Automatically Generating Problems and Solutions for Intelligent Tutoring Systems

A thesis submitted

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

by

Umair Z. Ahmed

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

March, 2019

CERTIFICATE

It is certified that the work contained in the thesis titled **Automatically Generating Problems and Solutions for Intelligent Tutoring Systems**, by **Umair Z. Ahmed**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare

Dr. Amey Karkare
As. Professor
Dept. of CSE, IIT Kanpur

Sumit

Dr. Sumit Gulwani
Adjunct Faculty
Dept. of CSE, IIT Kanpur
Research Manager
Microsoft Research Redmond

March, 2019

ABSTRACT

Name of student: **Umair Z. Ahmed** Roll no: **13111166**

Degree for which thesis is submitted: **Doctor of Philosophy**

Department: **Computer Science & Engineering**

Thesis title: **Automatically Generating Problems and Solutions for Intelligent Tutoring Systems**

Name of Thesis Supervisor:

1. **Dr. Amey Karkare**, Professor, Dept. of CSE, IIT Kanpur
2. **Dr. Sumit Gulwani**, Adjunct Faculty, Dept. of CSE, IIT Kanpur
   Research Manager, Microsoft Research Redmond

Month and year of thesis submission: **March, 2019**

The advent of Massive Open Online Courses (MOOCs) has made high quality education material accessible for all, at large scale and low cost. However, one of the important open challenges is in providing personalized help to students who are unable to solve a particular problem. Providing this personalized feedback manually is infeasible on a large scale and hence domain-specific Intelligent Tutoring Systems (ITS) are designed to automatically help guide the student.

In this thesis, we present a generic novel framework of ITS for automated (i) solution generation, where we generate the complete solution of a given problem; (ii) similar problem generation, where we search for other problems having solution similar to that of a given problem; and (iii) parameterized problem generation, where we create new problems satisfying given solution characteristics.

To achieve the above stated goals of solution and problem generation, we propose a novel 4-staged methodology of building ITS where (i) we first abstract the original problem into a symbolic representation, (ii) develop domain-specific system to solve this abstract problem, by (iii) leveraging offline computed data-structures, and finally

(iv) refine the abstract solution to the desired concrete solution.

We demonstrate this methodology by developing ITS targeting three diverse domains of (a) Natural deduction proofs for propositional logic, (b) Interesting starting positions of traditional board games, and (c) Compilation errors during introductory programming. Our usage of efficient offline symbolic computation achieves real-time solution and feedback generation, typically in the order of milli-seconds. Our modelling of problem generation as reverse of solution generation ensures that the hundreds of newly generated problems have the exact same abstract solution.

The solutions produced by our ITS are not only accurate, but also relevant for students; a metric often ignored in literature. We supplement the theoretical claim of relevance by deploying our introductory programming feedback tools during a live CS-1 (introductory to programming) course offering credited by 400+ novice first-year undergraduates at IIT-Kanpur. During this large scale randomized user study, we observed that students with access to feedback from our ITS tools are able to resolve programming errors faster, compared to the distinct set of students without similar access; with the performance advantage increasing with problem complexity. This benefit achieved is primarily logistical since students receiving feedback from ITS tools perform identical to human-tutored students when no feedback is provided.

Acknowledgements

First and foremost, I would like to begin by thanking God for giving me the strength and ability to satisfactorily complete one of the most interesting, challenging and rewarding journey of my life; and for introducing some amazing people along the way, without whose help and guidance this couldn't have been possible.

I would like to thank all my school and college teachers (including my parents, the very first teachers) who imparted invaluable lessons to me. Notably, my 10th class tution teacher Mr. Venkatesh, for introducing me to the "conceptual" way of learning; my 11th and 12th class computer-science (CS) teachers – Ms. Smitha and Ms. Rekha, for inspiring me to pursue an Under Graduate (UG) in CS, despite barely surviving a Java programming crash course during my 10th class; my UG teacher Dr. A. Srinivas, for motivating me to continue on my "quest for learning" by pursuing higher studies.

My very first encounter with building real world applications came during the extra-curricular courses organized by Manoj Bharadwaj, Manuel Paul and Karthik at my UG college. I am indebted to them for taking time off from their regular work and volunteering to make us better software engineers. Without their help, I would probably have been another run-of-the-mill engineer with text-book knowledge.

The credit for taking me "off-track" into the research world goes to Cohan Sujay Carlos, who introduced us to the fascinating research fields of statistical machine learning and natural language processing via a co-curricular UG course, while juggling with his startup at the same time. I would like to thank Dr. Monojit Choudhury and Dr. Kalika Bali for giving me an opportunity at Microsoft Research India (MSR-I),

and for introducing me to the research domain of Intelligent Tutoring System (ITS), which went on to become the topic of my doctoral research. My various interactions with the researchers and interns at MSR-I, one of the rare industrial research labs with an academic atmosphere, cemented my resolve to pursue a PhD.

I am indebted to Dr. Sumit Gulwani, my PhD co-supervisor, for believing in my (unproven) research potential and offering exciting research opportunities. Sumit's vision of using technology to aid students and instructors in educational setting convinced me to invest into designing effective tutoring systems as my PhD topic at IIT-Kanpur. Sumit's enthusiasm is infectious; whenever I got bogged down with the nitty-gritties after months of effort and rejections, and began to doubt the problem definition, our approach (or even myself), Sumit helped me see the bigger picture and made me feel like I am doing something worthwhile, making a difference.

I consider myself lucky to have had Dr. Amey Karkare as my PhD supervisor. Amey sir has been my strongest proponent right from day one of this long journey; he has always been approachable and supportive to me, during the good as well as difficult phases. Amey sir found the right balance between giving me sufficient leeway to explore, while gently guiding me towards the appropriate space/time. I look up to both my supervisors, who are not only great researchers but also amazing human beings. I am grateful to them for teaching me the necessary skills to become better at research, from selecting a good research problem to getting the most out of attending a research conference. Their imparted wisdom will continue to guide me for the rest of my research career, and in fact my life.

I would like to express my gratitude to various people and institutions who supported my research at IIT-Kanpur. Especially, my PhD thesis review committee for their valuable feedback; my research collaborators from whom I learnt a lot; IIT-Kanpur's Research-I and Microsoft Research India for supporting my scientific conference travels; IBM Research for awarding me a PhD fellowship to support my studies from 2017–2018; Dr. Amey Karkare and Rajdeep Das for successfully deploying the ambitious Prutor (online programming editor/tutor), that enabled

my research into programming ITS tools to provide instruction support for novice programmers; the 1000s of Prutor users (students, TAs, tutors and instructors) for making Prutor a success story and being patient with its technical snags during the early stages of development; Naman Bansal, my friend and project partner, for his critical support during my initial naive stage of PhD; Naman Bansal, Ayush Sekhari and Sanil Jain, the talented team of Logic-A (I regret that we could not take the project to its logical conclusion).

I would like to thank all my PhD student colleagues and friends, for helping me improve professionally as well as personally. The Hall-8 mohalla, for making me feel at home in a hostel. Tejas Gandhi, Sumit Kalra and the KD-109 lab for helping me navigate through the academic pitfalls. My brilliant KD-108 labmates and close friends, for tolerating my inquisitive (occasionally irritating) nature; Arpita Korwar and Rohit Gurjar, whose theoretical discussions were more insightful to me than entire courses; Sudhanshu Shukla, for being the industrious member of our lab, which encouraged me in turn to strive harder. Shubham Sahai, Garima Gaur, Abhishek Dang, Hrishikesh Terdalkar and Shubhangi Agarwal – thank you for being my family at IIT-Kanpur and not letting me miss my home. Our intense discussions on everything and anything to the random activities we pursued, helped me understand myself better and broadened my entire world view! My 5+ years of stay at IIT-Kanpur were enjoyable and memorable thanks to you all.

My Family and relatives have been a constant source of support, inspiring me to aim for greater heights. My maternal grandfather, Syed Ataulla, deserves a special mention for being my strongest advocate, celebrating my every big and small success.

Finally to my parents and Susheela, whom I cannot thank enough for staying by my side through this seemingly never ending arduous journey. You took care of me without complain, so that I can continue to focus on my research problems in peace and bring them to fruition. None of this would have been possible without your frequent sacrifices, unwavering love and constant support. I dedicate this dissertation to you three.

Contents

List of Tables

List of Figures

List of Publications

Large portions of the Chapters 2, 3, 4 and 5 have appeared in below publications. The work described in Chapter 6, a joint work with Dr. Amey Karkare, Dr. Nisheeth Srivastava (IIT Kanpur) and Dr. Renuka Sindhgatta (Queensland University of Technology), is under review and yet to be published.

[1] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. "Automatically Generating Problems and Solutions for Natural Deduction". In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI '13. Beijing, China: AAAI Press, 2013, pp. 1968–1975. ISBN: 978-1-57735-633-2. URL: http://dl.acm.org/citation.cfm?id=2540128.2540411.

[2] Umair Z. Ahmed, Krishnendu Chatterjee, and Sumit Gulwani. "Automatic Generation of Alternative Starting Positions for Simple Traditional Board Games". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI'15. Austin, Texas: AAAI Press, 2015, pp. 745–752. ISBN: 0-262-51129-0. URL: http://dl.acm.org/citation.cfm?id=2887007.2887111.

[3] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. "Compilation Error Repair: For the Student Programs, from the Student Programs". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET '18. Gothenburg, Sweden: ACM, 2018, pp. 78–87. ISBN: 978-1-4503-5660-2. DOI: 10.1145/3183377.3183383. URL: http://doi.acm.org/10.1145/3183377.3183383.

[4] Umair Z. Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. "Targeted Example Generation for Compilation Errors". In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2019. IEEE/ACM, 2019.

Chapter 1

Introduction

The advent of Massive Open Online Courses (MOOCs) has made high quality education material accessible for all, at large scale and low cost. However, one of the important open challenges in education at scale lies in providing personalized information to students solving a specific problem [1]. Given a student's attempt at a problem, this personalized information can vary from a binary correct/incorrect indicator to more involved feedback that hints at a solution. Manually generating such personalized information for every student is infeasible on a large scale, and hence domain-specific Intelligent Tutoring Systems (ITS) are designed to automate these tasks.

Large number of Intelligent Tutoring Systems (ITS) have been proposed in literature targeting variety of domains, ranging from elementary school subjects to advanced engineering courses. Since the concepts covered and the mistakes made by the students are often similar across course offerings, ITS tools can be designed to help automate repetitive tasks in education such as grade student submissions [2, 3], clarify frequent student misconceptions [4], generate practice problems [1], and generate feedback for incorrect solutions [5].

1.1 Problem and Solution Generation

Multiple instructional principles have been evaluated for efficient learning in literature, based on the complexity of the educational content [6]. Providing timely feedback is one such important instructional principle, and studies on tutoring systems indicate significant learning benefit when students are provided with some form of feedback [7].

Providing relevant feedback to students goes beyond indicating if their solution is correct or incorrect. Students with incorrect attempts prefer (and perform better) when provided with conceptual hints and counter-example based feedback (that indirectly reveals relevant solutions), over binary (correct/incorrect) feedback [5].

Similarly, when a student makes an incorrect attempt, providing worked examples for similar difficulty problems is a commonly recommended instructional principle for efficient learning [8]. Automatically generating these domain specific conceptual solutions and similar difficulty problems remains a challenging task.

In this thesis, we focus on the following repetitive tasks in education

1. *Solution Generation*: Given a student's incorrect attempt, the goal of ITS is to automatically generate a closest correct solution to the problem. This is important for several reasons. First, the generated solution can be used to provide feedback on the steps to fix the student's incorrect attempt. Second, it can be used to automatically grade a student's attempt, based on its distance from the closest correct solution. Third, it can be used to generate sample solutions for automatically generated problems, and hence determine their hardness level.

2. *Similar Problem Generation*: Given a student's incorrect attempt, new practice problems can be generated with solution similar to that of the original problem. Students can be then expected to learn from these example problems and their solutions, and transfer the knowledge to fix their original incorrect solution.

3. *Parameterized Problem Generation*: Given solution characteristics as input by

the instructor, ITS can automatically create new problems satisfying them, thereby saving teacher's time and effort on a tedious task. Additionally, these fresh problems can be used to prevent plagiarism in assignments where students can be provided with different problems of same difficulty level.

Generating problems of a specified difficulty level that exercise a given set of concepts can help create *personalized workflows* for students. On solving a problem correctly, the student may be presented with another problem that is more difficult than the previous one, or exercises a richer set of concepts. Conversely, a simpler problem can be presented to the student who fails to solve the previous one.

Recognizing this need, various ITS tools have been proposed to generate personalized feedback through means of solution generation, targeting K-8 arithmetic problems [9], high school geometry constructions [10], deterministic finite automata constructions [5], syntax errors in introductory programming [11], logical errors in introductory programming [12].

Similarly, many automated problem generation tools exist in literature, targeting various domains such as algebraic equations [13], high school trigonometry and calculus [14], high school geometry proofs [15], mathematical word problems [16], deterministic finite automata constructions [5], compilation errors in programming [17].

Challenges

Developing such ITS tools is a challenging task for several reasons.

- The systems are domain-specific, requiring intricate knowledge of the domain.
- The feedback provided should be largely automated, requiring minimal human intervention by expert instructor or novice student, for the ITS to scale at large.
- The feedback must be provided in real-time to help students resolve their errors in a live setting.

- The feedback provided should be relevant to students' specific mistake and desired solution.
- The new practice problems generated should be relevant to students' specific mistake and the difficulty level of original problem.
- Measuring the pedagogical value provided by such ITS is non-trivial, requiring usage of feedback tools by large number of students for long periods of time in real world setting.

In this thesis, we address the above challenges by proposing a novel framework of designing ITS, and developing problem and solution generation tools for three different diverse domains utilizing it. Our design methodology ensures live automated solution generation that works in real-time, and our modeling of problem generation as reverse of solution generation enables large number of relevant examples. Finally, we report on a large scale user study to measure the benefit of providing problem and solution based feedback to novice students.

1.2 Novel Framework for ITS Design

We propose a generic novel framework of developing ITS for problem and solution generation, consisting of the following common core components:

1. *Abstraction*: The original problem is first abstracted into a symbolic representation. Multiple different problems map to the same representation (many-to-one), in order to reduce the solution search space.
2. *Abstract Solver*: Accurate solvers are then developed to solve the abstract problem, which search for an abstract solution in the symbolic search space with the help of offline computed data structures.
3. *Offline Computed Data structure*: The symbolic reasoning required to solve the abstract problem is computed and stored in an offline phase. This reduces the

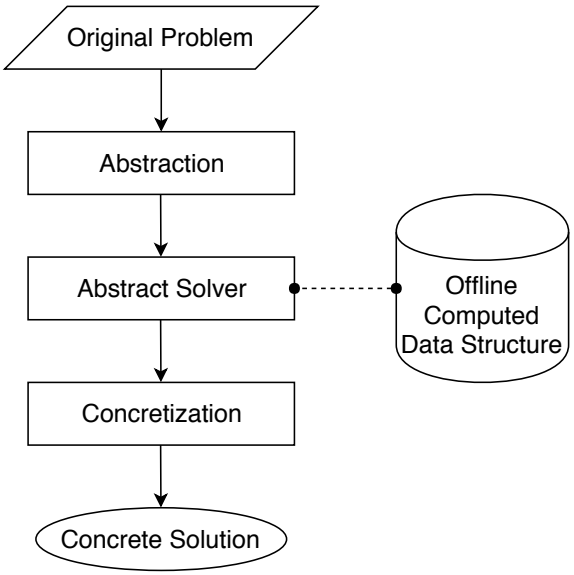


Figure 1.1: Solution Generation Framework

cost of abstract solver at runtime, and helps generate problems by performing reverse lookup on it.

4. *Problem Search*: A reverse search is performed on the offline computed data structure, where new abstract problems are generated having the same abstract solution as given input problem (or input characteristics).

5. *Concretization*: All representations in the abstract solution are replaced with concrete instantiations, to obtain the concrete solution to original problem. Concretization is an approximate process which is reverse of abstraction, where a single abstract representation can be replaced with multiple concrete ones (one-to-many).

Our framework enables real-time solution generation through usage of novel two-phased methodology that first searches for an abstract solution and then refines it to a concrete solution, combined with offline computed abstract reasoning. Figure 1.1 displays this arrangement of different *solution generation* components. Our *Similar problem generation* module involves first solving the given problem using solution generation module, followed by searching for new problems with similar solution in the offline computed data structure. This framework is shown in Figure 1.2. In the case of *parameterized problem generation*, the offline computed data structure

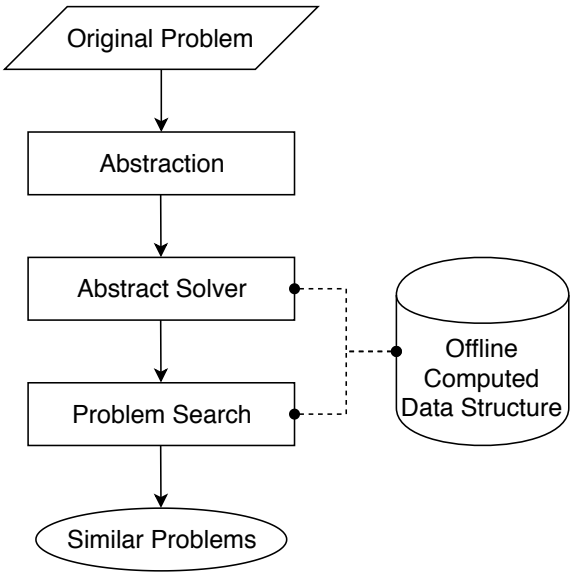


Figure 1.2: Similar Problem Generation Framework

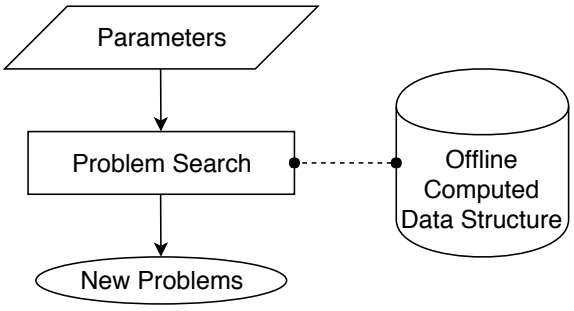


Figure 1.3: Parameterized Problem Generation Framework

Step	Proposition	Reason
P1	$a \lor (b \land c)$	Premise
P2	$a \rightarrow d$	Premise
P3	$d \rightarrow e$	Premise
1	$(a \lor b) \land (a \lor c)$	P1, Distribution
2	$a \lor b$	**1, Simplification**
3	$a \rightarrow e$	**P2, P3, Hypothetical Syllogism**
4	$b \lor a$	2, Commutativity
5	$\neg\neg b \lor a$	4, Double Negation
6	$\neg b \rightarrow a$	5, Implication
7	$\neg b \rightarrow e$	**6, 3, Hypothetical Syllogism**
8	$\neg\neg b \lor e$	7, Implication
C	$b \lor e$	8, Double Negation

Table 1.1: Natural deduction proof example, with application of inference rules highlighted in bold. The proof was generated by our solution generation module.

Premise 1	Premise 2	Premise 3	Conclusion
$a \equiv b$	$c \rightarrow \neg b$	$(d \rightarrow e) \rightarrow c$	$a \rightarrow (d \land \neg e)$

Table 1.2: New natural deduction problem generated by our ITS, having same abstract proof (bold highlighted steps) as Table 1.1.

is searched for new problems that satisfy the solution characteristics provided by instructor, as shown in Figure 1.3.

1.3 Our Intelligent Tutoring Systems

We developed ITS for three diverse domains, utilizing the generic framework described earlier. The solution generation modules of our ITS are not only accurate, but also relevant to the student's problem; a metric often ignored in literature. The problem generation modules can generate a large number of new problems having the desired solution.

Due to the domain-specific nature of each ITS, we cite relevant related work in their corresponding chapters.

1.3.1 ITS #1: Natural deduction for propositional logic

In this work, presented in Chapter 2 in detail, we developed an ITS to enable computer aided education for natural deduction [18]. Natural deduction is a method for establishing validity of propositional type arguments, where a conclusion is derived from a set of premises through a series of inference and replacement rule applications. It is typically taught as part of introductory course on logic.

Table 1.1 shows a natural deduction proof generated by our ITS, where conclusion C is derived from premises P1, P2 and P3. The solution generation module involves

i) abstracting the propositions with their truth table bit vector representation,

ii) computing an offline data structure called Universal Proof Graph (UPG) which records all possible inference rule applications on the abstractions,

iii) performing forward graph traversal search on the data structure, starting from premises until the conclusion is reached, to obtain an abstract solution to the problem, and lastly,

iv) refining the abstract proof to the desired concrete natural deduction proof by inserting appropriate replacement rule transformations.

On 279 benchmark problems obtained from 5 different logic textbooks, our ITS could solve 84% of them within 10 seconds.

Our ITS can also automatically generate hundreds of new problems having similar solution to these benchmark problems, within few minutes. For example, our ITS was able to automatically create 516 new natural deduction problems, whose proof requires the exact same inference rule applications (abstract solution) as the original problem in Table 1.1, albeit with different set of replacement rule transformations. These similar problems were obtained by performing a reverse *problem search* of abstract solution on the same *offline computed data structure* (UPG). Table 1.2 shows one such problem generated by our ITS, which can be provided as an assignment or a practice problem for students.

O				O
X		X		X
X		O		X
X	X	O	X	O
O	O	O	X	O

Table 1.3: Interesting starting board positions generated by our tool, for CONNECT-4 board game. Player 1 (X) has a guaranteed path to victory in 3 moves, if played efficiently.

1.3.2 ITS #2: Alternative starting positions for board games

Traditional board games, such as Tic-Tac-Toe and CONNECT-4 which have been taught to children for generations, play an important role in development of logical skills and social development. The traditional research question in such games has focused on who is the winner and what are the optimal strategies, when starting from the default empty board state. In this work, described in Chapter 3 in detail, we instead focus on generating alternative starting positions, which can be useful for teaching new board games to novitiates and make the plays more interesting.

Table 1.3 shows an interesting starting board positions for CONNECT-4, which requires 4 consecutive pieces in row, column or diagonal to win. In this board position, which was automatically generated by our tool, Player 1 (X) can win in 3 moves by playing optimally, irrespective of the choices made by Player 2 (O). Providing such fresh start states with customizable hardness and length of play, where strategies cannot be memorized, can motivate children in practice and mastery of the board game.

The problem generation module for this domain is built using the generic framework described earlier. Given solution characteristics as input, such as board game rules (board size, valid moves, winning condition) and #steps-to-win (j)

i) we first abstract all winning board position W_0 with a boolean formula,

ii) construct an offline Binary Decision Diagram (BDD) representing the set of all board positions W_j from which Player 1 (X) can win in j-moves,

iii) sample starting positions from W_j and search for interesting hard ones based on

```
1 #include<stdio.h>
2 int main(){
3    int x,x1,d;
4    // ...
5    d=(x-x1) (x-x1);
6    return d;
7 }
```

```
1 #include<stdio.h>
2 int main(){
3    int x,x1,d;
4    // ...
5    d=(x-x1)*(x-x1);
6    return d;
7 }
```

Figure 1.4: Erroneous program attempt by student (left), and its automatically generated repair (right) by our ITS. The compiler message read: *Line-5, Column-11: error: called object type 'int' is not a function or function pointer.*

Line#	Erroneous Example	Repaired Example
5	`amount = P (1 + T*R/100);`	`amount = P*(1 + T*R/100);`

Figure 1.5: Example problem suggested by our ITS, which suffers from the same error and desires the same repair as program in Figure 1.4.

winning probability of Min-Max simulation.

Our proposed problem generation methodology is applicable for all directed graph board games, and finds hundreds of interesting starting positions for various board games. As an example, for CONNECT-4 5×5 board game which has more than 69 million unique board positions, our tool found within few hours of runtime more than 200 interesting positions from which Player 1 can win in 4 moves by playing optimally. These 200+ positions are interesting since they are hard to win even with a depth-3 look ahead strategy. Since these number of interesting positions are a very small percentage of overall state space, any other enumerative approach is intractable on such large graph games.

1.3.3 ITS #3: Introductory programming compilation errors

Introductory programming is one of the most popular course offered, with the class sizes reaching more than 1000 students in some universities. While attempting programming assignments, novice students often encounter compile-time errors due to incorrect usage of syntax and types. Although compilers locate and report compilation errors in the program, their error messages are targeted towards expert programmers and hence often appear cryptic to beginners who struggle in resolving

them.

Figure 1.4 demonstrates an erroneous program, where the student has missed an asterisk "*" operator between round parenthesis. The compiler error message, which treats this error as an incorrect function invocation, is cryptic for beginners in understanding the mistake or the required fix. Our solution generation ITS, presented in Chapter 4, automatically generates the required "*" operator at the correct position.

The repair solution is obtained as follows

i) The program specific tokens (variables/literals) are first abstracted with their generic type tokens.

ii) An offline Recurrent Neural Network (RNN) is trained to predict a syntactically correct sequence of abstract tokens.

iii) Our abstract solver then locates suspicious abstract sequences and translates them to repaired abstract sequences (Seq2Seq), using the offline computed RNN.

iv) The resultant repaired sequence of abstract tokens is concretized, by replacing generic type tokens with specific variables/literal instantiations, to obtain the desired program repair.

The repairs performed by our solution generation ITS are not only accurate, but also relevant to actual repair desired by student. The deep network is trained on 15,000+ code pairs and achieves about 80% error repair accuracy on 4,500+ held out test set, with its top-3 predictions containing the exact same repair as students' for 74% of the test cases.

Given an erroneous program as input, our problem generation module

i) First abstracts it with their generic types.

ii) Locates suspicious abstract sequences and labels them with an error-repair class, using an offline trained *dense neural network* classifier, signifying the mistake made and fix desired by student.

Generation Stage	Chapter 2 Natural Deduction	Chapter 4 Compilation Error
Original problem (Input)	Premises and conclusion propositions	Erroneous program failing to compile
Abstraction	Propositions to truth-table bit-vectors	Program tokens to generic types
Offline computed data-structure	Universal Proof Graph (UPG)	Recurrent Neural Network (RNN)
Abstract Solver	Forward search on UPG	Sequence-to-Sequence prediction
Concretization	Abstract proof to concrete proof	Abstract repair to concrete repair
Solution (Output)	Natural deduction proof	Repaired program which compiles successfully

Table 1.4: Solution Generation Module

iii) Searches the 15,000+ training set repository for erroneous-repaired code pairs that belong to the same error-repair class predicted earlier. The top frequent such examples are then suggested as feedback to the student.

Our problem generation ITS, described in Chapter 5, can automatically suggest dozens of example programs given an erroneous program as input, within few milliseconds. On a held out test-set of 3,000+ programs, our ITS achieves 98% accuracy in correctly predicting the error-repair class in its top-3 prediction, and hence in suggesting the corresponding relevant examples. Figure 1.5 shows an example problem generated by our ITS, which suffers from the same error and desires same repair as program in Figure 1.4.

Summary

Tables 1.4, 1.5, and 1.6 summarize the unifying framework and the corresponding components used in ITS design for all three domains to enable *solution generation*, *similar problem generation*, and *parameterized problem generation* respectively.

Generation Stage	Chapter 2 Natural Deduction	Chapter 5 Compilation Error
Original problem (Input)	Premises and conclusion propositions	Erroneous program failing to compile
Abstraction	Propositions to truth-table bit-vectors	Program tokens to generic types
Offline computed data-structure	Universal Proof Graph (UPG)	Dense neural network
Abstract solver	Forward search on UPG	Error-Repair class prediction
Problem search	Backward search on UPG	Student code repository search
Similar problems (Output)	Propositions with similar abstract solution	Programs with similar compilation error and repair

Table 1.5: Similar Problem Generation Module

Generation Stage	Chapter 2 Natural Deduction	Chapter 3 Board Games
Parameters (Input)	#variables, #premises, maximum formula size, rule-set and #solution-steps	board size, valid moves, winning condition, #steps-to-win
Offline computed data-structure	Universal Proof Graph (UPG)	Binary Decision Diagram (BDD)
Problem search	Backward search on UPG	Sample vertices of specified hardness level
New problems (Output)	Propositions of specified characteristics	Interesting starting board positions of specified characteristics

Table 1.6: Parameterized Problem Generation Module

1.4 A Large Scale User Study for ITS #3

We supplement the theoretical claim of our feedback tools being relevant by describing results from deployment of our programming feedback tools (ITS #3) during a live CS-1 (introduction to programming) course credited by 400+ novice first-year undergraduates at IIT-Kanpur, a large public university. Students of the 2017–2018–II semester course offering were randomly partitioned to receive feedback from our two automated feedback tools of solution generation and example problem generation. While students from previous semester offerings, where human teaching assistants provided feedback, are used as baseline.

During this large scale randomized user study, we observed that students with access to feedback from our ITS tools are able to resolve programming errors more efficiently, compared to the distinct set of students without similar access; with the performance advantage increasing with problem complexity. However, the benefit achieved is primarily logistical since students receiving feedback from ITS tools performed identical to human-tutored students when no form of feedback (automated or manual) was provided. We also observed that feedback via repair solution and example problem have distinct non-overlapping relative advantages, for different categories of programming errors. Chapter 6 discusses these results in detail.

In the following Chapter 2, we describe our very first ITS #1 for natural deduction proof, based on the design principles mentioned earlier.

Chapter 2

Natural Deduction

Natural deduction, a method for establishing validity of propositional type arguments, helps develop important reasoning skills and is thus a key ingredient in a course on introductory logic. In this chapter we present two core components, namely solution generation and practice problem generation, for enabling computer-aided education for this important subject domain [18].

The key enabling technology used is an offline-computed data-structure called *Universal Proof Graph* (UPG) that encodes *all* possible applications of inference rules over all *small* propositions abstracted using their *bitvector-based truth-table representation*. This allows an efficient forward search in UPG for solution generation. More interestingly, this allows generating fresh practice problems of given solution characteristic by performing a backward search in UPG.

We report results on 279 natural deduction problems, obtained from various textbook exercises. Our solution generation procedure can solve many more problems, compared to the traditional forward-chaining based procedure. While our problem generation procedure can efficiently generate several variants with desired characteristics.

2.1 Introduction

Natural deduction [19] is a method for establishing the validity of propositional type arguments, where the conclusion of an argument is derived from the initial premises through a series of discrete steps. A proof in natural deduction consists of a sequence of propositions; each of which is either a premise or is derived from preceding propositions through application of some inference/replacement rule, and the last of which is the conclusion of the argument.

From a pragmatic perspective, natural deduction helps develop reasoning skills required to construct sound arguments and to evaluate the arguments of others. It instills a sensitivity towards formal component in language, a thorough command of which is necessary for clear, effective, and meaningful communication. Such a logical training provides a fundamental defense against the prejudiced attitudes that threaten the foundations of our democratic society [20].

From a pedagogical perspective, natural deduction gently introduces the usage of logical symbols, which carries forward into other more difficult fields such as algebra, geometry, physics, and economics. Natural deduction is typically taught as part of an introductory course on logic, which is a central component of college education and is generally offered to students of all disciplines regardless of their major. It is thus unsurprising that an introductory course on logic is listed on various Massive Open Online Course (MOOC) platforms, including Coursera [21], Open Learning Initiative [22], and Khan Academy [23].

2.1.1 Novel Technical Insights

Our observations in this work include:

- **Small-Sized Hypothesis**: Propositions that occur in educational contexts use a small number of variables and have a small size (Table 2.9). The number of such small-sized propositions is bounded (Figure 2.6).
- **Truth-Table Representation**: A proposition can be abstracted using its

truth-table, which can be represented using a bitvector representation [24]. This provides three key advantages: (i) It partitions small-sized propositions into a small number of buckets (Figure 2.7). (ii) It reduces the size/depth of a natural deduction proof tree, making it easier to search for solutions or generate problems with given solution characteristics. (iii) Application of inference rules over bitvector representation reduces to performing efficient bitwise operations.

- **Offline Computation**: The symbolic reasoning required to pattern match propositions for applying inference rules can be performed (over their truth-table bitvector representation) and stored in an offline phase. This has two advantages: (i) It alleviates the cost of symbolic reasoning by a large constant factor, by removing the need to perform any symbolic matching at runtime. (ii) It enables efficient backward search for appropriate premises starting from a conclusion during problem generation, which we model as a *reverse of solution generation*.

2.1.2 Contributions

This work makes the following contributions.

- We propose leveraging the following novel ingredients for building an efficient computer-aided education system for natural deduction: small-sized proposition hypothesis, truth-table based representation, and offline computation.
- We present a novel two-phased methodology for solution generation that first searches for an abstract solution and then refines it to a concrete solution.
- We motivate and define some useful goals for problem generation, namely similar problem generation and parameterized problem generation. We present a novel methodology for generating such problems using a process that is reverse of solution generation.
- We present detailed experimental results on 279 benchmark problems collected from various textbooks. Our solution generation algorithm can solve 84% of

Rule Name	Premise-1	Premise-2	Conclusion
Modus Ponens (MP)	$p \rightarrow q$	p	q
Modus Tollens (MT)	$p \rightarrow q$	$\neg q$	$\neg p$
Hypothetical Syllogism (HS)	$p \rightarrow q$	$q \rightarrow r$	$p \rightarrow r$
Disjunctive Syllogism (DS)	$p \vee q$	$\neg p$	q
Constructive Dilemma (CD)	$(p \rightarrow q) \wedge (r \rightarrow s)$	$p \vee r$	$q \vee s$
Destructive Dilemma (DD)	$(p \rightarrow q) \wedge (r \rightarrow s)$	$\neg q \vee \neg s$	$\neg p \vee \neg r$
Simplification (Simp)	$p \wedge q$		q
Conjunction (Conj)	p	q	$p \wedge q$
Addition (Add)	p		$p \vee q$

Table 2.1: Inference Rules

these problems, while the baseline traditional algorithm could only solve 57% of these problems. Our problem generation algorithm is able to generate few thousands of *similar problems* and *parameterized problems* on average per instance in a few minutes.

2.2 Problem Definition

Let $x_1, \ldots, x_n$ be n Boolean variables. A *proposition* over these Boolean variables is a Boolean formula consisting of Boolean connectives over these `variables`.

Definition 2.1 (Natural Deduction Problem) *A natural deduction problem is a pair* $(\{p_i\}_{i=1}^m, c)$ *of a set of propositions* $\{p_i\}_{i=1}^m$ *called* premises *and a proposition* c *called* conclusion. *A natural deduction problem is said to be well-defined if the conclusion is implied by all the premises, but not by any strict subset of those premises.*

Definition 2.2 (Natural Deduction Proof) *Let* $\mathcal{I}$ *and* $\mathcal{R}$ *be sets of inference rules and replacement rules respectively. A natural deduction proof for a problem* $(\{p_i\}_{i=1}^m, c)$ *is a step-by-step derivation of conclusion* c *from premises* $\{p_i\}_{i=1}^m$*, obtained through application of some inference rule from* $\mathcal{I}$ *or some replacement rule from* $\mathcal{R}$ *at each step.*

Rule Name	Proposition	Equivalent Proposition
De Morgan's Theorem	$\neg (p \wedge q)$	$\neg p \vee \neg q$
	$\neg (p \vee q)$	$\neg p \wedge \neg q$
Commutation	$p \vee q$	$q \vee p$
	$p \wedge q$	$q \wedge p$
Association	$p \vee (q \vee r)$	$(p \vee q) \vee r$
	$p \wedge (q \wedge r)$	$(p \wedge q) \wedge r$
Distribution	$p \vee (q \wedge r)$	$(p \vee q) \wedge (p \vee r)$
	$p \wedge (q \vee r)$	$(p \wedge q) \vee (p \wedge r)$
Double Negation	p	$\neg \neg p$
Transposition	$p \rightarrow q$	$\neg q \rightarrow \neg p$
Implication	$p \rightarrow q$	$\neg p \vee q$
Equivalence	$p \equiv q$	$(p \rightarrow q) \wedge (q \rightarrow p)$
	$p \equiv q$	$(p \wedge q) \vee (\neg p \wedge \neg q)$
Exportation	$(p \wedge q) \rightarrow r$	$p \rightarrow (q \rightarrow r)$
Tautology	p	$p \vee p$
	p	$p \wedge p$

Table 2.2: Replacement Rules

Inference and Replacement Rules

An *inference rule* $\boldsymbol{I}$ can be applied on one or more premises $\boldsymbol{Premises(I)}$ to derive a new intermediate proposition $\boldsymbol{Conclusion(I)}$. Table 2.1 lists down the inference rules commonly used in natural deduction textbook proofs. Unlike the inference rules, which are basic argument forms, a *replacement rule* is expressed as pair of logically equivalent statement forms, either of which can replace the other in a proof sequence. Table 2.2 lists down the common replacement rules.

Our system does not leverage any knowledge specific to one inference/replacement rule. The only interface that it requires of a rule is the capability to generate the target proposition from source propositions.

Example 2.3 *Consider the natural deduction problem* ($\{a \vee (b \wedge c),\ a \rightarrow d,\ d \rightarrow e\},\ b \vee e$)*. Table 2.3 shows a natural deduction proof for it with inference rule applications in bold.*

Step	Proposition	Reason
P1	$a \lor (b \land c)$	Premise
P2	$a \rightarrow d$	Premise
P3	$d \rightarrow e$	Premise
1	$(a \lor b) \land (a \lor c)$	P1, Distribution
2	$a \lor b$	**1, Simplification**
3	$a \rightarrow e$	**P2, P3, Hypothetical Syllogism**
4	$b \lor a$	2, Commutativity
5	$\neg\neg b \lor a$	4, Double Negation
6	$\neg b \rightarrow a$	5, Implication
7	$\neg b \rightarrow e$	**6, 3, Hypothetical Syllogism**
8	$\neg\neg b \lor e$	7, Implication
9	$b \lor e$	8, Double Negation

Table 2.3: Natural deduction proof example, with application of inference rules highlighted in bold

2.3 Universal Proof Graph

We start out by describing our key data-structure that is used for both solution and problem generation. It encodes all possible applications of inference rules over all propositions of small size, abstracted using their truth-table representation.

Definition 2.4 (Truth-Table Bitvector Representation) *Let q be a proposition over n Boolean variables. Its truth-table, which assigns a Boolean value to each of the 2^n possible assignments to the n Boolean variables, can be represented using a 2^n bitvector [24]. We denote this bitvector by $\tilde{q}$.*

For example, consider the proposition $a \rightarrow b$ over $n = 3$ Boolean variables $\{a, b, c\}$. Table 2.4 lists its truth-table, by assigning $2^3 = 8$ different possible Boolean value assignments to the 3 variables. Note that a truth-table representation does not distinguish between equivalent propositions such as a and $\neg\neg a$.

Definition 2.5 (Abstract Proof) *Let $\mathcal{I}$ be a set of inference rules. An abstract proof tree for a natural deduction problem $(\{p_i\}_{i=1}^m, c)$ is any step-by-step deduction for deriving $\tilde{c}$ from $\{\tilde{p}_i\}_{i=1}^m$ using the abstract version of some inference rule from $\mathcal{I}$ at each step. An abstract version of an inference rule $\tilde{I}$ has premises $\{\tilde{q} \mid q \in \mathit{Premises}(I)\}$*

a	b	c	$a \rightarrow b$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1
15	51	85	243

Table 2.4: Truth Table example of proposition $a \rightarrow b$ over $n = 3$ Boolean variables $\{a, b, c\}$. The truth-table consists of $2^3 = 8$ Boolean value assignments, with the last row containing the 8-bit integer equivalent of the bit-vector representation; the bottom-most row represents the Least Significant Bit (LSB).

