to reboot your computer after the install.

If you are not sure if you have the right version of .NET Framework installed or not, you can try running

Complexity Tutor, and it will give you an error message if you don’t have the right version.

WE CURRENTLY DO NOT SUPPORT MAC OR LINUX, BUT THERE ARE SOME OPTIONS IF YOU DO NOT HAVE WINDOWS:

1. UMass students enrolled in CS courses are eligible for a FREE copy of Windows, which will run on Macs too:
http://www.umass.edu/it/support/software/microsoft-imagine-no-cost-software-education-research#How to Obtain & Install Software

2. The UMass library will loan a Windows laptop for a 24 hour period. Go here for information:
https://www.library.umass.edu/services/computers/laptops/

3. If neither of these options work for you, please email me and we can discuss alternate arrangements.

STEP 3 - GET COMPLEXITY TUTOR

Download it from here:
https://people.cs.umass.edu/˜markml/CTutorCS250_F17.zip

Once it is downloaded, unzip it somewhere on your computer, and it will create a folder called "Complexity Tutor for CS 250".

At that point it is ready to run, and no further installation is needed. When you are finished with the experiment, you can simply delete the folder and the zip file and it will be completely removed from your computer .

Inside the folder, you will see a subfolder called " SubmissionData". This folder will store all the data you will submit to us after you use the software. Whenever you run Complexity Tutor, it will create video files recording your usage of the software, which will be stored inside the "SubmissionData" folder. Feel free to look at these videos, but do not alter them in anyway. Also, we ask that you not delete any of the files either, since we would like to see all your attempts at using the software.

In case you are interested, the videos are created with FFMPEG, an industry standard open source video transcoding tool.

Please DO NOT share the Complexity Tutor software with anyone else.

STEP 4 - TUTORIAL ON HOW TO USE COMPLEXITY TUTOR

Watch the following YouTube playlist:
https://www.youtube.com/playlist?list=PLDchYViZHp92uMNv-O9teUPALRkvkKS5q

These two quick videos will tell you what you need to know about how to use Complexity Tutor. If you would like to follow along with the problem demonstrated in the video, you can load the "Tutorial1.CTP" file in Complexity Tutor .

Here is an additional tip:
As your proof gets large, it may not fit entirely on the screen. That is okay because the Proof Space will automatically expand to give you more room and scroll bars, when you move stuff off the screen.

STEP 5 - DO THREE PROBLEMS IN COMPLEXITY TUTOR

Now, we get to the fun part!

In the "Complexity Tutor for CS 250" folder, you will find the following 3 Complexity Tutor Problem files:

"Problem1.CTP"

"Problem2.CTP"

"Problem3.CTP"

We want you to attempt each of these problems. Here’s how:

Run the "ComplexityTutor.exe" application. It will give you a dialog box, asking you to select a problem file. Choose one of the above mentioned files. You may need to wait a moment for the problem to load, especially the first time you are using Complexity Tutor. When the problem has loaded, you will be presented with a screen similar to what you saw in the tutorial videos.

In the problem, you will be asked to try to prove something. You should try to work on the problem until Complexity Tutor tells you that you have completed your proof, or until you get too frustrated to keep working on the problem.

When you feel you are done with the problem, close the Complexity Tutor application, and reload it to select another problem.

Feel free to attempt a problem multiple times. For instance, if you get stuck on one problem, you may wish to close

it and try another problem, and then come back to the original problem later. Unfortunately, Complexity Tutor does not yet have a feature to save partial progress, so if you do decide to go back to a problem you previously attempted, you will have to restart it from scratch. Also , Complexity Tutor does not let you have multiple problems open at once either, since we can only record you working on one problem at a time.

It is up to you how much time you decide to devote to using Complexity Tutor to solve these problems, however we ask that you at least attempt all three problems, and try to make the same amount of effort you would if these problems were given to you for a homework assignment.

To incentivize you to try your best to completely solve all three problems, we will give you more extra credit for each problem you manage to take down. Note the three problems will all be weighted evenly though.

VERY IMPORTANT: Please DO NOT discuss the problems or even anything about using the Complexity Tutor software with anyone else in the class. The normal collaboration policy for CS 250 does not apply to this study. For this study, no collaboration is permitted at all. If you have a technical problem with the software, you should email me at markml@cs.umass.edu rather than asking for help from

your fellow students. This will ensure I get the research data I need.

STEP 6 -- ZIPPING UP YOUR DATA

We need you to ZIP up the data, before you send it to us.

First, make sure the Complexity Tutor program has been CLOSED.

Next, navigate to the "Complexity Tutor for CS 250" folder. Inside it, you should see a subfolder called " SubmissionData". This folder contains all the data we need. We want you to create a ZIP of this "SubmissionData " folder...

In most versions of Windows, you can do so as follows:

1. Right-click the "SubmissionData" folder, and select "Send To", and then select "Compressed (zipped) folder".
2. A new ZIP file with the name "SubmissionData.zip" is created. You should rename this ZIP file to include your Participant ID.

Alternatively, you can use any 3rd party ZIP utility you prefer to do the above task (i.e. 7Zip, WinZip, etc.)

If you have any technical difficulties zipping your data, please send a reply to this email for troubleshooting.

STEP 7 -- UPLOADING YOUR DATA

Please go to the following webpage to submit your data:
https://people.cs.umass.edu/~markml/study_fall16/submitS17.html

You will be prompted for a username and password to access this page, found here:

Username: research311
Password: iknowalgorithms

Note that this is CASE-SENSITIVE.

The webpage will give you a box that allows you to Upload a file. Please do the following:

1. Click "Choose File" and select the ZIP file you previously created.

2. (VERY IMPORTANT) In the "Description" box, PLEASE TYPE YOUR PARTICIPANT ID!!!
3. Click the "Upload" button.
4. Wait while the file is being uploaded. Do not close the browser window. This may take several minutes.
5. When the file has been successfully uploaded, you should see the following message:

"Success. Your file has been uploaded, and the
folder owner has been notified."

If for some reason you do not see this message, and/or an extremely long passes and it seems to be stuck uploading the file, then I recommend you close your browser window and try again to repeat the upload process. There may be a problem with your Internet connection that has interrupted the upload.

If that does not work and you are still having trouble uploading your file, please send a reply to this email for troubleshooting.

STEP 8 -- FILL OUT A SHORT QUESTIONNAIRE

Please fill-out this questionnaire:

https://goo.gl/forms/72UPZwaLZs9NmesW2

STEP 9 -- ASSESSMENT QUIZ

You will be emailed a short assessment quiz to complete in early December, to evaluate your performance after having participated in this study. The quiz can be done at home , and should take about 15-30 minutes to complete.

FURTHER STEPS:

Once you have completed the above steps, you are eligible for the extra credit. Your instructor will notify you with the procedures for claiming your extra credit.

Please reply to this email if you have any additional questions.

A.4 Sample of directions emailed to control group (Fall 2017)

Hi there!

You are receiving this email, because you volunteered to participate in a research study on the effectiveness of

using a tutoring to teach theoretical computer science topics, such as what you are learning in CS 250.

If for some reason, you believe you have received this email by accident, please let me know.

You have been chosen to be a part of the Control Group in our study. To participate, please CAREFULLY READ and follow the directions below:

Please complete STEPS 1-5 below by THURSDAY, NOVEMBER 30. By doing so, you will be eligible for up to 5 extra credit points on your final exam:

-- 3 points just for correctly following the instructions in this email
-- 2 points based on your performance on the three problems you will be asked to solve

NOTE that all URLs listed in this email are CASE-SENSITIVE, so please make sure to type them correctly. If you encounter any problems, send email to markml@cs.umass.edu .

STEP 1 - GET A PARTICIPANT ID

You will first need to get a Participant ID for the study. Throughout the study, you will be identified by this Participant ID and NOT your actual name. This protects your privacy and gives you a high level of anonymity. As such, it is essential that you keep track of your Participant ID until you are finished with the study.