Step	**Truth-table**	**Reason**
P1	$\eta_1 = 1048575$	Premise
P2	$\eta_2 = 4294914867$	Premise
P3	$\eta_3 = 3722304989$	Premise
1	$\eta_4 = 16777215$	**P1, Simplification**
2	$\eta_5 = 4294923605$	**P2, P3, Hypothetical Syllogism**
3	$\eta_6 = 1442797055$	**1, 2, Hypothetical Syllogism**

Table 2.5: Abstract proof with truth-tables shown using a 32-bit integer representation

and conclusion $\tilde{q'}$*, where* $q' = \mathit{Conclusion}(I)$*. An abstract proof* A *is* minimal *if for every other abstract proof* A'*, the set of truth-tables derived in* A' *is not a strict subset of* A*.*

An abstract proof for a natural deduction problem is smaller than a natural deduction proof. This is because an abstract proof operates on truth-tables, i.e., equivalence classes of propositions, and hence does not need to encode the replacement steps to express equivalent rewrites within a class. Note that a natural deduction proof for a given problem, over inference rules $\mathcal{I}$ and replacement rules $\mathcal{R}$, can always be translated into an abstract proof for that problem over $\mathcal{I}$. However, translating an abstract proof into natural deduction proof depends on whether or not $\mathcal{R}$ contains sufficient replacement rules.

Example 2.6 *Consider the problem mentioned in Example 2.3 and its natural*

n	Total	s				
		1	2	3	4	5
1		2	10	1.3×10^2	2.4×10^3	4.6×10^4
2		4	52	1.5×10^3	5.9×10^4	2.5×10^6
3	∞	6	126	5.8×10^3	3.3×10^5	2.1×10^7
4		8	232	1.4×10^4	1.1×10^6	9.8×10^7
5		10	370	2.9×10^4	2.8×10^6	3.1×10^8

Table 2.6: Number of propositions over n variables and size at most s, excluding those with double or more negations.

n	Total	s				
		1	2	3	4	5
1	4	2	4	4	4	4
2	16	4	16	16	16	16
3	256	6	38	152	232	256
4	65,536	8	70	526	3,000	13,624
5	4,294,967,296	10	112	1,252	12,822	122,648

Table 2.7: Number of truth-tables over n variables and size at most s

deduction proof in Table 2.3. An abstract proof for the same problem is shown in Table 2.5, which consists only of inference rules from $\mathbf{I}$ *that were highlighted in bold in its natural deduction proof, along with interpreting each proposition* q *as its truth-table* $\tilde{q}$*.*

Definition 2.7 (Size of a Proposition) *We define* $Size(q)$*, the size of a proposition* q*, as the number of variable occurrences (operands) in it.*

$$\begin{aligned} Size(x) &= 1 \\ Size(\neg q) &= Size(q) \\ Size(q_1 \ \ op \ \ q_2) &= Size(q_1) + Size(q_2) \end{aligned}$$

For example, the size of proposition $(a \wedge b) \vee (\neg a \wedge c)$ is 4.

Let $P_{n,s}$ denote the set of all propositions over n variables that have size at most s. Table 2.6 shows the number of propositions over n variables as a function of s. Note that, even though the number of syntactically distinct propositions over n variables is potentially infinite, the number of such propositions that have constant size is bounded.

For example, the 10 distinct propositions for $n = 1$ variable and $s = 2$ size are $a,\ \neg a,\ a \wedge \neg a,\ \neg a \wedge a,\ a \vee \neg a,\ \neg a \vee a,\ a \rightarrow \neg a,\ \neg a \rightarrow a,\ a \equiv \neg a,\ \neg a \equiv a$.

Definition 2.8 (Size of a Truth-Table) *We define the size of a truth-table t to be the smallest size of a proposition q such that $\tilde{q} = t$. We select one such smallest-sized proposition as the* ***canonical proposition*** *for truth-table t.*

Let $T_{n,s}$ denote the set of all truth-tables over n variables that have size at most s. Table 2.7 mentions statistics related to the number of truth-tables in $T_{n,s}$ for various values of n and s. Note that the cardinality of $T_{n,s}$ is much smaller than the cardinality of $P_{n,s}$.

For example, although $n = 1$ variable and $s = 2$ size has $P_{1,2} = 10$ unique propositions, its number of unique truth tables is restricted to $T_{1,2} = 4$; specifically, the truth-tables $\{0, 1, 2, 3\}$ represented using 2-bit integer for canonical propositions $\{a \wedge \neg a,\ a,\ \neg a,\ a \vee \neg a\}$ respectively.

Definition 2.9 (Universal Proof Graph (UPG)) *A $(n, s, \mathcal{I})$ Universal Proof Graph is a hyper-graph whose nodes are labeled with truth-tables from $T_{n,s}$ and edges are labeled with an inference rule from $\mathcal{I}$.*

2.3.1 Algorithm

We now describe our algorithm for computing the $(n, s, \mathcal{I})$-UPG, which makes use of functions $Eval$ and $EvalS$ defined later.

1. The node set of $(n, s, \mathcal{I})$-UPG is $T_{n,s}$, the set of all truth-tables of size s. We compute $T_{n,s}$ by enumerating all propositions q of size s over n variables and adding $\tilde{q}$ to the node set. During this computation, we also maintain a reverse mapping called $Canonical$ that maps each truth-table to a canonical proposition.

2. The edge set computation involves adding a hyper-edge $\mathcal{E}$ for each inference rule $I \in \mathcal{I}$, for each state ρ that maps free variables in I to a truth-table in $T_{n,s}$.

For each inference rule $\boldsymbol{I} \in \boldsymbol{\mathcal{I}}$, where p_i is the i^{th} premise in $\boldsymbol{Premises}(\boldsymbol{I})$ and $q = \boldsymbol{Conclusion}(\boldsymbol{I})$, the i^{th} source node of the hyper-edge is $\boldsymbol{Eval}(p_i, \rho)$ and the target node is $\boldsymbol{Eval}(q, \rho)$. We add this edge only when

(i) $\boldsymbol{Size}(\boldsymbol{EvalS}(q, \rho)) \leq s$ and $\forall i\ \boldsymbol{Size}(\boldsymbol{EvalS}(p_i, \rho)) \leq s$, and

(ii) $\boldsymbol{EvalS}(q, \rho)$ and $\forall i\ \boldsymbol{EvalS}(p_i, \rho)$ can be rewritten into the corresponding canonical propositions using the given set of replacement rules.

This optimization avoids adding too many edges that result from too involved reasoning on large-sized propositions, which could lead to failure in later stage of solution generation (concretization) involving replacement rules. Each hyper-edge $\mathcal{E}$ is annotated with the set of all tuples $\{[\boldsymbol{EvalS}(p_1, \rho), \ldots, \boldsymbol{EvalS}(p_j, \rho)], \boldsymbol{EvalS}(q, \rho)\}$ obtained from any inference rule $\boldsymbol{I}$ and any state ρ that yields $\mathcal{E}$. This set is referred to as $\boldsymbol{PTuples}(\mathcal{E})$.

Eval

The function $\boldsymbol{Eval}(q, \rho)$ substitutes each free variable x in q by $\rho(x)$. Here $\&, \|, \sim$ denote bitwise-and, bitwise-or, and bitwise-not operators respectively.

$$
\begin{aligned}
\boldsymbol{Eval}(q_1 \wedge q_2, \rho) &= \boldsymbol{Eval}(q_1, \rho)\ \&\ \boldsymbol{Eval}(q_2, \rho) \\
\boldsymbol{Eval}(q_1 \vee q_2, \rho) &= \boldsymbol{Eval}(q_1, \rho)\ \|\ \boldsymbol{Eval}(q_2, \rho) \\
\boldsymbol{Eval}(\neg q, \rho) &= \sim \boldsymbol{Eval}(q, \rho) \\
\boldsymbol{Eval}(x, \rho) &= \rho(x)
\end{aligned}
$$

EvalS

The function $EvalS(q, \rho)$ substitutes each free variable x in q by $Canonical(\rho(x))$. That is:

$$\begin{aligned} EvalS(q_1 \wedge q_2, \rho) &= EvalS(q_1, \rho) \wedge EvalS(q_2, \rho) \\ EvalS(q_1 \vee q_2, \rho) &= EvalS(q_1, \rho) \vee EvalS(q_2, \rho) \\ EvalS(\neg q, \rho) &= \neg EvalS(q, \rho) \\ EvalS(x, \rho) &= Canonical(\rho(x)) \end{aligned}$$

2.3.2 Example

For example, consider the *Modus Ponens* (MP) rule: $(\{p \rightarrow q, p\}, q)$. A $(n = 4, s = 3, \mathcal{I} = \{MP\})$ UPG has 526 bit-vector nodes $\in T_{n=4,s=3}$ (from row-4 column-3 in Table 2.7). $\forall i \in T_{n=4,s=3}\ \forall j \in T_{n=4,s=3}$, its hyper edges are from bit-vector nodes $i \rightarrow j$ and i to j, provided the following condition is true $((i \rightarrow j) \wedge i) \rightarrow j$.

An example of variable assignment which satisfies the above condition is when $i = 15$ (representative proposition a), $j = 51$ (representative proposition b). Hence, an edge is added from node $i \rightarrow j$ ($15 \rightarrow 51 = \neg 15 \vee 51 = 243$, with representative proposition $a \rightarrow b$) and node i (15), to node j (51).

Since i and j are bit-vectors, efficient bit-wise `and` ($\wedge$), `or` ($\vee$), `not` ($\neg$) operations are used to compute the satisfiability of above condition. Note that $i \rightarrow j$ is rewritten as $\neg i \vee j$.

2.3.3 Key Ideas of the Algorithm

Note that a naive approach of computing the node set, by enumerating all truth-tables and filtering out truth-tables with size at most s, has two key challenges: (i) It is not easy to identify whether a given truth-table has small size. (ii) Furthermore, the total number of truth-tables over n variables is huge (2^{2^n}). Instead we compute $T_{n,s}$ by enumerating all small propositions one by one.

n	# Edges	Time (hours)
1	20	< 1
2	960	< 1
3	14,424	< 1
4	68,422	< 3
5	207,322	< 8

Table 2.8: UPG statistics for $s = 4$ and $n \leq 5$

Similarly, note that the naive approach to computing the edge set, by enumerating over all possible tuples of source nodes and target node and identifying if a given inference rule is applicable, has two key challenges: (i)It is not easy to identify whether a given inference rule is applicable at the truth-table representation, and (ii) the number of such tuples is huge. Instead we enumerate all edges corresponding to a given inference rule by enumerating over all possible applications of that rule.

2.3.4 Results

Table 2.8 describes the number of edges of UPGs and the time taken to compute them for various values of $n \leq 5$ and $s = 4$. We chose s to be 4 since most problems in practice have $s \leq 4$ (See Table. 2.9). We experimented with $n \leq 5$ since it allows use of standard 32-bit integer for bitvector representation of a truth-table.

2.4 Solution Generation

In this section, we discuss how to generate a solution (i.e., a natural deduction proof) to a natural deduction problem.

Definition 2.10 (Traditional Algorithm) *A naive approach for generating a minimal natural deduction proof, which performs a breadth-first search by starting from premises p_i and applying some inference or replacement rule at each step to generate a new proposition until the conclusion is reached.*

We next present our two-phase *UPG-based algorithm* that first computes an abstract proof using the UPG and then extends it to a natural deduction proof.

2.4.1 Algorithm

Given the natural deduction problem $(\{p_i\}_{i=1}^m, c)$, with n variables and max proposition size s, a set $\mathcal{I}$ of inference rules, and a set $\mathcal{R}$ of replacement rules, we do the following:

1. Construct an *abstract proof* for the problem $(\{p_i\}_{i=1}^m, c)$. This is done by performing a forward search for conclusion $\tilde{c}$ starting from $\{\tilde{p}_i\}_{i=1}^m$ in $(n, s, \mathcal{I}$-UPG. A breadth-first search over UPG is undertaken to compute this minimal abstract proof.

2. *Concretize* the abstract proof to a natural deduction proof. This is done by generating a set of *candidate propositions* for each node η in the abstract proof tree in a topological order as follows.

   For each premise node, the set of *candidate propositions* contains only the corresponding premise proposition. Let $\mathcal{E}$ be the hyper-edge from the UPG whose target node yielded η, and let $([q_1, \ldots, q_n], q) \in \boldsymbol{PTuples}(\mathcal{E})$. If for each i^{th} child node of η, any of the candidate propositions q_i' associated with it can be rewritten to q_i, then q is added as a candidate proposition for η.

   The rewriting of propositions is attempted by performing a bounded breadth-first search over transitive applications of replacement rules from $\mathcal{R}$. $[q_1', \ldots, q_n']$ is called the *source-witness tuple* for q and $[q_1, \ldots, q_n]$ is called the *target-witness tuple* for q.

3. The natural deduction proof is now obtained by iteratively selecting a *source proposition* and a *target proposition* for each node of abstract proof in reverse topological order. These propositions are obtained respectively from the source-witness tuple and target-witness tuple of the parent node. Each node in the abstract proof is then expanded with a series of replacement rules that convert the source proposition into the target proposition.

Step	Truth-table	Reason
P1	$\eta_1 = 1048575$	Premise
P2	$\eta_2 = 4294914867$	Premise
P3	$\eta_3 = 3722304989$	Premise
1	$\eta_4 = 16777215$	**P1, Simplification**
2	$\eta_5 = 4294923605$	**P2, P3, Hypothetical Syllogism**
3	$\eta_6 = 1442797055$	**1, 2, Hypothetical Syllogism**

Figure 2.1: Abstract proof with truth-tables shown using a 32-bit integer representation

Node	Candidate Proposition	Target Witness	Source Witness
η_1	q_{P1}		
η_2	q_{P2}		
η_3	q_{P3}		
η_4	q_2	$[q_1]$	$[q_{\mathrm{P1}}]$
η_5	q_3	$[q_{\mathrm{P2}}, q_{\mathrm{P3}}]$	$[q_{\mathrm{P2}}, q_{\mathrm{P3}}]$
η_6	q_7	$[q_6, q_3]$	$[q_2, q_3]$

Figure 2.2: Steps in converting abstract proof to natural deduction proof. Each source witness needs to be rewritten into the corresponding target witness, using replacement rules $\mathcal{R}$.

Step	Proposition	Reason
P1	$a \lor (b \land c)$	Premise
P2	$a \to d$	Premise
P3	$d \to e$	Premise
1	$(a \lor b) \land (a \lor c)$	P1, Distribution
2	$a \lor b$	**1, Simplification**
3	$a \to e$	**P2, P3, Hypothetical Syllogism**
4	$b \lor a$	2, Commutativity
5	$\neg\neg b \lor a$	4, Double Negation
6	$\neg b \to a$	5, Implication
7	$\neg b \to e$	**6, 3, Hypothetical Syllogism**
8	$\neg\neg b \lor e$	7, Implication
9	$b \lor e$	8, Double Negation

Figure 2.3: Natural deduction proof example, with application of inference rules highlighted in bold

2.4.2 Example

Consider the natural deduction problem mentioned earlier in Example 2.3 ($\{a \lor (b \land c),\ a \to d,\ d \to e\},\ b \lor e$).

n	s				
	2	3	4	5	6
1	2		1		
2	32	6	6		2
3	27	61	15	3	2
4	15	34	27	1	
5	3	30	20	2	
6		7	20	1	1
7		3	7		1
8		2	3	2	
9		1		1	1

Table 2.9: 339 benchmark natural deduction problems obtained from 5 textbooks [25, 26, 27, 20, 28], distributed across number of Boolean variables n and max size s of each proposition. Note that 82% of the problems have $s \leq 4$ and $n \leq 5$.

Figure 2.2 outlines how the above algorithm generates the natural deduction proof in Figure 2.3 from the abstract proof in Figure 2.1. Column "Candidate Proposition" lists a placeholder candidate propositions produced by step 2 of the solution generation algorithm, which is later identified by step 3 of the algorithm using the corresponding source/target witnesses.

2.4.3 Results

We report on results of our UPG-based algorithm over 279 benchmark problems collected from various textbooks, that have $n \leq 5$ and $s \leq 4$ (Table 2.9). These problems were picked across 21 different exercise sets, each requiring use of a specific set $\mathcal{I}$ of inference rules. Some exercise sets asked for a proof by contradiction, which we modeled into our framework by modifying conclusion to $false$ and adding the negation of the original conclusion as a premise. We obtained $(n, s, \mathcal{I})$-UPG as $\bigcup_{I \in \mathcal{I}} (n, s, \{I\})$-UPG instead of having to compute the UPG for each given subset $\mathcal{I}$ from scratch.

Our tool was able to generate an abstract proof for 88% of these problems, using a timeout of 10 seconds. Among these 88% of problems, our tool was able to concretize the abstract proof to a natural deduction proof for 96% instances, thereby achieving an overall success rate of 84%. In contrast, the overall success rate of the traditional

Total #Steps	Avg #Abstract Steps (j)	# Problems	% Success Traditional	Avg time UPG (sec)	Avg time Trad. (sec)
1	0.6	18	100	0.0	0.0
2	0.9	21	100	0.0	0.2
3	0.9	25	100	0.0	0.9
4	1.2	23	70	0.0	1.2
5	1.8	16	69	0.1	0.2
6	2.0	19	47	0.1	0.6
7	1.7	14	71	0.9	0.6
8	1.9	20	70	1.2	0.3
9	2.0	14	29	0.2	1.9
10	2.7	6	33	0.2	1.5
11	2.5	6	17	1.9	0.0
12	2.6	10	60	4.4	2.0
13	2.7	6	33	3.0	0.1
14	3.0	2	50	0.4	4.4
15	3.1	8	13	2.9	0.1
16	2.6	8	38	1.0	0.4
17	2.8	6	17	2.2	0.1
19	3.3	3	33	3.0	0.1
20	2.0	2	0	0.3	-
21	4.0	1	100	6.4	8.7
22	3.0	2	0	1.4	-
23	3.0	1	0	4.3	-
24	2.0	1	100	1.3	0.1
25	4.0	1	0	8.7	-
27	4.0	1	0	9.3	-

Table 2.10: Solution generation results, with timeout of 10 sec. Only those problems which UPG-based algorithm could solve overall (84% of 279 = 234) are listed here, grouped by the total number of steps in overall natural deduction proof. The performance of traditional algorithm against these problems is shown in column "% Success Traditional". The last two columns report on the average time-taken by both algorithms, only accounting for problems which could be solved.

breadth-first-search algorithm is 57%, using the same timeout of 10 seconds.

Table 2.10 shows the distribution of the 84% problems that the UPG-algorithm is able to solve overall, with respect to the total number of steps required to solve them. Column 4 "% Success Traditional" shows that the traditional algorithm is unable to solve several of these problems, especially those requiring larger number of steps, due to timing out. Column 2 shows that the average number of steps of the abstract proof is significantly smaller than the total number of steps in Column 1 for natural deduction proof; this partly explains the higher success rate of the UPG-based algorithm.

There are a small number of instances where the traditional algorithm works better than the UPG-based algorithm. This happens when the traditional algorithm is able to find an overall shorter proof that has more number of abstract steps than the proof found by the UPG-based algorithm (which has smallest number of abstract steps, but involves significantly large number of replacement steps).

The UPG-based algorithm does fail to solve certain problems. This is because it restricts its search to proofs where all the propositions have size $s \leq 4$. However, for certain problems, proofs involve intermediate propositions whose size is greater than 4. Note that this can happen even when the premises and conclusion have size at most 4.

2.5 Similar Problem Generation

We now show how to generate fresh problems with given solution characteristic. The key algorithmic insight here is to model problem generation as reverse of solution generation, and this backward search is enabled through UPG. We consider two goals, that of producing similar problems and parameterized problems.

Definition 2.11 (Similar Problems) *Two problems Q_1 and Q_2 are called **similar** if they entail a similar minimal abstract proof tree. Two abstract proof trees are said to be similar if they involve exactly the same order of inference rule applications.*

Generating similar problems can help address issues such as copyright and plagiarism, as discussed earlier in Section 2.1.

2.5.1 Algorithm

Let $\boldsymbol{P}$ be a problem with $\boldsymbol{n}$ variables and maximum proposition size $\boldsymbol{s}$ that needs to be solved using inference rules $\boldsymbol{\mathcal{I}}$. Our algorithm for generating problems that are similar to $\boldsymbol{P}$ involves the following steps.

1. Generate a minimal abstract proof $\boldsymbol{A}$ for $\boldsymbol{P}$, by performing a forward breadth-first search in the $(\boldsymbol{n}, \boldsymbol{s}, \boldsymbol{\mathcal{I}})$-UPG.

2. Find matches of $\boldsymbol{A}$ in the $(\boldsymbol{n}, \boldsymbol{s}, \boldsymbol{\mathcal{I}})$-UPG using a backtracking based backward search.

3. Replace each truth-table node in the match by its canonical proposition, to generate new problems with the same abstract solution as original problem $\boldsymbol{P}$.

2.5.2 Results

Table 2.11 presents statistics on the number of similar (but well-defined) problems that we produced, given a seed problem, along with the time taken to produce them and number of abstract proof steps $\boldsymbol{j}$ required to solve these. We discount all trivial problems generated, such as replacing a variable by any other variable or its negation.

2.5.3 Example

In Figure 2.4, we show 5 (out of 516) new similar problems generated for the last problem in Table 2.11, which is the same original problem from Example 2.3. Figure 2.6 shows a solution for the first new problem in Figure 2.4. Observe that the abstract version of this solution (i.e., the bold steps) are similar to the abstract version of the solution for the original problem, which is shown in Figure 2.5.

Original Premises	Original Conclusion	j	#Similar Problems	Time min:sec
$a \vee b,\ c,\ (c \wedge (a \vee b)) \rightarrow \neg d,\ \neg e \rightarrow d$	$\neg\neg e$	3	21,849	42:42
$a,\ (a \wedge b) \rightarrow (c \wedge d),\ b,\ d \rightarrow e$	e	4	9,290	30:28
$\neg a,\ (\neg a \wedge \neg b) \rightarrow (c \rightarrow d),\ \neg b,\ d \rightarrow e$	$c \rightarrow e$	3	5,306	17:56
$\neg(a \wedge b),\ (a \rightarrow \neg b) \rightarrow (\neg c \wedge \neg d),\ e \vee (c \vee d)$	e	2	4,001	5:56
$a \rightarrow (b \rightarrow c),\ \neg(b \rightarrow c),\ \neg d \vee a$	$\neg d$	2	2,908	0:06
$a \rightarrow (b \rightarrow (c \vee d)),\ b \wedge a,\ \neg d$	c	2	2,353	0:14
$\neg a \rightarrow b,\ a \rightarrow c,\ \neg c,\ b \rightarrow d$	$d \vee e$	4	1,248	5:17
$(a \rightarrow b) \wedge \neg c,\ (\neg a \rightarrow e) \vee c,\ (\neg b \rightarrow e) \rightarrow d$	d	5	274	1:18
$(c \vee b) \rightarrow \neg a,\ d \vee a,\ b$	d	3	122	0:02
$a \vee (b \wedge c),\ a \rightarrow d,\ d \rightarrow e$	$b \vee e$	3	516	1:52

Table 2.11: Number of similar problems generated and time taken to generate them for a few representative selection of problems. shown along with the number of inference steps j required to solve them. Note that the last row problem is the same as Example 2.3 problem, whose similar problem examples were shown in earlier Section 2.5.3.

2.6 Parameterized Problem Generation

Definition 2.12 ($(n, m, s, j, \mathcal{I})$-problem) *A $(n, m, s, j, \mathcal{I})$ problem is one that has (i) m premises involving n variables of size at most s, and (ii) that has a minimal abstract proof involving j steps, making use of only and all inference rules from set $\mathcal{I}$.*

Generating parameterized problems can be used to generate personalized workflows as discussed in Section 2.1.

2.6.1 Algorithm

The algorithm for generating $(n, m, s, j, \mathcal{I})$-problems that have n variables and maximum proposition size s involves performing a backtracking based backward search in $(n, s, \mathcal{I})$-UPG to find appropriate matches. To generate a concrete problem, we replace each truth-table node in the match by its canonical proposition.

Premise 1	Premise 2	Premise 3	Conclusion
$a \equiv b$	$c \rightarrow \neg b$	$(d \rightarrow e) \rightarrow c$	$a \rightarrow (d \wedge \neg e)$
$a \wedge (b \rightarrow c)$	$(a \vee d) \rightarrow \neg e$	$b \vee e$	$(a \vee d) \rightarrow c$
$(a \vee b) \rightarrow c$	$c \rightarrow (a \wedge d)$	$(a \wedge d) \rightarrow e$	$a \rightarrow e$
$(a \rightarrow b) \rightarrow c$	$c \rightarrow \neg d$	$a \vee (e \vee d)$	$e \vee (b \rightarrow a)$
$a \rightarrow (b \wedge c)$	$d \rightarrow \neg b$	$(c \equiv e) \rightarrow d$	$a \rightarrow (c \equiv \neg e)$

Figure 2.4: 5 (out of 516) new similar problems generated, given the Example 2.3 problem ($\{a \vee (b \wedge c),\ a \rightarrow d,\ d \rightarrow e\},\ b \vee e$) as seed.

Step	Proposition	Reason
P1	$a \vee (b \wedge c)$	Premise
P2	$a \rightarrow d$	Premise
P3	$d \rightarrow e$	Premise
1	$(a \vee b) \wedge (a \vee c)$	P1, Distribution
2	$a \vee b$	**1, Simplification**
3	$a \rightarrow e$	**P2, P3, Hypothetical Syllogism**
4	$b \vee a$	2, Commutativity
5	$\neg\neg b \vee a$	4, Double Negation
6	$\neg b \rightarrow a$	5, Implication
7	$\neg b \rightarrow e$	**6, 3, Hypothetical Syllogism**
8	$\neg\neg b \vee e$	7, Implication
9	$b \vee e$	8, Double Negation

Figure 2.5: Natural deduction proof of Example 2.3 problem ($\{a \vee (b \wedge c),\ a \rightarrow d,\ d \rightarrow e\},\ b \vee e$). Application of inference rules are highlighted in bold.

Step	Proposition	Reason
P1	$a \equiv b$	Premise
P2	$c \rightarrow \neg b$	Premise
P3	$(d \rightarrow e) \rightarrow c$	Premise
1	$(a \rightarrow b) \wedge (b \rightarrow a)$	P1, Equivalence
2	$a \rightarrow b$	**1, Simplification**
3	$(d \rightarrow e) \rightarrow \neg b$	**P3, P2, Hypothetical Syllogism**
4	$\neg\neg b \rightarrow \neg(d \rightarrow e)$	3, Transposition
5	$b \rightarrow \neg(d \rightarrow e)$	4, Double Negation
6	$a \rightarrow \neg(d \rightarrow e)$	**2, 5, Hypothetical Syllogism**
7	$a \rightarrow \neg(\neg d \vee e)$	6, Implication
8	$a \rightarrow (\neg\neg d \wedge \neg e)$	7, De Morgan's
9	$a \rightarrow (d \wedge \neg e)$	8, Double Negation

Figure 2.6: Natural Deduction Proof of a problem (first row of Figure 2.4) similar to Example 2.3 problem ($\{a \vee (b \wedge c),\ a \rightarrow d,\ d \rightarrow e\},\ b \vee e$). The abstract proof, highlighted in bold, is the exact same as original problem (Figure 2.5)

n	m	j	$\mathcal{I}$	**Num. of problems**	**Time min:sec**
2	2	1	MP	4	0:00
2	2	2	Simp, MP	6	0:02
3	3	2	DS, HS	145	0:00
3	3	3	DD, MT, Conj	188	0:00
3	2	4	Simp, Conj, MP	279	0:09
4	2	3	Simp, MP,	2060	1:52
4	3	3	Conj, MP, HS	6146	1:29
4	4	3	HS, MP	18132	9:52
5	3	2	Simp, MP	5628	8:04
5	3	2	HS, DS	5838	7:39

Table 2.12: Number of $(n, m, s, j, \mathcal{I})$-problems generated and time taken to generate them for few representative choices of $n, m, j, \mathcal{I}$, with size s fixed to 4.

Premise 1	**Premise 2**	**Premise 3**	**Conclusion**
$(a \rightarrow c) \rightarrow b$	$b \rightarrow c$	$\neg c$	$a \wedge \neg c$
$c \rightarrow a$	$(c \equiv a) \rightarrow b$	$\neg b$	$a \wedge \neg c$
$(a \equiv c) \vee (a \equiv b)$	$(a \equiv b) \rightarrow c$	$\neg c$	$a \equiv c$
$a \equiv \neg c$	$b \vee a$	$c \rightarrow \neg b$	$a \wedge \neg c$
$c \rightarrow a$	$a \rightarrow (b \wedge c)$	$c \rightarrow \neg b$	$\neg c$

Table 2.13: 5 (out of 145) new parameterized problems generated from the parameters in the 3rd row in Table 2.12 (namely, $n = 3$, $m = 3$, $s = 4$, $j = 2$, $\mathcal{I} = \{\mathsf{DS}, \mathsf{HS}\}$).

2.6.2 Results

Table 2.12 presents statistics on the number of well-defined $(n, m, s, j, \mathcal{I})$-problems produced for certain values of $(n, m, j, \mathcal{I})$, as obtained from some existing problems and $s = 4$. As before, we do not count duplicate problems that can be obtained by replacing a variable with any other variable or its negation.

2.6.3 Example

In Table 2.13, we show 5 (out of 145) new problems generated for $n = 3$, $m = 3$, $s = 4$, $j = 2$, and $\mathcal{I} = \{\mathsf{DS}, \mathsf{HS}\}$, which corresponds to the parameters in the third row in Table 2.12. All of these problems can be solved in $j = 2$ inference steps, using inference rule `DS` and `HS`.

2.7 Related Work

Natural Deduction

Several proof assistants have been developed for teaching natural deduction: MacLogic [29], Symlog [30], Jape [31], Hyperproof [32], Pandora [33], Proof Lab [34], and ProofWeb [35]. These tools primarily differ in proof checking, how proofs are visualized (notably whether proofs are trees or sequences), whether both forward and backward reasoning are supported, the availability of global, tactical and strategic help, and debugging facilities. Ditmarsch et al. [36] provide a nice survey of these tools, and we discuss few notable ones below.

Pandora [33] is a Java based tool that allows the user to reason both forwards and backwards, checking the rule application at each stage and providing feedback. Proof Lab [34] makes use of the AProS algorithm and can search for proofs using a combination of forward chaining, backward chaining, and contradiction. In contrast, we focus on generating proofs without contradiction, thereby staying true to what the original problem asks for.

ProofWeb [35] makes use of a state-of-the-art proof assistant Coq [37], which allows encoding of various tactics for proof generation. However, none of these systems report on performance metrics as we do. More significantly, our work makes an orthogonal point, namely how to speed up forward/backward and with/without contradiction search [1], by using offline computation (which exploits the small formula sized hypothesis) and a two-staged proof generation strategy that first computes an abstract proof and then the natural proof. These optimizations also pave way for generating fresh problems that is not addressed by any of these existing tools.

[1] Though we demonstrate this in the context of forward search without contradiction, our methodology is applicable more broadly.

Problem Generation

Singh et al. [14] describe a problem generation technology for generating problems that are similar in structure to a given algebraic identity proof problem. Their technology leverages continuity of the underlying domain of algebraic expressions, and uses extension of polynomial identity testing to check the correctness of a generated problem candidate on a random input. In contrast, the domain of Boolean expressions is non-continuous or discrete, and hence requires a different technology of checking the correctness or well-formedness of a problem candidate on all inputs. Furthermore, unlike Singh et al. [14], our technology also enables solution generation.

Andersen et al.[38] describe a problem generation technology for *procedural domain*, which includes problems commonly found in middle-school math curriculum such as subtraction and greatest common divisor computation. Their underlying technology leverages test input generation techniques [39] to generate problems that explore various paths in the procedure that the student is expected to learn. In contrast, we address problem generation for *conceptual domain*, where there is no step-by-step decision procedure that the student can use to solve a problem, which requires creative skills such as pattern matching.

Solution Generation

Gulwani et al. [10] describe a solution generation technology for ruler/compass based geometric construction problems, using a search technique like ours with some interesting similarities/differences. They perform a forward breadth-first search (similar to us) by repeatedly applying ruler/compass operations to reach the desired output geometric object from input geometric objects. The geometric object is represented using a concrete representation, which constitutes its *probabilistic* hash, in order to avoid symbolic reasoning. We similarly avoid symbolic reasoning, but by using an *abstract hash* (i.e., truth-table representation). Since inference rules cannot be directly applied on the abstract hash, we resort to leveraging offline computation (UPG). This provides the additional benefit of generating problems

with given solution characteristics.

2.8 Summary

Computer-aided instruction can raise the quality of education by making it more interactive and customized. In this work, we have provided the core building blocks, namely problem generation and solution generation, for the classic subject domain of natural deduction taught in an introductory logic course. This can free instructors from the burden of creating and generating sample solutions to assignment problems. An instructor can create few interesting seed problems and our tool can automatically generate variants of these problems with similar difficulty level.

The SAT solving and theorem proving communities have focused on solving large-sized problem instances in a reasonable amount of time. In contrast, our work innovates by developing techniques for solving small-sized instances in real-time. The small-sized assumption allows use of offline computation, and use of bitvector data structure helps alleviate the cost associated with symbolic reasoning. This paves way for some new applications, namely generation of human-readable proofs and new problems. Our solution generation technology can be extended to complete partial proofs or fix buggy proofs submitted by students.

In the next Chapter 3, we present an approach for generating interesting starting levels for traditional board games, with core components similar to parameterized problem generation in this Chapter. In the following Chapters 4 and 5, we present solution generation and problem generation feedback tools for programming, respectively. Then in Chapter 6, we measure the productivity impact of these tools when deployed in a real course offerings.

Chapter 3

Simple Traditional Board Games

Simple traditional board games, such as Tic-Tac-Toe and CONNECT-4, play an important role not only in the development of mathematical and logical skills, but also in the emotional and social development. In this chapter, we address the problem of generating targeted starting positions for such board games. This can facilitate new approaches for bringing novice players to mastery, and also leads to the discovery of interesting game variants.

In this chapter, we present an approach [40] that generates starting states of varying hardness levels for player 1 in a two-player board game, given rules of the board game, the desired number of steps required for player 1 to win, and the expertise levels of the two players. Our approach leverages symbolic methods and iterative simulation to efficiently search the extremely large state space. We present experimental results that include discovery of states of varying hardness levels for several of these simple grid-based board games. The presence of such states for standard game variants, like 4×4 Tic-Tac-Toe, opens up new games to be played that have never been played as the default start state is heavily biased.

3.1 Introduction

Board games involve placing pieces on a pre-marked surface or board, according to a set of rules in a turn based fashion. Some of these grid-based two-player games, like

Tic-Tac-Toe and CONNECT-4, have a relatively simple set of rules and yet, they are reasonably challenging for certain age groups. Such games that are easy to learn but difficult to master have been immensely popular across centuries.

Studies have shown that board games can significantly improve a child's mathematical ability [41], and early differences in mathematical ability persists into later stages of education [42]. Board games also assist with emotional and social development of a child [43]. They instill a competitive desire to master new skills in order to win, which could give a boost to their self confidence. Playing a game within a set of rules helps them develop social etiquette; taking turns, and being patient. Strategy is another huge component of games, where children learn causal effect by observing that decisions they make in the beginning of the game have larger consequences later on [44]. Playing board games can be beneficial for elderly people as well, helping them stay mentally sharp and hence less likely to develop Alzheimer [45].

3.1.1 Significance of Generating Fresh Starting States

Board games are typically played with a default start state; for example, empty board in case of Tic-Tac-Toe and CONNECT-4. However, there are following drawbacks in starting from the default starting state, which we use to motivate our goals.

Customizing hardness level of a start state

The default starting state for a certain game, while being unbiased, *might not be conducive for a novice player to enjoy and master the game.* Traditional board games in particular are easy to learn but difficult to master because these games have intertwined mechanics and force the player to consider large number of possibilities from the standard starting configurations. Players can achieve mastery most effectively if complex mechanics can be simplified and learned in isolation. Csikszentmihalyi's theory of flow [46] suggests that we can keep the learner in a state of maximal engagement by continually increasing difficulty to match the learner's increasing skill. Hence, we need an approach that allows generating start states of a specified

hardness level. This capability can be used to generate a progression of starting states of increasing hardness. This is similar to how students are taught educational concepts like addition through a progression of increasingly hard problems [38].

Leveling the playing field

The starting state for commonly played games is mostly unbiased, and hence *does not offer a fair experience for players of different skills.* The flexibility to start from other starting states that is more biased towards the weaker player can allow for leveling the playing field, and hence a more enjoyable game for both. Hence, we need an approach that takes as input the expertise levels of players and uses that information to associate a hardness level with a state.

Generating multiple fresh start states

A fixed starting state might have a *well-known conclusion.* For example, both players can enforce a draw in Tic-Tac-Toe while the first player can enforce a win in CONNECT-4 [47], starting from the default empty starting state. Players can *memorize certain moves* from a fixed starting state and gain undue advantage. Hence, we need an approach that generates multiple start states (of a specified hardness level). This observation has also inspired the design of Chess960 [48] (or Fischer Random Chess), which is a variant of chess that employs the same board and pieces as standard chess; however, the starting position of the pieces on the players' home ranks is randomized. The random setup renders the prospect of obtaining an advantage through memorization of opening lines impracticable, compelling players to rely on their talent and creativity.

Customizing length of play

People sometimes might be disinterested in playing a game if it takes *too much time to finish.* However, selecting non-default starting positions allow the potential of a shorter game play. Certain interesting situations might manifest only in states that

are typically not easily reachable from the start state, or require *too many steps*. The flexibility to start from such states might lead to more opportunities for practice of specific targeted strategies. Thus, we need an approach that can take as input a parameter for the number of steps that can lead to a win for a given player.

Experimenting with game variants

While people might be hesitant to learn a new game with completely different rules, it is quite convenient to modify the existing rules slightly. For example, instead of allowing for straight-line matches in each of row, column, or diagonal (RCD) in Tic-Tac-Toe or CONNECT-4, one may restrict the matches to say only row or diagonal (RD). However, *the default starting state of a new game may be heavily biased towards a particular player*; as a result that specific game might not have been popular. For example, consider the game of Tic-Tac-Toe ($m = 4, n = 4, k = 3$), where the goal is to make a straight line of $k = 3$ pieces, but on a 4×4 ($m \times n$) board. In this game, the person who plays first invariably almost always wins even with a naive strategy. Hence, such a game has never been popular. However, there can be non-default unbiased states for such games and starting from those states can make playing such games interesting. Hence, we need an approach that is parameterized by the rules of a game. This also has the advantage of experimenting with new games or variants of existing games.

To summarize, we need an approach to generate *multiple* start states of *specified hardness levels*, given *expertise levels of the players* and *length of plays*, for traditional *board games and their variants*.

3.1.2 Technique Overview

In this chapter, we address the problem of automatically generating *interesting* starting states (i.e., states of desired hardness levels) for a given two-player board game. Our approach takes as input the *rules of a board game variants* and the *desired number of steps required for player 1 to win*. It then generates multiple starting

O				O
X		X		X
X		O		X
X	X	O	X	O
O	O	O	X	O

Table 3.1: Interesting starting board positions generated by our tool, for CONNECT-4 board game. Player 1 (X) has a guaranteed path to victory in 3 moves, if played efficiently.

states of varying hardness levels (in particular, *easy, medium, or hard*) for player 1 for various *expertise level* combinations of the two players.