The anonymity also means that you are free to drop-out of the study at any time, and we won't know who has decided to do so.

However, if you want to receive the extra credit bonus you are entitled to for participating, you will need to reveal your Participant ID to the instructor to confirm your participation. However, the instructor will not keep this information, so there will be no permanent record linking you to your Participant ID after the course is over.

Please follow this link to generate a new Participant ID:
http://people.cs.umass.edu/~markml/participantID.php

VERY VERY IMPORTANT: Make sure you WRITE DOWN the Participant ID and DO NOT LOSE it. Every time you go to the above URL, it will generate a new ID!!!

STEP 2 - DO SOME PROBLEMS

As part of the Control Group, we would like you to do a short problem set in the same manner as you would normally complete homework problems in your CS 250 course .

We would additionally like you to record approximately how much time you spent working on each problem.

The problem set to do can be found here:
http://people.cs.umass.edu/~markml/ControlGroup_CS250.pdf

You should either scan or type your solutions, and prepare them as a PDF file. Make sure to write your Participant ID inside it as well.

STEP 3 -- CHECK YOUR PDF

You should have a PDF file for the solutions you write to the problem set. This PDF can be either a scanned copy of handwritten solutions or it can be typeset. Please make sure it is a PDF file though, and not any other format!

Also, make sure that you have included your Participant ID # instead of your name in the solutions.

Also, make sure you have written a time estimate for how much time you have spent on each problem.

STEP 4 -- UPLOAD YOUR DATA

Please go to the following webpage to submit your data: https://people.cs.umass.edu/~markml/study_fall16/submitS17.html

You will be prompted for a username and password to access this page, found here:

Username: research311
Password: iknowalgorithms

Note that this is CASE-SENSITIVE.

The webpage will give you a box that allows you to Upload a file. Please do the following:

1. Click "Choose File" and select your PDF file.

2. (VERY IMPORTANT) In the "Description" box, PLEASE TYPE YOUR PARTICIPANT ID!!!
3. Click the "Upload" button.
4. Wait while the file is being uploaded. Do not close the browser window. This may take several minutes.
5. When the file has been successfully uploaded, you should see the following message:

"Success. Your file has been uploaded, and the
folder owner has been notified."

If for some reason you do not see this message, and/or an extremely long passes and it seems to be stuck uploading the file, then I recommend you close your browser window and try again to repeat the upload process. There may be a problem with your Internet connection that has interrupted the upload.

If that does not work and you are still having trouble uploading your file, please send a reply to this email for troubleshooting.

STEP 5 -- GET FEEDBACK ON YOUR SOLUTIONS

Feedback on your solutions will be emailed to an anonymous email account at Dispostable.com.

To access the feedback, please go to http://www.dispostable.com.

Your anonymous email address will be Participant[YOUR PARTICIPANT ID]@dispostable.com. For instance, if your Participant ID was 987654321, then your anonymous email address would be Participant987654321@dispostable.com.

Type in your anonymous email address, and click "Check inbox >>". There is no password associated with this email address, and privacy is protected by the uniqueness of your Participant ID, so make sure to not share it with anyone.

Note that emails are automatically deleted from the Dispostable.comservers every 3 days, so make sure to check your Dispostable.comaccount regularly after November 3 until you receive your feedback.

STEP 6 -- ASSESSMENT QUIZ

You will be emailed a short assessment quiz to complete in early December, to evaluate your performance after having participated in this study. The quiz can be done at home, and should take about 15-30 minutes to complete.

FURTHER STEPS:

Once you have completed the above steps, you are eligible for the extra credit. Your instructor will notify you with the procedures for claiming your extra credit.

APPENDIX B

NP-COMPLETENESS PROBLEMS USED IN COMPLEXITY TUTOR EXPERIMENTS

B.1 Conceptual Problem 1

Let PATH $= \{\langle G, s, t\rangle \mid G$ is a graph with a path from s to $t\}$. We know PATH is in P. Prove that if P=NP, PATH is NP-Complete.

B.2 Conceptual Problem 2 (used in Spring 2018 only)

Note: *For this problem, you must use the Karp Reduction (pg. 473 of the textbook [87]) definition of polytime reductions. The definition is:* $Y \leq_p X$ ***if there is a polytime function*** $A(s)$ ***where*** $s \in Y \iff A(S) \in X$**.**

Show for all decision problems X and Y, if X is in NP and $Y \leq_p X$, then Y is also in NP.

B.3 Conceptual Problem 3 (used in Spring 2018 only)

Note: *For this problem, you must use the Karp Reduction (pg. 473 of the textbook [87]) definition of polytime reductions. The definition is:* $Y \leq_p X$ ***if there is a polytime function*** $A(s)$ ***where*** $s \in Y \iff A(S) \in X$**.**

Recall the definition of co-NP (pg. 496 of the textbook [87]):

A decision problem X is in co-NP if there is a decision problem Y in NP where $s \in X \iff s \notin Y$.

Show for all decision problems X and Y, if X is in co-NP and $Y \leq_p X$, then Y is also in co-NP.

B.4 Conceptual Problem 4 (used in Spring 2018 only)

Note: *For this problem, you must use the Karp Reduction (pg. 473 of the textbook [87]) definition of polytime reductions. The definition is:* $Y \leq_p X$ ***if there is a polytime function*** $A(s)$ ***where*** $s \in Y \iff A(S) \in X$.

A decision problem X is *co-NP-Complete* if X is in co-NP and for all Y in co-NP, $Y \leq_p X$.

Prove that if a decision problem X is both NP-Complete and co-NP-Complete then NP=co-NP. *HINT: You can assume the results you proved in Conceptual Problem 2 and Conceptual Problem 3.*

B.5 BIN-PACKING Reduction Problem

Assume PARTITION is NP-Complete and BIN-PACKING is in NP. Prove that BIN-PACKING is NP-Complete.

PARTITION

INSTANCE: A set X of positive integers.

PROBLEM: Is there a subset $S \subseteq X$ such that:

$$\sum_{x \in S} x = \sum_{x \in X/S} x$$

(i.e. X be partitioned into two sets with equal sums)

BIN-PACKING

INSTANCE: A set of reals Y from the domain [0, 1) and a positive integer K.

PROBLEM: Is there a partition of Y into $Y_1, \ldots, Y_K$ such that for every Y_i:

$$\sum_{y \in Y_i} y \leq 1$$

B.6 0/1-PROG Reduction Problem

Prove 0/1-PROG = $\{\langle$ Matrix A, Vector $b\rangle$ | There exists a Vector x consisting of elements in the set $\{0, 1\}$ such that $Ax \leq b$, where A is an m-by-n Matrix and b and x are length n vectors$\}$ is NP-Complete. *Hint: Use 3-SAT.*

APPENDIX C

LOGIC PROBLEMS USED IN COMPLEXITY TUTOR EXPERIMENTS

C.1 Pizza Problem

Emily and Catherine each have their own pizza. Using the given assumptions, prove that Emily is not lactose intolerant.

Given Assumptions:

1. Any pizza that has pepperoni also has cheese.
2. Either Catherine's pizza or Emily's pizza, or both, has pepperoni, but Catherine's pizza does not have cheese.
3. If someone is lactose intolerant, then their pizza does not have cheese.

C.2 Muddy Dog Problem

Suppose there are 3 dogs—Biscuit, Cardie and Duncan. Using the given assumptions, prove that Duncan is not muddy.