For example, Table 3.1 shows an interesting starting board positions for CONNECT-4, which requires 4 consecutive pieces in row, column or diagonal to win. The moves in CONNECT games are restricted by gravity; i.e, while a player has the freedom to place their piece on any column, the selected row must be the bottom most available empty cell. Player 1 (X) has the first turn to move followed by Player-2 (O), in an alternating fashion. In the Table 3.1 example, Player 1 (X) has only three valid moves: $\boldsymbol{C}_2\boldsymbol{R}_3, \boldsymbol{C}_3\boldsymbol{R}_1, \boldsymbol{C}_4\boldsymbol{R}_3$; where $\boldsymbol{C}$ denotes column number and $\boldsymbol{R}$ denotes row number of the empty cells. This board position was automatically generated by our tool in which Player 1 (X) can win in 2 turns (3 moves of Player 1 and 2 moves of Player 2) if they play optimally, irrespective of the choices made by Player 2 (O).

We formalize the exploration of a game as a strategy tree and the expertise level of a player as depth of this strategy tree. The hardness of a state is defined in terms of fraction of times player 1 will win, while playing a strategy of depth k_1 against an opponent who plays a strategy of depth k_2.

Our solution employs a novel combination of symbolic methods and iterative simulation to efficiently search for desired states. Symbolic methods are used to compute the winning set of player 1. These methods work particularly well for navigating a state space where the transition relation forms a sparse directed acyclic graph (DAG). Which is the case for board games, such as Tic-Tac-Toe and CONNECT-4, where a piece once placed on the board doesn't move. Minimax simulation is then used to identify the hardness of a given winning state. Instead of randomly sampling the winning set to identify a state of a certain hardness level,

we identify states of varying hardness levels in order of increasing values of k_1 and k_2. The key observation is that hard states are much fewer than easy states, and for a given k_2, interesting states for higher values of k_1 are a subset of hard states for smaller values of k_1.

3.1.3 Results Overview

While our general search methodology applies to any graph game, for our experimental results we focus on generating interesting starting states for *simple* board games and their variants. These are games whose transition relation forms a sparse DAG (as opposed to an arbitrary graph).

Generating starting states in simple and traditional games, as compared to games with complicated rules, is both more challenging and more relevant. First, in sophisticated games with complicated rules interesting states are likely to be abundant and hence easier to find, whereas finding interesting states in simple games is more challenging. Second, games with complicated rules are harder to master, whereas simple variants of traditional games (such as larger or smaller board size, or changing winning conditions from RCD to RD) are easier to adopt.

We experimented with Tic-Tac-Toe, CONNECT, Bottom-2 (a new game that is a hybrid of Tic-Tac-Toe and CONNECT) games and several of their variants, with modified winning condition such as RD or RC instead of RCD. We were able to generate several starting states of various hardness levels for various expertise levels and number of winning steps. Two important findings of our experiments are: (i) discovery of starting states of various hardness levels in these traditional board games, especially in games such as Tic-Tac-Toe 4×4 where the default start state is heavily biased; and (ii) these states are rare and thus require a non-trivial search strategy like ours to find them.

3.1.4 Contributions

- We introduce and study a novel aspect of graph games, namely generation of starting states. These starting states are of varying hardness levels, parameterized by look-ahead depth of the strategies of the two players, the graph game description, and the number of steps required for winning.

- We present a novel search methodology for generating desired initial states. It involves combination of symbolic methods and iterative simulation to efficiently search a huge state space.

- We present experimental results that illustrate the effectiveness of our search methodology. We produce a collection of initial states of varying hardness levels for standard games as well as their variants (thereby discovering some interesting variants of the standard games in the first place).

3.1.5 Chapter Outline

This chapter is organized as follows. We start out with a formal background of graph games and then formally state our problem definition in Section 3.2. We present our search methodology for generating desired initial states in Section 3.3. We then present a framework for describing (rules of) new board games that are like Tic-Tac-Toe or CONNECT-4 (or similar simple variants) in Section 3.4. We describe experimental results for several instantiations of this framework in Section 3.5. We describe related work in Section 3.6 and then conclude in Section 3.7.

3.2 Problem Definition

In this section, we first present some necessary background related to mathematical model of graph games and recall some basic results in Section 3.2.1. We then describe the notion of hardness in Section 3.2.2 and formally describe our problem in Section 3.2.3.

3.2.1 Background on Graph Games

Graph games

An *alternating graph game* (for short, *graph game*) $G = ((V, E), (V_1, V_2))$ consists of a finite graph G with vertex set V, a partition of the vertex set into player-1 vertices V_1 and player-2 vertices V_2, and edge set $E \subseteq ((V_1 \times V_2) \cup (V_2 \times V_1))$. The game is alternating in the sense that the edges of player-1 vertices go to player-2 vertices and vice-versa.

The game is played as follows: the game starts at a starting vertex u_0; if the current vertex is a player-1 vertex, then player 1 chooses an outgoing edge to move to a new vertex; if the current vertex is a player-2 vertex, then player 2 does likewise. The winning condition is given by a target set $T_1 \subseteq V$ for player 1; and similarly a target set $T_2 \subseteq V$ for player 2. If the target set T_1 is reached, then player 1 wins; if T_2 is reached, then player 2 wins; else we have a draw if no more moves are possible (terminal state is reached).

Example

The class of graph games provides the mathematical framework to study board games like Chess or Tic-Tac-Toe. For example in standard Tic-Tac-Toe, the vertices of the graph represent the board configurations and whether it is the turn of player 1 ($\times$) or player 2 ($\circ$) to play next. The set T_1 (respectively T_2) is the set of board configurations with three consecutive $\times$ (resp. $\circ$) in a row, column, or diagonal.

Classical game theory result

A classic result in the theory of graph games [49] shows that for every graph game with target sets T_1 and T_2 for both players, from every starting vertex, exactly one of the following three conditions hold:

1. Player 1 can enforce a win no matter how player 2 plays. That is, there is a strategy for player 1 to ensure winning against all possible strategies of the

opponent.

2. Player 2 can enforce a win no matter how player 1 plays.

3. Both players can enforce a draw; player 1 can enforce a draw no matter how player 2 plays, and player 2 can enforce a draw no matter how player 1 plays.

In the mathematical study of game theory, the theoretical question (which ignores the notion of hardness) is as follows: given a designated starting vertex u_0, determine whether case (1), case (2), or case (3) holds. In other words, the mathematical game theoretic question concerns the best possible way for a player to play, to ensure the best possible result.

Let the set W_j be the set of vertices such that player 1 can ensure win within j-moves. The winning set $W^1 = \bigcup_{j \geq 0} W_j$ is the set of vertices of player 1, where player 1 can win in any number of moves. Analogously, we define W^2. The classical game theory question is then reiterated as follows: given a designated starting vertex u_0, decide whether u_0 belongs to W^1 (player-1 winning set) or to W^2 (player-2 winning set) or to $V \setminus (W^1 \cup W^2)$ (both players draw ensuring set).

3.2.2 Notion of Hardness

The above game theoretic question ignores two critical aspects that we are interested in. (1) The notion of hardness: the theoretic question is concerned with optimal strategies irrespective of hardness; and (2) The problem of generating different starting vertices. The hardness notion we consider for our purpose is the depth of the search tree a player explores, which is a standard metric in artificial intelligence.

Search Tree

Consider a player-1 vertex u_0. The *search tree* of depth 1 is a tree rooted at u_0 of player 1, whose children are vertices u_1 of player 2. Further, every vertex u_1 contains vertices u_2 of player 1 as its children; with these u_2 vertices forming the leaves of

search tree of depth 1. That is, for $u_0, u_2 \in V_1$ and $u_1 \in V_2$, $(u_0, u_1) \in E$ (there is an edge from u_0 to u_1) and $(u_1, u_2) \in E$.

This gives us the search tree of depth 1, which intuitively corresponds to exploring one round of the play. The search tree of depth $k+1$ is defined inductively from the search tree of depth k, where we first consider the search tree of depth 1 and replace every leaf with a search tree of depth k. The depth of the search tree denotes the depth of reasoning (analysis depth) of a player. The search tree for player 2 is defined analogously.

Reward Function

For every vertex u of the game, we associate a reward $r(v)$; a number that denotes how "close" it is to the winning vertex of a player. This reward function r is game specific.

For example, consider the standard Tic-Tac-Toe (m=3, n=3, k=3) game, where the goal is to make a straight line of 3 consecutive pieces in row, column or diagonal, on a 3×3 board. Given a board state of Tic-Tac-Toe (3,3,3), the reward function r for player 1 is defined as:

1. If the board position is winning for player 1, then it is assigned reward $+\infty$
2. Else if it is winning for player 2, then it is assigned reward $-\infty$
3. Otherwise it is assigned the score as follows: let n_1 (resp. n_2) be the total number of two consecutive pieces for player 1 (resp. player 2), which can be extended to satisfy the winning condition. Then the reward is the difference $n_1 - n_2$

Intuitively, the number n_1 represents the number of possibilities for player 1 to win, n_2 represents the number of possibilities for player 2, and their difference represents how favorable the board position is for player 1.

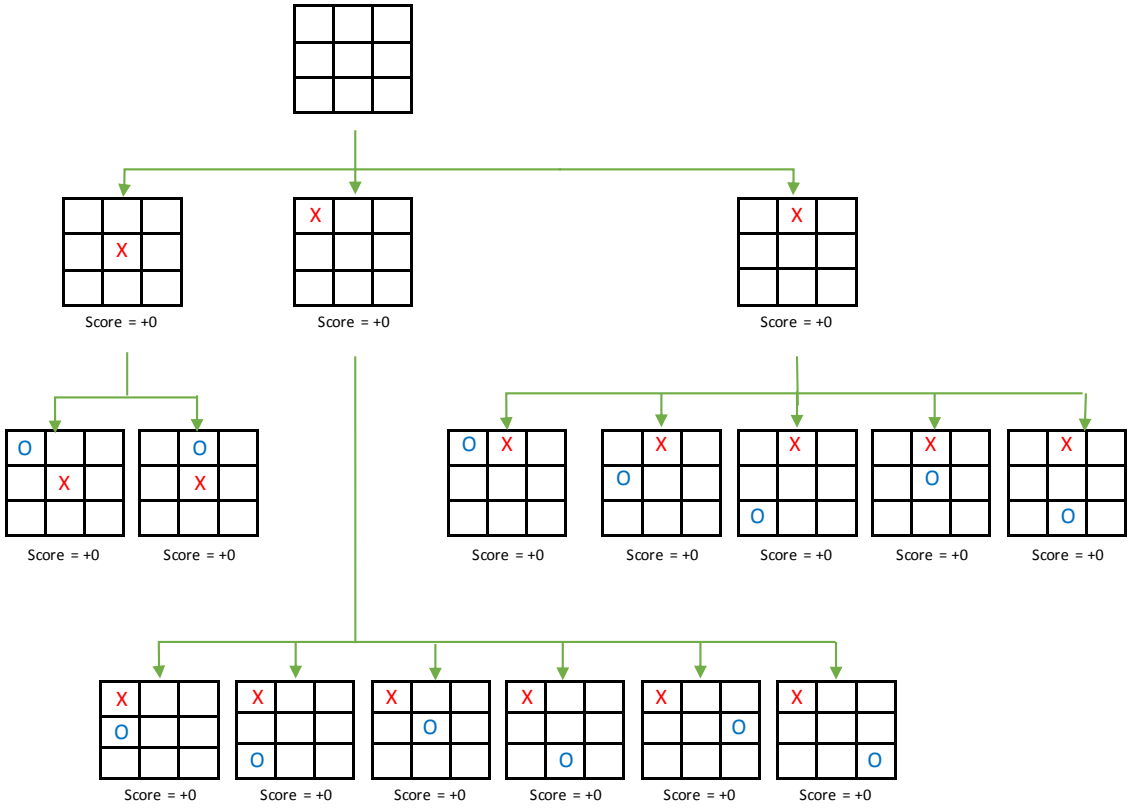


Figure 3.1: Illustration of depth $k_1 = 1$ tree exploration for Tic-Tac-Toe, for the blank starting state. The bottom-most leaves are assigned scores using reward function, and their parent's reward is calculated using *min-max* strategy. Since all possible moves for $k_1 = 1$ have equal score of +0, any valid board positions is chosen at random.

Search tree exploration

Given the current position vertex u, a *depth-k strategy* of a player constructs the search tree of depth k rooted at u, and evaluates this tree bottom-up using the classical *min-max* reasoning (or backward induction).

First, all the leaf vertices of depth k tree are assigned rewards, using a pre-defined rewards function r. Then, each parent vertex is assigned the maximum (resp. minimum) reward of its children, if the parent vertex is a player 1 (resp. player 2) vertex. This process continues until we reach the root note u, where the strategy chooses *uniformly at random* among its children with the highest reward.

For example, consider the standard Tic-Tac-Toe (3,3,3) with the blank starting state, where no player has placed any piece on the 3×3 board. If we consider the depth-1 strategy, then the strategy chooses all board positions uniformly at random;

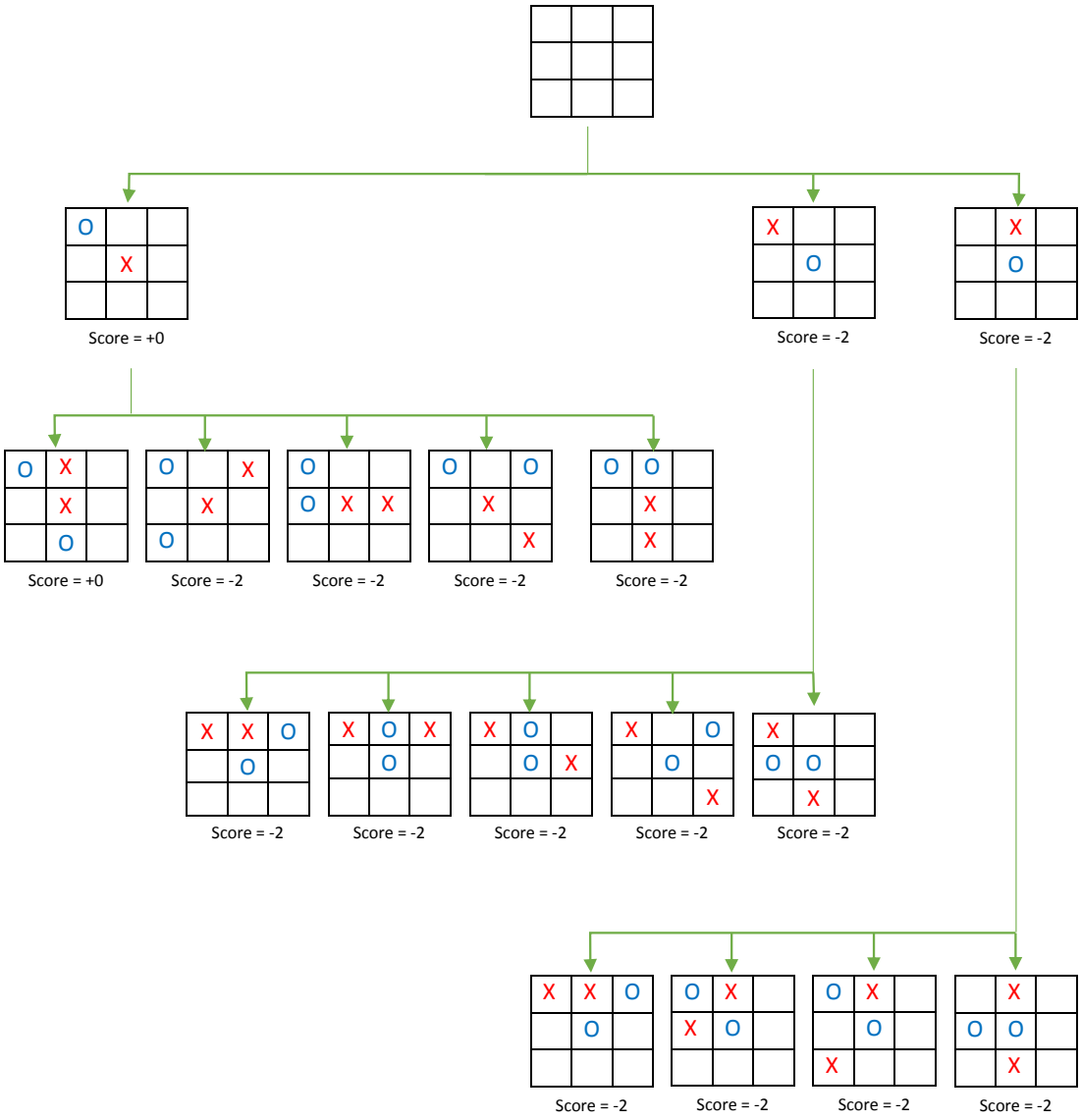
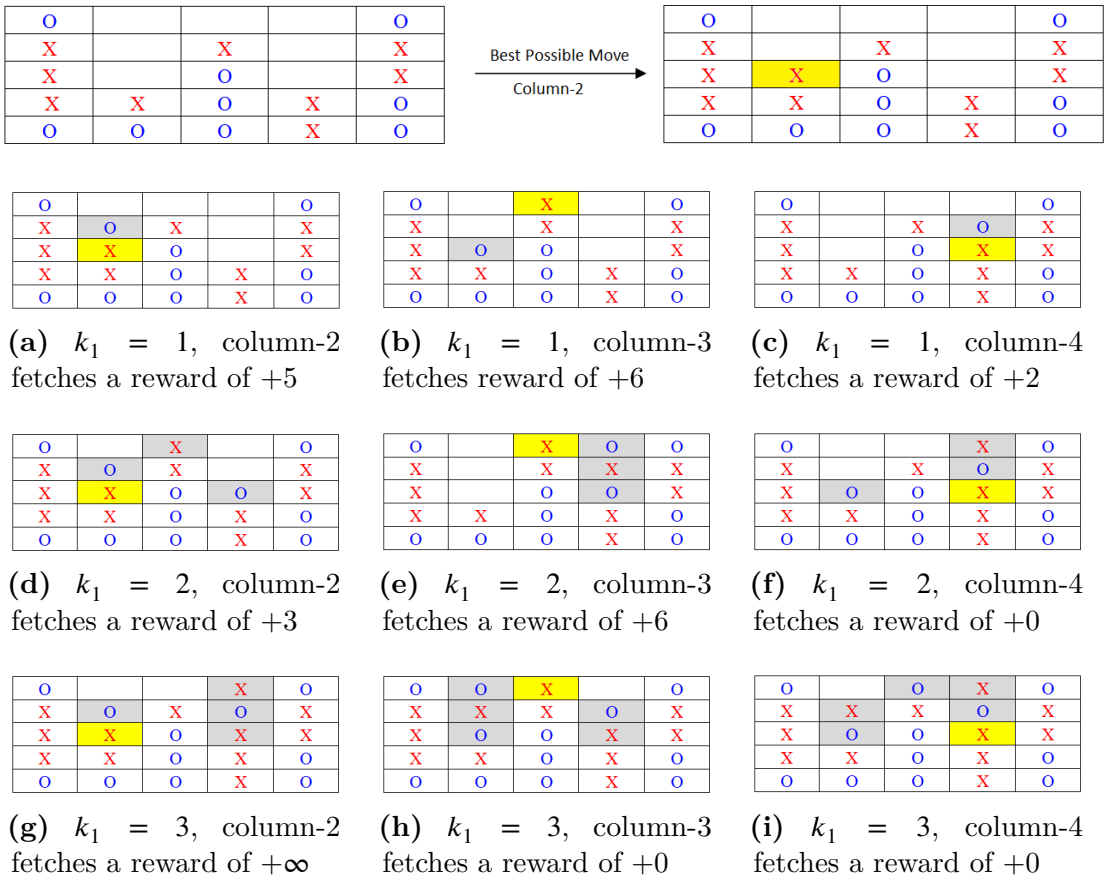


Figure 3.2: Illustration of depth $k_1 = 2$ tree exploration for Tic-Tac-Toe, for the blank starting state. The bottom-most leaves are assigned scores using reward function, and their parent's reward is calculated using *min-max* strategy. In this figure, every choice of player 1 is followed by the optimal choice of opponent and they are collapsed to a single state. Since the center position has a higher score of $+0$, the $k_1 = 2$ strategy always chooses this move and considers all other positions to be equal with score of -2.

a depth-2 strategy chooses the "center" position and considers all other positions to be equal; a depth-3 strategy chooses the center and also recognizes that the next best choice is one of the four corners. This example is illustrated in the Figure 3.1 and 3.2 for depth 1 and depth 2 strategy, respectively.

Figure 3.3 illustrates the depth-k_1 strategy tree exploration, on an interesting



(a) $k_1 = 1$, column-2 fetches a reward of $+5$

(b) $k_1 = 1$, column-3 fetches reward of $+6$

(c) $k_1 = 1$, column-4 fetches a reward of $+2$

(d) $k_1 = 2$, column-2 fetches a reward of $+3$

(e) $k_1 = 2$, column-3 fetches a reward of $+6$

(f) $k_1 = 2$, column-4 fetches a reward of $+0$

(g) $k_1 = 3$, column-2 fetches a reward of $+\infty$

(h) $k_1 = 3$, column-3 fetches a reward of $+0$

(i) $k_1 = 3$, column-4 fetches a reward of $+0$

Figure 3.3: Illustration of depth-k_1 strategy exploration on a interesting CONNECT-4 RCD starting board position, with player × turn as current player. The figure shows how different depth-k_1 strategies choose the best available position to mark on a Connect-4 RCD. The three depth-k_1 strategies ($k_1 = 1, 2, 3$) play as player-× and assign a score to each of the three available positions (column-$2, 3, 4$) by looking k_1-turns ahead. In each sub-figure, the position with yellow-background is the one chosen for exploration and the positions with grey-background are the predicted moves of how the game might turn out after k_1-turns. As observed in Figure 3.3g, only $k_1 = 3$ strategy is able to foresee that marking column-2 would lead player-X to a winning state and also conclude that the other column choices will lead to a draw. Where as, $k_1 = 1, 2$ incorrectly choose column-3 as the best position to mark, hence making this starting position a *category*-3 state (easy for $k_1 = 3$ but hard for $k_1 = 1, 2$).

CONNECT-4 starting board position. Only $k_1 = 3$ is able to foresee that column-2 is the correct winning choice, while $k_1 = 1, 2$ choose sub-optimal column leading to draw game. Note that the rewards assigned to leaf vertices are based on the vertex itself, without using any look-ahead; and the look-ahead is captured by the classical min-max tree exploration. As the depth increases, the strategies become more intelligent for the game.

Outcomes and Probabilities

Given a starting vertex u, a depth-k_1 strategy σ_1 for player 1, and depth-k_2 strategy σ_2 for player 2, let O be the set of possible outcomes. In other words, O is the set of possible plays given σ_1 and σ_2 from u, where a play is a sequence of vertices.

The strategies are randomized because they choose a random child with lowest-/highest value (alternating min-max depending on player turn) during the search tree exploration from starting vertex, and hence they define a probability distribution over the set of outcomes denoted as $\Pr_u^{\sigma_1,\sigma_2}$. That is, we define $\Pr_v^{\sigma_1,\sigma_2}(\rho)$ as the probability of play ρ occurring in the set of outcomes O given the strategies, and this probability distribution is used to formally define the notion of hardness we consider.

3.2.3 Formalization of Problem Definition

We consider several board games (such as Tic-Tac-Toe, CONNECT-4, and their variants), and our goal is to obtain starting positions of different hardness levels, which is characterized by strategies of different depths.

Hardness

Formally, we define hardness as follows. Consider a starting vertex $u \in W_j$ that is winning for player 1 within j-moves plus one winning move (i.e., $j+1$ moves for player 1 and j moves of player 2). Let σ_1 and σ_2 be a depth-k_1 strategy for player 1 and depth-k_2 strategy for player 2, respectively. Let $O_1 \subseteq O$ be the set of plays that belong to the set of outcomes and is winning for player 1. Let $\Pr_v^{\sigma_1,\sigma_2}(O_1) = \sum_{\rho \in O_1} \Pr_v^{\sigma_1,\sigma_2}(\rho)$ be the probability of the winning plays.

Then, the (k_1, k_2) hardness classification of a starting vertex u is defined as follows:

1. if player 1 wins at least 2/3 times, i.e $\Pr_u^{\sigma_1,\sigma_2}(O_1) \geq \frac{2}{3}$; then we call the starting vertex-u *easy (E)*
2. if player 1 wins at most 1/3 times, i.e $\Pr_u^{\sigma_1,\sigma_2}(O_1) \leq \frac{1}{3}$; then we call it *hard (H)*
3. otherwise, vertex u belongs to *medium (M)* hardness class.

Remark 1. In the definition above, we set the probability boundaries at $\frac{1}{3}$ and $\frac{2}{3}$, to divide the interval $[0, 1]$ symmetrically in regions of E, M, and H, and present our results based on these. These probabilities could be easily changed and further experimented with. Our goal is to consider various games and identify vertices belonging to different categories, such as hard for depth-k_1 vs. depth-k_2 but easy for depth-(k_1+1) vs. depth-k_2, for small values of k_1 and k_2.

Remark 2. In this work we consider classical min-max reasoning for tree exploration. A related notion is Monte Carlo Tree Search (MCTS) which in general converges to min-max exploration, but can take longer time. However, this convergence is much faster in our setting, since we consider simple games that have great symmetry, and explore only small-depth strategies.

3.3 Search Strategy

We now describe the search strategy used for generating starting positions of different hardness levels.

3.3.1 Overall methodology

Generation of j-steps win set

Given a game graph $G = ((V, E), (V_1, V_2))$ along with target sets T_1 and T_2 for player 1 and player 2, respectively, our first step is to compute the set of vertices W_j such that player 1 can win within j-moves. For this, we define two kinds of predecessor operators: one predecessor operator for player 1, which uses existential quantification over successors, and one for player 2, which uses universal quantification over successors.

Given a set of vertices X, let $\mathsf{EPre}(X)$ (called existential predecessor) denote the set of player-1 vertices that has an edge to X; i.e., $\mathsf{EPre}(X) = \{u \in V_1 \mid$ there exists $v \in X$ such that $(u, v) \in E\}$. In other words, it is possible for player 1 to reach X from $\mathsf{EPre}(X)$ in one step.

And let $\mathsf{APre}(X)$ (called universal predecessor) denote the set of player-2 vertices that has all its outgoing edges to X; i.e., $\mathsf{APre}(X) = \{u \in V_2 \mid \text{ for all } (u,v) \in E \text{ we have } v \in X\}$. In other words, irrespective of the choice of player 2 the set X can be reached from $\mathsf{APre}(X)$ in one step.

The computation of the set W_j is defined inductively as follows:

$W_0 = \mathsf{EPre}(T_1)$. Given a vertex of W_0, player 1 can reach winning state T_1 in a single move.

$W_{i+1} = \mathsf{EPre}(\mathsf{APre}(W_i))$. From W_i player 1 can win within i-moves, and from $\mathsf{APre}(W_i)$ irrespective of the choice of player 2 the next vertex is in W_i; and hence $\mathsf{EPre}(\mathsf{APre}(W_i))$ is the set of vertices such that player 1 can win within $(i+1)$-moves.

Exploring vertices from W_j

The second step is to explore vertices from W_j, starting with small values of j. Consider a vertex $v \in W_j$, a depth-k_1 strategy for player 1 and a depth-k_2 strategy for player 2. Then, we play the game multiple times with the starting vertex as v, to find out the hardness level of vertex v with respect to our (k_1, k_2)-classification defined earlier.

Note that for vertices in W_j, player 1 has a guaranteed winning strategy with j-moves; and hence starting states of desired game length can be generated by changing the value of j.

Two key issues

There are two main computational issues associated with the above approach in practice. The first issue is related to the large size of the state space (number of vertices) of the game, which makes representing and analyzing the game graph explicitly, using enumerative approaches, computationally infeasible. For example, the size of the state space of Tic-Tac-Toe 4×4 game is 6,036,001; and a CONNECT-4 5×5 game is 69,763,700 (above 69 million). Any enumerative method would not work for such large game graphs.

The second issue is related to exploring the vertices from W_j. If W_j has a lot of witness vertices, then playing the depth-k_1 vs. depth-k_2 game multiple times from all of them will be computationally expensive. Hence we need a computationally inexpensive initial metric, to guide the search of vertices from W_j.

We solve the first issue with *symbolic methods*, and the second one by *iterative simulation*.

3.3.2 Symbolic methods

In this section, we discuss the symbolic methods used to analyze games with large state spaces. The key idea is to represent the games symbolically using variables, instead of explicit state space representation. In particular, we use BDD (binary decision diagrams) [50], which can efficiently represent a set of vertices in terms of boolean formula, which in turn can be represented as a rooted DAG. We used the tool CuDD [51] for representing the board game state space symbolically using BDDs, and the tool supports standard operations on BDDs, such as EPre and APre.

Symbolic representation of vertices

In symbolic methods, a game graph is represented by a set of variables $x_1, x_2, \ldots, x_n$ such that each one of them takes values from a finite set (e.g., $\times$, $\circ$, and blank symbol); and each vertex of the game represents a valuation assigned to the variables.

For example, the symbolic representation of the game of Tic-Tac-Toe of board size 3×3 consists of ten variables $x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1} \ldots, x_{3,3}, x_{10}$, where the first nine variables $x_{i,\ell}$ denote the symbols in the board position (i, ℓ) and the symbol is either $\times$, $\circ$, or blank; and the last variable x_{10} denotes whether it is player 1 or player 2's turn to play. Note that the vertices of the game graph not only contains the information about the board configuration, but also additional information such as the turn of the players.

To illustrate how a symbolic representation is efficient, consider the set of all valuations to boolean variables $y_1, y_2, \ldots, y_n$ where the first variable is true, and the

second variable is false. An explicit enumeration requires to list 2^{n-2} valuations, where as a boolean formula representation $y_1 \wedge \neg y_2$ is succinct. Symbolic representation with BDDs exploit such succinct representation for sets of vertices, and are used in many applications, e.g. hardware verification [50].

Symbolic encoding of transition function

The transition function (or the edges) are also encoded in a symbolic fashion. Instead of specifying every edge explicitly, the symbolic encoding allows to write a program over the variables to specify the transitions. The tool CuDD accepts such a symbolic description and constructs a BDD representation of the game.

For example, for Tic-Tac-Toe, a program to describe the symbolic transition maintains a set U of positions on the board that are already marked. As the game progresses, at every move it receives an input (i, ℓ) from the set $\{(a, b) \mid 1 \leq a, b \leq 3\} \setminus U$ of remaining board positions, from the player of the current turn. This position (i, ℓ) is then added to the set U, and the variable $x_{i,\ell}$ is assigned value $\times$ or $\circ$ (depending on whether it was player 1 or player 2's turn).

This gives the symbolic description of the transition function.

Symbolic encoding of target vertices

The set of target vertices T_1 and T_2 are encoded as a boolean formula as well. For example in Tic-Tac-Toe, T_1, the set of target vertices for player 1 is given by the following boolean formula:

$$\exists i, \ell.\ 1 \leq i, \ell \leq 3$$

$$((x_{i,\ell} = \times \ \wedge \ x_{i+1,\ell} = \times \ \wedge \ x_{i+2,\ell} = \times)$$

$$\vee\ (x_{i,\ell} = \times \ \wedge \ x_{i,\ell+1} = \times \ \wedge \ x_{i,\ell+2} = \times)$$

$$\vee\ (x_{2,2} = \times \ \wedge \ ((x_{1,1} = \times \ \wedge \ x_{3,3} = \times) \ \vee \ (x_{3,1} = \times \ \wedge \ x_{1,3} = \times))))$$

$\wedge$ Negation of above with $\circ$ to specify player 2 not winning

The above formula states that either there is some column ($x_{i,\ell}, x_{i+1,\ell}$ and $x_{i+2,\ell}$)

that is winning for player 1; or a row ($x_{i,\ell}, x_{i,\ell+1}$ and $x_{i,\ell+2}$) that is winning for player 1; or there is a diagonal ($x_{1,1}, x_{2,2}$ and $x_{3,3}$; or $x_{3,1}, x_{2,2}$ and $x_{1,3}$) that is winning for player 1; and player 2 has not won already.

To be precise, we also need to ensure that the BDD represents all valid board configurations (reachable vertices from the empty board). This is achieved by taking an intersection of the above formula BDD with valid board configurations BDD, to obtain the target set T_1.

Symbolic computation of W_j

The symbolic computation of W_j is as follows: given the boolean formula for the target set T_1, we obtain the BDD representation of T_1. With the help of CuDD, we designed a tool which takes as input a BDD representing set X and supports the operation to return the BDD for $\mathsf{EPre}(X)$ and $\mathsf{APre}(X)$. Thus we obtain the W_j symbolically, using the computation mentioned earlier in Section 3.3.1, in terms of EPre and APre operations on target set T_1.

3.3.3 Iterative simulation

In the previous section, we described how to use symbolic methods to deal with the large state space problem. We now describe a computationally inexpensive (but approximate) procedure to aid sampling of vertices as starting position candidates, of a given hardness level.

Given a starting vertex v, a depth-k_1 strategy for player 1, and a depth-k_2 strategy for player 2, we need to consider the tree exploration of depth $\max\{k_1, k_2\}$ to classify the hardness of v. Hence if either of the strategy is of high depth, then it is computationally expensive to compute the hardness. Thus we need a preliminary metric that can be computed relatively easily for small values of k_1 and k_2, as a guide for vertices to be further explored in depth. We use a very *simple metric* for this purpose. The hard vertices are rarer than the easy vertices, and thus we rule out easy ones quickly using the following approach:

If k_1 is large

Given a strategy of depth k_2, the set of hard vertices for higher values of k_1 are a subset of the hard vertices for smaller values of k_1. Thus we iteratively start with smaller values and proceed to higher values of k_1, only for vertices that are already hard for smaller values of k_1.

For example, given W_j of a board game, to find the set of hard vertices for $k_1 = 3$ against $k_2 = 1$, we first find those states which are hard for $k_1 = 1$ against $k_2 = 1$. That is, for each $v \in W_j$, a game of depth-$k_1 = 1$ vs. depth-$k_2 = 1$ is simulated starting from v board state. Since the number of hard states are rare, all easy states are filtered out and only the remaining few hard states for $k_1 = 1$ are considered for a new round of simulation, for $k_1 = 2$ against $k_2 = 1$. Similarly, only the hard states for $k_1 = 2$ are taken up for evaluation of $k_1 = 3$ against $k_2 = 1$ strategy.

Evaluating a board position for $k_1 = 1$ against $k_2 = 1$ takes few seconds due to the very few paths to explore, and any state that is easy for $k_1 = 1$ should be easy for higher values of k_1.

If k_2 is large

In this case, the above subset rule is not applicable; since any state that is easy against lower values of opponent k_2, need not be easy against higher level of opponent k_2.

Here we exploit the following intuition: Given a strategy of depth k_1, a vertex which is hard for high value of k_2 is likely to show indication of hardness in small values of k_2. Hence we consider the following approach. For each vertex in W_j, we assign a number (called *score*), based on the performance of the depth-k_1 strategy and a small depth (such as $k_2 = 1$) strategy of the opponent. This score indicates the fraction of games won by the depth-k_1 strategy against the opponent strategy of small depth.

The vertices that have low score according to this metric are then iteratively simulated against larger depth-k_2 strategies of the opponent, to obtain vertices of

different hardness level. This heuristic serves as a simple metric to explore vertices for large value of k_2, starting with small values of k_2.

3.4 Framework for Board Games

We now consider the specific problem of board games. In this section, we describe a framework to specify several variants of two-player grid-based board games such as Tic-Tac-Toe, CONNECT-4, and several new variants. Note that, although our implementation of symbolic methods works for the class of traditional board games and their variants, our methodology is applicable to the general class of graph games.

3.4.1 Parameters

Our framework allows three different parameters to generate variants of board games.

1. *Board Size*: For example, the board size could be 3×3; or 4×4; or 4×5 and so on.

2. *Winning Condition*: we experiment with 4 different cases of winning condition

   (a) RCD: denoting the player wins if the moves are in a line along a row (R), a column (C), or the diagonal (D).

   (b) RC: line must be along a row or column, but diagonal lines are not winning.

   (c) RD: winning line must be along a row or diagonal, but not column.

   (d) CD: winning line must be along column or diagonal, but not row.

3. *Valid Moves*: At any point in the game, the players can choose any available column (i.e., column with at least one empty position); after which, they are restricted according to the following setting

   (a) Full gravity: once a player chooses a column, the move is fixed to be the lowest available position in that column.

(b) Partial gravity-ℓ: once a player chooses a column, the move can be one of the bottom-ℓ available positions in the column. Note that full gravity is a special case of partial gravity, with $\ell = 1$.

(c) No gravity: the player can choose any of the available positions in the column.

Using the above parameters, standard Tic-Tac-Toe is described as (i) board size: 3×3; (ii) winning condition: RCD; and (iii) valid moves: no-gravity. Whereas in CONNECT-4, the winning condition is still RCD but valid moves are restricted to full-gravity.

Our framework allows describing multiple new variants of the previous classical games, such as Tic-Tac-Toe in a board of size 4×4 but diagonal lines are not winning (RC); and Bottom-2 (partial gravity-2) which is a crossover between Tic-Tac-Toe and CONNECT games in terms of moves allowed.

Tic-Tac-Toe, Bottom-2 and CONNECT-3 require 3 consecutive positions to be marked for a player to win, while CONNECT-4 requires 4 consecutive positions.

3.4.2 Features

In this work, we provide theoretical approach and implementation results for generating starting vertices (or board positions) of different hardness levels (if they exist), for the class of board games described above. The main features that our implementation supports are: (i) Generation of starting vertices of different hardness level, if they exist; (ii) Playing against opponents of different levels.

We have experimented with opponent depth strategy of $k_2 = 1, 2$ and 3 values. Typically in the board games we considered, depth-3 strategies are sufficiently intelligent and hence we do not explore larger values of k_2. Thus, a learner (beginner) can consider starting with our generated board positions of various hardness levels, and play against opponents of different skill level; in order to hone her ability to play the game, and be exposed to new combinatorial challenges of the game.

3.5 Experimental Results

The main aim of our experimental results is to investigate the existence of interesting starting vertices and their prevalence in simple traditional game variants, for various combinations of expertise levels and winning rules (RCD, RC, RD, and CD), for j small lengths of plays. Moreover, the computation time for searching these states should be reasonable.

Our key finding is that while such vertices do exist, they are rare for most of our game variants; thus their discovery is an important finding and illustrates the significance of our non-trivial search strategy. Interesting starting vertices, which are hard to win in j-steps against against a depth-$k_2 = 3$ strategy of the opponent, exist in Tic-Tac-Toe for depth-$k_1 = 1$ strategies, in Bottom-2 for depth-$k_1 = 1, 2$ strategies, and in CONNECT-4 for depth-$k_1 = 1, 2, 3$ strategies.

Furthermore, we observe the existence of interesting vertices in Tic-Tac-Toe game variants over 4×4 board size, where the default blank start vertex is uninteresting due to its heavy winning bias for player 1.

We next briefly describe our experimental results and important findings, along with some example board configurations.

3.5.1 CONNECT Games

Table 3.2 presents results for CONNECT-3 and CONNECT-4 games, against depth-3 strategies of the opponent. The first column represents the type of the game (CONNECT-3 or CONNECT-4) and the board size (either 4×4 or 5×5). The second column denotes the size of the overall state space of the game. The third column $j = 2, 3$ denotes whether we explore from W_2 or W_3. In our experiments, we explore vertices from W_2 and W_3 set only, as the set W_4 ($j = 4$ turns from winning state W_0) is almost always empty. That is, if there is a winning starting position, it belongs to either W_1, W_2 or W_3; with W_1 states being trivial for even a simple $k_1 = 1$ player 1 strategy.