Given Assumptions:

1. Either Cardie or Biscuit, but not both, went in the pond, and if Cardie is not muddy then Biscuit is muddy
2. A dog is wet if and only if it went in the pond or it went in the swamp
3. Every muddy dog went in the swamp, and every dog who went in the pond is not muddy

4. Not all of the three dogs are wet

C.3 Murder Mystery Problem

Professor Plum (p), Miss Scarlet (s), Mr. Green (g) and Colonel Mustard (m) are suspects in a murder investigation. The police have determined that one of them did a murder in either the Ballroom (b), the Dining Room (d) or the Hall (h)—with either the Knife (k), the Rope (r) or the Candlestick (c).

Referring to the predicate and set definitions below, use the given assumptions to prove who did the murder, where they did it, and with what weapon:

Let set $S = \{p, s, g, m\}$—the finite set of suspects.

Let set $W = \{k, r, c\}$—the finite set of weapons.

Let set $L = \{b, d, h\}$—the finite set of locations.

Let predicate $M(x, y, z)$ mean "suspect x did the murder in location y with weapon z".

Let predicate $L(x, y)$ mean "suspect x was in location y during the murder".

Let predicate $T(x, y)$ mean "suspect x testified they were in location y during the murder".

Let predicate $O(x, z)$ mean "suspect x owns weapon z".

Let predicate $Q(x)$ mean "suspect x told the truth".

Given Assumptions:

1. $\forall x : \forall y : [(\exists x' : \exists z' : M(x', y, z')) \land L(x, y)] \to [\exists z' : M(x, y, z')]$

2. $\forall x : \forall y : \forall y' : y = y' \lor \neg L(x, y) \lor \neg L(x, y')$

3. $\forall x : \forall y : Q(x) \to [T(x, y) \to L(x, y)]$

4. $\forall y : [\exists x : \exists z : M(x, y, z)] \to [\forall x : \forall x' : x = x' \lor \neg L(x, y) \lor \neg L(x', y)]$

5. $\forall x : \forall y : \forall z : M(x, y, z) \to \neg O(x, z)$

6. Only one suspect did not tell the truth.
7. Plum testified he was in the Hall during the murder.
8. Scarlett testified she was in the Hall during the murder.
9. Green testified he was in the Dining Room during the murder.
10. Mustard owns the Knife.
11. Scarlett owns the Rope.
12. Plum owns the Candlestick.
13. The murder was done in the Hall.
14. The murder was not done with the Knife.
15. During the murder, Mustard was in the same location as Plum.

APPENDIX D

EXAMS QUESTIONS USED IN EMPIRICAL STUDY

This appendix reproduces with minor edits some actual exam questions that were given to students in the courses studied, and which were analyzed for the research of this dissertation. Some minor typographical errors have been corrected from the original source material, but those errors were unlikely to have substantially affected how students performed when answering the questions.

D.1 Spring 2017 midterm exam questions

On this exam, there were three related questions, concerning first-order prepositional logic. These questions used the following definitions:

Let D be a finite set of dogs consisting of exactly the four distinct dogs Cardie (c), Duncan (d), Mia (m), and Whistle (w).

Let Z be a finite set of languages consisting of exactly the five distinct languages Chinese (C), English (E), French (F), Latin (L), and Spanish (S).

Let T be the unary relation on D defined so that $T(x)$ means "dog x is a terrier".

Let R be the binary relation from D to Z defined so that $R(x, y)$ means "dog x responds to commands in language y".

Question 1 (20 points)

Translate each statement as indicated, using the set of dogs $D = \{c, d, m, w\}$, the set of languages $Z = \{C, E, F, L, S\}$, the predicate $T(x)$ meaning "dog x is a terrier", and the predicate $R(x, y)$ meaning "dog x responds to commands in language y". In

general, x is used as a variable of type "dog" (S) and y is used as a variable of type "language" (Z), but this should also be clear by the usage of variables in predicates.

- (to English) Statement I: $\neg(R(c, F) \rightarrow R(c, S)) \wedge \neg(\neg R(c, L) \vee R(c, S))$
- (to symbols) Statement II: Cardie responds to commands both in French and in Latin.
- (to English) Statement III: $\forall y : (y \neq C) \leftrightarrow (\exists x : R(x, y))$
- (to symbols) Statement IV: There is exactly one dog, Whistle, who responds to commands in Spanish.
- (to English) Statement V: $\forall x : \forall y : R(x, y) \rightarrow R(x, E)$
- (to symbols) Statement VI: Unless Duncan is a terrier, he responds to commands in Chinese.
- (to English) Statement VII: $\exists x : \forall y : \neg T(x) \wedge (R(x, y) \leftrightarrow (y = E))$
- (to symbols) Statement VIII: Any dog who does not respond to commands in English must be a terrier.

Question 2 (10 points)

This question uses the sets, definitions, and predicates above, and the statements from Question 1.

Prove that if Statement I is true, Statement II must be true as well. You may use either a truth table or a deductive argument.

Question 3 (20 points)

This question also uses the sets, definitions, and predicates from above and the statements from Question 1.

Prove, using any or all of Statements I through VII, that Statement VIII is true. Do not assume anything about the English meaning of the predicates, except what you are given in the statements. Make your use of quantifier proof rules clear.

(Hint: If you have an arbitrary dog x, you may divide into the four cases $x = c$, $x = d$, $x = m$, and $x = w$. It is possible to solve this problem with or without Proof By Contradiction.)

D.2 Spring 2017 final exam questions

Two parts of Question 2, concerning first-order prepositional logic, were considered individually from this exam.

Question 2 deals with the following scenario. All of the dogs in the neighborhood are avid birdwatchers. One day, a set of five dogs met after their morning walks to compare their observations of five possible bird species. The set D of dogs consists exactly of Arly (a), Cardie (c), Duncan (d), Mia (m), and Whistle (w). The set S of species consists exactly of Bluebird (B), Crow (C), Heron (H), Mallard (M), and Woodpecker (W). The relation $R \subseteq (D \times S)$ is defined so that $(x, y) \in R$ means "dog x observed a bird of species y".

Question 2a (10 points)

In the following five statements, variables are of type "dog" or of type "bird species". Translate each of these five statements as indicated.

- (to symbols) Statement I: There is a species that was observed by all the dogs.
- (to English) Statement II: $\forall z : R(a, z) \leftrightarrow (z = C)$
- (to symbols) Statement III: Every dog other than Arly observed at least two different species of bird.

- (to English) Statement IV: $\forall x : [(x \neq a) \wedge (x \neq c)] \rightarrow [(\forall y : R(c, y) \rightarrow R(x, y)) \wedge (\exists z : R(x, z) \wedge \neg R(c, z))]$
- (to symbols) Statement V: Duncan was the one and only dog who observed a Woodpecker, and Whistle was the one and only dog who observed a Bluebird.

Question 2b (20 points)

Assuming that Statements I-V from Question 2a are all true, prove that some dog observed a Heron. You may use either English or symbols, but make your use of quantifier rules clear.

D.3 Fall 2017 midterm exam questions

On this exam, there were two related questions, concerning first-order prepositional logic. These questions used the following definitions:

Let A be a finite set of animals consisting of exactly the five distinct animals Cardie (c), Duncan (d), Floyd (f), Scout (s), Whistle (w).

Let D be the unary relation on A defined so that $D(x)$ means "x is a dog".

Let F be the unary relation on A defined so that $F(x)$ means "x lives on the farm".

Let R be the unary relation on A defined so that $R(x)$ means "x is a retriever".

Let M be the binary relation on A defined so that $M(x, y)$ means "animal x met animal y during the morning walk". Note that two animals could be together on the walk without meeting *during* it.

Question 1 (15 points)

Translate each statement as indicated, using the set of animals $A = \{c, d, f, s, w\}$, the predicate $D(x)$ meaning "animal x is a dog", the predicate $F(x)$ meaning "animal x lives on the farm", the predicate $R(x)$ meaning "animal x is a retriever", and the

predicate $M(x, y)$ meaning "animal x and animal y met during the morning walk". Note that two animals might be together for the entire morning walk but not meet during it. All these are also defined above. Note that variables and constants of type "animal" are in small letters, and predicates are in capital letters.