Game	State Space $\lvert V\rvert$	j	Win Cond	No. of States $\lvert W_j\rvert$	Sampling	$k_2 = 3$								
						$k_1 = 1$			$k_1 = 2$			$k_1 = 3$		
						E	M	H	E	M	H	E	M	H
CONNECT-3	4.1×10^4	2	RCD	110	All	*	24	5	*	3	0	*	0	0
4×4	6.5×10^4		RC	200	All	*	39	9	*	23	5	*	0	0
	7.6×10^4		RD	418	All	*	36	17	*	25	4	*	0	0
	6.5×10^4		CD	277	All	*	41	24	*	27	21	*	0	0
CONNECT-3		3	RCD	0	-									
4×4			RC	0	-									
			RD	18	All	*	0	0	*	0	0	*	0	0
			CD	0	-									
CONNECT-4	6.9×10^7	2	RCD	1.2×10^6	Random	*	184	215	*	141	129	*	0	0
5×5	8.7×10^7		RC	1.6×10^6	Random	*	81	239	*	70	186	*	0	0
	1.0×10^8		RD	1.1×10^6	Random	*	106	285	*	151	82	*	0	0
	9.5×10^7		CD	5.3×10^5	Random	*	364	173	*	209	96	*	0	0
CONNECT-4		3	RCD	2.8×10^5	Random	*	445	832	*	397	506	*	208	211
5×5			RC	7.7×10^5	Random	*	328	969	*	340	508	*	111	208
			RD	8.0×10^5	Random	*	398	1206	*	464	538	*	179	111
			CD	1.5×10^5	Random	*	146	73	*	171	110	*	120	72

Table 3.2: CONNECT-3 & CONNECT-4 results against depth-3 strategy of opponent. The third column (j) denotes whether we explore from W_2 or W_3. The sixth column denotes sampling method used, to select starting vertices; if $|W_j|$ is small, "All" vertices are explored, else "Random" sampling of first 5000 vertices from W_j are explored. The E, M, and H columns report the number of easy, medium, or hard vertices found among the sampled vertices. For each $k_1 = 1, 2, 3$ the sum of E, M, and H columns is equal to the number of sampled vertices, and * denotes the number of remaining vertices; that is, $E = |W_j| - M - H$. Observe that $|W_j|$ is small fraction of $|V|$, which illustrates the significance of our symbolic methods in finding these. Also, observe that vertices labeled medium and hard are a small fraction of the sampled vertices, which illustrates the significance of our efficient iterative sampling strategy.

Game	State Space $\lvert V\rvert$	j	Win Cond	No. of States $\lvert W_j\rvert$	Sampling	$k_2 = 3$								
						$k_1 = 1$			$k_1 = 2$			$k_1 = 3$		
						E	M	H	E	M	H	E	M	H
3×3	4.1×10^3	2	RCD	20	All	*	5	0	*	1	0	*	0	0
	4.3×10^3		RC	0	-									
	4.3×10^3		RD	9	All	*	2	1	*	3	0	*	0	0
	4.3×10^3		CD	1	All	*	0	0	*	0	0	*	0	0
3×3		3	RCD	0	-									
			RC	0	-									
			RD	0	-									
			CD	0	-									
4×4	1.8×10^6	2	RCD	193	All	*	12	26	*	0	2	*	0	0
	2.4×10^6		RC	2709	All	*	586	297	*	98	249	*	0	0
	2.3×10^6		RD	2132	All	*	111	50	*	18	16	*	0	0
	2.4×10^6		CD	1469	All	*	123	53	*	25	8	*	0	0
4×4		3	RCD	0	-									
			RC	90	All	*	37	31	*	0	0	*	0	0
			RD	24	All	*	1	2	*	0	0	*	0	0
			CD	16	All	*	6	4	*	1	0	*	0	0

Table 3.3: Bottom-2 results against depth-3 strategy of opponent. The third column (j) denotes whether we explore from W_2 or W_3. The E, M, and H columns report the number of easy, medium, or hard vertices found among the sampled vertices. For each $k_1 = 1, 2, 3$ the sum of E, M, and H columns is equal to the number of sampled vertices, and * denotes the number of remaining vertices; that is, $E = |W_j| - M - H$. Observe that $|W_j|$ is small fraction of $|V|$, which illustrates the significance of our symbolic methods in finding these. Also, observe that vertices labeled medium and hard are a small fraction of the sampled vertices, which illustrates the significance of our efficient iterative sampling strategy.

Game	State Space $\lvert V\rvert$	j	Win Cond	No. of States $\lvert W_j\rvert$	Sampling	$k_2 = 3$								
						$k_1 = 1$			$k_1 = 2$			$k_1 = 3$		
						E	M	H	E	M	H	E	M	H
3×3	$5.4{\times}10^3$	2	RCD	36	All	*	14	2	*	0	0	*	0	0
	$5.6{\times}10^3$		RC	0	-									
	$5.6{\times}10^3$		RD	1	All	*	0	0	*	0	0	*	0	0
	$5.6{\times}10^3$		CD	1	All	*	0	0	*	0	0	*	0	0
3×3		3	RCD	0	-									
			RC	0	-									
			RD	0	-									
			CD	0	-									
4×4	$6.0{\times}10^6$	2	RCD	128	All	*	6	2	*	0	0	*	0	0
	$7.2{\times}10^6$		RC	3272	Lowest-100	*	47	22	*	0	0	*	0	0
	$7.2{\times}10^6$		RD	4627	Lowest-100	*	3	2	*	0	0	*	0	0
	$7.2{\times}10^6$		CD	4627	Lowest-100	*	3	2	*	0	0	*	0	0
4×4		3	RCD	0	-									
			RC	0	-									
			RD	4	All	*	0	0	*	0	0	*	0	0
			CD	4	All	*	0	0	*	0	0	*	0	0

Table 3.4: Tic-Tac-Toe results against depth-3 strategy of opponent. The third column (j) denotes whether we explore from W_2 or W_3. The sixth column denotes sampling method used, to select starting vertices; if $|W_j|$ is small, "All" vertices are explored, else "Lowest-100" sampling of 100 least scored vertices (according to iterative simulation score) from W_j are explored. The E, M, and H columns report the number of easy, medium, or hard vertices found among the sampled vertices. For each $k_1 = 1, 2, 3$ the sum of E, M, and H columns is equal to the number of sampled vertices, and * denotes the number of remaining vertices; that is, $E = |W_j| - M - H$. Observe that $|W_j|$ is small fraction of $|V|$, which illustrates the significance of our symbolic methods in finding these. Also, observe that vertices labeled medium and hard are a small fraction of the sampled vertices, which illustrates the significance of our efficient iterative sampling strategy.

The fourth column denotes the winning condition (RCD, RD, RC, CD). The fifth column denotes the number of starting states found in W_j. The last three columns describe the analysis of the W_j states with respect to depth-$k_1 = 1, 2, 3$ strategies for player 1, and depth-3 strategy of the opponent. The sixth "sampling" column describes whether we analyzed *all* the states in W_j, or a *random* 5000 starting states from it owing to large size of W_j.

Starting states from W_j are classified into Easy (E), Medium (M) or Hard (H) categories, by simulating a game between the depth-k_1 vs the depth-k_2 strategy, for 30 different runs. If player 1 wins

1. More than $\frac{2}{3}$ times (20 times), then the starting state is counted under Easy (E)

2. Less than $\frac{1}{3}$ times (10 times), then the starting state is counted under Hard (H)

3. Otherwise, the starting state is identified as Medium (M)

An interesting finding from these results is that in CONNECT-4 games with board size 5×5, for all winning conditions (RCD, RD, CD, RC), starting vertices exist for easy, medium, and hard categories, for $k_1 = 1, 2$ and 3, when $j = 3$. That is, even in much smaller board size (5×5 as compared to the traditional 7×7), we discover interesting starting positions for CONNECT-4 games and its simple variants.

3.5.2 Bottom-2 Games

Table 3.3 shows the results for Bottom-2 (partial gravity-2) against depth-3 strategies of the opponent. The meaning of the entries are similar to the ones described earlier for CONNECT games. In contrast to CONNECT games, we observe that interesting starting vertices (medium or hard) do not exist for depth-$k_1 = 3$ strategies of player 1.

3.5.3 Tic-Tac-Toe Games

The results for Tic-Tac-Toe games are shown in Table 3.4. For Tic-Tac-Toe 4×4 games, the strategy exploration is expensive since a tree of depth-3 requires analysis of 10^6 nodes. Hence using the iterative simulation techniques, we first assign a score to all vertices and further explore only the bottom hundred vertices; that is, hundred vertices with the lowest score according to our iterative simulation metric.

In contrast to CONNECT games, we observe that interesting vertices exist only for depth-$k_1 = 1$ strategies, but not for depth-$k_1 = 2$ and depth-$k_1 = 3$ strategies.

3.5.4 Results Summary

Game	Category-1	Category-2	Category-3	Category-4
Tic-Tac-Toe	All variants	3x3: only RCD 4x4: All j=2 variants		
Bottom-2	All variants	3x3: only RD 4x4: All variants	4x4: All j=2 variants	
CONNECT-3	All variants	All j=2 variants	All j=2 variants except RCD	
CONNECT-4	All variants	All variants	All variants	All j=3 variants

Table 3.5: Summary of interesting states for various games

In Table 3.5, we summarize the different games and the existence of category i states in such games. A state is denoted as *category-i* state if it is easy for depth-i strategy, but not easy for depth-$(i-1)$ strategy.

Our first key finding is the *existence of vertices of different hardness levels* in various traditional board games. We observe that in Tic-Tac-Toe games, only board positions that are hard for $k_1 = 1$ (Category-2) exist. In particular, and more interestingly, such states also exist in board of size 4×4. Since the default (blank) starting vertex in 4×4 Tic-Tac-Toe games is heavily biased towards the player 1 who starts first, they have been believed to be uninteresting for ages; the discovery

of interesting starting vertices by us could make them playable for enthusiasts. With the slight variation of allowable moves (Bottom-2), we obtain board positions that are hard for $k_1 = 2$. In Connect-4 we obtain vertices that are hard for $k_1 = 3$, even on our smaller board size variant of 5×5.

The second key finding of our results is that *the number of interesting vertices is a negligible fraction of the huge state space.* For example, while Bottom-2 RCD game with board size 4×4 has state space size $|V|$ of over 1.8 million, it has only two positions which are hard for $k_1 = 2$. Similarly, CONNECT-4 5×5 RCD games with state space of more than sixty-nine million has around two hundred hard vertices for $k_1 = 3$, among the five thousand vertices sampled randomly from W_3. Since the size of W_j in this case is 2.8×10^5, the potential number of hard vertices could be twelve thousand (among the sixty-nine million overall state space size). In other words, since the interesting positions are quite rare, a naive approach of randomly generating positions and measuring its hardness would be impractical, akin to searching for a needle in a haystack. Thus there is need for a non-trivial search strategy like ours (Section 3.3), which we implement and demonstrate.

We remark that the default (blank) starting vertex of Tic-Tac-Toe 3×3 and Connect-4 5×5, does not belong to any winning set W_j. However in Tic-Tac-Toe 4×4, it belongs to the winning set W_2 and is easy for all depth strategies.

Additional results for depth-$k_2 = 2$ strategy of the opponent are provided in the Appendix A.1.

Running Time

The generation of W_j for $j = 2$ and $j = 3$ took between 2–4 hours per game variant. Note that this is a one-time computation for each game. The evaluation time to classify a starting state as E, M, or H for depth-$k_1 = k_2 = 3$ strategies of both players, playing 30 times from a board position takes on average

- 12 seconds for CONNECT-4 games with board size 5×5
- 47 seconds for Bottom-2 games with board size 4×4

- 25 minutes for Tic-Tac-Toe games with board size 4×4

Note that the size of the state space is around 10^8 for CONNECT-4 games with board size 5×5, and testing the winning nature of a state takes at least few seconds. Hence an explicit enumeration of the state space cannot be used to obtain W_j in reasonable time; in contrast, our symbolic methods succeed to compute W_j efficiently. Figure 3.3 lists a category-3 state for CONNECT-4 RCD, which is easy for $k_1 = 3$ strategy, but hard for $k_1 = 1, 2$ strategies of player 1.

3.5.5 Example board position

O		X	
X		O	X
		X	O
O		X	O

(a) Tic-Tac-Toe RC, $k_1 = 1$

		X	
X			
	O		
O	X	O	

(b) Tic-Tac-Toe CD, $k_1 = 1$

	X		
	O		X
			O

(c) Bottom-2 RCD, $k_1 = 2$

		X	
X		O	
O	O	X	

(d) Bottom-2 RC, $k_1 = 2$

O				O
X		X		X
X		O		X
X	X	O	X	O
O	O	O	X	O

(e) CONNECT-4 RCD, $k_1 = 2$

O		O		
X		X	X	
O		O	X	
O		O	X	
O	X	O	X	X

(f) CONNECT-4 RD, $k_1 = 3$

Figure 3.4: Some "Hard" starting board positions generated by our tool, for variety of games and different expertise level k_1 of player 1. The opponent expertise level k_2 is fixed at 3. Player 1 (X) can win in 2 steps for games (a)-(e) and in 3 steps for game (f).

In Figure 3.4, we present examples of several board positions that are hard to win (winning probability $< \frac{1}{3}$) for strategies of certain depth. In all the figures, player-X is the current player against opponent of depth-3 strategy. All these board positions were discovered automatically through our experiments.

3.6 Related Work

Tic-Tac-Toe and Connect-4

In prior work, Tic-Tac-Toe has been generalized to different board sizes, match length [52], and even polyomino matches [53], to find variants that are interesting from the default blank start state. Existing research has focussed on establishing which of these games have a winning strategy [54, 55, 56]. In contrast, we show that even simpler variants can be interesting if we start from certain specific states. Existing Connect-4 research has focussed on establishing a winning strategy from the default starting state for the first player [47]. In contrast, we study how easy or difficult it is to win from new winnable starting states, given different expertise levels. BDDs have been used earlier to represent board games [57], and perform Monte Carlo Tree Search (MCTS) run with Upper Confidence bounds applied to Trees (UCT). In such usage, BDDs are instantiated to find the number of states explored by a single agent. In our setting we have two players, and use BDDs to compute the winning set.

Level generation

Our proposed technique, which generates starting states given certain parameters (namely, expertise levels of players, number of steps to win and search tree depth), can be used to generate different game levels by simply varying the choice of the parameters along some partial/total order. Here we discuss some related work from the area of level generation.

The problem of level generation has been studied for specific games. Goldspinner [58] is a level generation system for KGoldrunner, a puzzle game with dynamic elements. It uses a genetic algorithm to generate candidate levels, and uses simulation to evaluate dynamic aspects of the game. We similarly use simulation to evaluate the dynamic aspect, but use symbolic methods instead to generate candidate states; also, our system is parameterized by game rules.

Most other related work has been restricted to games that lack opponent and dynamic content, such as Sudoku [59, 60]. Smith et al. [61] used answer-set programming to generate levels for Refraction, an educational puzzle game, that adhered to pre-specified constraints written in first-order logic. Similar approaches have also been used to generate levels for platform games [62]. In these approaches, designers must explicitly specify constraints on the generated content; e.g., the tree needs to be near the rock and the river needs to be near the tree. In contrast, our system takes as input rules of the game and does not require any further help from the designer.

A similar model is used by Andersen et al. [38], which applies symbolic methods (namely, test input generation techniques) to generate various levels for DragonBox (an algebra-learning video game that became the most purchased game in Norway on the Apple App Store [63]). In contrast, we use symbolic methods for generating valid start states, and use simulation for estimating their hardness level.

Problem Generation

Automatic generation of fresh problems can be a key capability in intelligent tutoring systems [1]. The technique for generation of algebraic proof problems [14] uses probabilistic testing to guarantee the validity of a generated problem candidate (from abstraction of the original problem) on random inputs, but offers no guarantee of the hardness level. In our work, simulation can be linked to this probabilistic testing approach with hardness level guarantee, where validity is guaranteed by symbolic methods.

The technique for generation of natural deduction problems, presented in previous Chapter 2, and the work on geometry proof problems [15] involves a backward existential search over the state space of all possible proofs for all possible facts to generate problems with a specific hardness level. In contrast, we employ a two-phased strategy of backward and forward search; backward search is necessary to identify winning states, while forward search ensures hardness levels. Furthermore, our state transitions alternate between different players, thereby necessitating alternate

universal vs. existential search over transitions.

3.7 Summary

In this chapter, we study a novel aspect of graph games, namely automatically generating interesting starting states (W_j) of varying hardness levels (E, M, H), parameterized by look ahead depth of both player 1 ($k_1 = 1, 2, 3$) and player 2 ($k_2 = 1, 2, 3$), the graph game description and number of winning steps ($j = 2, 3$).

Our novel search methodology, a combination of symbolic methods and iterative simulation, allows us to efficiently search the vast search space of our board game variants. Our experiment results report on the two important findings: (i) the *existence of vertices of different hardness levels* in various traditional board games, and (ii) *the number of interesting vertices is a negligible fraction of the huge state space*. Thus, there is a need for non-trivial search strategy such as ours, to find these rare starting states.

Interesting starting states that require few steps to play and win are often published in newspapers for sophisticated games like Chess and Bridge. These are usually obtained from database of past games. In contrast, we show how to automatically generate such states, albeit for simpler traditional games that form a Directed Acyclic Graph. This is an important result in itself since the existence of such states, even in simple traditional games, has not been explored before. Such interesting states can aid novitiates master the popular traditional games by keeping them engaged, given the customizable length of play and guarantee of a win if played efficiently.

Chapter 4

Compilation Error Repair

CS–1, the Introductory to Programming course, is one of the most popular courses with class sizes reaching 1000+ students in some universities [64], and up to hundreds of thousands on Massive Open Online Courses (MOOCs) [65]. The Taublee survey [66] reports that the number of undergraduates enrolling for computer-science (CS) majors in the United States (US) universities has steadily increased for ten consecutive years, with an additional 11.4% new students added in the 2017–2018 year per university on average. This double-digit growth is estimated to continue for the near future. At the same time, an increasing number of non-CS major students are opting for lower-level (such as introductory programming) CS courses as well [66]. This increased enrollments is reportedly overwhelming the CS faculties on how to effectively impart quality education for all, requiring them to either raise significant amount of resources to meet the demand, or turn away enthusiastic students [67]. One of the fundamental challenge when operating at such massive class sizes lies in providing personalized feedback to students who are completely new to programming. These students are also referred to as novice programmers in the rest of our thesis.

While attempting their programming assignments, students run into compilation errors and logical errors. Compilation errors are those errors that are detected and reported by a compiler, usually when the program does not confine to the syntax of programming-language grammar. While logical errors are those errors due to which the program does not match some pre-defined specification, such as test-cases

containing input and its desired output. In order to help students who get stuck while programming, dozens of automated code repair tools are proposed every year [68]. These tools take the incorrect student solution (also referred to as erroneous or buggy code), and provide a complete or partial hint to help repair the error. This hint is typically in the form of direct code-edits, or textual/visual feedback to aid in identifying the error or fix.

The field of automated code repair has traditionally focused on fixing logical errors, upon being given requirement specifications (such as test-suites), while not paying much attention to compilation errors [69]. This was deemed acceptable, given advancements in compiler techniques which made identifying and fixing compile-time errors by experienced developers relatively straight-forward. However, compile-time errors pose a major learning hurdle for students of introductory programming courses. Compiler error messages, while accurate, are targeted at seasoned programmers and seem cryptic to beginners [70], thereby being an impediment to effective programming. Also, the error messages returned by standard compilers are often generic and only with experience do programmers figure out the common causality of such errors and the appropriate repair. Unfortunately, this issue has been largely ignored by compiler designers and better error messages are usually a low priority feature [70]. For those beginning to learn a new language, and hence unfamiliar with programming constructs of that language, compile time errors can be very confusing and time consuming to fix [71]. This is especially true for those who lack prior experience in any form of programming, learning their very first language.

Figures 4.1 and 4.2 illustrate actual attempts by students in the very first lab session of an introductory course on C programming, as well as the actual fixes proposed by our method TRACER. The captions list the messages returned by the Clang compiler [72], a popular compiler for the C language. It is clear that the compiler error messages in these examples are not very informative for a student who has just been introduced to the concepts of formal programming with datatypes and operators.

```
1 #include<stdio.h>
2 int main(){
3  int a;
4  scanf("%d", a );
5  printf("ans=%d",
6   a+10);
7  return 0;
8 }
```

```
1 #include<stdio.h>
2 int main(){
3  int a;
4  scanf("%d", &a );
5  printf("ans=%d",
6   a+10);
7  return 0;
8 }
```

Figure 4.1: Left: erroneous program, Right: fix by TRACER. The compiler message read: *Line-4, Column-9: warning: format '%d' expects argument of type 'int *', but argument 2 has type 'int'.*

```
1 #include<stdio.h>
2 int main(){
3    int x,x1,d;
4    // ...
5    d=(x-x1) (x-x1);
6    return d;
7 }
```

```
1 #include<stdio.h>
2 int main(){
3    int x,x1,d;
4    // ...
5    d=(x-x1)*(x-x1);
6    return d;
7 }
```

Figure 4.2: Left: erroneous program, Right: fix by TRACER. The compiler message read: *Line-5, Column-11: error: called object type 'int' is not a function or function pointer.*

In the first case, the error message does not provide any valuable feedback to a student unfamiliar with the concept of pass-by-reference vs pass-by-value. It may be frustrating to the student that a format that is valid for printf (%d, followed by variable name) is being flagged as an error for the scanf invocation. In the second case, the compiler error message may actually mislead the student since the error in the program is the trivial omission of an arithmetic operator whereas the compiler is interpreting it as an illegal function invocation, simply because parentheses are involved.

These gaps arise since compiler error messages are written for expert programmers and assume that the programmer has a thorough understanding of advanced concepts such as variable addresses, pointers and function invocation. However, these concepts are generally covered much later in a programming course. Thus, a novice programmer is unable to comprehend the error message or the cause of the error. In our course offering at IIT Kanpur, around half of the students made similar errors and although the fix is simple in all these cases, students did require the help of a teaching assistant

to understand the error messages and apply the fixes.

We do observe in our course offerings that the time taken by students to fix common compile-time errors decreases over time, as they get more comfortable with grammar of the language and adapt to the compiler messages. However, this takes a considerable amount of effort and supervision from teaching assistants who have to not only help students correct mistakes, but also help them understand the cause of the error.

With a move towards Massive Open Online Courses (MOOCs) where thousands of students may enroll, it is infeasible to provide human assistance in this manner, even in the initial phase. Moreover, there is evidence [71, 73] that across such programming courses, the errors made by students in the initial phases are quite similar. This points to a lucrative potential for automating this repetitive and monotonous role played by teaching assistants.

The work of Traver [70] attempts to address this problem by offering more informative error messages that would aid programmers in easier diagnoses. There has also been work on designing custom compilers for novice users [74]. These compilers aim for better error recovery and correction than standard compilers, so as to offer better feedback. However, these require a significant amount of effort from compiler designers, and the effort has to be replicated for every compiler being used. Moreover, some studies [75] have shown that additional information in the form of enhanced error messages does not seem to be very helpful, especially for novice programmers.

In this chapter, we address the problem by proposing a pedagogically-inspired program repair tool called TRACER (**T**argeted **R**ep**A**ir of **C**ompilation **ER**rors) [76], a system for performing repairs on compilation errors, aimed at introductory programmers.

4.1 Chapter Outline

This chapter is organized as follows: We start by outlining our proposal and contributions in this Section 4.1. Then in Section 4.2 we describe the processes adopted to prepare data to train the learning algorithms present in TRACER. Section 4.3 presents details of various modules that constitute TRACER. Section 4.4 presents a brief overview of the deep learning techniques used by TRACER. Section 4.5 explains the experimental setup, followed by Section 4.6 which presents the results of an extensive evaluation of TRACER on programs taken from an actual introductory programming course. Section 4.7 then concludes the discussion by presenting a more detailed literature review and outlining some future directions for this effort.

4.1.1 Technique Proposal

In this chapter we present TRACER, a method for automatically generating template repairs for buggy programs that face compilation errors. This problem has generated significant interest recently. However, the guarantees offered by existing works [11, 73, 77] are rather modest and simply offer compilable code in a large fraction of instances.

In comparison, the design of TRACER is based on a realization that the goal of program repair in pedagogical settings is not to simply eliminate compilation errors using any means possible (such as performing trivial fixes by deleting the error lines), but rather to reveal the omissions to the students, so that they may learn how to correct similar errors by themselves in the future [78]. In fact, it may be argued that offering the exact repair to the student, in the form of a compilable program which many existing works do [73, 11, 17], defeats the purpose of learning and may even pose challenges in course evaluations.

To ameliorate the problem, TRACER offers *targeted* corrections that pinpoint the source of the error, as well as recommend the fix actually desired by the student, thereby offering compilable code as a *by-product*. The techniques adopted by

TRACER for program repair are motivated by a common observation [17, 73] that in practice, programmers (especially novice programmers) use a rather small subset of rules and productions of the entire programming language grammar. Thus, it should be possible to repair an erroneous program by mapping it to a similar program (past code obtained from students themselves) that is known to be compilable, and hence (at the very least) syntactically correct. Figures 4.1 and 4.2 demonstrate some of the fixes TRACER can successfully apply to actual programs.

To this end, TRACER adopts a modular, four-phased methodology for code repair which involves

1. **Repair Localization**: TRACER locates the line(s) where repair must be performed.

2. **Code Abstraction**: TRACER abstracts the erroneous code lines, by replacing program specific tokens with their generic types.

3. **Abstract Code Repair Prediction**: TRACER attempts to identify the intent of student and recommends an abstracted form of the repair to the erroneous line(s), based on sequence-to-sequence prediction techniques that use recurrent-neural-networks (RNNs).

4. **Concretization**: TRACER finally converts the abstract repair into actual code that can be compiled.

4.1.2 Our Contributions

The above described approach presents a significant departure from existing works [11, 73, 77], which also adopt deep learning techniques such as recurrent-neural-networks (RNNs), but in a very *monolithic* manner. It is common in existing works to simply feed the entire erroneous program into a deep network and expect repairs as an output. In contrast, TRACER's modular approach to error repair offers several key advantages over the current state-of-the-art.

1. To the best of our knowledge, TRACER is the first system to be able to successfully reproduce the exact fix expected by the student on an erroneous program, and not merely reduce compilation errors.

2. Even when comparing compilation repair accuracy, TRACER offers success rates that are far superior to the state-of-the-art.

3. TRACER offers repairs in both abstract and concretized forms. Depending on the learning objectives set by the instructor, either may be offered to students. In particular, the concretized code may be redacted and just the abstract form offered if it is desired that the student identify the form of the error from this feedback rather than simply receive corrected code. This is not possible with existing techniques.

4. Each of the four phases can be improved upon independently, or be replaced with a different technique, to increase the overall accuracy of the system.

The modular structure of TRACER also helps it harness the power of deep learning techniques in a focused manner. In particular, the abstraction phase implicitly performs a *vocabulary compression* step that greatly eases the working of neural networks.

Note that although the TRACER system is presented for a C-programming environment, versions of TRACER may be readily developed for other languages as well. As we shall see in future sections, the language specific components of the system are minimal, as is the manual effort required to port the system to a new programming language.

4.2 Data Preparation

TRACER learns to recommend error repairs by observing real student mistakes and fixes. To train TRACER, we obtained student programming submissions from the 2015–2016 fall semester offering of an Introductory to C programming course (CS1)

at the Indian Institute of Technology Kanpur (IIT-K). Below we describe the steps taken to create a training subset for TRACER from these submissions.

4.2.1 Raw Data Collection

Our data was collected using Prutor [79], an online system that captures intermediate versions of student program/code, in addition to the final submissions. These programs are recorded while students attempt weekly assignments under the invigilation of teaching assistants. Prutor is capable of taking snapshots of the student code at every compilation request, as well as at regular intervals. Thus, the system gives us a sequence of programs that tracks the progress of each student in solving a question.

4.2.2 Source-Target Pair Identification

Given this raw data, we filtered the sequence of programs to obtain *source-target* program pairs as follows. We identified successive snapshots of the student code (say C_t and C_{t+1}) such that

1. C_t is a code attempt by student at time t
2. C_{t+1} is the code attempt by same student for the same question, at time $t+1$ (immediately following C_t)
3. Compilation of C_t resulted in at least one compilation error
4. Compilation of C_{t+1} did not produce any compilation errors

Then, C_t is called the *Source Program* and C_{t+1} is called the *Target Program*. The difference between the target and source program is the set of changes performed by the student on source (buggy) program, in order to compile the program successfully. These program pairs were further categorized into those pairs where the programs C_t and C_{t+1} differ at a single line (single-line edits) and those that differed on multiple lines (multi-line edits). Ostensibly, the former correspond to programs where the error was confined to a single line whereas in the latter, the error (and hence the

Lab	# Programs	Topic
Lab 1	524	Hello World
Lab 2	1,643	Simple Expressions
Lab 3	1,015	Simple Expressions, printf, scanf
Lab 4	901	Conditionals
Lab 5	1,104	Loops, Nested Loops
Lab 6	1,098	Integer Arrays
Lab 7	1,232	Character Arrays (Strings) and Functions
Lab 8	1,023	Multi-dimensional Arrays (Matrices)
Lab 9	660	Recursion
Lab 10	466	Pointers
Lab 11	453	Algorithms (sorting, permutations, puzzles)
Lab 12	726	Structures (User-Defined data-types)
Exam	3,427	Mid-term and End-term programming tests
Practice	9,003	Practice problems released regularly
ALL	23,275	Total # single-line error programs

Table 4.1: Single-line compilation error dataset (singleL)

student edits) were present on multiple lines. For single-line edit program pairs, the line in C_t that was modified is called the *Source Line* and the corresponding line in C_{t+1} is called the *Target Line*.

For example, in Figure 4.1 (respectively 4.2), line number 4 (respectively line number 5) in the left and the right hand programs is the source and the target line for that pair of programs. While we only use single-line edit program pairs in training TRACER, Section 4.3.5 reports on the technique to handle multi-line edit programs as well.

We observe that while we train TRACER only on genuine source-target pairs (actual attempts by students), other learning approaches such as DeepFix [73] use program pairs created by artificially introducing errors into a correct program. This alternate approach can be tedious to create and manage. Moreover, it is not clear if this helps the system predict realistic corrections desired by students.

4.2.3 Dataset Statistics

The 2015–2016 fall semester offering of CS1 was credited by 400+ first year undergraduate students at IIT–K university. One of the main graded component of the

course was weekly programming assignments (termed *Labs*). These assignments had a specific theme every week, as described in Table 4.1, so as to test the concepts taught in the class so far. We saved the progressive versions of the attempts students made, towards the goal of passing as many pre-defined test-cases as possible. Multiple submission attempts were allowed, with only the last submission being graded.

For each of these labs, we pick ($\boldsymbol{C}_t$, $\boldsymbol{C}_{t+1}$) program pairs as our (source, target) dataset, where $\boldsymbol{C}_t$ is a version of a student program that fails to compile (source), and $\boldsymbol{C}_{t+1}$ is a later version of the attempt by the same student that compiles successfully (target). The second column of Table 4.1 lists the number of such program pairs found in our dataset requiring single-line edit/repair, for each corresponding lab in first column.

From student submissions across the entire run of the semester-long course, we obtained 23,275 and 17,451 source-target program pairs having single-line and multi-line edits respectively. As mentioned before, we utilize only single-line edit pairs for the training phase. These correspond to programs that require edits to a single line in the erroneous program. However, despite being trained on only single-line edit pairs, TRACER is able to seamlessly handle programs requiring repairs on multiple lines as well by simply invoking TRACER repeatedly to fix each individual line separately, as described in Section 4.3.5.

We refer the reader to Table 4.2 for an overview of the compilation errors encountered in our dataset of 23,275 program pairs requiring single-line edits, and Figure 4.3 for an initial look at the performance of TRACER on this dataset. Table 4.2 lists the different types of compilation errors that were present in at least 50 submissions in our dataset, ranked in decreasing order of frequency. Each compilation error code refers to a generalized error message returned by Clang [72] compiler. For example, individual error messages (returned by Clang) such as "Expected ⬚", "Expected ⬚ after expression", "Expected ⬚ at the end of declaration", … are all subsumed by the error-code **E1** – *expected* ⬚. Where, ⬚ is a placeholder for program specific token such as '{', ';', variable-name, …

Code	Error Message	Code	Error Message
E1	expected ⬚	E9	too few args to func call
E2	undeclared ⬚	E10	expected decl or statement
E3	expected expression	E11	called object not a func
E4	extraneous ⬚	E12	invalid digit in const
E5	incorrect assignment	E13	too many args to func call
E6	re-definition of ⬚	E14	return from a void function
E7	invalid operands	E15	statement not in loop/switch
E8	incorrect pointer/struct	OTH	Others

Table 4.2: Top frequent compilation error codes

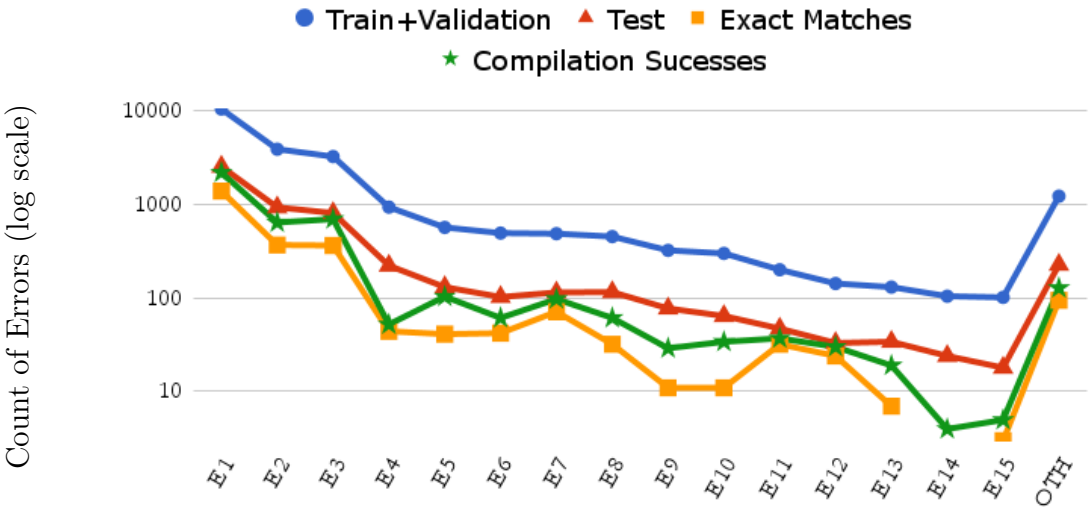


Figure 4.3: Compilation error accuracy plot, with y-axis in log-scale. TRACER has high accuracy across different kinds of compilation errors.

Figure-4.3 then charts the frequency of various error types in the entire single-line *training set* (blue ∘) and the held out *test set* (red △). The *compilation success* (green ∗) plot depicts what number of errors of each type was TRACER able to successfully rectify in the test set, such that the resultant code compiles. The orange plot (□) indicates how many times the top recommendation of TRACER was the same *exact match* as the (abstracted) fix expected by the student for that error. In other words, *exact match* ⊆ *compilation success* ⊆ *test set* [1]. Despite this extremely stringent criterion of exactly matching with the students' fix, TRACER's top recommended

[1]This holds true assuming perfect concretization module. In practice, a small percentage of abstract predictions, some of which exactly match students' abstract fix, cannot be successfully converted to a compilable concrete code by our concretization module (refer Section 4.3.4).

fix has an *exact match* accuracy (also referred to as *abstraction match* or *prediction* accuracy in the thesis) of more than 40% for most error types. If we consider its top-3 recommendations instead, TRACER predicts the exact same fix for a much higher 74% of errors, on average across error types. We also note that TRACER excels not only on frequent error types such as missing identifiers (E1), but also on rare error types that have less than 200 instances in the dataset (Eg, E11: 78% correct, E12: 90% correct).

4.3 TRACER

In this section, we describe the technical details of the TRACER system. Most of this discussion will focus only on program pairs that have single-line edits. Section 4.3.5 will describe on how to adapt TRACER to handle multi-line edit programs. To recollect, in single-line edit program pairs we can identify a source and a target line, with target line being the fix that student applied to rectify the error in source line.

TRACER treats compilation-error repair as a sequence prediction problem. Given a source line, TRACER interprets it as a sequence of tokens, and attempts to predict a new sequence of tokens, that hopefully correspond to the target line. The framework of recurrent neural networks (RNNs) is utilized to implement this. However, several augmentations are required for this strategy to succeed.

In particular, given a faulty program, the task of localizing the repair is in itself a non-trivial task. Some of the existing work, such as DeepFix [73], expect a neural network to jointly perform localization and correction. However, our results show that this monolithic approach can overwhelm the underlying neural network architecture. To remedy this, TRACER performs localization as a separate, modular step. A side advantage of delinking repair localization from error correction is that TRACER is able to use different techniques for localization and correction, which gives it more freedom for fine tuning.

4.3.1 Repair Localization

While the compiler error messages does report the exact line number of the code which resulted in an error state during compilation, this isn't necessarily the same line where repair needs to be performed. This problem has been addressed by prior work in different ways: HelpMeOut [17] targets the exact line number reported by compiler, sk_p [77] performs a brute force replacement starting from the first statement to the last, while DeepFix [73] trains a deep-network on the entire buggy source-program (encoded with line-numbers) to generate a ranked list of potential lines to focus their repair on.

The strategy adopted by TRACER for repair localization is based on a useful observation: in our dataset of introductory C programs where single-line edits were performed by students, the location of the edits lay very close to the line where the compiler flagged an error. In 87.79% of the cases, the students' source-target pairs were located at a distance of <= 1 from the line number reported in the compiler error message i.e. immediately above, immediately below, or at the compiler-reported line. This suggests a surprisingly simple strategy for repair localization: obtain a line number l from the compiler error message (recall that we are considering single-line edit programs for now) and simply consider the line numbers $l-1$, l, $l+1$ as candidate lines to attempt repair.

What is more surprising is that this simple approach achieves almost the same repair localization accuracy as much more involved techniques in literature. For instance, DeepFix [73] utilize a deep network to perform repair localization and report 87.5% accuracy in correctly identifying the location of the error (in one of the top-5 predicted lines). TRACER achieves slightly higher accuracy of 87.79% on the same dataset, using a simpler technique.

We observe that majority of the ~ 12% repair localization failures are due to incorrect opening/closing of braces and variable undeclared issues. That is, for these two cases, the erroneous lines reported by compiler (error-localization) need not align with those lines where the repair is performed (repair-localization). Figure 4.4

```
1 #include<stdio.h>
2
3 int main(){
4  printf("Hello");
5  }
6
7  return 0;
8 }
```

Line #	Error ID	Compiler Message
7	E1	expected identifier or "("
8	E4	extra closing brace "}"

Figure 4.4: Repair localization failure example. Line 5 in the above buggy program contains a spurious closing brace, which needs to be removed to repair it. Compiler reports the line 7 and 8 as erroneous instead, which doesn't match with the repair location.

demonstrates one such buggy program.

4.3.2 Source and Target Abstraction

The second point of departure that TRACER makes from the state-of-the art is in applying a pre-processing step before feeding the sequence of tokens to deep neural networks, and a post-processing step after prediction. That is, TRACER is trained to identify and predict the fixes over abstracted program tokens, instead of learning over the original student programs.