- (to symbols) Statement I: Floyd, who is not a dog, met every animal who does not live on the farm.
- (to English) Statement II: $\forall x : \neg R(x) \vee D(x)$
- (to symbols) Statement III: It is not the case that if Floyd lives on the farm, than Duncan met Cardie.
- (to English) Statement IV: $[\forall z : \neg M(x, x)] \wedge [\forall y : \forall z : M(y, z) \rightarrow M(z, y)]$
- (to symbols) Statement V: Cardie and Duncan met exactly the same animals, and they met all the animals who live on the farm.
- (to English) Statement VI: $\neg F(w) \wedge [\exists x : R(x) \wedge F(x) \wedge M(x, w)]$
- (to symbols) Statement VII: Whistle met every animal who lives on the farm.

Question 2 (30 points)

These questions use the sets, definitions, and predicates above, and the statements from Question 1.

a. (10 points) Use Statements I, II, and III to infer propositional statements about the propositions $D(f)$, $F(f)$, and $R(f)$. Use propositional methods (a truth table, or deductive or equational proof rules) to determine the truth of these three propositions, assuming *only* that Statements I, II, and III are true.

b. (10 points) Assuming that Statements I, IV and V are true, use propositional and predicate proof rules to prove Statement III. Do not assume the truth of

any of the other statements. You may use English, symbols, or a combination, as long as your argument is clear.

c. (10 points) Assuming that Statements I, II, III, IV, V, and VI are all true, use propositional and predicate proof rules to prove Statement VII. Do not assume the truth of any of the other statements. You may use English, symbols, or a combination, as long as your argument is clear.

D.4 Fall 2017 final exam questions

Question 1, concerning first-order prepositional logic, was considered from this exam. Question 1 deals with the following scenario:

The web site *WeRateDogs*™ gives numerical ratings of animals based on photographs, and publishes these at `twitter.com/dog_rates`. They provided ratings for a set A of six animals, consisting exactly of Cardie (c), Duncan (d), Floyd (f), Mia (m), Pushkin (p), and Tib (t). Thus, there is a function r from A to $\mathbb{N}$ where $r(x)$ is the rating of animal x.

There are a number of additional predicates defined on A:

- $E(x)$ means "animal x is enormous".
- $ML(x, y)$ means "animal x is much larger than animal y" and this is *defined* to mean "$E(x) \wedge \neg E(y)$".
- $D(x)$ means "animal x is a dog".
- $P(n)$ means "natural number n is prime".

Question 1 (30 points)

This question deals with the scenario described above, and with six statements about a set of animals A, consisting of exactly the six animals Cardie (c), Duncan

(d), Floyd (f), Mia (m), Pushkin (p), and Tib (t). It uses the function $r : A \mapsto \mathbb{N}$ and the predicates E, ML, D, and P defined above.

a. (10 points) Translate each of these six statements as indicated.

- (to symbols) Statement I: Floyd, who is enormous, received a rating of 7, and no enormous animal received a higher rating.
- (to English) Statement II: $\forall x : P(r(x)) \wedge ((r(x) > 10) \leftrightarrow D(x))$
- (to symbols) Statement III: Tib received the same rating as some animal much larger than herself.
- (to English) Statement IV: $\forall x : \exists y : (x \neq y) \wedge (r(x) = r(y))$
- (to symbols) Statement V: Pushkin received a rating that was less than some other animal's rating and greater than some other animal's rating.
- (to English) Statement VI: $\neg \forall x : \forall y : (D(x) \wedge D(y)) \rightarrow (r(x) = r(y))$

b. (10 points) Assuming that Statements I-VI are all true, determine exactly which of the six animals are dogs, and prove your answer.

c. (10 points) Assuming that Statements I-VI are all true, prove that some dog received a rating of 13 or more.

APPENDIX E

POSTTEST QUIZZES

E.1 NP-Completeness posttest quiz used in Spring 2018

- This quiz should take about 15-30 minutes to complete.
- Please complete it in a single sitting.
- Do not consult your textbook, classmates, Internet, or other resources when completing the quiz.
- You must submit a quiz to receive extra credit, but your performance on the quiz will not affect how much extra credit you receive or affect your course grade in anyway.

DIRECTIONS:

There are 5 Questions. Only the last one requires you to write a proof.

For Questions 1-4, you are given an assertion along with an *attempted proof* of the assertion. The attempted proof may or may not be correct. If the attempted proof is correct, write "CORRECT" for that problem and move on.

If the proof is not correct and has flaws or bugs, underline any and all parts of the proof that are wrong, and write an explanation for why they are wrong.

Question 1

Given decision problems X and Y, show that if X is not in P and $X \leq_p Y$, then Y is not in P.

Attempted Proof:

Since X is not in P, X must be NP-Complete. And since X reduces to Y, it follows that Y is also NP-Complete. Therefore, Y is not in P.

Question 2

We define co-P as follows: A decision problem X is in *co-P* if there is a decision problem Y in P where $s \in X \iff s \notin Y$.

Prove that P=co-P.

Attempted Proof:

Suppose we have a decision problem X in co-P. Therefore, there is a decision problem Y in P where $s \in X \iff s \notin Y$. Since Y is in P, there is polytime decider algorithm for Y, which we can refer to as decide-Y. If we simply run decide-Y and invert the result it gives, then this is a polytime decider algorithm for X. Thus X is in P.

So $X \in$ co-P $\implies X \in P$. Now we prove the converse:

Suppose we have a decision problem X in P. Now, consider the decision problem Y where $s \in Y \iff s \notin X$. We know that Y is in co-P. From what we proved above, Y must also be in P. But this means that X is in co-P.

So $X \in$ P $\implies X \in$ co-P.

We have shown $X \in$ P $\iff X \in$ co-P, which means P and co-P are equal.

Question 3

Prove that if no NP-Complete problems are in NP $\cap$ co-NP then P $\neq$ NP.

Attempted Proof:

Since there are no NP-Complete problems in NP $\cap$ co-NP, we know that no problems from NP belong to NP $\cap$ co-NP. This in turn implies that NP $\cap$ co-NP is empty and that no problems from NP belong to co-NP. But then that implies NP $\neq$ co-NP. To complete the proof, we show that NP $\neq$ co-NP implies P $\neq$ NP.

We do so by contrapositive, showing that P = NP would imply NP = co-NP.

Assuming, P=NP, if a given decision problem X is in NP then it must also be in P, and we know that its opposite (swapping 'Yes' and 'No' outputs) must also be in P and therefore in NP. This would mean that X is in co-NP. So NP is a subset of co-NP.

Also assuming P=NP, if a given decision problem X is in co-NP, then its opposite is in NP, which is also in P, which implies X itself is in P. And thus X is in NP. Hence, co-NP is a subset of NP.

If NP is a subset of co-NP and co-NP is a subset of NP, then NP = co-NP.

Question 4

Uncle Scrooge is writing his will, and wants Huey, Duey, Louie and Abigail to each receive an inheritance of equal value. In other words, he has a list of items to split between them, each item with a monetary value, and he wants to split the items into 4 sets that each sum to the same amount. Show that determining if this is possible is NP-Complete.

Attempted Proof:

Given a set of integers, the problem is to determine if we can split the set into 4 subsets with equal sum. This problem is obviously in NP because if we were given a particular assignment of the inheritance of Huey, Duey, Louie and Abigail, we could easily check that the assignment is fair, by just adding up the value of each nephews inheritance and comparing. This takes linear time.

To show that the problem is actually NP-Complete, we will reduce to a similar problem, PARTITION. The problem of PARTITION is "Given a set of integers, determine if we can split the set into 2 subsets with equal sum". We already know that PARTITION is NP-Complete.