Techniques such as recurrent neural networks operate with a static *vocabulary*. As a result, supplying student programs directly to these networks requires all possible identifier/literal names possibly used by students to be included in the vocabulary. This not only blows up the vocabulary size, but also creates problems for extending the approach to newer offerings of the course where students may use hitherto unused identifier names and literals.

To remedy this problem, TRACER takes source-target line pairs and processes them by replacing all literals and identifier/variables with abstract tokens representing their corresponding types. The types are inferred using LLVM [72] (the back-end for the Clang compiler suite), a standard static analysis tool.

However, there are exceptions to the above rule. The names of keywords and some

standard identifiers/library-functions such as `{printf, scanf, malloc, NULL}` are retained during abstraction. User defined function names are replaced by a generic token `FUNC`. A special token called `INVALID` is used for those literals and identifiers for which static analysis is unable to reveal the type.

The character/string literals are abstracted out to remove all text. Single/double quotes `{', "}`, as well as any format specifiers such as `{%d, %s, %f}` and character escape-sequences such as `{\n, \\}` are retained. For example, line number 5 in Figure 4.1 would be abstracted out as

`printf("ans=%d", a + 10);` → `printf("%d", INT + LITERAL_I);`

These exceptions to the abstraction rule exist since several errors made by students, especially in the initial days, involve the `printf` and `scanf` functions where it is crucial to retain the format string as is, to be able to identify and fix the error. For instance, for the programs in Figure 4.1, the source and target lines would be abstracted as follows

	Line		**Abstraction**
Source	`scanf("%d",a);`	→	`scanf("%d",INT);`
Target	`scanf("%d",&a);`	→	`scanf("%d", & INT);`

These abstracted lines are called respectively, the *Abstract Source Line* and the *Abstract Target Line*. For sake of simplicity, we will often refer to them as simply the *Source* and the *Target*. Table 4.3 lists several actual source and target pairs from our dataset.

We note that this technique of abstracting out literals and variables is prevalent in literature. However, previous works such as DeepFix [73] replace *all* identifiers/variables, including string-literals, with generic distinct tokens (an un-typed `ID_x`). This precludes their ability to fix semantic-errors (which may be type-specific) and errors involving formatted strings (e.g. `{printf,scanf}` errors) which are very commonly faced by beginners. TRACER performs a much more nuanced abstraction that retains type-specific information which helps it address a much wider range of errors.

For example, consider the buggy program statement `printf("%f",);`. Due to

#	Source-line and Source-abstraction	Target-line and Target-abstraction
1	**d = (x-x1)(x-x1);** INT = (INT - INT)(INT - INT);	**d = (x-x1)*(x-x1);** INT = (INT - INT)*(INT - INT);
2	**p + 'a' = p;** CHAR + 'LITERAL_C' = CHAR;	**p = p + 'a';** CHAR = CHAR + 'LITERAL_C';
3	**printf(,c,);** printf(,INT,);	**printf(c);** printf(INT);
4	**if(a[i]==a[j] && i!(==)j)** if(ARRAY[INT] == ARRAY[INT] && INT !(==) INT)	**if(a[i]==a[j] && !(j==i))** if(ARRAY[INT] == ARRAY[INT] && !(INT == INT))
5	**printf("%d", a + b/6));** printf("%d",INT + INT/LITERAL_I));	**printf("%d", a + (b/6));** printf("%d",INT + (INT/LITERAL_I));
6	**{ while(a > 0){** { while(INT > LITERAL_I){	**while(a > 0){** while(INT > LITERAL_I){

Table 4.3: Examples of source-target pairs for single-line errors. Each row describes an actual error case found in our data. The first column (#) lists the row-index, and second column describes the (erroneous) source line and its abstraction. The third column describes the target line (extracted from repairs attempted by the student) and its abstraction.

the presence of format specifier `%f`, TRACER correctly predicts the insertion of a float type variable after the comma (`,`). While methods, such as DeepFix [73], which ignore the additional information present inside string-literals, predict the insertion of a variable of most commonly observed type in their training dataset (typically integer).

4.3.3 Abstract Source-Target Line Translation

As mentioned before, TRACER performs repair by treating the abstract source line as sequences of tokens and attempting to predict the abstract target sequence using it. The abstracted tokens include reserved keywords such as `{while, double}`, standard library functions such as `{printf, scanf}` and abstract tokens such as `{INT, FUNC, FLOAT}` as discussed in the previous subsection. This sequence translation is performed using Recurrent Neural Networks (RNNs) which are described in detail in Section 4.4. In fact, for any source sequence, the RNN architecture is able to provide multiple

suggestions for the target sequence.

This is beneficial in allowing a graceful degradation of the system. In our experiments, we observed that even if the top-ranked repair presented by the RNN is not the most appropriate, usually the second or the third ranked suggestions do correspond to the most appropriate fix. TRACER uses a finely tuned encoder-decoder model with attention and Long Short-Term Memory (LSTM) mechanisms enabled. Details of parameter settings and tuning for our method are described in Section 4.6.

4.3.4 Target Recommendation and Concretization

TRACER offers recommended repairs in two formats – abstract and concrete. The abstract recommendations are obtained directly from the RNN framework by employing a beam search to obtain 5 target sequences with the highest scores. Note that these abstract sequences still contain abstract tokens like `{INT, FUNC, FLOAT}`.

Now, the abstract recommendations may themselves be offered to students as hints and constitute valuable feedback. It may be argued that if given the actual corrected code, the student has no incentive to explore why did s/he make an error and what was the key to fixing the error. However, if only the abstract form of the repair/solution is provided as a hint, then the student is compelled to map the abstraction onto his/her own program which, in many cases, reveals what was the error in the source program, thus fulfilling several didactic goals.

However, TRACER can go one step further. Given recommended repairs in abstract form, TRACER can generate non-abstract versions of these repairs that can be applied to the original buggy program to produce compilable code. This is done by performing *concretization*, a process which approximately reverses the abstraction step. This has potential application in auto-grading, where marks can be given to incorrect student program attempts that fail to compile, based on some "distance" metric from closest compiling program.

For concretization, TRACER uses standard map-and-store based techniques to put back literals and identifiers of appropriate type into the abstract recommenda-

tion to obtain a program that can be compiled. For this purpose, the Edlib [80] tool is used to compute the sequence alignment of the source-abstraction with recommended-abstraction (by the RNN framework), based on edit distance. This sequence alignment is represented as a *cigar* string, a compact representation consisting of a chain of operators; primarily `Match (=), Insert (I), Delete (D)` and `Mismatch/Replace (X).`

Once TRACER generates the recommended-abstraction, this is aligned with the source-abstraction of student program to generate the *cigar* string. Each successful match (`=`) in recommended-abstraction is replaced with corresponding concrete-code (non-abstract) from source-line. For mis-matches (`X`) and inserts (`I`), TRACER replaces the recommended-abstract type with the closest concrete variable of the corresponding type, encountered earlier in the program. To achieve this, TRACER maintains a symbol table which stores the association of types and variable/literals. When an abstract token is marked as mis-match (`X`) or insert (`I`), the symbol table lookup returns the most recently used/declared concrete token, just before the error was encountered. An example of the concretization step is presented in Table 4.4.

Note that, inserting/replacing the abstract tokens with previously concrete code tokens is an approximate process, and there is no guarantee that the resultant code will compile, even if TRACER generates the exact same abstraction as the students' abstract target line (fix). For eg, a `FUNC` abstract token doesn't capture additional information such as the number of arguments required by the user-defined function. If this `FUNC` is replaced by a concrete function invocation with incorrect number of parameters, then the resultant code will fail to compile. The concretization success rate can be taken closer to 100% in future by maintaining extra meta-information during the abstraction stage, to handle such corner case issues. We observe that our concretization technique is able to convert the abstract target line (abstraction of students' fix) to compilable code for 95% of cases in our dataset.

source-line	xyz	=	4	
source-abstraction	INVALID	=	LITERAL_I	
recommended-abstraction	INT	=	LITERAL_I	;
Cigar (alignment path)	X	=	=	I
concrete-line	i	=	4	;

Table 4.4: An example of the *concretization* process. The incorrect source line is `xyz=4` where `xyz` has not been declared and a semi-colon is missing. TRACER correctly generates the abstract form of the repair `INT = LITERAL_I;`. However, multiple concretizations, all of whom produce valid compilable code, are possible. For example, if the identifiers `i,j` were declared as integer variables, then `i=4;` and `j=4;` are both valid concretizations.

4.3.5 Multiple Error Lines

Although the discussion so far has focused on programs where the fix is required on a single line, TRACER can be adapted to repair programs with errors present across multiple lines as well.

In our dataset, we observed that a large portion of programs having errors on multiple lines can be interpreted as multiple instances of single line errors, i.e. the errors in multiple lines of these programs are not correlated and fixes may be applied to the lines individually. Hence, given a multi-line error program, TRACER does the following

1. **Repair Localization**: TRACER fetches all the source-lines flagged by compiler for error, as well as the lines just above and just below those lines.

2. **Code Abstraction**: TRACER obtains source-abstractions for the above source-lines.

3. **Abstract Code Repair Prediction**: TRACER executes the RNN model on these source-abstractions to get the top-5 recommended-abstractions for each line.

4. **Concretization**: TRACER finally refines each of the recommended-abstractions to generate concrete, repaired code, and checks if the compiler error associated with that particular line disappears on applying the repair. If so, the concrete-code is retained as the fix for the corresponding source-line.

Performance Metric	Description
Prec@k	Percentage of abstract source-target pairs where the top–k abstract recommendations by TRACER contains the abstract target line.
SPrec@k	Percentage of abstract source-target pairs where the top–k abstract recommendations by TRACER contains any abstract target line that has the same abstract source line.

Table 4.5: Performance Metric Description. TRACER uses some of the most stringent performance metrics to ensure that students receive relevant recommendations to correct the actual mistake they are making, and not merely remove compilation errors.

We consider a *multi-line* repair done as outlined above to be successful only if all compiler errors are resolved as a result. While we test and report on results of TRACER on programs with errors in multiple lines, only the single line (singleL) dataset of program pairs is used during training stage.

4.3.6 Performance Measures

We evaluate TRACER's performance with the metrics used in information retrieval and recommendation systems. These are some of the most unforgiving performance measures and we are not aware of their prior use in the program repair domain. The first performance measure we use is Precision at the Top (dubbed Prec@k). For every source-target pair, Prec@k gives a unit reward if the abstract target line is a part of the top–k abstract recommendations returned by TRACER. Note that Prec@1 is an extremely stringent measure that accepts nothing but the exact solution expected by the student (in abstract form).

We also report performance on *Smoothed* Precision at the Top (SPrec@k). This performance measure takes into account the fact that upon abstraction, multiple source lines may map to the same abstract source line and consequently, that abstract source line may now map to multiple abstract target lines. Table 4.5 describes these performance measures.

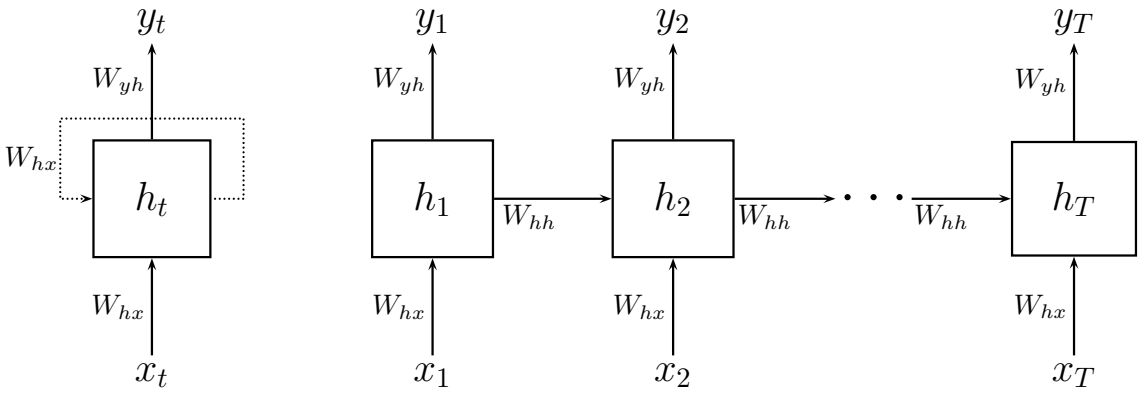


Figure 4.5: The schematic of a recurrent neural network [76].

4.4 Recurrent Neural Networks

Recurrent neural networks (RNNs) have emerged as the preferred learning model in several areas that involve sequence modelling tasks such as natural language processing, speech recognition, image captioning, etc [81]. RNNs differ from classical neural networks by maintaining an internal state and feedback loops into the network. This statefullness of the model and the ability to pass that state onto itself allow RNNs to effectively model sequences of data. Figure 4.5 shows the schematic of an RNN performing sequence translation [76].

Let $X = \{x_1, x_2, \ldots, x_T\}$ and $Y = \{y_1, y_2, \ldots, y_T\}$ denote the input and output sequence of tokens for RNN, respectively, over a shared alphabet set Σ. That is, for any sequence number $t \geq 1$, we have $x_t, y_t \in \Sigma$. In their simplest form, RNNs are trained to learn a language model by predicting the next token in a sequence correctly. An output token y_t is predicted as a function of the input sequence observed so far $X = \{x_1, x_2, \ldots, x_t\}$.

In order to perform this prediction, the RNN maintains an internal state $h_t \in H$. Usually $H \subset \mathbb{R}^k$ is a set of real vectors of dimensionality k. Let the size of the vocabulary be $|\Sigma| = S$. Each token $x_t \in \Sigma$ is represented as a d-dimensional vector (which we also denote as x_t by abusing notation). This representation is usually learned using techniques such as Word-to-Vec [82].

The prediction y_t is defined as a function of the hidden state h_t, and the hidden

state h_t itself evolves as a function of the previous hidden state h_{t-1} and the current input x_t. This is formally defined as follows:

$$h_t = f_H(W_{hx}x_t + W_{hh}h_{t-1})$$
$$y_t = \text{SELECT}(f_O(W_{yh}^\top h_t)),$$

where f_O and f_H encode *activation* functions that map reals to reals (when applied to vectors, the activation functions act in a coordinate-wise manner), and $W_{yh} \in \mathbb{R}^{S\times k}$, $W_{hh} \in \mathbb{R}^{k\times k}$, $W_{hx} \in \mathbb{R}^{k\times d}$ are matrices. The SELECT operation simply chooses the coordinate of a vector with the largest value. Since $f_O(W_{yh}^\top h_t)$ is a $|\Sigma| = S$-dimensional vector, the SELECT operation returns a token in Σ. For a more detailed introduction to the RNN framework, we refer the reader to blogs by Britz [83] and Karpathy [81].

Unequal Sequence Lengths

A specific hurdle encountered while using RNNs for program repair is the input and output sequences being of unequal lengths. This can arise due to the fix in incorrect program requiring insertions and/or deletions of tokens (for example, see Figures 4.1 and 4.2). The Encoder-Decoder model [84, 85, 86] is a generic solution to this problem that employs two components, an encoder and a decoder. The encoder operates by using the input sequence to generate a sequence of hidden states.

$$h_t = f_H(W_{hx}x_t + W_{hh}h_{t-1})$$

At the end of the sequence, an intermediate representation, known as the *context vector* is computed as a function of the hidden state sequence $\{h_t\}$.

$$c = q(h_1, \ldots, h_T)$$

The decoder generates a fresh sequence of (still hidden) states $\{s_t\}$ using the context vector and previous hidden states. These states are then used to generate an output

sequence $\{y_t\}$

$$p(y_t|y_1, \ldots, y_{t-1}, c) = g_O(y_{t-1}, s_t, c)$$

until a special delimiter token $\langle \text{eos} \rangle$ is generated, indicating a termination (or end) of the output sequence.

Handling Long Sequences

Another significant hurdle to training RNNs on long sequences, such as those we encounter in our program repair application, is the problem of vanishing or exploding gradients [87]. An elegant fix to this problem is the Long Short Term Memory (LSTM) model [88] and its variants that overcome the problem by replacing hidden states with a gating mechanism, which allows gradients to flow freely in the backpropagation-through-time algorithm.

The problem emerges in a different form when dealing with long output sequences, where a single context vector becomes insufficient to predict an output sequence of arbitrary length. *Attention mechanisms* [89] overcome this problem by identifying for each output token y_t, a specific part of input sequence that is most relevant for predicting y_t. This is done by employing a separate context vector c_t for predicting the output token y_t instead of a uniform one. These context vectors are generated by a separate neural network that is trained jointly with the RNN. Typically, a weighted combination of the hidden states of the encoder are used to generate these context vectors. For more details, we refer the reader to some excellent tutorials on LSTMs [90] and attention mechanisms [91].

Usefulness of the Repair Localization Step

Even with attention mechanisms, we found RNNs with Encoder-Decoder models to struggle when trained on entire programs. This is because the decoder in such situations is heavily stressed to identify the relevant subset of the input sequence for every output token, thereby decreasing performance while increasing training and prediction time. However, the modular approach adopted by TRACER, that first

identifies a very small (often 3) set of sentences to focus on, greatly improves the ability of the RNN framework to analyze and suggest relevant local fixes.

4.5 Experimental Setup

All experiments were performed on a system with an Intel® Core™ i7 CPU 930 @ 2.80GHz × 8 CPU with 8GB RAM, and an NVIDIA GeForce® GTX 760 GPU with 2GB GPU Memory.

Dataset

Our dataset contained a total of $23,275$ source–target single line pairs (refer Table 4.1), which were further divided into training (70%), validation (10%) and test (20%) sets. Validation and test examples were randomly picked from the dataset. The most frequent 150 and 140 tokens were chosen for the source and target vocabulary, respectively for training the RNN i.e. $|\Sigma| = 150$ for the input vocabulary and 140 for the output vocabulary. Tokens not in vocabulary were replaced by a special token `UNK`. The maximum length of source and target sequences was set to 80 and 82, respectively.

Training

We trained a supervised Encoder-Decoder model with attention mechanism to translate the source (abstract) line to target (abstract) line. This model was built using an open source Torch implementation of a standard sequence-to-sequence model, where the encoder-decoder are LSTMs[2].

The models were trained by minimizing the class negative likelihood loss. We conducted an extensive grid search to set various hyper-parameters: number of hidden layers for both Encoder and Decoder in $\{1, 2, 3, 4\}$, size of the LSTM hidden state in $\{50, 100, 200, 250, 300, 350\}$, and word embedding size in $\{50, 100, 150, 200, 250, 300\}$. Word embeddings were learnt jointly with network parameters. We tried both

[2] `https://github.com/harvardnlp/seq2seq-attn`

unidirectional and bidirectional RNNs with reverse source side setting. The best model based on validation perplexity was found to be a bidirectional Encoder with 2 hidden layers, a hidden state size of $k = 300$, word embedding size $d = 100$. Reversing the source sequence gave poor results.

Apart from this, we used a mini-batch size of 32 for training, standard stochastic gradient descent with an initial learning rate 1 and learning rate decay of 0.5 after the 9^{th} epoch for a total of 35 epochs. We initialized the hidden state of the Decoder at time 0 with the last hidden state of the Encoder, initialized other parameters randomly in range $[-0.1, 0.1]$, clipped gradients at magnitude 5, and used a dropout probability of 0.3 between LSTM layers.

Prediction

We performed a beam search with beam size 50 to select the best 20 target sequences. Out of these 20 the top 5 unique predictions were used for measuring performance according to the measures described in Section 4.3.6. We also performed these experiments in cross validation mode with 5 random train-validation-test splits of the data, and the result was found to be very similar to the one reported below.

4.6 Evaluation

In this section, we report on the prediction accuracy (how close are the repairs to students' fix), overall accuracy (how many programs from test-set could be repaired to compile successfully), and the time taken (for generating these repairs) by TRACER.

Prediction Accuracy

Table 4.6 reports the prediction accuracies of TRACER on a test set of $4,578$ programs, obtained from the ~20% held out set of our overall collection of $23,275$ source-target single line pairs (refer Section 4.5). TRACER is able to correctly predict the exact abstract target repair as its top suggestion (Prec@1) in 59.60% of

k	Prec@k	SPrec@k
1	59.60	68.61
2	64.90	72.73
3	66.61	73.85
4	67.85	74.73
5	68.32	75.15

Table 4.6: Prediction accuracy of TRACER on 4,578 (~20%) held out test set programs. TRACER excels on the challenging Prec@k and SPrec@k metrics (refer Section 4.3.6).

the instances. If we include the top 3 recommendations of TRACER, then this figure increases to 66.61% for Prec@3 and further to 73.85% for SPrec@3.

Figure 4.3 shows that TRACER consistently achieves high levels of Prec@1 accuracy in predicting the exact student repair, across various kinds of compilation errors. It has lower success rate on just a few error types among the top-15 frequent ones, such as *E9: too few args to function call* and *E13: too many args to function call*. This is because presently, TRACER cannot predict the correct number of parameters required by function calls in the *target*. Extending TRACER to maintain additional meta-information can fix this limitation (refer Section 4.3.4).

Table 4.7 lists a few interesting examples from our dataset on which TRACER was able to provide correct and relevant fixes as its top-recommendation. For the first two examples, TRACER makes an accurate prediction, correctly suggesting the insertion of an arithmetic operator in example #1, and swapping the *Left* and *Right* operands of assignment expression in example #2. For rest of the examples, its top recommendation does not match the students' fix. However, notice that TRACER's recommendation is still an appropriate fix that produced compilable code in most cases.

For instance, consider the example in row #3. Although the target code (students' fix) will compile successfully with warnings, TRACER's top suggestion is the most appropriate fix for the corresponding error. The target code results in a compiler warning: *incompatible integer to pointer conversion passing 'int' to parameter of type 'const char *'*. While TRACER's suggestion does not generate any compiler warnings and captures the students' intent better. However, our evaluation metric

#	Source-line and Source-abstraction	Target-line and Target-abstraction	TRACER's Top Prediction	EM	CS
1	**d = (x-x1) (x-x1);** INT = (INT - INT)(INT - INT);	**d = (x-x1)*(x-x1);** INT = (INT - INT)*(INT - INT);	**d = (x-x1)*(x-x1);** INT = (INT - INT)*(INT - INT);	✓	✓
2	**p + 'a' = p;** CHAR + 'LITERAL_C' = CHAR;	**p = p + 'a';** CHAR = CHAR + 'LITERAL_C';	**p = p + 'a';** CHAR = CHAR + 'LITERAL_C';	✓	✓
3	**printf(,c,);** printf(,INT,);	**printf(c);** printf(INT);	**printf("%d",c);** printf("%d",INT);	✗	✓
4	**if(a[i]==a[j] && i!(==)j)** if(ARRAY[INT] == ARRAY[INT] && INT !(==) INT)	**if(a[i]==a[j] && !(j==i))** if(ARRAY[INT] == ARRAY[INT] && !(INT == INT))	**if(a[i]==a[j] && i!=(j))** if(ARRAY[INT] == ARRAY[INT] && INT != (INT))	✗	✓
5	**printf("%d", a + b/6));** printf("%d",INT + INT/LITERAL_I));	**printf("%d", a + (b/6));** printf("%d",INT + (INT/LITERAL_I));	**printf("%d", a + b/6);** printf("%d",INT + INT/LITERAL_I);	✗	✓
6	**{ while(a > 0){** { while(INT > LITERAL_I){	**while(a > 0){** while(INT > LITERAL_I){	**{ while(a > 0){** { while(INT > LITERAL_I){	✗	✗

Table 4.7: Examples of source-target pairs for single-line errors. Each row describes an actual error case found in our data. The first column (#) lists the row-index, and second column describes the (erroneous) source line and its abstraction. The third column describes the target line (extracted from repairs attempted by the student) and its abstraction. The fourth column describes the abstract and concrete versions of the top-ranked repair suggested by TRACER. Finally, the last 2 columns indicate whether TRACER's top recommendation Exactly Matches the target-abstraction (EM) and whether the resultant concrete-code Compiled Successfully (CS). Note that in rows #4 and #5, although TRACER's top recommendation is semantically equivalent to the target, the (EM) metric still counts it as failure for not exactly matching student's repair.

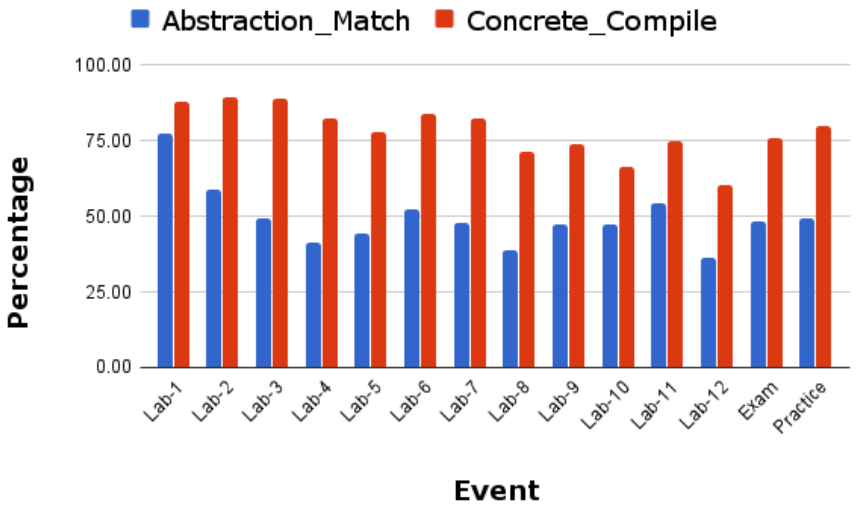


Figure 4.6: Accuracy of TRACER across labs, on single-line test dataset. TRACER is robust in handling compilation errors occurring across all labs. Even on erroneous programs dealing with advanced concepts such as pointers and recursion, TRACER is able to consistently achieve high repair rate.

still considers it a failure due to mismatch with the student's repair.

In examples #4 and #5, TRACER predicts an abstract fix that is semantically equivalent to the desired fix but we nevertheless penalize it due to our stringent evaluation criteria of an exact match with the student repair. TRACER arguably produces a simpler fix than what the student devised for examples #4 and #5, since it learns from frequent error patterns found in multiple student submissions. In future, better scoring mechanisms can be developed to evaluate the predictions. Nonetheless, even with such stringent metrics, TRACER is able to achieve high accuracy of 59.6 % for prediction@1 (Prec@1).

For the example #6 in Table 4.7, TRACER produces the *source* verbatim since it is unable to figure out the desired number of opening braces from local context. In future, we plan to incorporate global context as well to predict such fixes.

End-to-End Repair

Apart from reporting the success rate of TRACER using our pedagogically relevant metrics of Prec@k and SPrec@k, we also report on TRACER's overall End-to-End repair accuracy. This overall accuracy is a conventional metric commonly used in

Dataset	#Programs	#Compiler-Errors	TRACER Repair %
Single–Test	4,578	4,853	79.27
Multiple	17,451	24,255	43.67
DeepFix	6,971	16,743	43.97

Table 4.8: Overall repair accuracy of TRACER on three different datasets. Single–Test is the held out test set of program pairs requiring edit on a single line. Multiple is the set of programs requiring repairs on multiple lines. DeepFix is the test dataset used by tool DeepFix [73]. On all three datasets, TRACER gives compilable code on a large fraction of programs.

literature, indicating the percentage of programs that the tool repaired to compile successfully. In the context of TRACER, it is referred to as End-to-End since it is the overall accuracy achieved after performing all 4 stages of repair: Localization → Abstraction → Prediction → Concretization.

Table 4.8 reports the compilation success rates of the multiple-error approach, outlined in Section 4.3.5, on our single-line (errors requiring fix at a single line), multiple-line (errors requiring fix at multiple different lines), and on the dataset obtained from DeepFix[3] [73]. TRACER is able to fix 79.27% of the single-line test set (referred to as *Single–Test*), while being relevant at the same time (as demonstrated by predication accuracy in Table 4.6). TRACER is also able to successfully repair about 44% of programs with multiple-line errors obtained from our course (denoted as *Multiple* in Table 4.8) and on the collection of programs used in DeepFix [73]. Note that DeepFix, the previous state-of-the-art, was able to achieve only 27% compilation success rate on the same set of programs.

In Figure 4.6, we demonstrate the overall end-to-end repair accuracy of TRACER on our dataset of single-line errors, across weekly lab assignments (refer to Table 4.1 for individual lab topic). The *Abstraction–Match* legend denotes how many times our recommended-abstraction *exactly* matched with the target-abstraction (abstraction of fix performed by same student). *Concrete–Compile* denotes what % of the erroneous programs was TRACER able to repair (End-to-End) and compile.

[3] `https://www.cse.iitk.ac.in/users/karkare/prutor/prutor-deepfix-09-12-2017.db.gz`

Time Taken

During our experiments, invoking the deep-network takes the maximum amount of time in this entire setup, while the abstraction/concretization and compilation steps take an order of milli-seconds time on average. Hence to ameliorate this, we cache the frequently looked-up abstraction translations during testing phase. This reduces the time taken for the overall repair process to just 1.66 seconds on average per erroneous program. Although the training phase requires order of few hours time to train the deep-network, this a one time process (per programming language).

4.7 Discussion

To the best of our knowledge, TRACER is the first approach to perform targeted repair where we learn and predict the exact fix desired for a students' programming error, instead of predicting it as a side-effect of not matching with a generic correct grammar. Prior works [73, 11, 77] first create a (deep neural network) model by observing a large number of generic correct programs. Then given a new erroneous program, they attempt to generate a repair which requires minimal changes to the source program, utilizing the generic correct program model. Also, these prior work do not use strong criterion (such as `Prec@k` used by TRACER) to evaluate the quality of repairs generated. For example, simply deleting the erroneous line or replacing it with a trivially correct statement will make the error go away in a large fraction of buggy programs, but this may not be an acceptable fix in a pedagogy setting.

The approach used by DeepFix [73] to repair common C programming errors comes closest to TRACER [76]. DeepFix learns a sequence-to-sequence neural network on the entire concrete program and features a repair localization module which, with the help of an Oracle, can attempt to resolve multi-line errors by making multiple passes. TRACER on the other hand focuses on multiple instances of single-line errors and achieves a high compilation accuracy, while being relevant to the

students' own target fix. In fact on the very dataset used by DeepFix, with errors occurring on multiple different lines, TRACER offers significantly higher compilation success rates than DeepFix.

HelpMeOut [17] is another related tool for helping students fix compilation errors. Instead of providing the actual fix desired by student, HelpMeOut suggests relevant examples of errors and repairs. This is achieved by searching a database of similar errors previously encountered by other students. It provides both the original erroneous line of the previous student, as well as the new repaired line that resulted in successful compilation. However, unlike TRACER, this repair is picked from a database and is not tuned to the erroneous source program. TRACER instead learns the repair from the submissions of the other students, and suggests the exact abstract/concrete fix tuned for the erroneous source program.

A recent user study [78] demonstrates that offering semantic/logical program repairs to expert human graders (teaching assistants) can help decrease the grading time. A similar positive impact on (automated) grading of erroneous programs with compilation errors is reported by GradeIT [2], which uses simple rewrite rules to repair compilation errors. The repairs generated by TRACER, along with systems such as *GradeIT*, can further aid such automated grading tasks.

It will be worth to investigate in future if re-write rules can be learnt automatically from the dataset, so that they are automatically applied on future incorrect programs to repair them.

Limitations

Our experiments confirm that TRACER performs very well on errors where repair can typically be obtained by looking at a local context; for example, on restructuring an expression. However, TRACER does not yet take into account the global context; for example, the number of arguments of a function defined away from its call, opening/closing an unmatched brace, undeclared variables etc. A separate new technique can then be developed to run in tandem with TRACER, to handle some

of these frequently occurring global context errors. Future versions of TRACER are planned to have these features to widen its scope.

Also, our evaluation metrics currently do not discount minor syntactic variations, e.g. `{a != b}` vs `{!(a == b)}`, while computing the mismatch between tool repair and the student repair. We believe a more moderate performance measure will give us a more realistic picture of the performance of TRACER and other such error-correction tools. Finally, it is worthwhile to combine TRACER with semantic repair tools such as Prophet [92] to further refine the repairs.

4.8 Summary

In this chapter, we presented TRACER, a tool to generate targeted compilation error repairs aimed at novice programmers. TRACER invokes a novel combination of tools from programming language theory and deep learning to offer accurate recommendations for fixes. On 4,500+ test programs with single-line errors, its top three recommendations include the students' desired repair 74% of the time. Even on 17,000+ programs with errors on multiple lines, TRACER offers a compilation success rate of more than 40%, requiring only several milliseconds to make its recommendations.

Although this chapter focused on the C programming language, TRACER can be easily ported to other programming languages such as C++, Java, Python; given sufficient training data. TRACER's performance on stringent correctness criterion strengthens our claim that repairs suggested by it are close to the fixes performed by the students themselves. As a result, these targeted repairs are of more didactic value than the compiler produced error messages, and repairs returned by previous tools that merely offer a certain compilation success rate.

High accuracy and real-time feedback generation make TRACER suitable for live deployment as a virtual teaching assistant for programming courses offered at a massive scale. In Chapter 5, we present another compilation error feedback tool called TEGCER, which aids students in fixing compilation errors by providing

generative feedback in the form of examples. In Chapter 6, we deploy both TRACER (discriminative feedback tool), and TEGCER (generative feedback tool) in a live offering of the C programming course, and compare their efficacy in helping novice programmers.

Chapter 5

Compilation Error Example Generation

In the previous Chapter 4, we talked about TRACER; a compilation error repair tool whose code repairs are not only accurate, but also relevant to students' own fix. The automated repair tools that exist in literature typically generate the correct *concrete* code as their final output [69]. Going a step further, TRACER can offer the repair solution either as *concrete* code (the correct code which compiles) or *abstract* code (where the variables and literals are replaced with their type information instead). While revealing either of these correct solutions could be invaluable as a feedback to students in certain situations, such as in the case of automated grading, it is unclear if providing it during the programming exercise could aid in effective learning.

In this chapter, we propose a new feedback tool called TEGCER (**T**argeted **E**xample **G**eneration for **C**ompilation **ER**rors) [93]. The goal of TEGCER is to provide an alternative feedback to students who encounter compilation errors in their program. This feedback is in the form of examples of fixes performed by other students, albeit on a different program, when faced with similar error previously. Such an approach is a departure from most of the recent work in literature, which instead tend to focus on producing the desired repair/solution [73, 11, 17], help improve the descriptive error message [94], or vary the error message's structure and

```
1  #include<stdio.h>
2
3  int main(){
4      int c, a=3, b=2, i;
5      c = (a-b) (a+b);
6
7      for(i=0, i<c, i++)
8          printf("%d" i);
9
10     return 0;
11 }
```

		Compiler Message
5	E_{15}	Called object type 'int' is not a function
7	E_7	expected ';' in 'for' statement specifier
8	E_1	expected ')'

(a) Buggy program and error message

```
1  #include<stdio.h>
2
3  int main(){
4      int c, a=3, b=2, i;
5      c = (a-b)*(a+b);
6
7      for(i=0; i<c; i++)
8          printf("%d", i);
9
10     return 0;
11 }
```

(b) TRACER's repair

Figure 5.1: Sample buggy program and its fix generated by TRACER.

Line#	Eg#	Buggy Examples	Repaired Examples
5	1	`amount = P (1+T*R/100);`	`amount = P*(1+T*R/100);`
	2	`ke = 1/2*m (v1*v1-v2*v2);`	`ke = 1/2*m*(v1*v1-v2*v2);`
7	1	`for(i=0, i<n; i++){`	`for(i=0; i<n; i++){`
	2	`for(i=0; i++){`	`for(i=0;; i++){`
8	1	`printf("%s" str);`	`printf("%s", str);`
	2	`printf("%d" c);`	`printf("%d", c);`

Figure 5.2: Example feedback by TEGCER, on Figure 5.1a buggy program.

placement [75].

Figure 5.1a reports an erroneous code attempt by student, and Figure 5.1b presents its fix generated by TRACER repair tool. The buggy program in Figure 5.1a encounters the following errors. In line #5 and #8, the student forgets to use an asterisk operator '*' and comma separator ',', respectively. While in line #7, the student mistakenly uses comma separator, instead of the semi-colon separator ';' demanded by for-statement syntax. TRACER's output correctly predicts these fixes, and generates a compilable code after applying the repair.

However, if the student is provided with concrete repair output of TRACER

as feedback, then it is plausible for the student to simply copy the answer without understanding the error causation or its fix. Providing the abstract code repair of TRACER could help mitigate some of these concerns. But instead, if students were provided with multiple examples of *source* code (code with similar error) and *target* code (its repaired code) generated by TEGCER as feedback, then the students can be expected to learn the general cause of that particular compilation error and its potential fixes. After which they can proceed to apply the acquired knowledge on his/her own code and repair it.

Figure 5.2 lists top-2 buggy-repaired example pairs suggested by TEGCER for the same buggy program in Figure 5.1a, for each erroneous line. These examples are from different programs, albeit having similar error and desired repair as the original buggy program in Figure 5.1a.

There have been extensive studies in literature on how providing relevant worked examples can help students learn effectively in a pedagogical setting, for example by Renkl et al. [95]. This idea of using code examples from other students as compilation error feedback is not new. The closest related work to ours is HelpMeOut [17], where students can query a central repository to fetch example erroneous-repaired code pairs of other students, that suffer from compilation errors similar to their own code. This repository of errors is created and maintained manually by students themselves. In contrast, TEGCER passively learns from mistakes made by students of previous offerings, and provides feedback for students in new future offerings of course. Although TEGCER does not offer an explanation for compilation error, such as the human generated one in HelpMeOut, it suggests the closest relevant erroneous-repaired code pairs automatically, without requiring any manual contribution. Also, the distance metric used by both differ significantly. HelpMeOut ranks the set of examples based on the error similarity and user provided ratings. While TEGCER ranks the examples based on the similarity of mistake made and ideal fix desired by the student.

5.1 Chapter Outline

This chapter is organized as follows: Section 5.2 starts by describing our dataset of $15,000+$ buggy programs, obtained from novice student programmers, and highlights the different kinds of compilation errors encountered in them. Then in Section 5.3, we provide further insight into our aligned dataset of buggy-repair pairs, so as to identify the abstract line and its error-repair class, for each buggy program.

Section 5.4 presents an overview of the deep learning techniques used by TEGCER, which learns and predicts labelled error-repair class given an abstract error line. The accuracy of our classifier is reported here as well. Section 5.5 then describes the working of TEGCER in its entirety, and the various modules that it comprises of. We also demonstrate TEGCER's relevant example feedback on buggy programs of actual students, along with explanations. Section 5.6 then concludes the discussion and outlines some future directions for this effort.

Technique Proposal

The design of TEGCER is largely inspired by the methodology of TRACER [76], where the neural network component is changed to predict the *class* of repairs instead of the exact repair. This *class* of repair denotes the set of abstract program tokens that need to be added/removed from the erroneous code to correct it. Since TEGCER has to only predict the class of fix required, as opposed to generating the complete fix at the exact position by TRACER, its precision-recall scores are considerably higher.

To achieve this end, TEGCER adopts a modular, four-phased methodology for generating code examples which involves

1. **Repair Localization**: TEGCER first locates the line(s) where repair must be performed.

2. **Code Abstraction**: TEGCER then abstracts the erroneous code lines, by replacing program specific tokens with their generic types.

3. **Error-Repair Class Prediction**: TEGCER classifies the abstract error code lines, based on the fixes required in terms of insertion/deletion of abstract program tokens. The classifier used is a dense neural network, trained to identify the intent of students on being given error code lines.

4. **Example Suggestion**: TEGCER finally suggests the top frequent erroneous programs and their repaired code, as observed in its training dataset, requiring the same class of fixes.