The reduction is as follows: Given a PARTITION solver, determine how to split the set into two subsets. Then, if the solver was successful, recursively run the solver on each of the two subsets. If we are successful, we will have created four subsets of equal sum, a solution to the inheritance problem. If not, there is no way to divide the inheritance. So to be able to divide the inheritance, we need to be able to solve PARTITION.

Therefore, the inheritance problem is NP-Complete.

Question 5

Let PRIMES = $\{\langle p \rangle \mid p$ is a prime number$\}$. We know PRIMES is in P. Prove that if P=NP, PRIMES is NP-Complete.

E.2 First-order logic posttest quiz used in Fall 2017

- This quiz will not affect your grade at all, so please do not cheat! Please do not consult friends or any other resources while attempting this quiz.
- If you are struggling with the quiz, it is okay to submit a blank sheet with just your Participant ID, since knowing how many people struggled with the quiz is beneficial to my research. However, if you can write something to explain why you are struggling that would be even better.

Merlin (m), Sabrina (s) and Harry (h) are children who may have received some presents for Christmas. *Did Merlin receive a unicorn for Christmas?*

Using the given definitions and assumptions, determine whether or not Merlin received a unicorn for Christmas, and prove it!

Definitions:

Let set $C = \{m, s, h\}$ – the set of children

Let predicate $S(y, x)$ mean "Santa delivered present y to child x".

Let predicate $R(x, y)$ mean "Child x received present y for Christmas".

Assumptions:

1. $\forall x : \forall y : \forall y' : y = y' \vee \neg R(x, y) \vee \neg R(x, y')$

2. $\forall y : [\exists x : S(y, x)] \rightarrow [\forall x : \forall x' : x = x' \vee \neg R(x, y) \vee \neg R(x', y)]$

3. There is a present that both Merlin and Sabrina received for Christmas.

4. Santa delivered a unicorn to a child.

BIBLIOGRAPHY

[1] Abrahams, Paul. *Machine verification of mathematical proofs*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1963.

[2] Ahmed, Umair Z., Gulwani, Sumit, and Karkare, Amey. Automatically generating problems and solutions for natural deduction. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (2013), AAAI Press, pp. 1968–1975.

[3] Aho, Alfred V., and Ullman, Jeffrey D. *Foundations of Computer Science*. Computer Science Press, Inc., New York, NY, USA, 1992.

[4] Albert, Elvira, Arenas, Puri, Genaim, Samir, and Puebla, Germán. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning 46*, 2 (Feb 2011), 161–203.

[5] Anderson, John R, Boyle, C Franklin, Corbett, Albert T, and Lewis, Matthew W. Cognitive modeling and intelligent tutoring. *Artificial intelligence 42*, 1 (1990), 7–49.

[6] Anderson, John R., Boyle, C. Franklin, and Yost, Gregg. The geometry tutor. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1* (1985), Morgan Kaufmann Publishers Inc., pp. 1–7.

[7] Anderson, J.R. *Language, memory, and thought*. The experimental psychology series. L. Erlbaum Associates, 1976.

[8] Anderson, J.R. *The architecture of cognition*. Harvard University Press, Cambridge, Massachusetts, 1983.

[9] Anderson, J.R. *Rules of the Mind*. L. Erlbaum Associates, 1993.

[10] Anderson, J.R. *The Architecture of Cognition*. Cognitive science series. Erlbaum, 1996.

[11] Andrews, Peter B. Refutations by matings. *IEEE Trans. Comput. 25*, 8 (Aug. 1976), 801–807.

[12] Andrews, Peter B. Transforming matings into natural deduction proofs. In *5th Conference on Automated Deduction Les Arcs, France, July 8–11, 1980* (Berlin, Heidelberg, 1980), Wolfgang Bibel and Robert Kowalski, Eds., Springer Berlin Heidelberg, pp. 281–292.

[13] Andrews, Peter B., Bishop, Matthew, Issar, Sunil, Nesmith, Dan, Pfenning, Frank, and Xi, Hongwei. TPS: A theorem-proving system for classical type theory. *Journal of Automated Reasoning 16*, 3 (Jun 1996), 321–353.

[14] Arora, S. Reductions, codes, PCPs, and inapproximability. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science* (1995), IEEE Computer Society, pp. 404–413.

[15] Arora, Sanjeev, and Barak, Boaz. *Computational Complexity: A Modern Approach*, 1st ed. Cambridge University Press, New York, NY, USA, 2009.

[16] Autexier, Serge. The CoRe calculus. In *Automated Deduction – CADE-20* (Berlin, Heidelberg, 2005), Robert Nieuwenhuis, Ed., Springer Berlin Heidelberg, pp. 84–98.

[17] Autexier, Serge, and Dietrich, Dominik. Atomic metadeduction. In *KI 2009: Advances in Artificial Intelligence* (Berlin, Heidelberg, 2009), Bärbel Mertsching, Marcus Hund, and Zaheer Aziz, Eds., Springer Berlin Heidelberg, pp. 444–451.

[18] Autexier, Serge, Dietrich, Dominik, and Schiller, Marvin. Cognitive tutoring in mathematics based on assertion level reasoning and proof strategies (extended abstract). In *THedu'11, Workshop associated to CADE-23. THedu - CTP Components for Educational Software (ThEdu-11), located at Conference on Automated Deduction, July 31, Wroclaw, Poland* (7 2011), Pedro Quaresma and Ralph-Johan Back, Eds., no. 2011/001 in CISUC Technical Report, Center for Informatics and Systems, University of Coimbra, Portugal, University of Coimbra, Portugal, pp. 11–15.

[19] Baader, Franz, and Nipkow, Tobias. *Term Rewriting and All That.* Cambridge University Press, 1998.

[20] Bachmair, Leo. *Canonical Equational Proofs.* Birkhauser, 1991.

[21] Barnes, Tiffany. Game2Learn Lab Research. `http://eliza.csc.ncsu.edu/research.html`. Accessed: 2019-05-01.

[22] Barnes, Tiffany, and C. Stamper, John. Automatic hint generation for logic proof tutoring using historical data. *Educational Technology & Society 13* (01 2010), 3–12.

[23] Battista, Giuseppe Di, Eades, Peter, Tamassia, Roberto, and Tollis, Ioannis G. *Graph drawing: algorithms for the visualization of graphs.* Prentice Hall PTR, 1998.

[24] Bell, Chester, and Newell, Allen. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.

[25] Ben-Amram, Amir M. On decidable growth-rate properties of imperative programs. In *Proceedings International Workshop on Developments in Implicit Computational complExity, DICE 2010, Paphos, Cyprus, 27-28th March 2010.* (2010), pp. 1–14.

[26] Ben-Sasson, Eli, and Wigderson, Avi. Short proofs are narrow — resolution made simple. *J. ACM 48*, 2 (Mar. 2001), 149–169.

[27] Benzmüller, Christoph, Dietrich, Dominik, Schiller, Marvin, and Autexier, Serge. Deep inference for automated proof tutoring? In *KI 2007: Advances in Artificial Intelligence* (Berlin, Heidelberg, 2007), Joachim Hertzberg, Michael Beetz, and Roman Englert, Eds., Springer Berlin Heidelberg, pp. 435–439.

[28] Benzmüller, Christoph, and Vo, Quoc Bao. Mathematical domain reasoning tasks in natural language tutorial dialog on proofs. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2* (2005), AAAI'05, AAAI Press, pp. 516–522.

[29] Bertot, Yves, and Castran, Pierre. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[30] Bibel, W. An approach to a systematic theorem proving procedure in first-order logic. *Computing 12*, 1 (Mar 1974), 43–55.

[31] Bledsoe, W.W. Splitting and reduction heuristics in automatic theorem proving. *Artificial Intelligence 2*, 1 (1971), 55 – 77.