Contributions

Our main contribution is the theory and implementation of TEGCER, a tool to automatically suggest relevant programming examples as feedback to students, without requiring any human assistance. To the best of our knowledge, TEGCER is the first automated tool of this kind. In the extremely challenging task of predicting a single correct label for buggy program from 212 unique error-repair classes, TEGCER achieves very high accuracy of 87% on its first prediction (Pred@1).

From a dataset of 15,000+ buggy programs, we identified 6,700+ unique compilation error-groups (***EG***s) made by novice students, and 200+ different class of error-repairs (***C***s) performed by them to fix the errors. This dataset will be released in public domain to help further research.

5.2 Compilation Errors

Compilation errors can occur due to a variety of reasons, ranging from syntax errors which occur when the source-code does not follow the pre-defined rules/grammar of language, to linker errors where a compiler is unable to resolve references to other source-files/libraries.

Error ID	Message	$\square_1$	$\square_2$
E_1	Expected $\square_1$	)	
E_2	Expected $\square_1$ after expression	;	
E_3	Use of undeclared identifier $\square_1$	sum	
E_4	Expected expression		
E_5	Expected identifier or $\square_1$	(	
E_6	Extraneous closing brace ($\square_1$)	}	
E_7	Expected $\square_1$ in $\square_2$ statement	;	for
E_8	Expected $\square_1$ at end of declaration	;	
E_9	Invalid operands to binary expression ($\square_1$ and $\square_2$)	int *	int
E_{10}	Expression is not assignable		

Table 5.1: The top–10 frequent individual compilation errors (Es), listed in the decreasing order of frequency, as observed in our course offering. Program specific tokens have been replaced with $\square$, and the last columns shows a sample concrete value for the same.

5.2.1 Unique Errors

During our course offering of Introductory to C Programming at Indian Institute of Technology Kanpur (IIT–K) university, where 400+ students attempt 40+ different programming assignments, students' code attempts trigger 250+ unique kinds of compilation error messages. Table 5.1 lists the top-10 frequent error messages returned by Clang compiler [72], a popular compiler for C programming language. The messages are generalized by replacing any program specific tokens (demarked by Clang within single/double quotes) with $\square$. One example substitution for each $\square$ is provided in the table as well.

Note that the arrangement of top-frequent error-IDs in Table 5.1 is slightly different compared to the one listed earlier in Table 4.2. Since TRACER does not utilize the compiler reported error-messages for predicting the repair, the error-messages were grouped together in previous chapter based on common keywords (refer Chapter 4.2.3) for ease of presentation to the reader. We found that passing the compiler reported error messages as input to the neural network of TEGCER (refer Section 5.4.1) helps increase its prediction accuracy. Table 5.1 hence retains the individual compiler reported errors instead of clustering them based on similarity. We plan to incorporate this improvement in the next version of TRACER as well.

```
1 #include<stdio.h>
2
3 int main(){
4  int x,y,;
5
6
7  return 0;
8 }
```

		Compiler Message
4	E_5	expected identifier or "("

(a) EG_{10}

```
1 #include<stdio.h>
2
3 int main(){
4  printf("Hello");
5  }
6
7  return 0;
8 }
```

		Compiler Message
7	E_5	expected identifier or "("
8	E_6	extra closing brace "}"

(b) EG_7

Figure 5.3: Example programs for (a) error-group EG_{10} containing error E_5 (b) error-group EG_7 containing errors $E_5 \wedge E_6$

5.2.2 Error Groups

A buggy program can contain one or more compilation errors, a collection of which is called compilation Error Group (abbreviated as **EG**). We define an error-group primarily for ease of presentation to the reader, where a single Error-Group ($\boldsymbol{EG}$) ID can be used interchangeably to represent the set of Error ($\boldsymbol{E}$) IDs present in a buggy program.

The bug in the program (and hence its fix) is characterized by the combination of errors occurring together. For example, consider the two erroneous code attempts in Fig 5.3 which fail to compile. Fig 5.3a program has an additional comma "," on line-4 before the semi-colon, and the compiler reports error $\boldsymbol{E_5}$ on line-4. While Fig 5.3b program has an additional closing brace "}" on line-5, and the compiler reports 2 different errors – $\boldsymbol{E_5}$ and $\boldsymbol{E_6}$ on line-7 and line-8 respectively. Even though the individual error of Fig 5.3a program ($\boldsymbol{E_5}$) is a subset of errors ($\boldsymbol{E_5 \wedge E_6}$) found in Fig 5.3b program, the bug and hence its fix is unrelated.

During our course offering, students made on average 35,000+ code attempts with compilation failure, encountering 250+ unique individual compilation errors ($\boldsymbol{E}$s) and 6700+ unique error-groups ($\boldsymbol{EG}$s). Table 5.2 lists the top frequently observed EGs. $\boldsymbol{EG_1}$ (resp $\boldsymbol{EG_{100}}$) is the top-frequent (resp 100^{th} frequent) compilation error-group,

Error Group ID	Error IDs	#Programs
EG_1	E_3	5,965
EG_2	E_4	4,156
EG_3	E_1	3,488
EG_4	E_2	3,018
EG_5	E_{35}	1,135
EG_6	$E_3 \wedge E_4$	949
EG_7	$E_5 \wedge E_6$	777
EG_8	E_9	763
EG_9	$E_1 \wedge E_4$	762
EG_{10}	E_5	603
...		
EG_{19}	E_7	339
...		
EG_{100}	$E_3 \wedge E_4$	35
ALL	–	35,000+

Table 5.2: The top frequent compilation error-groups (EGs), observed in our course offering. EG_i represents the i^{th} frequent error-group.

encountered by 5,965 (resp 35) programs compiled by students. We demonstrate later, in Chapter 6, that feedback tool performance (and hence student performance) is affected by the set of errors (EGs) that a program triggers, rather than individual errors (Es). For example, we show that feedback tools have higher accuracy on buggy programs with error E_5 (EG_{10}), than those programs where both E_5 and E_6 are encountered together (EG_7).

As observed in Figure 5.4, the frequency of the error groups (*EG*s) encountered by novice programmers in our course offering follows a heavy tailed distribution. In other words, there is a sharp decrease in the number of programs affected by an error group (*EG*s), as we proceed through a list of *EG*s sorted by their frequency count. Similar observation has been made by others as well [71, 17]. In our entire course offering where students made 35,000+ failure code attempts, the top-8 frequent *EG*s accounted for more than 50% of all the student programs failing to compile. Out of the 6700+ unique *EG*s, only the top-240 *EG*s repeat in 10 or more different programs.

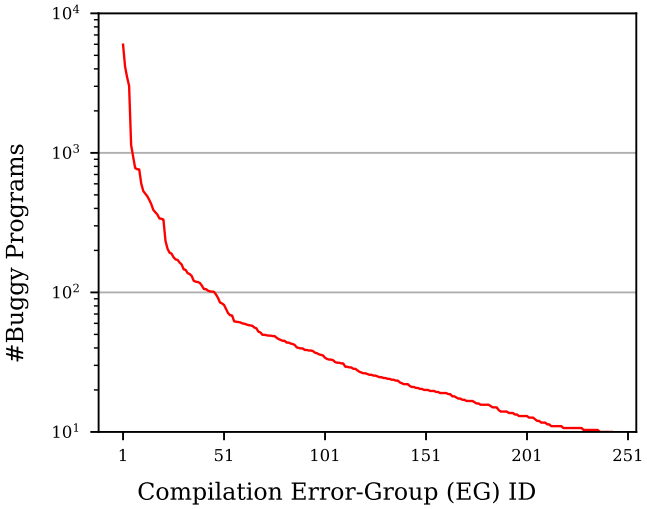


Figure 5.4: Frequency distribution of compilation error-groups (***EG***s). The plot depicts the number of buggy program (y-axis, in $\log_{10}$ scale) attempts by students in our course offering, that failed to compile due to presence of ***EG*** ID (x-axis).

5.3 Error Repair Classes

The previous section gave an overview about the different kinds of compilation errors encountered by novice students in our entire course offering. In this section, we present the different kinds of repairs performed by the same students to fix their program. These repairs are in the form of insertion/deletion [1] of program tokens until all compilation errors are resolved.

Single-Line Dataset

In order to identify the different kinds of repairs students make, we train TEGCER on the same single-line edit dataset as TRACER, described earlier in Section 4.2.3. This single-line edit dataset, obtained from the 2015–2016 fall semester offering of CS1 at IIT–K university, consists of buggy-repaired program pairs that require editing a single line to fix the compilation error. Programs requiring multiple-line edits can often be treated as multiple instances of single-line edits, as mentioned earlier in Section 4.3.5.

This single-line edit dataset is obtained from the collection of failure code attempts

[1] replacement is treated as insertion+deletion of program tokens

by 400+ students, who attempted 12 weekly guided labs, 2 different exams, and few practice problems of their own, as shown earlier in Table 4.1. A total of $23,275$ buggy programs were found from the entire semester such that (i) the student program failed to compile and (ii) the same student edited a single-line in the buggy program to repair it. The single-line of buggy program that was changed is referred to as *source-line* and the corresponding line in repaired program is referred to as *target-line*. The *source-line* and *target-line* program tokens are then replaced with generic type tokens, using the abstraction technique described in Chapter 4.3.2, to obtain *source-abstraction* and *target-abstraction* respectively. For the sake of simplicity, *source-abstraction* and *target-abstraction* are henceforth referred to as *source* and *target* respectively.

Repair Tokens

Given the *source* and *target* abstract line, we can define the *repair tokens* (***R***s) in terms of the set of insertion and deletion of tokens required in *source*, to obtain the desired *target*.

	Source		**Target**
Line	`printf("Ans=%d",a)`	$\longrightarrow$	`printf("Ans=%d",a);`
Abstraction	`printf("%d",INT)`	$\longrightarrow$	`printf("%d",INT);`
Repair		`+;`	

Table 5.3: Example source-target pair, requiring insertion of "`;`" as repair. The compiler reports error E_2 (expected $\square_1$ after expression) for the source-line.

For example, consider the source and target abstraction shown in Table 5.3. The difference between the target and source tokens is a single "`;`" (semi-colon) token. That is, a "`;`" was added by student in the source/buggy program to obtain the target/repaired program. Hence, its repair tokens are `{+;}`, with the "`+`" indicating an insertion of token in source.

Consider another example shown in Table 5.4. Here, the student has made a compilation error by using an undeclared variable "`xyz`", and rectifies the error by replacing it with an integer variable "`a`" instead. In other words, an `INVALID` token

	Source		Target
Line	`b=xyz;`	→	`b=a;`
Abstraction	`INT=INVALID;`	→	`INT=INT;`
Repair	`+INT -INVALID`		

Table 5.4: Example source-target pair, requiring replacement of "`INVALID`" token with "`INT`" token as repair. The compiler reports error E_3 (use of undeclared identifier $\square_1$) for the source-line.

was replaced in source with an `INT` token, to get the desired target. Hence, the repair tokens are `{+INT -INVALID}`, with the "`-`" indicating deletion of token in source.

We use difflib [2], a standard python programming library package, to obtain the repair tokens in terms of difference between source and target pairs. Note that, repeated insertion (or deletion) of a unique token is accounted for only once. For example, even if a program requires two separate semi-colons to be inserted, its set of repair tokens consists of a single semi-colon `{+;}`.

Identifying Error Repair Class

Given a buggy source program with compilation errors (***E***s), and requiring a set of repair tokens (***R***s) to fix it, its error-repair *class* (***C***) is simply the merged set of errors and repairs {***E***s ***R***s}.

```
1 #include<stdio.h>
2
3 int main(){
4     int i=0;
5
6     for(i=0; i<10, i++){
7       printf("%d",i);
8     }
9 }
```

		Compiler Message
6	E_7	expected ';' in 'for' statement

(a) Buggy program with error E_7

```
1 #include<stdio.h>
2
3 int main(){
4     int i=0;
5
6     for(i=0; i<10; i++){
7       printf("%d",i);
8     }
9 }
```

Class	
C_{10}	{E_7 +; -, }

(b) Correct program, with fix ∈ C_{10}

Figure 5.5: Example buggy program and its fix belonging to error-repair class C_{10}.

Figure 5.5 lists an example buggy program and its class. In this example, the

[2] `https://docs.python.org/2.7/library/difflib.html#difflib.SequenceMatcher`

Class ID	Class (C)		#Programs
C_1	E_2	`+;`	3,888
C_2	E_3	`+INT -INVALID`	731
C_3	E_1	`+}`	519
C_4	E_1	`+)`	475
C_5	$E_5 \wedge E_6$	`-}`	418
C_6	E_8	`+;`	404
C_7	E_1	`+,`	389
C_8	E_{10}	`+== -=`	309
C_9	E_1	`+;`	305
C_{10}	E_7	`+; -,`	299
	...		
C_{211}	E_{47}	`-, -INT`	10
C_{212}	$E_1 \wedge E_5 \wedge E_{47}$	`-;`	10
ALL		–	15,579

Table 5.5: Top frequent Error-Repair Classes (Cs) from single-line repair dataset. The class is a combination of error-ID (Es) and repair tokens (Rs). A total of 212 repair classes having at least 10 buggy programs are considered.

source program has an error in line-6, where a student mistakenly uses "`,`" in place of "`;`" inside the for-statement specifier, triggering a compilation error E_7 (the 7^{th} most frequent error). The program in Figure 5.5 is then labelled with class {E_7 `+; -,` }. Since this class is the 10^{th} most frequent one encountered in our dataset of single-line edits, it is labelled as C_{10}.

In a similar fashion, we labelled our entire dataset of $23,275$ programs, that require repairs on a single-line, with their error-repair classes (Cs). This was done automatically by obtaining the compilation errors (Es) using Clang compiler [72] and the repair tokens (Rs) using difflib python package.

Table 5.5 lists the top frequent error-repair classes (Cs), along with the number of buggy programs belonging to that C. A total of 212 Cs were found having 10 or more buggy programs. From our single-line dataset of $23,275$ buggy programs, a total of $15,579$ programs belong to one of these 212 Cs. In other words, only these $15,579$ buggy programs and their 212 Cs form the training dataset for TEGCER. While the remaining classes are unused due to lack of sufficient training examples.

As seen from Table 5.5, C_1 {E_2 `+;` } is the top frequent class, containing $3,888$

EG_{ID}	#Cs	Top-3 C_{ID}	Top-3 Classes (C)		#Programs
EG_1	24	C_2	E_3	`+INT -INVALID`	731
		C_{11}	E_3	`+ARRAY -INVALID`	286
		C_{16}	E_3	`+LITERAL_INT -INVALID`	150
EG_2	38	C_{15}	E_4	`+INT`	190
		C_{17}	E_4	`+LITERAL_INT`	145
		C_{18}	E_4	`-[ -]`	138
EG_3	28	C_3	E_1	`+}`	519
		C_4	E_1	`+)`	475
		C_7	E_1	`+,`	389
EG_4	5	C_1	E_2	`+;`	3,888
		C_{58}	E_2	`+if`	52
		C_{79}	E_2	`+; -:`	36
EG_7	2	C_5	$E_5 \wedge E_6$	`-}`	418
		C_{78}	$E_5 \wedge E_6$	`+{`	41
EG_{10}	3	C_{22}	E_5	`-,`	109
		C_{205}	E_5	`-;`	10
		C_{206}	E_5	`-int`	10

Table 5.6: Top-3 Classes (Cs) for few of the frequent error-group (EGs)

buggy programs which encounter compilation error E_2 (expected $\square_1$ after expression) and require one or more insertion of semi-colon "`+;`" to fix it. C_{212} $\{E_1 \wedge E_5 \wedge E_{47}$ `-;` $\}$ is the least frequent class, containing 10 buggy programs which encounter compilation errors $E_1 \wedge E_5 \wedge E_{47}$ and require one or more deletion of semi-colon "`-;`" to fix it.

Table 5.6 then lists the classes (Cs), grouped by frequent compilation error groups (EGs). For each EG, the second column (#Cs) denotes the total number of different repairs possible, with the third and fourth column listing the top-3 frequent class ID (C-ID) and class (C) respectively. Finally, the number of buggy programs (#Programs) belonging to each C is provided in the last column.

As seen from Table 5.6, the top frequent compilation error group (EG_1), consisting of error E_3 (use of undeclared identifier $\square_1$), has 24 different unique classes. Its top-3 frequent classes, C_2, C_{11} and C_{16}, suggest replacing an undeclared variable of type `INVALID` with a variable/literal of type `INT`, `ARRAY` and `LITERAL_INT` respectively. Similarly, the 2^{nd} (EG_2) and 3^{rd} (EG_3) most frequent compilation error groups require a wide variety of repairs; a total of 38 and 28 classes respectively. Where, EG_2

consists of error E_4 (expected expression) and EG_3 consists of error E_1 (expected $\square_1$). In comparison, the 4^{th} most frequent error group (EG_4), consisting of error E_2 (expected $\square_1$ after expression), has just 5 different types of unique repairs. Almost all of these small number of classes suggest insertion of "`;`" (semi-colon) to the buggy program, with some variation.

Additional examples of classes are presented later in the discussion Section 5.6.

5.4 Error-Repair Classifier

We experimented with various neural network setups which given a *source* (buggy abstract line), predicts the relevant error-repair class it belongs to. In this section, we report on the data pre-processing techniques used, different arrangements of neural network layers, and their precision-recall scores.

5.4.1 Data Preparation

Recall that (Section 5.3) we obtained $15,579$ source-target program pairs labelled with 212 unique classes. Although both abstracted source (buggy) and target (repaired) lines are available in this single line dataset, only the source lines were used for training and testing the classifier. In other words, TEGCER generates example based feedback, given only the buggy source program of students as input.

The labelled dataset of $15,579$ source programs is split in the ratio of $70\% : 10\% : 20\%$ for training, validation and testing purposes respectively. Before the abstracted source line is supplied to neural network for learning, we apply different pre-processing techniques to help the network generalize better.

Input Tokens

Firstly, the source sequence is split into unigram and bigram tokens. For example, consider the source line in Table 5.4. The source "`INT=INVALID;`" contains four unigram tokens {`INT`, `=`, `INVALID`, `;`} and three bigrams {`INT_=`, `=_INVALID`,

`INVALID_;`}. Special character tokens such as "`\n`", "`!`", "`!=`", … are all retained. Neural networks trained on natural languages typically ignore special symbols, but these are important from programming language perspective.

The compilation errors (***E***s) associated with each source line are passed to the neural network as well. These are prefixed to the original source line, separated by a special separator-token. For the example source line in Table 5.4, which suffers from error $\boldsymbol{E}_3$, the sequence of input tokens passed to neural network are: {`<ERR>`, $\boldsymbol{E}_3$, `<UNI>`, `INT`, `=`, `INVALID`, `;`, `<BI>`, `INT_=`, `=_INVALID`, `INVALID_;`, `<EOS>`}.

In our dataset of $15,579$ source lines, the vocabulary size is observed to be $1,756$. That is, a total of $1,756$ unique input tokens (unigrams, bigrams and ***E***s) exist in our dataset.

Input Encoding

Each input token is then vectorized using *tokenizer* [3], a text pre-processor provided by the *Keras* deep learning library [96]. Depending on the type of neural network used, the input tokens are either replaced with a unique decimal integer indicating word sequence number, or with a binary representation indicating existence of token.

More involved encoding techniques such as frequency or tf-idf [97] gave poor results, due to the extreme paucity of data for most classes; More than 30% of the 212 classes have just 10 examples to train and test from. For more information on data encoding techniques, we refer the reader to Brownlee's blog [98].

When an integer encoding is used, the sequences are padded by dummy values using the *pad_sequence* [4] pre-processor provided by *Keras*. This is done to ensure that the length of sequences is the same for all input encodings. Since the maximum sequence length in our dataset was found to be 277 tokens, all examples are padded up to contain 277 tokens.

[3] `https://keras.io/preprocessing/text/#tokenizer`
[4] `https://keras.io/preprocessing/sequence/#pad_sequences`

Output Encoding

Similar to the input encoding, output labels require encoding with numbers as well, for neural network to be able to predict them. The goal of the neural network is to predict a single class, out of the 212 different labels available, that the buggy program belongs to. Hence, the class labels are encoded with one-hot binary encoding, using the *to_categorical* [5] utility provided by *Keras*.

5.4.2 Neural Network Layers

The goal of the neural network classifier is to predict the correct label, given the buggy source. It achieves this by fine tuning the weight assignment between the various visible and hidden layers of network, until a pre-defined loss metric is minimized [99]. There has been extensive study in literature on finding the neural network setup best suited to represent computer programs. The survey by Allamanis et al. [100] lists a large number of such networks designed by prior work for different tasks, categorized on multiple metrics.

For our classification task, we experimented with some of the popular complex deep networks, such as Long Short Term Memory (LSTM) models with embeddings [88] and Convoluted Neural Network (CNN) models with max-pooling [101]. These complex models need to train a large number of parameters, in the order of millions of different weights, capturing patterns in the entire sequence of source tokens. For which, massive amounts of training data is required, typically in the order of thousands per class.

However, in our labelled dataset of $15,000+$ programs tagged with 212 labels, only the largest class has more than $1,000$ programs. 67 of these classes (31% of total 212) have just 10 examples each, for both training and testing the classifier. Due to which all the complex neural networks fail to generalize on this (relatively) tiny dataset of student errors, recording prediction accuracy in the range of $30-40\%$.

[5] `https://keras.io/utils/#to_categorical`

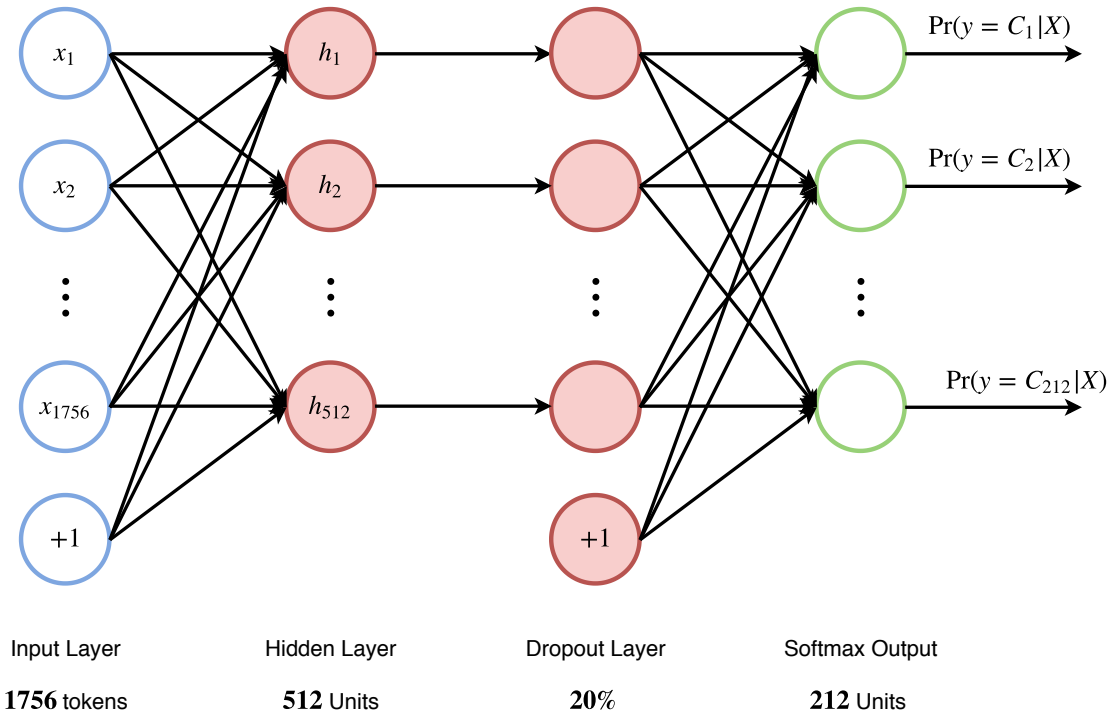


Figure 5.6: Dense Neural Network of 512 hidden units with 20% dropout, used to classify 1,756 input tokens with 212 class labels.

Dense Neural Network

To overcome the issue of small class size, we turn towards simpler neural network model for our class predictions. In particular, we found that a dense neural network with single hidden layer is best suited for our task. A dense network is a fully connected network, where each neuron is connected to all the neurons of previous layer.

Figure 5.6 describes the network arrangement of our dense classifier. The first layer is an input layer, which consists of 1756 binary encoded input source tokens (refer to Section 5.4.1). The second layer is the single hidden layer of 512 units, densely connected with the previous input layer. This is followed by a dropout layer [102], which randomly drops 20% of the input neurons to 0 during the training stage, to avoid overfitting the model on training dataset. Finally, our last layer is an output layer containing 212 units (one for each class). This output layer is densely connected with the previous hidden layer.

Parameters

The *Keras* [96] framework was used to build our dense neural network [6] classifier. The following parameters, including the hidden layer size, were selected based on the performance of classifier on the 10% validation set.

We used Rectified Linear Unit (ReLU), a non-linear activation function, between the hidden layers. A softmax activation function is used in the last output layer to predict the probabilities of all 212 classes, given the set of input tokens. For more information on the dense network and various activation functions, we refer the reader to the CS-231n Stanford course notes [103].

The training phase of classifier involves assigning weights between the input–hidden and hidden–output layers. These weights are learnt using a stochastic gradient descent algorithm, which minimizes a pre-defined objective/loss function over the 70% training data set. We used categorical cross-entropy loss function, a standard logarithmic loss function for multi class problem. Also, Adam [104], an adaptive moment estimator function, was chosen to optimize the learning process, leading to faster and stable convergence.

The peak accuracy of our model on the 10% validation data set was obtained after training for 6 epochs. Beyond which, the model begins to heavily overfit on training dataset, due to the small number of examples for most classes. The training phase typically lasts a few seconds, while predicting a class during the testing phase requires few milliseconds.

5.4.3 Accuracy

We report on the overall accuracy of classifier, as well as precision-recall scores of individual classes, on our held out 20% test dataset. The latter analysis is necessitated due to the highly skewed distribution of classes in our dataset, where the top-10 classes, out of the 212 unique ones, account for 50%+ of the total test-cases.

[6] `https://keras.io/layers/core/`

Classifier	Pred@1	Pred@3	Pred@5
Dense Neural Network	87.06%	97.68%	98.81%

Table 5.7: Pred@k accuracies for the Dense Neural Network classifier

Overall Accuracy

We measure the overall performance using a Pred@k metric, which denotes the percentage of test-cases where the top-k class predictions contains the actual class. The classifier returns a probability score for each class on being given a buggy source, $Pr(y = C_j|X)$. Which can be sorted in descending order to select the top-k results as predicted classes.

Table 5.7 reports on the accuracy of dense network in predicting one of the 212 classes on 20% held out test-set of 3,098 buggy source lines, for various values of k. When we consider only the top prediction (Pred@1) by the dense classifier, the predicted class is exactly same as the actual class in 87.06% of the cases. If we are to sample the top-3 predictions instead, then the actual class is present in one of these three predictions for 97.68% of the cases. In other words, for majority of the test cases (3026 out of 3098), dense neural network predicts the correct class in its top-3.

Top Frequent Classes

The dense neural network achieves high accuracy across majority of the 212 classes. In order to demonstrate this, we look at the precision-recall scores of individual classes. The *precision* score indicates that when the classifier labels a source line with a class C_j, then how reliable is that class-j labelling. While the *recall* score of a class C_j is used to measure, out of all the C_j labelled data in test set, how many of them were misclassified. Note that these precision-recall scores consider only the top prediction (Pred@1) of classifier.

More formally, the precision and recall scores of a class C_j are defined as follows:

$$\text{Precision}(j) = \frac{\#\text{Cases where both } \textit{Predicted} \text{ and } \textit{Actual} \text{ class is } j}{\#\text{Cases where } \textit{Predicted} \text{ is } j}$$

C_{ID}	Error-Repair Class (C)	#Train	#Test	Precision	Recall	Top Incorrect Prediction	
						Incorrect Class	#Test
C_1	E_2 `+;`	3,110	778	0.99	0.99	E_2 `+if`	4
C_2	E_3 `+INT -INVALID`	585	146	0.79	0.79	E_3 `+LITERAL_INT -INVALID`	7
C_3	E_1 `+}`	415	104	0.91	0.95	E_1 `-{`	2
C_4	E_1 `+)`	380	95	0.77	0.84	E_1 `-(`	10
C_5	$E_5 \wedge E_6$ `-}`	334	84	1.00	1.00	–	
C_6	E_8 `+;`	323	81	0.97	0.95	E_8 `+; -,`	4
C_7	E_1 `+,`	311	78	1.00	0.96	E_1 `+}`	3
C_8	E_{10} `+== -=`	247	62	0.97	0.98	E_{10} `+( +)`	1
C_9	E_1 `+;`	244	61	0.97	0.98	E_1 `+)`	1
C_{10}	E_7 `+; -,`	239	60	1.00	0.97	E_7 `+;`	2
	...						
C_{211}	E_{47} `-, -INT`	8	2	1.00	1.00	–	
C_{212}	$E_1 \wedge E_5 \wedge E_{47}$ `-;`	8	2	1.00	1.00	–	

Table 5.8: Precision and Recall Scores of Dense Neural Network on top frequent Cs. The columns #Train and #Test indicates the total number of source lines used for training/validation and testing of the classifier, respectively. For each actual class C, *Top Incorrect Prediction* column lists the top incorrectly predicted class, along with the count of test-cases in which this mis-classification occurred.

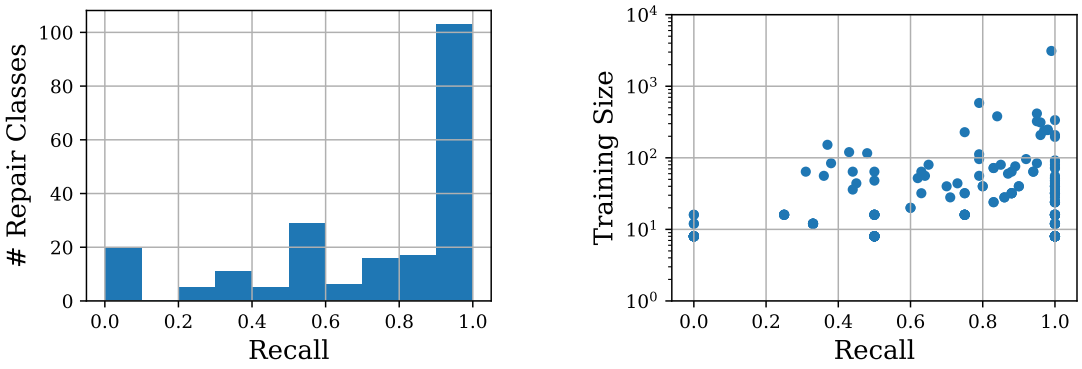
$$\text{Recall}(j) = \frac{\#\text{Cases where both } \textit{Predicted} \text{ and } \textit{Actual} \text{ class is } j}{\#\text{Cases where } \textit{Actual} \text{ class is } j}$$

In Table 5.8, we list the precision and recall scores of the dense neural network for the top frequent classes (C). As seen from the table, the dense classifier enjoys high precision and recall scores, of greater than 0.9, across most of the listed classes. Further, even on the extremely rare classes of C_{211} and C_{212}, that offer just 8 buggy examples to train and validate from, the neural network is able to generalize successfully and achieve 100% precision/recall score.

Out of the total 12 repair classes listed in Table 5.8, only two of them, C_2 and C_4, suffer from (relatively) weaker recall scores of 0.79 and 0.84, respectively. In other words, out of the 146 (resp. 95) test cases having label C_2 (resp. C_4), the classifier is able to correctly predict 79% (resp. 84%) of them. The last column, which lists the top most incorrect prediction made by classifier, suggests that this is a valid confusion due to the nature of compilation error and its repair. There are multiple different ways to repair a buggy source program, and hence the source can belong to multiple classes. However from our error-repair dataset (Section 5.3), we can observe only the single class of repair performed by actual student, and hence we treat the student's repair as the unique correct solution. This limitation of allowing only a single correct label causes a decrease in precision/recall scores, when multiple repair classes are applicable for a buggy source.

From Table 5.8, the class C_2 $\{E_3$ `+INT -INVALID`$\}$ is most confused with $\{E_3$ `+LITERAL_INT -INVALID`$\}$. This is intuitive since, in C programming language, a variable is often interchangeable with a literal of the same type. Similarly, the class C_4 $\{E_1$ `+)`$\}$ is incorrectly predicted as class $\{E_1$ `-(`$\}$ 10 different times, since the repair for incorrectly balanced parenthesis could be to either add more closing parenthesis, or remove few of the existing open ones. While choice of the repair is dictated by logical correctness, either of these two can fix the compilation error.

This suggests a larger issue, where an erroneous line can belong to multiple classes, since it can be repaired in multiple different ways. Since our erroneous-repaired



(a) Histogram of 212 classes' recall scores

(b) Effect of classes' training size on recall scores

Figure 5.7: Dense neural network recall scores for 212 classes

dataset captures only one form of repair performed by actual student, we are forced to treat the problem as multi-class, instead of multi-label. This limitation accounts for the lower precision/recall scores observed, in few of the classes.

Individual Classes

In Figures 5.7a and 5.7b, we analyze the recall scores across all 212 classes. From Figure 5.7a it is seen that, our dense neural classifier achieves high accuracy, across majority of the classes. The recall score is >= 0.5 for more than 80% (171/212) of classes, and >= 0.9 for more than 100 classes.

Figure 5.7b then presents the recall scores of all 212 classes, plotted against their training sample size. While the classifier achieves 100% recall score on classes across multiple training sizes, including those with just 8 training examples, the recall scores drop to <= 0.5 (resp. 0.0), only when the training size is below 200 (resp. 20). In other words, having more training examples for small sized classes could help improve their prediction scores.

5.5 TEGCER

In this section, we present TEGCER (**T**argeted **E**xample **G**eneration for **C**ompilation **ER**rors) in its entirety. Given a buggy program, TEGCER uses a four-phased

methodology to generate example buggy programs having similar error and repair as the original one.

5.5.1 Repair Localization

Given an erroneous code, TEGCER locates the line(s) where repair must be performed, by relying on the exact line number reported by the compiler, similar to related example generation work of HelpMeOut [17]. While TEGCER is trained only on single-line edit program dataset, during the testing phase, programs with errors on multiple-lines are treated as multiple instances of single-line errors

This is a departure from existing state-of-art repair tools, such as TRACER (Section 4.3.1) and DeepFix [73], which employ a "search, repair & test" strategy to achieve localization accuracy of 86%. Such a strategy is not applicable for example generation, due to its very nature of being subjective. TEGCER achieves localization accuracy of 80% on the same dataset, by focusing on only compiler reported lines.

For example, consider the buggy program in Figure 5.5. Here, the compiler reports an error in the for-statement, present on line number 6. TEGCER relies on the compiler reported line number, and generates examples having similar error-repair pairs as the original for-statement.

5.5.2 Code Abstraction

As a second step, TEGCER pre-processes the compiler reported erroneous code lines, by replacing variables/literals with their abstract generic types. This stage greatly reduces the load on neural networks and helps it generalize better, by performing implicit vocabulary compression. The neural network is better equipped to predict repairs based on type information, instead of finding loose patterns through user defined names.

The abstraction module used by TEGCER is exactly similar to the one employed by TRACER, as described in Chapter 4.3.2. Tables 5.3 and 5.4 list two examples of this abstraction process.

5.5.3 Error-Repair Class Prediction

In the third step, TEGCER assigns one of the 212 error-repair class ($\boldsymbol{C}$) to each of the abstract code lines, obtained in previous step. This is done using the dense neural network classifier, trained offline as described in Section 5.4.2. The class label indicates the closest type of error in buggy program and its desired fix, in terms of insertion/deletion of abstract program tokens.

TEGCER is able to achieve significantly higher prediction score of 97.68% as opposed TRACER's 66.61%, when considering the top-3 suggestions (Pred@3) of both the compilation feedback tools. This is because TEGCER only needs to only predict the type of tokens that should be inserted/deleted from buggy code, as opposed to the additional challenging task that TRACER has to perform in figuring out the exact position where the edit should be applied.

5.5.4 Example Suggestion

Given an erroneous student program, TEGCER assigns error-repair class label for each compiler reported line, and suggests $\boldsymbol{N}$ number of relevant examples for each class. These example programs are chosen as follows: (i) from our dataset of $15,579$ buggy programs, we filter all those programs that have the same class of error-repair as student's buggy program, and (ii) The top-$\boldsymbol{N}$ frequent erroneous programs per class are suggested back to the student.

For each example, only the error code, or only the repaired code, or both error–repair code pairs, can be provided to the student depending on the instructor preference. In prior work of HelpMeOut [17], both the error and its repair example are provided as feedback to the student. Also, the choice of number of examples, $\boldsymbol{N}$, is parameterized and should typically be set to 3 or more, since students might need to observe multiple different examples in order to learn the general pattern of mistake made and its desired fix.

Table 5.9 lists few buggy source lines, and the top example feedback provided

#	Source-Line and Target-Line	Predicted Class	TEGCER's Top Example Erroneous and Repaired	Relevant
1	`d = (x-x1) (x-x1);` `d = (x-x1)*(x-x1);`	C_{32} $\{E_{15}$ `+*` $\}$	`amount = P (1+T*R/100);` `amount = P*(1+T*R/100);`	✓
2	`printf("Ans=" a);` `printf("Ans=%d", a);`	C_{7} $\{E_{1}$ `+,` $\}$	`printf("%s" str);` `printf("%s", str);`	✓
3	`if (a!(==)0)` `if (a!=0)`	C_{148} $\{E_{1}$ `+!= -!` $\}$	`while (n ! 0)` `while (n != 0)`	✓
4	`for(j=0, j<10, j++)` `for(j=0; j<10; j++)`	C_{10} $\{E_{7}$ `+; -,` $\}$	`for(i=0, i<n; i++){` `for(i=0; i<n; i++){`	✓
5	`b=xyz+1;` `b=a+1;`	C_{2} $\{E_{3}$ `+INT -INVALID` $\}$	`scanf("%d", &a);` `scanf("%d", &n);`	✓
6	`printf("%d", a));` `printf("%d", a);`	C_{40} $\{E_{19}$ `-)` $\}$	`printf("(%f,%f"), x,y);` `printf("(%f,%f)", x,y);`	✓
7	`printf["%d", a];` `printf("%d", a);`	C_{210} $\{E_{62}$ `+( +) -[ -]` $\}$	`sort[rsum, n];` `sort(rsum, n);`	✓
8	`} else {` `else {`	C_{71} $\{E_{5} \wedge E_{6}$ `+{` $\}$	`else` `else {`	✓
9	`p+'a' = p;` `p = p+'a';`	C_{8} $\{E_{10}$ `+== -=` $\}$	`if (a = b)` `if (a == b)`	✗

Table 5.9: Sample buggy source lines and the top example generated by TEGCER. The second column lists erroneous source lines and actual repairs performed by the same student. The third and fourth column then describe the top class and top example predicted by TEGCER, respectively. Finally, the last column indicates whether the predicted example feedback is relevant to the student's buggy program.

by TEGCER. Both the error (source-line) and its repair (target-line) are provided in the second column. Similarly, the before repair and after repair code lines are provided for the suggested example, belonging to the predicted class.

In the first two rows #1 and #2 of Table 5.9, the student fails to realize that that an asterisk '*' operator and a comma separator ',', respectively, is required between the two expressions. While in rows #3 and #4, the student has misunderstood the inequality operator '`!=`' and semi-colon separator in "`for`" statement, respectively.

The suggested examples by TEGCER for these top four rows reflects the same confusion and desired repair, as the given student's buggy source line. The compiler reports cryptic error messages for row #1 E_{15}`: called object type "int" is not a function`, row #2 E_1`: expected ')'`, and row #3 E_1`: expected ')'`. Hence, providing TEGCER's examples as feedback to novice students could be more useful over these compiler messages, which are neither relevant to student's error nor to the desired repair.