[32] Boolos, George S., Burgess, John P., and Jeffrey, Richard C. *Computability and Logic*, 4 ed. Cambridge University Press, 2002.

[33] Boolos, G.S., Burgess, J.P., and Jeffrey, R.C. *Computability and Logic*. Computability and Logic. Cambridge University Press, 2002.

[34] Braine, Martin D. On the relation between the natural logic of reasoning and standard logic. *Psychological Review 85*, 1 (1978), 1–21.

[35] Brauner, Paul, Houtmann, Clement, and Kirchner, Claude. Principles of superdeduction. In *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science* (Washington, DC, USA, 2007), LICS '07, IEEE Computer Society, pp. 41–50.

[36] Brockschmidt, Marc, Emmes, Fabian, Falke, Stephan, Fuhs, Carsten, and Giesl, Jürgen. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst. 38*, 4 (Aug. 2016), 13:1–13:50.

[37] Brown, John Seely. Process versus product: A perspective on tools for communal and informal electronic learning. *Journal of Educational Computing Research 1*, 2 (1985), 179–201.

[38] Brünken, Roland, Moreno, Roxana, and Plass, Jan L. *Cognitive Load Theory.* Cambridge University Press, 2010.

[39] Catrambone, Richard. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of experimental psychology: General 127*, 4 (1998), 355.

[40] Cheng, Patricia W, and Holyoak, Keith J. Pragmatic reasoning schemas. *Cognitive Psychology 17*, 4 (1985), 391 – 416.

[41] Collins, Allan, and Brown, John Seely. The computer as a tool for learning through reflection. In *Learning Issues for Intelligent Tutoring Systems*, Heinz Mandl and Alan Lesgold, Eds. Springer US, New York, NY, 1988, pp. 1–18.

[42] Cook, Stephen A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1971), STOC '71, ACM, pp. 151–158.

[43] Cooper, Graham, and Sweller, John. Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of educational psychology 79*, 4 (1987), 347.

[44] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[45] Cosmides, Leda, and Tooby, John. Cognitive adaptations for social exchange. In *The adapted mind: Evolutionary psychology and the generation of culture*, Jerome H. Barkow, Leda Cosmides, and John Tooby, Eds. Oxford University Press, New York, 1992, ch. 3, pp. 163–228.

[46] Denny, Paul, Luxton-Reilly, Andrew, and Simon, Beth. Evaluating a new exam question: Parsons problems. In *Proceedings of the fourth international workshop on computing education research* (2008), ACM, pp. 113–124.

[47] Dietrich, Dominik, and Buckley, Mark. Verification of proof steps for tutoring mathematical proofs. In *Proceedings of the 2007 Conference on Artificial Intelligence in Education: Building Technology Rich Learning Contexts That Work* (Amsterdam, The Netherlands, The Netherlands, 2007), IOS Press, pp. 560–562.

[48] Dijkstra, Edsger W., and Scholten, Carel S. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, Heidelberg, 1990.

[49] Douce, Christopher, Livingstone, David, and Orwell, James. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput. 5*, 3 (Sept. 2005).

[50] Epp, Susanna S. The role of logic in teaching proof. *The American Mathematical Monthly 110*, 10 (2003), 886–899.

[51] Ericson, Barbara Jane. *Evaluating the effectiveness and efficiency of Parsons problems and dynamically adaptive parsons problems as a type of low cognitive load practice problem.* PhD thesis, Georgia Institute of Technology, 2018.

[52] Ericsson, K.A., and Simon, H.A. *Protocol Analysis: Verbal Reports as Data.* A Bradford book. Bradford Books, 1993.

[53] Falkner, Nickolas, Sooriamurthi, Raja, and Michalewicz, Zbigniew. Puzzle-based learning for engineering and computer science. *Computer 43*, 4 (2010), 20–28.

[54] Fitch, F.B. *Symbolic Logic: An Introduction.* Ronald Press Company, 1952.

[55] Flores-Montoya, Antonio, and Hähnle, Reiner. Resource analysis of complex programs with cost equations. In *Programming Languages and Systems* (Cham, 2014), Jacques Garrigue, Ed., Springer International Publishing, pp. 275–295.

[56] Forgy, Charles L. OPS5 user's manual. Tech. Rep. CMU-CS-81-135, Department of Computer Science, Carnegie Mellon University, 1981.

[57] Forišek, Michal. On the suitability of programming tasks for automated evaluation. *Informatics in education 5*, 1 (2006), 63–76.

[58] Fuchs, Norbert E, Kaljurand, Kaarel, and Kuhn, Tobias. Attempto controlled english for knowledge representation. In *Reasoning Web*. Springer, 2008, pp. 104–124.

[59] Garey, Michael R., and Johnson, David S. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[60] Gentzen, Gerhard. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift 39*, 1 (Dec 1935), 176–210.

[61] Giesl, Jürgen, Brockschmidt, Marc, Emmes, Fabian, Frohn, Florian, Fuhs, Carsten, Otto, Carsten, Plücker, Martin, Schneider-Kamp, Peter, Ströder, Thomas, Swiderski, Stephanie, and Thiemann, René. Proving termination of programs automatically with AProVE. In *Automated Reasoning* (Cham, 2014), Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, Eds., Springer International Publishing, pp. 184–191.

[62] Greer, Jim, and McCalla, Gordon. A computational framework for granularity and its application to educational diagnosis. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1* (San Francisco, CA, USA, 1989), IJCAI'89, Morgan Kaufmann Publishers Inc., pp. 477–482.

[63] Gries, David. *The Science of Programming*, 1st ed. Springer-Verlag, Berlin, Heidelberg, 1987.

[64] Gries, David, and Schneider, Fred B. *A Logical Approach to Discrete Math.* Springer-Verlag, Berlin, Heidelberg, 1993.

[65] Gries, David, and Schneider, Fred B. A new approach to teaching mathematics. Tech. Rep. TR94-1411, Department of Computer Science, Cornell University, 1994.

[66] Gries, David, and Schneider, Fred B. Equational propositional logic. *Information Processing Letters 53*, 3 (1995), 145 – 152. The calculational method.

[67] Griggs, Richard A., and Cox, James R. The elusive thematic-materials effect in Wason's selection task. *British Journal of Psychology 73*, 3 (1982), 407–420.

[68] Guglielmi, Alessio. Deep inference. In *All About Proofs, Proofs for All.* College Publications, 2015.

[69] Haghighi, Aria, and Klein, Dan. Prototype-driven grammar induction. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics* (2006), Association for Computational Linguistics, pp. 881–888.

[70] Hamalainen, Wilhelmiina. Problem-based learning of theoretical computer science. In *Frontiers in Education, 2004. FIE 2004. 34th Annual* (2004), IEEE, pp. S1H–1.

[71] Herbrand, J. *Recherches sur la théorie de la démonstration.* (Prace Towarzystwa Nauk. Warsz). imprimerie J. Dziewulski, 1930.

[72] Herstein, I.N. *Topics in algebra.* Xerox College Pub., 1975.

[73] Hobbs, Jerry R. Granularity. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1* (San Francisco, CA, USA, 1985), IJCAI'85, Morgan Kaufmann Publishers Inc., pp. 432–435.

[74] Hollingsworth, Jack. Automatic graders for programming classes. *Commun. ACM 3*, 10 (Oct. 1960), 528–529.

[75] Hopcroft, John E., Motwani, Rajeev, and Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[76] Huang, Xiaorong. Reconstruction proofs at the assertion level. In *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings* (1994), pp. 738–752.

[77] Jaśkowski, S. *On the Rules of Suppositions in Formal Logic.* Studia logica. Nakładem Seminarjum Filozoficznego Wydziału Matematyczno-Przyrodniczego Uniwersytetu Warszawskiego, 1934.

[78] Johnson-Laird, Philip N. *Mental Models: Towards a Cognitive Science of Language, Inference, and Consciousness*. Cognitive science series. Harvard University Press, 1983.

[79] Johnson-Laird, Philip N. Mental models and human reasoning. *Proceedings of the National Academy of Sciences 107*, 43 (2010), 18243–18250.

[80] Jonassen, David H. Objectivism versus constructivism: Do we need a new philosophical paradigm? *Educational Technology Research and Development 39*, 3 (1991), 5–14.

[81] Jones, F. Burton. The Moore method. *The American Mathematical Monthly 84*, 4 (1977), 273–278.

[82] Jones, Neil D. *Computability and Complexity: From a Programming Perspective*. MIT Press, Cambridge, MA, USA, 1997.

[83] Kahl, Wolfram. CalcCheck: A proof checker for teaching the "logical approach to discrete math". In *Interactive Theorem Proving* (Cham, 2018), Jeremy Avigad and Assia Mahboubi, Eds., Springer International Publishing, pp. 324–341.

[84] Karp, Richard M. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[85] Keet, C. M. A taxonomy of types of granularity. In *2006 IEEE International Conference on Granular Computing* (May 2006), pp. 106–111.

[86] Kfoury, A. J., Arbib, Michael A., and Moll, Robert N. *Programming Approach to Computability*. Springer-Verlag, Berlin, Heidelberg, 1991.

[87] Kleinberg, Jon, and Tardos, Eva. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[88] Koedinger, Kenneth. *Tutoring Concepts, Percepts, and Rules in Geometry Problem Solving*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1991.

[89] Koedinger, Kenneth R, and Anderson, John R. Reifying implicit planning in geometry: Guidelines for model-based intelligent. *Computers as cognitive tools* (2013), 15.

[90] Korf, Richard E. Macro-operators: A weak method for learning. *Artificial intelligence 26*, 1 (1985), 35–77.

[91] Laird, John E, Newell, Allen, and Rosenbloom, Paul S. Soar: An architecture for general intelligence. *Artificial intelligence 33*, 1 (1987), 1–64.

[92] Lämmel, R., and Verhoef, C. Semi-automatic grammar recovery. *Softw. Pract. Exper. 31*, 15 (Dec. 2001), 1395–1448.

[93] Larkin, Jill H, McDermott, John, Simon, Dorothea P, and Simon, Herbert A. Models of competence in solving physics problems. *Cognitive science 4*, 4 (1980), 317–345.

[94] Laursen, S. L., Hassi M.-L. Kogan M. Weston T. J. Benefits for women and men of inquiry-based learning in college mathematics: A multi-institution study. *Journal of Research in Mathematics Education 45*, 4 (2014), 406–418.

[95] Lay, S.R. *Analysis: With an Introduction to Proof*. Prentice Hall, 2000.

[96] Levin, Leonid Anatolevich. Universal sequential search problems. *Problemy Peredachi Informatsii 9*, 3 (1973), 115–116.

[97] Lipton, Richard Jay. Why is discrete math hard to teach? `https://rjlipton.wordpress.com/2015/12/27/why-is-discrete-math-hard-to-teach/`, Dec. 2015. Accessed: 2019-05-01.

[98] Manktelow, K. I., and Evans, J. St B. T. Facilitation of reasoning by realism: Effect or non-effect? *British Journal of Psychology 70*, 4 (1979), 477–488.

[99] Marion, Jean-Yves. A type system for complexity flow analysis. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on* (2011), IEEE, pp. 123–132.

[100] Martelli, Alberto, and Montanari, Ugo. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst. 4*, 2 (Apr. 1982), 258–282.

[101] Martin, Philippe. Knowledge representation in CGLF, CGIF, KIF, frame-CG and formalized-English. In *International Conference on Conceptual Structures* (2002), Springer, pp. 77–91.

[102] McCarthy, John. Computer programs for checking mathematical proofs. *Recursive Function Theory, Proceedings of Symposia in Pure Mathematics 5* (1962).

[103] McCartin-Lim, Mark, Woolf, Beverly, and McGregor, Andrew. Connect the dots to prove it: A novel way to learn proof construction. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2018), SIGCSE '18, ACM, pp. 533–538.

[104] McCune, William. Prover9 manual version 2009-11a. `https://www.cs.unm.edu/~mccune/prover9/manual/2009-11A/`. Accessed: 2019-05-01.

[105] McDonald, James. The EXCHECK CAI system. *University-level Computer-assisted Instruction at Stanford: 1968-1980* (1981).

[106] McGraw-Hill. McGraw-Hill Education acquires Redbird Advanced Learning, a digital personalized learning provider for K-12. Press Release, September 2016.

[107] McMath, David, Rozenfeld, Marianna, and Sommer, Richard. A computer environment for writing ordinary mathematical proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning* (Berlin, Heidelberg, 2001), Robert Nieuwenhuis and Andrei Voronkov, Eds., Springer Berlin Heidelberg, pp. 507–516.

[108] Michalewicz, Zbigniew, and Michalewicz, Matthew. Puzzle-based learning. In *Proceedings of the 2007 AaeE Conference* (2007), pp. 1–8.

[109] Miller, Dale A. *Proofs in higher-order logic.* PhD thesis, Carnegie Mellon University, 1983.

[110] Mitkov, Ruslan. Outstanding issues in anaphora resolution. In *International Conference on Intelligent Text Processing and Computational Linguistics* (2001), Springer, pp. 110–125.

[111] Morrison, Briana B., Margulieux, Lauren E., and Guzdial, Mark. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (New York, NY, USA, 2015), ICER '15, ACM, pp. 21–29.

[112] Mostafavi, Behrooz, and Barnes, Tiffany. Determining problem selection for a logic proof tutor. In *Educational Data Mining 2013* (2013).

[113] Naumowicz, Adam, and Korniłowicz, Artur. A brief overview of Mizar. In *Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2009), Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, Eds., Springer Berlin Heidelberg, pp. 67–72.

[114] Newell, Allen. Production systems: Models of control structures. In *Visual information processing*. Elsevier, 1973, pp. 463–526.

[115] Newell, Allen. *Unified Theories of Cognition.* Harvard University Press, Cambridge, MA, USA, 1990.

[116] Newell, Allen, Shaw, J. C., and Simon, H. A. Empirical explorations of the logic theory machine: A case study in heuristic. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability* (New York, NY, USA, 1957), IRE-AIEE-ACM '57 (Western), ACM, pp. 218–230.

[117] Newell, Allen, and Simon, H.A. *Human problem solving.* Prentice-Hall, 1972.

[118] Nipkow, Tobias, Wenzel, Markus, and Paulson, Lawrence C. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[119] North Dakota State University, American Mathematical Society. Mathematics genealogy project.

[120] Paas, Fred G. Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. *Journal of educational psychology 84*, 4 (1992), 429.

[121] Paas, Fred GWC, and Van Merriënboer, Jeroen JG. Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach. *Journal of educational psychology 86*, 1 (1994), 122.

[122] Parsons, Dale, and Haden, Patricia. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52* (2006), Australian Computer Society, Inc., pp. 157–163.

[123] Pease, Adam, and Murray, William. An english to logic translator for ontology-based knowledge representation languages. In *Natural Language Processing and Knowledge Engineering, 2003. Proceedings. 2003 International Conference on* (2003), IEEE, pp. 777–783.

[124] Perkins, Douglas. Strategic proof tutoring in logic. Master's thesis, Carnegie Mellon University, 2007.

[125] Pfenning, Frank. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, Jan. 1987.

[126] Polk, T. A., and Newell, A. Modeling human syllogistic reasoning in Soar. In *The Soar Papers (Vol. 1)*, Paul S. Rosenbloom, John E. Laird, and Allen Newell, Eds. MIT Press, Cambridge, MA, USA, 1993, pp. 627–633.

[127] Post, Emil L. Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics 65*, 2 (1943), 197–215.