For row #5, the student uses an undeclared variable whose fix, as suggested by the top example, is to replace the undeclared variable with an integer variable.

Rows #6 and #7 deal with bracketing issues in `printf` function call. For row #6, the student has mistakenly added an additional closing parenthesis "`)`", and TEGCER's example suggests its deletion. While in row #7, a student unfamiliar with the correct type of bracketing in "`printf`" function call, used square brackets "[ ]" instead of the round-brackets/parenthesis "( )". TEGCER's example, of replacing the parenthesis in a user-defined function call "`sort`", is highly relevant with the exact same confusion and desired repair.

And in row #8, TEGCER recognizes that the error is to do with unmatched braces. Although its predicted class, and hence suggested example, suggests adding an additional brace, the student in test dataset removed a brace instead. We still count this example as relevant, since either solution is valid and it does point to the student the need to adjust the braces.

Row #9 demonstrates the limitation of our dense neural network, which relies on

unigrams, bigrams and compilation error messages to predict the class. In this case, the buggy source line is `p+'a'=p;`, and the student is confused between the lvalue (left) and rvalue (right) of assignment operator. The input tokens passed to the neural network are {`<ERR>`, E_{10}, `<UNI>`, `CHAR`, `+`, `'`, `LITERAL-CHAR`, `'`, `=`, `CHAR`, `;`, `<BI>`, `CHAR_+`, `+_'`, `'_LITERAL-CHAR`, `LITERAL-CHAR_'`, `'_=`, `=_CHAR`, `CHAR_;`, `<EOS>`}.

As we can see, there is very little indication from these unigram and bigram tokens, that the issue is due to presence of multiple tokens in lvalue of assignment. In fact, examples belonging to a frequently occurring class, C_8, have very similar unigram/bigram tokens, where the confusion is between usage of assignment "`=`" and equality "`==`" operator. Hence, the classifier incorrectly predicts the class C_8 {E_{10} `+== -=`} for the row #9 source line, suggesting replacement of "`=`" with "`==`". Using more complex neural networks, which capture the entire sequence information, could help resolve this issue. However, these would require much larger training dataset.

5.6 Discussion

In this section, we discuss some of the important features and limitations of TEGCER, our approach to generate example based feedback for compilation errors.

Class as Error-Repair pair

Since TEGCER's aim is to ultimately provide relevant examples through which students can learn and transfer the repair, it is important for the class to capture both the type of repair as well as the type of error. If classes are assigned to programs on the sole basis of repair tokens, then programs suffering from different mistakes will be incorrectly grouped together.

Consider the two erroneous programs in Table 5.10. In both the programs, the repair tokens suggest replacing comma '`,`' with semicolon '`;`'. However, the relevant examples that can be used as feedback are different for both. Program #1 requires examples which teaches about the necessity of semicolon separator, at the end of every statement. Where as program #2 requires examples that refresh the student on

	Program #1	**Program #2**
Erroneous	`int a=0,`	`for(i=0, i<10; i++)`
Repaired	`int a=0;`	`for(i=0; i<10; i++)`
Compilation Error	E_8: expected ';' at the end of declaration	E_7: `expected ';' in for statement specifier`
Repair Tokens	`-, +;`	`-, +;`

Table 5.10: Compilation error and repair tokens of two erroneous programs.

syntax of for-statement specifier, with semicolon separators between each component. Both of these errors belong to different class of examples, and hence the compilation error is an integral part of the class.

Unseen Error

Further, the design of class is flexible in accommodating new kinds of mistakes that students would make, unobserved in our training dataset. To further elaborate, consider an unlikely buggy line "`printf("%d" xyz!0);`", where the student has made 3 different mistakes. Namely, a comma separator '`,`' is missing between both expressions, the inequality operator '`!=`' is incorrectly written as '`!`', and "`xyz`" is an undeclared variable of type "`INVALID`". The compiler reports just a single error E_1: `expected ')'` on this particular line.

Our neural network classifier has never come across this combination of errors during its entire learning phase. Nonetheless, it is able to capture the errors better than a standard compiler, and predict relevant classes for all 3 mistakes individually. The top-4 classes predicted by TEGCER are C_{148} {E_1 `+!= -!` }, C_2 {E_3 `+INT -INVALID`}, C_{35} {E_{12} `+INT -INVALID`}, and C_7 {E_1 `+,` }. Hence, TEGCER can be used to successfully generate relevant feedback on unseen errors, by sampling multiple top-N classes. Notice that TEGCER does not restrict itself to the (incorrect) compiler reported error E_1, and this freedom allows it to match buggy program with relevant repair, by borrowing from those repairs observed in different errors E_3 and E_{12}.

```
1 #include<stdio.h>
2
3 int main(){
4     int i;
5     i = 10
6 }
```

		Compiler Message
5	E_2	expected $\square_1$ after expression

(a) Buggy program with error E_2

```
1 #include<stdio.h>
2
3 int main(){
4     int i;
5     i = 10;
6 }
```

	Class
C_1	$\{E_2$ +; $\}$

(b) Correct program, with fix $\in C_1$

Figure 5.8: Example buggy program and its fix belonging to class C_1.

```
1 #include<stdio.h>
2
3 int main(){
4     int i
5     i = 10;
6 }
```

		Compiler Message
4	E_8	expected $\square_1$ at end of declaration

(a) Buggy program with error E_8

```
1 #include<stdio.h>
2
3 int main(){
4     int i;
5     i = 10;
6 }
```

	Class
C_6	$\{E_8$ +; $\}$

(b) Correct program, with fix $\in C_6$

Figure 5.9: Example buggy program and its fix belonging to class C_6.

Local Context

Compiler Line Number: Note that, matching braces is a tricky issue due to its global context, and TEGCER's performance is directly dependant on compiler. In some cases, the compiler reports the line number where it encountered brace error, which is not the same as the line to perform the repair. In which case, TEGCER would fail to predict the correct class, and hence the error-repair examples.

Duplicate Classes

In Figures 5.8 and 5.9, the compiler reports different errors E_2 and E_8, based on whether the line is an expression or declaration (respectively). However, both these buggy programs require an insertion of ";" at the end of line, whose error-repair could be understood by students on being given the same set of examples. Similar

classes such as these can be merged together, to achieve improved precision/recall and hence suggest more relevant examples to students. In future, we plan to develop an automated (or semi-automatic) technique to achieve this.

Note that such a technique would be non-trivial and cannot rely solely on the desired repair (insertion/deletion of token). For example, the fix in Figure 5.5, which requires a additional ";" token inside the for-statement, is different in nature from that required in Figures 5.8 and 5.9. Hence, the examples suggested to student as feedback should be different as well.

Multi Label

Conversely, erroneous programs can belong to multiple classes as well. Since the input erroneous program is an incomplete specification of the desired repair, predicting the students' intention can become intractable in certain situations. Consider the buggy-repaired code pairs in Figure 5.10 and 5.11, where both the buggy programs are exactly the same. While both the repairs (Figure 5.10b and 5.11b) are valid, the desired fix (and hence the relevant example) depends on student's intention.

Treating this problem as multi-label, as opposed to multi-class problem, can help ameliorate this issue. However, the exponential number (power-set) of possible labels makes this a challenging problem, typically leading to a decrease in classification accuracy [105]. Multiple methods, such as by Guo et al. [106] and Zhang et al. [105], have been proposed to increase the multi-label accuracy by capturing inter-dependencies among labels. We plan to explore the same in a future version of TEGCER.

Unigram and Bigram Features

Our usage of (relatively simpler) unigram/bigram features sometimes fails to capture the error-repair class that an erroneous program belongs to; as demonstrated in row #9 example in Table 5.9.

Using more complex neural network models, such as LSTMs which capture the

```
1 #include<stdio.h>
2
3 int main(){
4     int i=0;
5     int j=;
6 }
```

	Compiler Message	
5	E_4	expected expression

(a) Buggy program with error E_4

```
1 #include<stdio.h>
2
3 int main(){
4     int i=0;
5     int j=i;
6 }
```

Class	
C_{15}	$\{E_4$ +INT $\}$

(b) Correct program, with fix $\in C_{15}$

Figure 5.10: Example buggy program and its fix belonging to class C_{15}.

```
1 #include<stdio.h>
2
3 int main(){
4     int i=0;
5     int j=;
6 }
```

	Compiler Message	
5	E_4	expected expression

(a) Buggy program with error E_4

```
1 #include<stdio.h>
2
3 int main(){
4     int i=0;
5     int j=0;
6 }
```

Class	
C_{17}	$\{E_4$ +LITERAL_INT $\}$

(b) Correct program, with fix $\in C_{17}$

Figure 5.11: Example buggy program and its fix belonging to class C_{17}.

entire sequence of tokens and long term dependencies between them, can help resolve such issues in future. However, the added complexity would require 100s of additional training examples for each individual class, to learn the general pattern of error. Our current dense network trained on unigram-bigram tokenization achieves a Pred@3 accuracy of 97.68% on our dataset of $15,000+$ erroneous programs, indicating that such long term dependencies affect a very small number of classes. Hence we believe our proposed modeling is one of the best suited for the task of example generation, until a significantly larger dataset can be collated.

5.7 Summary

In this chapter, we presented TEGCER, a new kind of feedback tool that automatically suggests relevant examples to novice programmers, who are struggling with compilation errors. TEGCER's example generation does not require manual

intervention, unlike prior related work of HelpMeOut [17] which require manual tagging by students. Also, the different examples provided compel the student to learn from the general pattern of error and its repair, in order to transfer the knowledge to fix their own erroneous program. This is a paradigm shift compared to repair tools such as TRACER, which attempt to directly fix the student's erroneous program and provide the generated repair as feedback, potentially loosing out on the pedagogical value of automated feedback.

The automated example generation by TEGCER is based on the realization that, while students make multiple types of compilation errors on various programming assignments, the kind of repairs performed by them is limited to addition/deletion of few abstract program tokens. To this end, we first collate a large dataset of $15,000+$ aligned student programs with compilation errors, and automatically label all with their error-repair class $\boldsymbol{C}$, which represents the combination of error and desired repair type. Next, a simple dense neural network classifier is trained to predict the desired class, using the unigram tokens, bigram tokens and compilation error messages of buggy programs. In the challenging task of predicting a single label out of 212 different classes, our trained classifier achieves high Pred@3 accuracy score of 97.68%, indicating that the labelling is sound and error patterns are generalizable. Finally, the top representative examples, that suffer from the same error-repair class as student's, are suggested back as feedback.

We believe that similar example generation based feedback tool can be developed for the logical mistakes made by students, using the novel framework proposed in this thesis. Given a buggy program that suffers from logical errors, a repository of error-repair can be searched for semantically equivalent repairs desired by the buggy program. These filtered repairs, undertaken by other students on their own programs, can then be provided as hints. The primary challenge in such an approach would be to define the semantic (or abstract) representation and semantic equivalence of programs.

In the next Chapter 6, we undertake a large user study of 400+ first year

undergraduate students. Both TRACER, a state-of-the-art program repair tool, and TEGCER, a state-of-the-art example generation tool, are deployed live for an entire course offering of introductory programming (CS-1) course. The performance of students with access to these feedback tools is compared against a baseline, in order to analyze which tool has better pedagogical value.

Chapter 6

User Study of Novice Programmers

In this chapter, we report empirical results from a randomized controlled experiment, performed to assess the efficacy of automated feedback tools in helping novice programmers. Our experimental group consists of 480 students, who were randomly assigned to receive feedback from two separate automated tools – TRACER and TEGCER. These feedback tools were designed using well known instructional principle for efficient learning, where feedback is provided either through (i) relevant solution, or through (ii) generated examples that guide towards the solution. We compare the performance of students with access to these automated tools, against student performance when tutored by human teaching assistants, albeit from a different offering of the same course at the same university.

We found that automated feedback allows students to resolve errors in their code more efficiently than students receiving manual feedback. This advantage disappears when all forms of feedback are withdrawn, and therefore does not correspond to improved conceptual understanding of compiler errors and its resolution. In other words, the advantage appears to correspond with timely instructional delivery via repair and examples.

We also found that the performance advantage of automated feedback over

human tutors increases with problem complexity, and that feedback via example and via specific repair have distinct, non-overlapping relative advantages for different categories of programming errors.

6.1 Introduction

There is active interest in aiding students struggling with compile-time errors (also known as compiler or compilation errors). These errors are caught and reported by the compiler, before the program is even executed. They typically occur due to the incorrect use of language syntax, or on importing incorrect files/libraries. These are, in fact, the types of errors most likely to be made by novitiates into programming and therefore, are of prime interest for instruction support in introductory programming courses.

Compiler errors are considered trivial to locate and fix by expert programmers, since the compiler reports the line-number in code where the error occurs and a short message explaining the error it encountered. Likely for this very reason, text in compiler error messages are targeted towards experienced programmers, and appear cryptic to beginners [70]. Multiple studies have shown that students who have just begun to learn programming often struggle with compilation errors [107, 71].

Recognizing the need to help novices, multiple approaches have been proposed to enhance the error messages, to make them more informative and suitable for novice programmers. As seen in the systematic literature review by Reilly et al. [108], these enhancements are typically achieved by manually analyzing the top frequent errors made by students and the erroneous code associated with it. Then additional information is added to the existing compiler error message by an expert, such as the usual causation of its fixes, with the process being repeated for different programming languages.

These techniques typically suffer from coverage, with a large number of infrequent errors being left out due to the manual effort involved. Moreover, the effect of enhanced error messages on student performance is inconclusive [109]. While some

studies demonstrate lower number of student errors per enhanced error message [110], other conflicting studies claim that improved messages are ineffective in helping students [109, 111], with no significant measurable effect being observed. Lack of attention to the error messages could be one of the reasons for this effect [109], with another study [112] proposing to make the error messages more engaging by introducing humour in them.

Owing to the above limitations in enhancing error messages, alternative forms of feedback have been explored which are scalable in nature. In particular, there has been significant interest recently in automatically suggesting fixes for compilation errors to students [76, 73, 77, 11], thus providing feedback by the way of repair. These tools learn from the repeated mistakes made by historical student submissions and eliminate manual effort in the current offering, making it possible to scale-up to massive class sizes.

While such automated program feedback efforts are all conceptually laudable, it is not empirically clear whether they actually help students learn better or not. As we discuss in greater detail below, empirical data differ in direction of the learning effect, and measure different and incompatible metrics. The existing studies reporting positive effects are typically small scale ($N < 50$), and report on average time-taken metrics, which fails to capture whether student performance gains are truly correlated with student learning.

To address these limitations in the literature surrounding automated feedback systems for learning programming, we conducted a large ($N = 480$) randomized controlled experiment, replacing human tutors with automated feedback systems for the first seven weeks of the introductory programming course at Indian Institute of Technology Kanpur (IIT-K), a large public university. We measured programming performance improvement using multiple metrics longitudinally. We found significant gains in error-resolution during programming assignments as a consequence of this intervention, over human-assisted baselines. The extent of improvement observed was greater for difficult errors than easier ones. We also found that programming

performance improvements seen during feedback tool use remained intact, once the tools were withdrawn at the midway point of the course. The remainder of this chapter describes our methods and results in more detail.

6.2 Feedback Tools

A program can have one or more compilation errors, a collection of which is called error group. Fig 6.1 shows a student code having 3 different errors. As observed in our course and prior work [76, 71, 110], the frequency count of the kind of errors novice programmers make is an exponentially decreasing function. During our course offerings, students make 100s of different kinds of errors, with the top-8 frequent error groups accounting for more than 50% of them.

There have been multiple instructional principles evaluated for efficient learning in literature, based on the complexity of the educational content [6]. Providing timely feedback is one such important instructional principle, and studies on tutoring systems for programming language indicate significant learning benefit when students are provided some form of feedback [7].

In this chapter, we focus on two particular feedback strategies relevant for novice programmers; i) error correction, and ii) example-based feedback where learners solve problems by analogy [95].

6.2.1 Discriminative Feedback: Repair

Students with compilation errors are provided discriminative feedback in the form of correct code, known as repair. To generate this code repair we used TRACER [76], the current state-of-the-art compilation error repair tool, presented in earlier Chapter 4. Given an erroneous program, TRACER (i) locates the error lines (ii) abstracts the variables and literals with their types (iii) predicts the relevant abstract repair using a deep learning sequence-to-sequence model, and (iv) converts back the repair from abstract representation to concrete code.

TRACER boasts 80% repair accuracy for single-line errors, and 44% accuracy on multiple-line errors, while being relevant - its top recommended fix is the exact same as students' 68% of the times.

6.2.2 Generative Feedback: Examples

Motivated by the approach used in TRACER, we designed a feedback tool called TEGCER, as described earlier in Chapter 5. TEGCER predicts examples of incorrect-correct code pairs, which suffers from the same mistake and desires same repair as the given erroneous program. Given an erroneous program, TEGCER (i) locates the error lines (ii) abstracts the error line (iii) predicts the class of error-repair which this abstract line belongs to (iv) suggests back the top frequent training examples observed for this particular class.

In the challenging task of predicting a single correct label of erroneous program, out of 212 different classes representing the error and repair, TEGCER achieves high Pred@3 accuracy score of 98%. TEGCER has high recall scores across majority of the classes, with recall score >= 0.9 for about half of the them; even on the classes that had just 8 code examples to learn from.

Figure 6.1 demonstrates the feedback generated by both these tools on a sample program having multiple errors. Both the tools typically generate relevant feedback within 1 second. Although the course and feedback tools are targeted towards C programming language, our approach and results generalize to other programming languages as well.

6.3 Experimental Setup

We conducted our study as a randomized controlled trial (RCT) at IIT-K, large public university, during the 2017–2018–II (spring-semester) course offering of the first programming course (CS-1). This is a common core course at this university, offered across two semesters to all first-year undergraduates. One of the major

Figure 6.1: Our custom IDE Prutor [79] deployed for programming assignments, with repair and example feedback tools integrated. Top-right *editor* pane contains the erroneous student code, and bottom-right *console* pane displays the compiler errors reported by Clang [72]. Only one of the feedback types (left pane: top or bottom) is shown in *tutor* pane, depending on the students' group assignment.

Lab #	Topic	Sample Question
LAB-01	Input/Output	Compute simple interest
LAB-02	Conditionals	Check leap year
LAB-03	Iterations	Check prime
LAB-04	Nested Iterations	Sum of primes
LAB-05	Functions	Predict day
LAB-06	Arrays	Remove duplicates
LAB-07	Matrices	Compute determinant

Table 6.1: Weekly programming lab topics

components of this course are weekly programming assignments, where students attempt 3 programming questions of varied difficulty every week, from the university computer lab. These *labs* are conducted from 14:00 to 17:15 hours (~3 hours long) on one particular day of the week. The students are not allowed to discuss among themselves, or use the internet to solve these questions. Only hand-written notes are allowed for reference. The score/marks allotted to students' submission is directly proportional to the number of instructor-designed test-cases they pass. In the regular running of the course, 40+ tutors, who are CS post-graduate students at the same university, remain physically present during these labs to help guide the undergraduates.

Table 6.1 lists the topic of assignments, along with one sample assignment question, during the first seven weekly labs.

6.3.1 Prutor: Online Programming Editor

At this university, the CS-1 course assignments are developed and submitted by students on Prutor [79]; an internal website accessible through a web-browser (client), with the majority of the computations such as program compilation and execution occurring on a dedicated server. This website hosts a custom built Integrated Development Environment (IDE) targeted for education, along with various administrative features including account management, display of grading rubric etc.

Specifically appropriate for our purpose, Prutor maintains a complete record of compilation requests by each student, including a unique anonymized ID per student,

week # of the course, student code, compilation errors it triggered (if any), and the timestamp.

6.3.2 Experimental Groups

480 students credited the course during 17–18–II semester offering, and they were divided into 2 experimental groups randomly using odd-even roll numbering. During the weekly lab conducted from 14:00 to 17:15 on any one particular day, 242 of these students received feedback from TRACER, our repair-hint tool; while the remaining 238 students received feedback from TEGCER, our example-hint tool. The number of students with odd roll numbering is not equal to those with even roll numbering due to administrative reasons, such as course drops. Human tutors were asked to desist from helping students resolve compilation errors, unless they had been unable to resolve them using our automated feedback suggestions for more than 5 minutes. For 7 weeks, these students continued to receive immediate feedback from the same tool that they initially received from.

Figure 6.1 shows a screenshot of the user-interface when a student compiles an erroneous program resulting in compiler errors. The right-pane is an *editor* where students write their code. The bottom-pane contains the *console* tab where compiler errors and warnings from a standard C compiler are displayed. The left-pane contains the *tutor* tab which displays feedback from our hint tool, depending on the students' control group. In the *tutor* tab, the first set of feedback are the repair-hints for the sample code, and second set of feedback right below it shows the example-hints for the same sample code. Note: Only one of these hints is shown to a student, depending on his/her control group. We have shown both these feedback together in this figure for ease of comparison.

The compilation errors encountered by the sample program of Figure 6.1 are listed in the *console* tab. The error messages for line #5 and line #8 are cryptic, with the compiler treating them as an incorrect function invocation and misplaced bracket, respectively. This can be even more confusing for novice students, learning

Semester	Feedback Type	#Students
16–17–II	Manual	439
17–18–I	Manual	453
17–18–II	Repair	242
17–18–II	Example	238

Table 6.2: # Students enrolled in different course offerings

their first programming language, who would then require human tutor assistance to understand the error and then fix it. On the other hand, as seen in the *tutor* tab, both our repair and example tool produce relevant feedback for all three erroneous lines; with TRACER directly suggesting the desired repair by student, and TEGCER suggesting multiple examples of correct syntax that hint at the desired repair.

Whenever a compilation error occurs, the *console* and *tutor* tabs are activated automatically. By default, either a single fix (per line) or the single best example (per line) is shown to students belonging to the repair or example groups respectively. Further, students who receive the example hints can choose to generate the next best examples, up to a maximum of 10 per line, by clicking on the button under *More?* column. For instance, in the Figure 6.1, for the example hints (second set of compilation hints in tutor pane), the student has requested one additional example for line #7 and two additional examples for line #9, by utilizing the *More?* option.

6.3.3 Control Groups

The previous two offerings of the course at the same university, albeit by different instructors, are used as baseline. Students in these two semesters used the same browser based IDE, Prutor, to complete their programming tasks, but without any feedback tool deployed. They relied on the compiler error messages and manual help by human tutors in order to resolve both compilation and logical errors. Table 6.2 lists the number of students enrolled in all three offerings.

All course offerings followed the same syllabus and weekly lab settings, as shown in Table 6.1. Different sets of lab assignments were framed for each offering. That is, the repair and example group students received the same questions, which were different

from other baselines. After the students get 7 labs (or weeks) of programming practice, they appear for a mid-semester lab examination where they submit their code on the same browser-based editor. For this exam event, neither feedback-tools nor human tutors provided any help on compilation errors. Students wrote their code under the invigilation of tutors, where discussion with other students or usage of any reference materials was disallowed.

6.4 Results

In this section, we measure the utility of automated feedback tools using a suite of existing metrics. Further, we propose a pair of new measurements that provide additional clarity on the value of these tools to novice programmers.

Previously proposed metrics for assessing feedback efficacy include the number of errors students make over time [17], and the average time-taken by students to fix the errors [78]. Both metrics, in isolation, are inadequate to demonstrate true feedback efficacy.

Multiple factors unrelated to feedback tools can attributed to improvements, such as problem difficulty. Since our experimental and control groups had to work on different set of assignment problems, comparing only the overall average results for these groups would be insufficient, since the set of errors made could be different. Also, the existing average metrics could be dominated by frequently occurring errors, such as semi-colon missing which is trivial to solve; or by top student performers, who do not need any additional feedback to resolve their errors. We show below how we engage with these concerns in our analysis.

6.4.1 Number of Errors over Time

RQ1: Do students with access to feedback tools make lesser #errors over time?

In the top row of Figure 6.2, we report the number of errors that students make in the 16-17-II, 17-18-I course offerings (without feedback) and the 17-18-II course

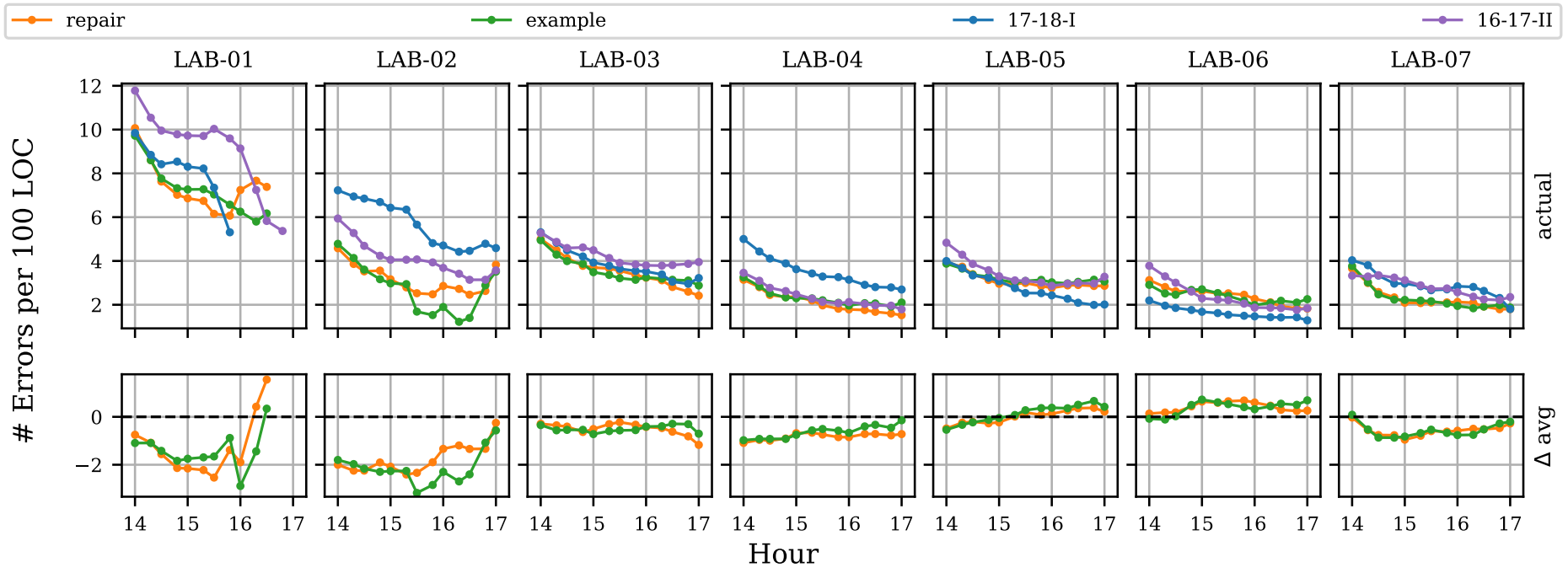


Figure 6.2: Plot depicting the #errors per 100 Lines of Code (y-axis) that students make over time (x-axis), during weekly lab event scheduled between 14:00 to 17:15 hours. The bottom row of plots depict the difference between #errors made by students with access to feedback tools and the average #errors of both previous offerings.

offerings (with repair and example feedback tools deployed). Each sub-plot focuses on one of the seven weekly labs, held between 14:00 to 17:15 hour (x-axis). The topic of these labs are mentioned in Table 6.1.

Each point in the seven subplots in the top row of Figure 6.2 refers to the #errors (y-axis) that students made for that particular tool/semester, in a 30 minute window at a particular time (x-axis) during lab. These #errors are normalized over #Lines-Of-Code (LOC) written by student, since the questions and total #students vary across offerings.

For example, consider the Lab-01 of 16-17-II course offering, where students made a total of 9,217 compilation requests between 14:00 to 14:30, out of which 1,523 programs had compiler errors. The programs that failed to compile had a total of 12,930 lines of code, which gives us **0.118** #normalized-errors (1523/12930). This is represented by the left-most purple dot in Figure 6.2, where x-axis=14 and y-axis=11.8. The second purple dot at y-axis=10.5 represents the #normalized-errors for the time-frame 14:15 to 14:45. The rest of the plot proceeds similarly.

From Figure 6.2, it can be observed that students tend to make more #errors at the beginning of lab, which steadily decreases until the end of lab. There is also a natural endogenous decline in errors across labs, where students eventually reach an average of 2-4 errors per 100 LOC, through practice and tutor assistance. Not only does this trend hold in our experimental group sample during 17-18-II, but students seem to be reaching the stable 2-4 errors per 100 LOC state faster, suggesting that our feedback tools are adequately replacing human tutor assistance in previous offerings.

The seven plots in the bottom row of Figure 6.2 depict the difference between #errors made by students with access to feedback tools and the average #errors of both the previous offerings. For example, for Lab-01 plot at 14:00 hour, the number of normalized-errors per 100 lines of code is 10.0, 9.7, 9.8 and 11.8 for repair, example, 17-18-I and 16-17-II groups respectively. That is, the average number of normalized-errors for the baseline group is 10.8, with the average difference between

repair and baseline being $10.0 - 10.8 = -0.8$ and $9.7 - 10.8 = -1.1$ between example and baseline group. The same is depicted as orange and green point (respectively) on the bottom plot for Lab-01, x-value = 14. Generally speaking, negative values of the Y-axis for the bottom row graphs indicate circumstances where the experimental groups appear to have an advantage over the human-tutored control groups, in terms of the number of errors manifested per 100 lines of code.

From these plots, a strong protective effect is seen for automated feedback tools at the outset of the course. During Lab-1 and Lab-2, students with feedback tools make an average of 1-2 fewer errors per 100 lines of code, as compared to human-tutored controls. This difference is no longer prominent beyond the second week, with automated feedback groups performing statistically identical with the human-tutored controls thereafter.

6.4.2 Time-Taken and Number of Attempts

RQ2: Do students with access to feedback tools resolve errors faster?

From the logs recorded by Prutor, our web-browser based IDE, we can calculate the time-taken by students to fix their compilation errors. This is defined as the time elapsed from the first occurrence of a compilation error, to the next successful compilation made by student. Similarly, we also extract the number of attempts it took to fix this error, defined as the number of unsuccessful compilation requests made by student before finally resolving all compilation errors.

For example, consider a student's code which ran into compilation errors $\boldsymbol{E}_1$ and $\boldsymbol{E}_2$, i.e $\boldsymbol{E}_1 \wedge \boldsymbol{E}_2$ at 14:10. Further, let us assume that the student made 3 different compilation requests after this, which resulted in errors $\boldsymbol{E}_1 \wedge \boldsymbol{E}_2$ at 14:11, $\boldsymbol{E}_1$ at 14:12, and finally $\varnothing$ (a successful compilation) at 14:13. Then we conclude that the student took 180 seconds and 3 attempts to resolve the compilation error which occurred at 14:10. In other words, the intermediate errors are accounted for by the #attempts and their individual time-stamps are ignored.

Table 6.3 summarizes cohort level statistics for our control and experimental

Semester	16–17–II	17–18–I	17–18–II	17–18–II
Feedback Type	Manual	Manual	Repair	Example
#Students	439	453	242	238
#Success Compile	155,135	161,326	99,445	97,763
#Failure Compile	13,454	15,026	7,306	8,624
Time-Taken (AVG)	85	103	75	78
Time-Taken (SD)	137	155	123	132
#Attempt (AVG)	2.11	2.20	1.88	1.99
#Attempt (SD)	1.95	2.10	1.69	1.86

Table 6.3: Total # successful and failure compilation-requests made by students during the 7-week labs across various offerings, along with the time-taken and #attempts to fix the failures.

groups. We only consider attempts with time taken lower than 15 minutes, to filter out cases where human tutors or other external events intervened in the problem solving process, such as student moving away from desk.

The cohort level statistics are directionally consistent with the hypothesis that automated feedback tools allow students to fix compilation errors more efficiently - using lesser time, and taking fewer attempts. Pairwise t-tests evaluating differences between the experimental and control cohorts, for both time taken and attempt counts, are all statistically significant at $p < 10^{-5}$; after Bonferroni correction to account for multiple comparisons.

While the differences point in the right direction and are heavily statistically significant, the size of the effects range from 0.05 to 0.19 for time taken, and 0.06 to 0.16 for attempt count. Thus, these conventional cohort-level analyses, by virtue of the large sample size of our study, reliably demonstrate a small improvement in resolving compilation errors for automated feedback tools, when compared to performance while assisted by human tutors.

In other words, while the average number of errors over time (Section 6.4.1) made by experimental group (students with access to our feedback tools) is similar to that of the control group after 2 weeks of deployment, the time-taken analysis (reported in this Section) demonstrates that the tools are indeed helpful in resolving the errors

Time-Taken (t)	#Errors Survived	$S(t)$
0	100	100/100 = 1.00
25	70	70/100 = 0.70
75	30	30/100 = 0.30
100	0	0/100 = 0.00

Table 6.4: Survival probability estimate for time-taken on hypothetical data.

faster.

6.4.3 Error Survival Probability for Time-Taken

RQ3: Do students with access to feedback tools have a lesser probability of their errors surviving, after spending an equivalent amount of time?

Seeking to go beyond summary assessments of cohort-level performance, we turned to survival analysis for better representation and analysis of the tens of thousands of compilation errors which students made in our course offerings. In particular, we estimate an error survival function that models the probability of an error surviving beyond a specified time. Survival functions are commonly used in the areas of system reliability, social sciences and medical research. For example, they are used to model the probability of a cancer patient surviving at time t, while under the effect of a treatment-X [113].

Survival probability function, commonly denoted as S, is formally defined as

$$S(t) = Pr(T > t)$$

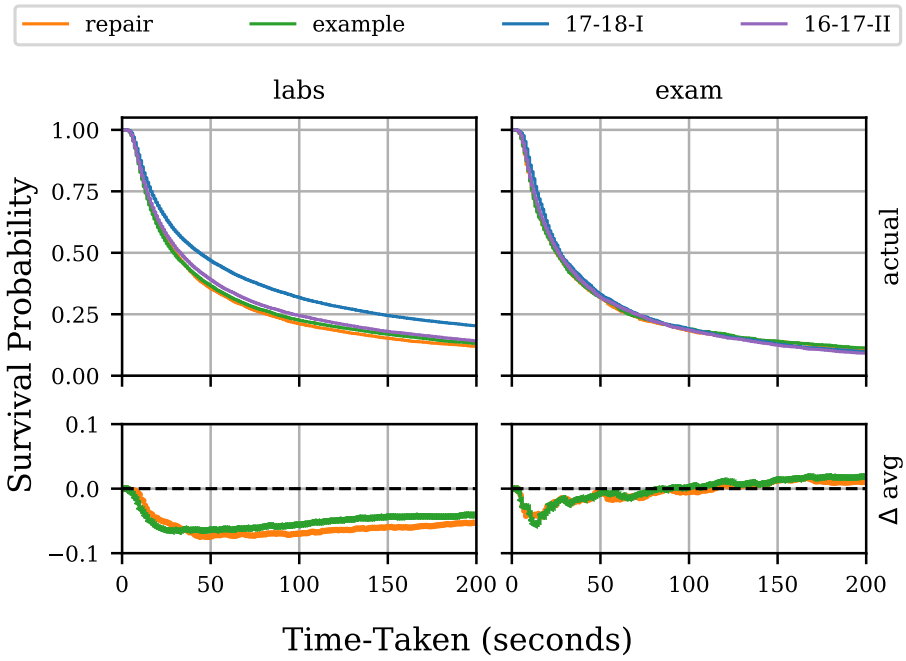
where Pr stands for probability, T is a random variable denoting the time of "death" of an event, and t is a specified time. We used the popular *Kaplan-Meier* estimator [114] from *lifelines* [115] Python package to model this survival function on our data.

For example, consider a hypothetical scenario where students make a total of 100 compilation errors during a lab. 30 of these errors are resolved at 25^{th} second, 40 of them are resolved at 75^{th} second, and the remaining 30 errors take exactly 100 seconds to resolve. Then, the survival probabilities for this example are defined

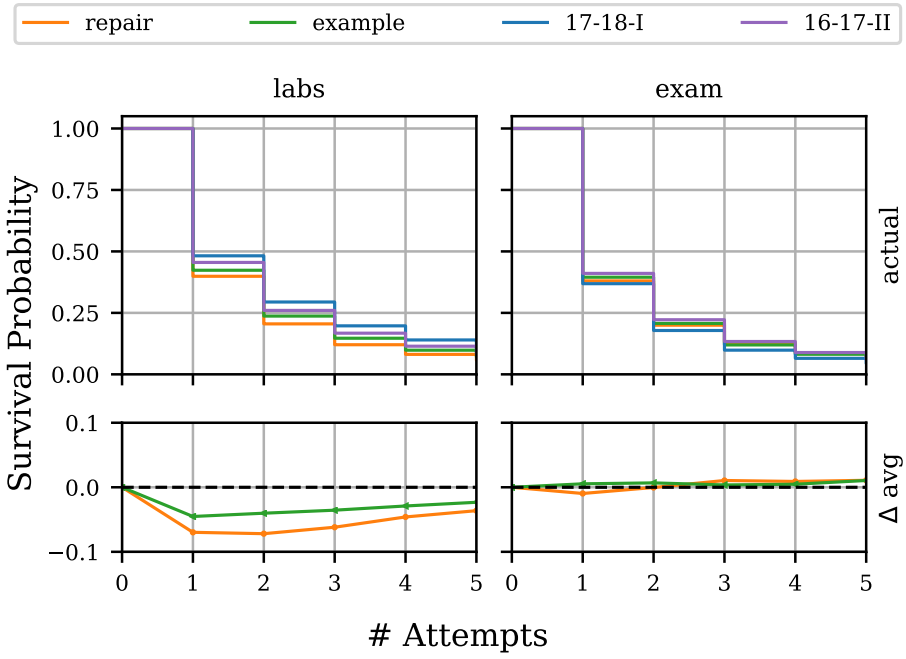
according to Table 6.4. Survival probability at time 0 second, $S(t = 0)$, is 1.00 since all errors persist when student has spent 0 seconds to fix the error. $S(t = 25)$ is 0.70 since 30 of the errors got resolved at 25^{th} second, and hence 70 errors survived after 25 seconds. Similarly, $S(t = 75)$ is 0.30 since only 30 errors survived after 75 seconds. Finally, $S(t = 100)$ is 0.00 since in our small hypothetical dataset, all observed errors got resolved within 100 seconds.

Figure 6.3a plots the survival function of time-taken data, for all errors from all students in the experiment, where survival probabilities (y-axis) are shown against time-taken (x-axis) to resolve compiler errors in different control groups. The top-left plot depicts the probability of an error surviving beyond certain time during the 7-week *labs*. The survival probability reaches 0.5 for repair, example, 17-18-I and 16-17-II groups at time 29.5, 28.7, 43.1 and 32.3 seconds respectively. In other words, students of these particular cohorts require at least this amount of time to resolve 50% of the compiler errors. The top-right plot depicts the survival probability of an error during the *exam* event, when no feedback (tool or human) was provided to students.

The bottom row graphs represents the difference of survival probabilities between repair/example (feedback) group and the average of previous semesters (baseline). From the bottom-left plot of Figure 6.3a, it can be observed that the feedback group of students have lower error survival probabilities than the baseline, averaged across all labs. In other words, as compared to the students in baseline group, the students with access to feedback tools have a lesser probability of their errors surviving after spending an equivalent amount of time to resolve them. To interpret the graphical result concretely, for example, the repair group students have a 7.5% additional chance of resolving a compiler error after having spent 50 seconds on the problem, compared to human tutored students. While we had expected *a priori* that feedback using examples would require more time on task than pointed repair suggestions, this analysis reveals that, for our domain at least, feedback using examples is equally time-effective as feedback using specific repair suggestions.