[128] Prawitz, Dag. *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, 1965.

[129] Prensky, Marc. *Digital Game-based Learning*. McGraw-Hill, 2001.

[130] Ravaglia, Raymond, Alper, Theodore, Rozenfeld, Marianna, and Suppes, Patrick. Successful pedagogical applications of symbolic computation. In *Computer-Human Interaction in Symbolic Computation*. Springer, 1998, pp. 61–88.

[131] Riazanov, Alexandre, and Voronkov, Andrei. The design and implementation of vampire. *AI Commun. 15*, 2,3 (Aug. 2002), 91–110.

[132] Rips, Lance J. Cognitive processes in propositional reasoning. *Psychological Review 90*, 1 (1983), 38–71.

[133] Robinson, Alan, and Voronkov, Andrei, Eds. *Handbook of Automated Reasoning*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[134] Robinson, J. A. A machine-oriented logic based on the resolution principle. *J. ACM 12*, 1 (Jan. 1965), 23–41.

[135] Scheines, Richard, and Sieg, Wilfried. An Experimental Comparison of Alternative Proof Construction Environments, August 1993.

[136] Schiller, Marvin R. G. *Granularity Analysis for Tutoring Mathematical Proofs*. PhD thesis, Saarland University, Saarbrücken, Germany, 2010.

[137] Schwitter, Rolf. English as a formal specification language. In *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on* (2002), IEEE, pp. 228–232.

[138] Schwitter, Rolf. The jobs puzzle: Taking on the challenge via controlled natural language processing. *Theory and Practice of Logic Programming 13*, 4-5 (2013), 487501.

[139] Schwonke, Rolf, Renkl, Alexander, Krieg, Carmen, Wittwer, Jörg, Aleven, Vincent, and Salden, Ron. The worked-example effect: Not an artefact of lousy control conditions. *Computers in Human Behavior 25*, 2 (2009), 258–266.

[140] Shahaf, Dafna, and Amir, Eyal. Towards a theory of AI completeness. In *In Proc. of 8th International Symposium on Logical Formalizations of Commonsense Reasoning (Commonsense'07), in AAAI Spring Sympo* (2007).

[141] Sieg, Wilfried. AProS: Background and evolution. `http://www.phil.cmu.edu/projects/apros/index.php?page=overview&subpage=background`. Accessed: 2019-05-01.

[142] Sieg, Wilfried, and Byrnes, John. Normal natural deduction proofs (in classical logic). *Studia Logica: An International Journal for Symbolic Logic 60*, 1 (1998), 67–106.

[143] Sieg, Wilfried, and Field, Clinton. Automated search for Gödel's proofs. *Annals of Pure and Applied Logic 133*, 1 (2005), 319 – 338. Festschrift on the occasion of Helmut Schwichtenberg's 60th birthday.

[144] Simon, Donald Lee. *Checking Number Theory Proofs in Natural Language*. PhD thesis, The University of Texas at Austin, 1990.

[145] Sinn, Moritz. *Automated Complexity Analysis for Imperative Programs*. PhD thesis, TU Wien, Faculty of Informatics, Wien, Austria, 2016.

[146] Sipser, Michael. *Introduction to the Theory of Computation*, 1st ed. International Thomson Publishing, 1996.

[147] Smith, R. L., Smith, N. W., and Rawson, F. L. Construct: In search of a theory of meaning. Tech. Rep. 238, Institute for Mathematical Studies in the Social Sciences, Stanford University, 10 1974.

[148] Smith, Robert L., and Blaine, Lee H. A generalized system for university mathematics instruction. In *Proceedings of the ACM SIGCSE-SIGCUE Technical Symposium on Computer Science and Education* (New York, NY, USA, 1976), SIGCSE '76, ACM, pp. 280–288.

[149] Stevenson, Andrew, and Cordy, James R. A survey of grammatical inference in software engineering. *Science of Computer Programming 96* (2014), 444 – 459. Selected Papers from the Fifth International Conference on Software Language Engineering (SLE 2012).

[150] Suppes, Patrick. *Axiomatic Set Theory.* Dover Publications, Inc., New York, 1972.

[151] Suppes, Patrick. Computer-assisted instruction at Stanford. In *Man and Computer: Proceedings of International Conference, Bordeaux, 1970.* Karger, Basel, 1972, pp. 298–330.

[152] Suppes, Patrick, Smith, Robert, and Beard, Marian. University-level computer-assisted instruction at stanford: 1975. *Instructional Science 6* (04/1977 1977), 151–185.

[153] Sutcliffe, Geoff. The CADE ATP system competition–CASC. *AI Magazine 37*, 2 (2016).

[154] Sweller, John, and Cooper, Graham A. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and instruction 2*, 1 (1985), 59–89.

[155] Sweller, John, Van Merrienboer, Jeroen JG, and Paas, Fred GWC. Cognitive architecture and instructional design. *Educational psychology review 10*, 3 (1998), 251–296.

[156] Taft, Darryl K. U.S. tops China in programming, but lags in math, logic. *eWeek* (Oct. 2011). `http://www.eweek.com/development/u.s.-tops-china-in-programming-but-lags-in-math-logic`. Accessed: 2019-05-01.

[157] Trafton, J Gregory, and Reiser, Brian J. The contributions of studying examples and solving problems to skill acquisitiony. In *Proceedings of the 15th Annual Conference of the Cognitive Society* (1993), pp. 1017–1022.

[158] Tucker, Allen B., Kelemen, Charles F., and Bruce, Kim B. Our curriculum has become math-phobic! In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2001), SIGCSE '01, ACM, pp. 243–247.

[159] Turing, Alan M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society 2*, 1 (1937), 230–265.

[160] Van Merriënboer, Jeroen JG. Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of educational computing research 6*, 3 (1990), 265–285.

[161] Van Merriënboer, Jeroen JG, and De Croock, Marcel BM. Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research 8*, 3 (1992), 365–394.

[162] van Rossum, Guido. Glue it all together with Python. In *Workshop on Compositional Software Architecture, 1998* (1998).

[163] VanLehn, K. *Mind Bugs: The Origins of Procedural Misconceptions.* A Bradford book. MIT Press, 1990.

[164] Verma, Rakesh M. A visual and interactive automata theory course emphasizing breadth of automata. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2005), ITiCSE '05, ACM, pp. 325–329.

[165] W3Schools.com. Browser display statistics. `https://www.w3schools.com/browsers/browsers_display.asp`. Accessed: 2019-05-01.

[166] Wack, Benjamin. *Typage et déduction dans le calcul de réécriture.* PhD thesis, Université Henri Poincaré-Nancy I, 2005.

[167] Wason, Peter C. Reasoning. In *New Horizons in Psychology*, B. Foss, Ed. Harmondsworth: Penguin Books, 1966, pp. 135–151.

[168] Wong, Wing-Kwong, Yin, Sheng-Kai, Yang, Hsi-Hsun, and Cheng, Ying-Hao. Using computer-assisted multiple representations in learning geometry proofs. *Journal of Educational Technology & Society 14*, 3 (2011).

[169] Woolf, Beverly Park. *Building Intelligent Interactive Tutors: Student-centered Strategies for Revolutionizing e-Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[170] Wos, Larry, and Pieper, Gail W. *A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[171] Xing, Cong-Cong. Proof diagrams: A graphical tool for assisting set proofs. *J. Comput. Sci. Coll. 22*, 5 (May 2007), 70–77.

[172] Xue, Ping, Poteet, Steve, Kao, Anne, Mott, David, and Braines, Dave. Constructing controlled english for both human usage and machine processing. *RuleML2013@ Challenge, Human Language Technology and Doctoral Consortium* (2013).

[173] Zinn, Claus. *Understanding informal mathematical discourse.* PhD thesis, Institut fur Informatik, Universitat Erlangen-Nurnberg, 2004.