(a) Error survival probability against Time-Taken



(b) Error survival probability against #Attempts

Figure 6.3: Error survival-probability (y-axis) plots against (a) time-taken and (b) #attempts (x-axis) to resolve compiler errors, across various control groups. For both the figures, the left-side plot depicts that when feedback tools are active during *labs* event, the survival probability of time and #attempts required to resolve compiler errors is slightly lower for repair (orange-line) and example (green-line) feedback tools. When the feedback tools were withdrawn during *exam* event, no side effect was observed.

From the bottom-right plot, it can be observed that all groups have a similar survival probability (y-axis) for any given time-taken (x-axis) during the *exam* event. This is in a way reassuring, in that the identical performance of the experiment and control group students, once the feedback tools are removed, suggest that the groups are ability-matched to a considerable degree. It is also revealing, and from a pedagogical perspective perhaps disappointing, that the advantages of automated feedback appear to be primarily logistical. Students who received automated feedback for seven weeks seem to be no better at diagnosing and repairing compilation errors on their own than students with human tutoring. The advantage seen during lab events, therefore, does not appear to correspond with improved understanding of compiler errors. In other words, the performance improvement during labs appears to correspond simply with effective delivery of repair instructions and example suggestions via automated feedback, and not with conceptual improvement.

On the other hand, it is also reassuring that automated tools which provide relevant feedback can substitute human tutoring without negative effects. Especially when there is lack of expert human tutors at scale, for example in a MOOC setting with thousands of student enrollments.

6.4.4 Error Survival Probability for Attempt Count

RQ4: Do students with access to feedback tools have a lesser probability of their errors surviving, after equivalent number of attempts?

The top-left graph in Figure 6.3b depicts the survival probabilities (y-axis) against the #compilation-attempts to resolve the error (x-axis) by students from various groups, during the 7-week *lab* event. For the repair, example, 17-18-I and 16-17-II groups, the survival probability of an error after one compilation attempt is 0.4, 0.42, 0.48 and 0.45 respectively. In other words, the repair (respectively example) feedback group has 6.5% (respectively 4.5%) higher chance of resolving an error in a single attempt, as compared to the average survival probability of students in baseline group.

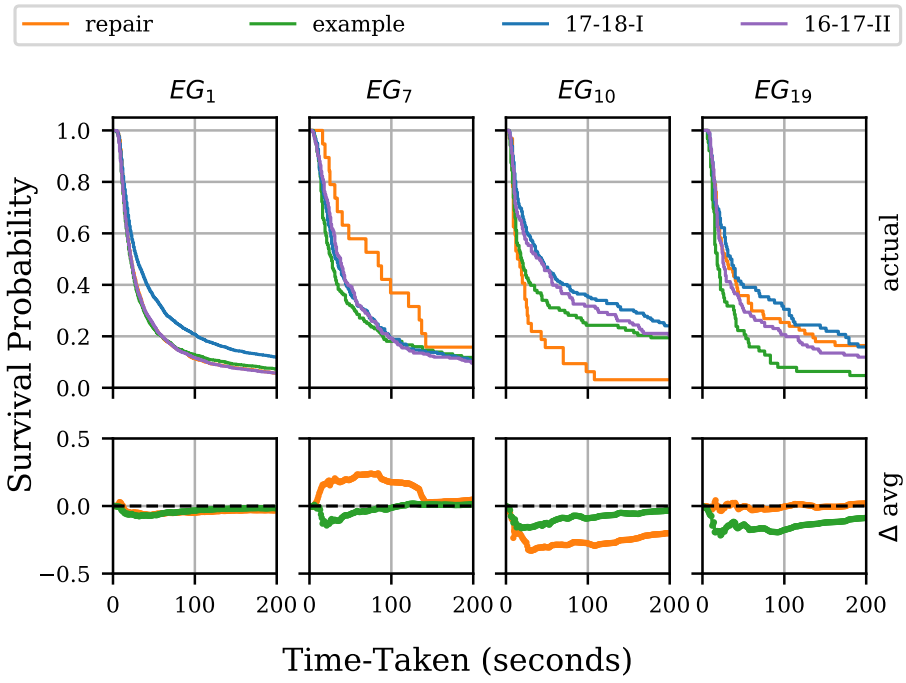


Figure 6.4: Error survival probability (y-axis) against time-taken (x-axis) plots for specific errors which display large divergence.

Thus, Figure 6.3b tells a similar story as Figure 6.3a, with number of attempts at error correction per problem as the unit of measurement, instead of time-taken. This reinforces the benefit of the tool since we reach the same conclusion from two different metrics. Also, as in the case of time-taken analysis, no significant change is observed in the survival probability of different groups during the *exam* event, when all forms of feedback are disallowed.

6.4.5 Survival probability of Specific Error Types

RQ5: What are some of the interesting errors, where one group performs better over the others?

Based on the earlier analysis, it is apparent that automated feedback tools do help students resolve compilation errors. In this section, we further analyze the specific types of errors wherein our feedback tools help the most (or least) as opposed to the baseline. Such findings could potentially assist in tailoring feedback type to error category in adaptive tutoring systems in future. The large sample size and longitudinal design of our study permits us to answer questions about individual

```
1 #include<stdio.h>
2
3 int main(){
4  printf("Hello");
5  }
6
7  return 0;
8 }
```

		Compiler Message
7	E_5	expected identifier or "("
8	E_6	extra closing brace "}"
		Repair Feedback
8		"Delete this line"
		Example Feedback
8		"Delete this line"
8		`// }`

(a) EG_7

```
1 #include<stdio.h>
2
3 int main(){
4  int x,y,;
5
6
7  return 0;
8 }
```

		Compiler Message
4	E_5	expected identifier or "("
		Repair Feedback
4		`int x,y;`
		Example Feedback
4		`int n;`
4		`float a, b;`

(b) EG_{10}

```
1 #include<stdio.h>
2
3 int main(){
4  for(int j=0, j<10; j++)
5   printf("%d", j);
6
7  return 0;
8 }
```

		Compiler Message
4	E_7	expected ";" in "for" statement
		Repair Feedback
4		`for(int j=0; j<10; j++)`
		Example Feedback
4		`for(i=0; i<n; i++)`
4		`for(i=1; i<=n; i++)`

(c) EG_{19}

Figure 6.5: Sample programs for specific errors - (a) EG_7 where feedback tools perform poorly, (b) EG_{10} where both tools have lower error survival, and (c) EG_{19} where examples seems to work better. The top repair tool fix prediction, and the top-2 example suggestions are also shown for each case.

error categories with statistical confidence.

To this end, we plot the error survival probability against time-taken for each individual error separately. The concept of error ($\boldsymbol{E}$) and error-groups ($\boldsymbol{EG}$) has been defined earlier, in Chapter 5.2.1 and 5.2.2 respectively. Fig 6.4 shows the survival probability graphs for the most frequent error group $\boldsymbol{EG}_1$, and for three error groups which demonstrate some of the largest divergence between the feedback and baseline groups - $\boldsymbol{EG}_7$, $\boldsymbol{EG}_{10}$, $\boldsymbol{EG}_{19}$.

$\boldsymbol{EG}_1$, the most frequent error group, contains the error $\boldsymbol{E}_3$: *use of undeclared identifier "ID"*. This is typically triggered when students use a variable without declaring it first, and the students are able to resolve this comfortably with or without feedback (for time-taken $t = 50$ seconds, the survival probability is 0.23 for almost all groups).

From Figure 6.4, students with repair tool feedback seem to demonstrate higher survival probability for error group $\boldsymbol{EG}_7$. This error group is typically encountered when students inadvertently add extra closing brace "}". Figure 6.5a shows a simple code that triggers these errors, along with the compiler message and feedback from tools. One of the major drawbacks of both the repair and example tool is that they focus only on the compiler reported line-number, and hence they are unable to pinpoint the correct fix: deletion of spurious "}" on line #5. Instead, both incorrectly suggest deleting the brace on line #8. This seems to affects repair group students adversely.

For the $\boldsymbol{EG}_{10}$ error group in Fig 6.4, both feedback tools demonstrate faster error resolution. At time t=50 seconds, the error survival probability is 0.15 for repair tool as compared with 0.46 average probability for baseline semesters. That is, there is a 31% lower chance for $\boldsymbol{EG}_{10}$ errors to survive for repair group, after 50 seconds of effort. Fig 6.5b shows a sample code which encounters this error. In this example, the student has a stray comma on line #4, but the compiler message seems cryptic while suggesting to either add another variable after the comma or to delete it. The top-2 examples proposed suggest both these scenarios, while the repair tool suggests

deletion of this comma.

$\boldsymbol{EG}_{19}$ is another interesting error group, where the example control group demonstrates the lowest survival probability, performing considerably better than the repair group and baseline. Fig 6.5c lists a simple example code for this error group, where the student is confused about for-loop syntax. While both the compiler message and repair feedback suggest the replacement of comma "," with a semi-colon ";" after loop initialization, example feedback tool suggests this using similar examples, along with a variety of other ways to write a for-loop specifier. The top-2 examples demonstrate the concept of 0-to-(N-1) and 1-to-N iteration. On requesting further examples, example tool suggests different ways of writing loops with empty-initialization, empty-condition or empty-increment mode, which is perhaps helping students master the for-loop syntax better, as compared to the other groups.

A general hypothesis congruent with these case samples seems to be that, feedback by example appears to work better for compiler errors by commission, viz. errors generated because of ignorance of the correct syntax; while feedback by specific repair suggestions appears to be better for compiler errors by omission, viz. situations where the student already knows the correct syntax, but has inadvertently made a mistake. Further investigation is naturally needed to concretely test this hypothesis, both in this specific context and more generally. In its general form, this hypothesis could be of considerable interest to education research in natural and artificial language learning, where the difference between learning from positive examples and from corrective feedback is extremely salient [116].

6.4.6 Potential Performance Improvement

RQ6: What is the improvement provided by feedback tools, analyzed across individual errors of varying difficulty?

As seen in Section 6.4.3, the survival analysis of our data quantifies the extent of improvement for the average compilation of the average student at 5-10% over the controlled baselines. But this average picture glosses over the heterogeneity in

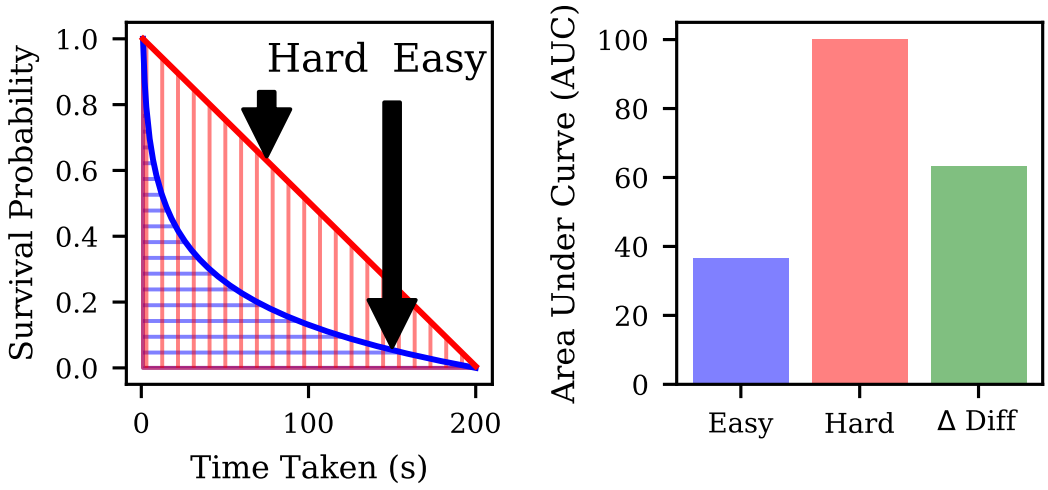


Figure 6.6: Area Under Curve (AUC) for hypothetical data. The blue curve has lesser error survival probability, and hence is easier compared to the red curve. This performance improvement is quantified as the difference between their AUCs.

problem difficulty involved. Some compilation errors are harder to find and debug than others. This matters because any form of feedback, even a cursory self-check, should prove adequate in resolving simple errors, for example a missing ';' at the end of a statement. The true test for a feedback mechanism lies in its ability to assist programmers resolve more complex errors.

In order to view the overall picture of performance improvement across the different error groups, we developed a new metric based on the Area-Under-Curve (AUC) of the error survival probability curves. The basic intuition here was that, errors which are intrinsically harder to resolve will have shallower error survival probability curves, and hence a larger area under this curve. Thus, we use survival probability AUC as a proxy for problem hardness. A hypothetical survival probability plot and its AUC calculation is shown in Fig 6.6.

Concretely, let $\boldsymbol{AUC_B}$ represent the average AUC of survival probability plots against time-taken for 16-17-II and 17-18-I baselines semesters. Let $\boldsymbol{AUC_F}$ represent the best (minimum) AUC of survival curves for repair and example tool, when tools were deployed during 17-18-II semester. Then, the area under survival curve for baseline $\boldsymbol{AUC_B}$ represents the hardness of a compiler-error.

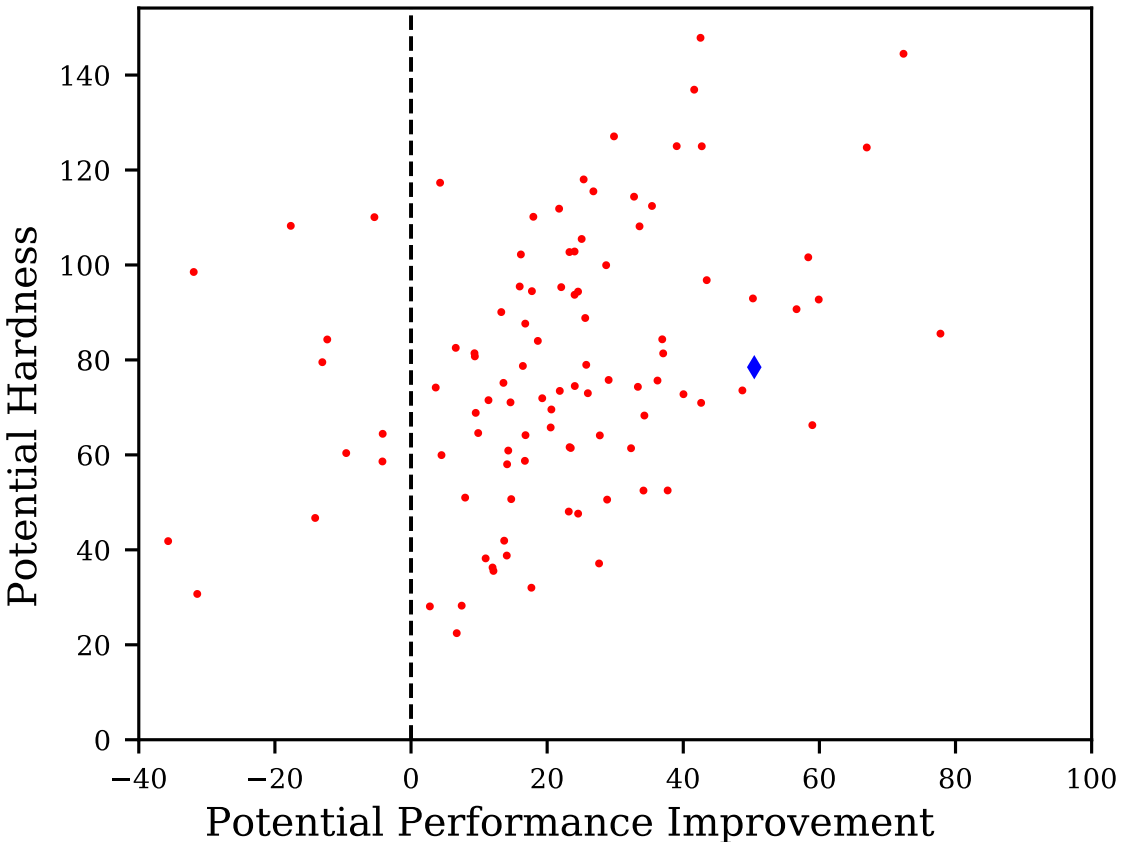


Figure 6.7: Analyzing potential performance improvement of feedback tools on 17-18-II semester students for time-taken to resolve Top-100 frequent compilation-errors, against the hardness of the error.

Also, as we have already seen above, large differences in performance would correspond with large divergences of the survival function curves between the control and test groups; with the test group survival probability curve lower than the control group curve if the feedback tools are effective. The larger the gap between these two curves, the greater the potential performance improvement offered by the tools. And hence we can quantify the difference between the baseline and feedback survival curve area, $\boldsymbol{AUC_B - AUC_F}$, to represent the potential performance improvement of students when feedback tools was deployed.

Having defined these two variables, we select the top-100 frequent compiler error-groups: $\boldsymbol{EG}_1$ to $\boldsymbol{EG}_{100}$. These error groups occur at a frequency of between 5964 to 35 times on average, in the 3 offerings of our course during 7-week *lab* event. Then, we plot survival probability curves (y-axis, 0.0 to 1.0) against time-taken (x-axis. 0s to 200s) for each of these individual compiler error group ($\boldsymbol{EG}$) separately.

The relationship between problem hardness and improvement attributable to automated feedback is plotted in Figure 6.7, with potential hardness on y-axis against potential performance improvement on x-axis. Positive x-axis indicates that the compiler-error has a lower survival chance for students with feedback-tools, as compared to their previous semester peers.

To translate between this graph and our earlier results, consider the Figure 6.4 denoting error survival curve for EG_{10}. The AUC for repair, example, 17-18-I and 16-17-II is 28.03, 61.78, 82.47 and 74.46 respectively. Then, $AUC_B = (82.47 + 74.46)/2 = 78.46$ and $AUC_F = min(28.03, 61.78) = 28.03$. Hence, EG_{10} is represented in the performance improvement Figure 6.7 by a blue-diamond point, at y-value $= AUC_B = 78.46$ and x-value $= AUC_B - AUC_F = 50.43$.

From Figure 6.7, we observe that there is performance improvement on usage of either repair or example feedback tool, across most compiler error groups. But more importantly, improvements are greater for the harder errors, suggesting that tool usage is more helpful than human tutor help for errors that are more complex.

6.5 Related Work

D'antoni et al. [5] develop and deploy an Intelligent Tutoring System which provides automated feedback for teaching automata construction to students. Three different types of feedback are provided when student designs an incorrect automata; binary (correct/incorrect), conceptual hints which propose strategies for fixing it, and counter-example hints which suggests specific string inputs that the students' automata should/shouldn't accept. They report that the students prefer and perform better on conceptual and counter-example based feedback over binary feedback (N=377, 1-week session). We observe similar effect in the programming domain for fixing compilation errors. While the compiler provides with additional details such as line-number and error message, in those instances where the message is cryptic for beginners, its utility is closer to just binary correctness.

Given the commonality of CS1 courses and growing enrollment on MOOCs, a

variety of feedback systems have been published to aid novice programmers at large scale. Some systems such as *CodeOpticon* [4] propose user-interfaces that aid tutors to manually intervene in live coding sessions, when they observe students making mistakes (N=226, 30 minute session). Other systems such as *OverCode* [117] propose visualizers to semi-automatically clusters the submissions, which the instructors can use later to provide personalized feedback per cluster, after the students have turned in their assignments (N=12, 60 minute session). Since the instructors or tutors have to manually analyze the code and provide feedback, scaling these systems requires additional effort by experts.

In the work by Yi et al. [78], state-of-the-art automated semantic repair tools designed for expert programmers are used to provide feedback on student assignments. These tools are able to generate only partial repairs on majority of the student assignments, and handing out these partial repairs doesn't seem to help them resolve logical bugs in other students' code (N=263, 15 minute session). In comparison, we provide relevant repair (or example) feedback on students' own code, using tools specialized in fixing compilation errors of introductory programming.

On the compilation error repair front, there is active research on how to better enhance error messages to suit novice programmers [94]. Meanwhile, other studies claim that additional information doesn't help students significantly [109], and that placement and structuring of the message is more crucial [75]. We believe these techniques are complementary to ours.

In terms of the underlying example feedback tools' algorithm, an approach very similar to ours was proposed under the name of *HelpMeOut* [17], where a social recommender system is proposed in which students can contribute to a growing database of error-repair example pairs, for each unique compilation error. These before-after examples, along with students' explanation, are then fetched from a database when another student encounters the same error message. The authors use self-reported survey after 39 person hours and # database hit/miss metrics to gauge performance gain.

6.6 Discussion

As we see above, existing user studies on the effectiveness of automated feedback are frequently conducted on relatively small study groups, or on sizeable groups over short period of time (order of minutes); primarily in observational settings and usually to make a case for the tool developed by the same authors conducting the user study. Thus, the generalizability of such user studies is difficult to assess.

In this chapter, we conduct a user study impervious to such challenges, so as to rigorously verify the pedagogical value of automated feedback in introductory programming. Our randomized controlled experiment involved 10,000+ person hours of user-study. Additionally, in an improvement over existing studies which frequently report isolated measures of efficacy, we report a comprehensive panel of both conventional and novel metrics to verify the efficacy of automated feedback tools.

The large size of our study sample, the RCT nature of our experimental design and the granular nature of our analysis permits a more nuanced appreciation of the value and limitations of automated feedback tools. Specifically, we find that these tools assist students in fixing compilation errors more rapidly than human tutors, but that this advantage is primarily logistical. In the absence of these tools, students learning programming assisted by these tools are no more effective at repairing compilation errors than human-tutored students. Thus, we conclude that such automated feedback tools cannot, standing alone, be considered effective from a pedagogical perspective.

At the same time, these tools have their own advantages in the actual task appointed to them - helping students fix errors in their code. As we show in our analyses, these tools prove increasingly superior to human tutors precisely in helping students solve rare, complex errors. This is a reasonable finding, since the rarity of such complex errors reduces the probability that any individual human tutor will have seen them before, whereas the computational resources of the automated tool

suffers from no such experiential limitation.

Finally, our results also open the tantalizing possibility of designing systems to rationally adapt feedback modalities to individual students' needs. While we have not established this conclusively, our results do suggest that feedback by example is a superior technique when a student's error is caused by ignorance of syntax, whereas feedback by repair suggestion is better when errors are inadvertent. Further investigating this hypothesis and finding ways to categorize programming errors into these two categories automatically should lead to the design of such adaptive interface feedback systems.

Limitations

While we have sought to minimize possible confounds in our experimental design, it is appropriate to point out some that we could not control. Most importantly, our baseline/control group are students of a different offering, from the experimental cohort. Due to administrative reasons and ethical concerns, we could not deny all forms of feedback to a new third group of students, randomly picked from the same offering. Given our time improvements observed, lack of automated feedback could put these students at relative academic disadvantage, compared to other groups. Nonetheless, we believe that the baseline comparisons are valid and provide valuable insights, owing to compulsory nature of CS-1 course at our university, having the same syllabus and teaching material across offerings. Where applicable, we take precaution in normalizing the metrics, such as with #students or #Lines-of-Code, to account for variance in #students and problem difficulty. The identical performance observed between experimental and control groups during *exam* event, once feedback tools are removed, suggests that our groups are ability-matched to a considerable degree.

The student cohort from semester 2017-2018-I seems to in general have higher error survival probabilities. This could partially be explained by the fact that they undertake this programming course in their very first semester at the university, as

opposed to students from the other control and test groups, who took this course in their second semester at the university. Students in the second semester might be expected to have more experience in the use of computer basics, as well as be better adjusted to scholastic expectations in college.

Also, our time-taken definition might not always accurately reflect the time-taken to resolve compilation error, since the student could be making logical improvements along with fixing the compilation bugs, in the same time-period. However, this is a random effect across control and baseline groups and hence should not affect our conclusions.

Chapter 7

Conclusion and Future Work

Recent advances in Intelligent Tutoring Systems (ITS) have made personalized education for the masses a reality in the near future. While significant effort is required in building ITS for each domain, we have proposed core building blocks to aid in the design and development of solution and problem generation modules. These modules are not only indispensable in MOOCs (Massive Open Online Courses) setting, but are also useful in traditional classroom setting. The problem and solution generation modules can automate the repetitive and tedious task of instructors in creating assignment problems, grading student submissions, and providing feedback on incorrect solutions. From the students' point of view, solution generation modules can provide real-time hints on incomplete solutions, and problem generation modules can create hundreds of alternate fresh practice problems.

Some of the ideas presented in this work are more broadly applicable. Breaking solution search in two parts (abstract solution and its refinement) is a general concept that might apply to many other subject domains. Our modeling of problem generation as reverse of solution generation is a general concept that might apply to problem generation in other subject domains. Our usage of offline computed data structures is a general concept that is applicable in other educational settings, enabled by our *small-sized hypothesis*; an observation that problems and solutions are typically small in educational setting. Hence all abstract solutions can be observed and recorded in an offline phase. We demonstrate the practicality of these ideas by

developing ITS for three diverse domains.

In educational setting, there can be multiple correct solutions to a given problem, with only some of them being desired by instructors and students. Acknowledging this difference, our TRACER and TEGCER tools propose and report on new relevance metric, which captures the percentage of cases where the tool's suggestion exactly matches the student's desired solution. We hope for more widespread adoption of these stringent metrics borrowed from information retrieval, as opposed to the common practice of reporting on %success, the percentage of cases where the tool generates any one of the correct solution. The primary goal in a pedagogy setting is to improve learning, and not to achieve the highest accuracy in generating ineffective correct solutions. Our proposed metrics attempt to capture this goal.

Our analysis of feedback tool efficacy through error survival curves and area under these curves, as opposed to the traditional average metrics prevalent in feedback tool literature, is broadly applicable and can be used to efficiently represent variations in experimental and control cohorts of large scale user studies.

Our large scale user study results demonstrate that automated tools can aid students in solving their problems more effectively, by means of providing timely relevant solution and example based feedback. These feedback tools were found to have no lasting impact, positive or otherwise, on their withdrawal. This suggests relevant feedback tools are equivalent to human tutoring, while being scalable in massive classroom settings like MOOCs.

Future Work

Conducting additional large scale user studies of ITS where feedback generation technology exists can help reinforce our positive results; ideally with a more tightly controlled group, from the same course offering as the experimental group. We plan to deploy our natural deduction and board-games ITS to measure learning improvement in novitiates, when provided with relevant solution and example based feedback. This would help investigate whether our conclusions are transferrable

across domains.

In this thesis, we focused on providing relevant solution and example based feedback to aid learning. Multiple forms of instruction principles have been evaluated for efficient learning in literature [6], depending on various factors including complexity and student's mastery of the educational content. Some of the prominent feedback mechanism used by instructors and teaching assistants [8], including the CS-1 introductory programming course offering at IIT-Kanpur, are (i) reveal (provide the complete solution), (ii) partially reveal (show a partial solution, which the student has to complete), (iii) reveal and justify (after revealing the solution, ask the student to justify each individual part), (iv) elicit and reveal (pose a series of questions which eventually lead to a solution), (v) obfuscate and reveal (the correct solution is mixed with incorrect solutions, such as in the form of multiple-choice options), (vi) verbose explanation, (vii) example based (where correct solution for similar problems is revealed). Since the goal of automated feedback tools is to mimic human tutor behaviour at large scale, it should be capable of automatically switching between different modes of feedback, depending on the situation, for efficient learning. In other words, instead of fixing a student to a particular form of feedback such as repair/example based, we plan to investigate the learning effect of providing dynamic type of automated feedback, based on student needs.

While our user-study focused on measuring the utility of solution and problem generation modules to aid students, in future we plan to quantify its benefit to the teachers as well. Our problem generation technology can be used for automated assignment creation, based on the topics covered and difficulty setting. Our solution generation technology can be extended to automatically grade student submissions, in terms of its distance from the closest correct solution. User studies can be conducted to capture teacher's satisfaction on various functional and non functional parameters. A recent user study by Yi et al. [78] found that providing partial repairs on incorrect student program attempts helps the teaching assistants grade faster, without affecting the final scores received by students. This is a promising results which needs to be

explored further, in a larger setting and for other domains.

Our positive results encourage the development and deployment of ITS for other educational domains. One such closely related domain is that of logical error repair for introductory programming. Introductory programming is one of the most popular courses offered by universities, and students often spend more time understanding and fixing logical errors compared to compilation errors. The user study by Yi et al. [78] claims that providing solution based feedback does not help students resolve logical errors in programs written by other students. However, new feedback tools can be designed for logical errors that generate relevant solutions and example based feedback on students' own code, in a live setting. This feedback tool can be developed utilizing the generic framework proposed in this thesis, which involves core components of abstraction, abstract solution, concretization and offline-computation. The primary challenge in such an approach would be to define the semantic (or abstract) representation and semantic equivalence of programs.

Not every educational domain can be mapped into our framework for solution and problem generation, consisting of abstraction, abstract solution, problem search and concretization phase. For example, consider the domain of trigonometry proofs. Trigonometry proofs can be considered similar to natural deduction proof style where rewrite rules are repeatedly applied on a starting premise, until a specific term is generated. Unlike natural deduction proofs, trigonometry proofs don't have any apparent abstraction to help reduce its intractable search space. Hence, new methodologies might need to be developed to enable both solution generation and similar problem generation. This is an area for further investigation.

Appendix A

A.1 Simple Traditional Board Games

In this section, we present additional results for Chapter 3. Tables A.1, A.2 and A.3 lists interesting starting states for CONNECT-3 & CONNECT-4, Bottom2 and Tic-Tac-Toe respectively, against depth-$k_2 = 2$ opponent strategy. Observe that $|W_j|$ is small fraction of $|V|$, which illustrates the significance of our symbolic methods in finding these. Also, observe that vertices labeled medium and hard are a small fraction of the sampled vertices, which illustrates the significance of our efficient iterative sampling strategy.

Game	State Space $\lvert V\rvert$	j	Win Cond	No. of States $\lvert W_j\rvert$	Sampling	$k_2 = 2$								
						$k_1 = 1$			$k_1 = 2$			$k_1 = 3$		
						E	M	H	E	M	H	E	M	H
CONNECT-3	4.1×10^4	2	RCD	110	All	*	24	5	*	3	0	*	0	0
4×4	6.5×10^4		RC	200	All	*	39	9	*	23	5	*	0	0
	7.6×10^4		RD	418	All	*	38	14	*	24	4	*	0	0
	6.5×10^4		CD	277	All	*	44	21	*	27	17	*	0	0
CONNECT-3		3	RCD	0	-									
4×4			RC	0	-									
			RD	18	All	*	0	0	*	0	0	*	0	0
			CD	0	-									
CONNECT-4	6.9×10^7	2	RCD	1.2×10^6	Random	*	183	202	*	148	115	*	0	0
5×5	8.7×10^7		RC	1.6×10^6	Random	*	70	237	*	75	181	*	0	0
	1.0×10^8		RD	1.1×10^6	Random	*	116	268	*	144	77	*	0	0
	9.5×10^7		CD	5.3×10^5	Random	*	357	133	*	200	95	*	0	0
CONNECT-4		3	RCD	2.8×10^5	Random	*	445	832	*	384	497	*	227	166
5×5			RC	7.7×10^5	Random	*	328	969	*	328	506	*	93	196
			RD	8.0×10^5	Random	*	398	1206	*	477	501	*	177	79
			CD	1.5×10^5	Random	*	146	73	*	168	44	*	87	19

Table A.1: CONNECT-3 & CONNECT-4 results against depth-2 strategy of opponent. The third column (j) denotes whether we explore from W_2 or W_3. The sixth column denotes sampling method used, to select starting vertices; if $|W_j|$ is small, "All" vertices are explored, else "Random" sampling of first 5000 vertices from W_j are explored. The E, M, and H columns report the number of easy, medium, or hard vertices found among the sampled vertices. For each $k_1 = 1, 2, 3$ the sum of E, M, and H columns is equal to the number of sampled vertices, and * denotes the number of remaining vertices; that is, $E = |W_j| - M - H$.

Game	State Space $\lvert V\rvert$	j	Win Cond	No. of States $\lvert W_j\rvert$	Sampling	$k_2 = 2$								
						$k_1 = 1$			$k_1 = 2$			$k_1 = 3$		
						E	M	H	E	M	H	E	M	H
3×3	4.1×10^3	2	RCD	20	All	*	5	0	*	1	0	*	0	0
	4.3×10^3		RC	0	-									
	4.3×10^3		RD	9	All	*	2	1	*	3	0	*	0	0
	4.3×10^3		CD	1	All	*	0	0	*	0	0	*	0	0
3×3		3	RCD	0	-									
			RC	0	-									
			RD	0	-									
			CD	0	-									
4×4	1.8×10^6	2	RCD	193	All	*	14	25	*	0	2	*	0	0
	2.4×10^6		RC	2709	All	*	572	288	*	89	245	*	0	0
	2.3×10^6		RD	2132	All	*	104	48	*	13	6	*	0	0
	2.4×10^6		CD	1469	All	*	127	49	*	18	5	*	0	0
4×4		3	RCD	0	-									
			RC	90	All	*	38	27	*	0	0	*	0	0
			RD	24	All	*	0	2	*	0	0	*	0	0
			CD	16	All	*	6	3	*	1	0	*	0	0

Table A.2: Bottom-2 results against depth-2 strategy of opponent. The third column (j) denotes whether we explore from W_2 or W_3. The E, M, and H columns report the number of easy, medium, or hard vertices found among the sampled vertices. For each $k_1 = 1, 2, 3$ the sum of E, M, and H columns is equal to the number of sampled vertices, and * denotes the number of remaining vertices; that is, $E = |W_j| - M - H$.

Game	State Space $\lvert V\rvert$	j	Win Cond	No. of States $\lvert W_j\rvert$	Sampling	$k_2 = 2$								
						$k_1 = 1$			$k_1 = 2$			$k_1 = 3$		
						E	M	H	E	M	H	E	M	H
3×3	$5.4{\times}10^3$	2	RCD	36	All	*	14	0	*	0	0	*	0	0
	$5.6{\times}10^3$		RC	0	-									
	$5.6{\times}10^3$		RD	1	All	*	0	0	*	0	0	*	0	0
	$5.6{\times}10^3$		CD	1	All	*	0	0	*	0	0	*	0	0
3×3		3	RCD	0	-									
			RC	0	-									
			RD	0	-									
			CD	0	-									
4×4	$6.0{\times}10^6$	2	RCD	128	All	*	8	0	*	0	0	*	0	0
	$7.2{\times}10^6$		RC	3272	Lowest-100	*	48	21	*	0	0	*	0	0
	$7.2{\times}10^6$		RD	4627	Lowest-100	*	3	2	*	0	0	*	0	0
	$7.2{\times}10^6$		CD	4627	Lowest-100	*	3	2	*	0	0	*	0	0
4×4		3	RCD	0	-									
			RC	0	-									
			RD	4	All	*	0	0	*	0	0	*	0	0
			CD	4	All	*	0	0	*	0	0	*	0	0

Table A.3: Tic-Tac-Toe results against depth-2 strategy of opponent. The third column (j) denotes whether we explore from W_2 or W_3. The sixth column denotes sampling method used, to select starting vertices; if $|W_j|$ is small, "All" vertices are explored, else "Lowest-100" sampling of 100 least scored vertices (according to iterative simulation score) from W_j are explored. The E, M, and H columns report the number of easy, medium, or hard vertices found among the sampled vertices. For each $k_1 = 1, 2, 3$ the sum of E, M, and H columns is equal to the number of sampled vertices, and * denotes the number of remaining vertices; that is, $E = |W_j| - M - H$.

References

[1] Sumit Gulwani. "Example-Based Learning in Computer-Aided STEM Education". In: *Commun. ACM* (Aug. 2014).

[2] Sagar Parihar et al. "Automatic Grading and Feedback Using Program Repair for Introductory Programming Courses". In: *ITiCSE '17*. 2017, pp. 92–97.

[3] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. "Automated Grading of DFA constructions". In: *IJCAI*. 2013.

[4] Philip J Guo. "Codeopticon: Real-time, one-to-many human tutoring for computer programming". In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. ACM, 2015, pp. 599–608.

[5] Loris D'antoni et al. "How can automatic feedback help students construct automata?" In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 22.2 (2015), p. 9.

[6] Kenneth R Koedinger, Albert T Corbett, and Charles Perfetti. "The Knowledge Learning Instruction framework: Bridging the science-practice chasm to enhance robust student learning". In: *Cognitive science* 36.5 (2012), pp. 757–798.

[7] Albert T Corbett and John R Anderson. "Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 2001, pp. 245–252.

[8] Barak Rosenshine. "Principles of Instruction: Research-Based Strategies That All Teachers Should Know." In: *American educator* 36.1 (2012), p. 12.

[9] Molly Q Feldman et al. "Automatic Diagnosis of Students' Misconceptions in K-8 Mathematics". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM. 2018, p. 264.

[10] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. "Synthesizing geometry constructions". In: *PLDI*. 2011.

[11] Sahil Bhatia and Rishabh Singh. "Automated correction for syntax errors in programming assignments using recurrent neural networks". In: *arXiv preprint arXiv:1603.06129* (2016).

[12] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. "Automated Feedback Generation for Introductory Programming Assignments". In: *PLDI*. 2013.

[13] Erik Andersen, Sumit Gulwani, and Zoran Popovic. "A trace-based framework for analyzing and synthesizing educational progressions". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2013, pp. 773–782.

[14] Rohit Singh, Sumit Gulwani, and Sriram Rajamani. "Automatically Generating Algebra Problems". In: *AAAI*. 2012.

[15] Christopher Alvin, Sumit Gulwani, Rupak Majumdar, and Supratik Mukhopadhyay. "Synthesis of Geometry Proof Problems". In: *AAAI*. 2014.

[16] Oleksandr Polozov et al. “Personalized mathematical word problem generation”. In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[17] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. “What Would Other Programmers Do? Suggesting Solutions to Error Messages”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2010, pp. 1019–1028.

[18] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. “Automatically Generating Problems and Solutions for Natural Deduction”. In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI ’13. Beijing, China: AAAI Press, 2013, pp. 1968–1975. ISBN: 978-1-57735-633-2. URL: http://dl.acm.org/citation.cfm?id=2540128.2540411.

[19] G. Gentzen. “Investigations into logical deduction”. In: *American philosophical quarterly* 1.4 (1964), pp. 288–306.

[20] Patrick J Hurley. *A Concise Introduction to Logic*. 11th. Wadsworth Publishing, 2011.

[21] Coursera. *Introduction to Logic*. URL: https://www.coursera.org/course/intrologic.

[22] Open Learning Initiative. *Logic & Proofs*. URL: http://oli.cmu.edu/courses/free-open/logic-proofs-course-details.

[23] Khan Academy. *Logical Reasoning*. URL: https://www.khanacademy.org/math/geometry/logical-reasoning/.

[24] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 2011.

[25] Frances Howard-Snyder, Daniel Howard-Snyder, and Ryan Wasserman. *The Power of Logic*. 4th. McGraw-Hill, 2008.

[26] Merrie Bergmann. *The Logic Book*. 5th. McGraw-Hill, 2008.

[27] Arnold vander Nat. *Simple formal logic. With common-sense symbolic techniques.* London: Routledge, 2010.

[28] Jerry Cederblom and David Paulsen. *Critical Reasoning*. 7th. Wadsworth Publishing, 2011.

[29] Roy Dyckhoff et al. *MacLogic*. URL: http://www.cs.st-andrews.ac.uk/~rd/logic/mac.

[30] Robert E. Tully Frederic D. Portoraro. *Logic with Symlog; Learning Symbolic Logic by Computer*. Prentice Hall, Englewood Cliffs USA, 1994.

[31] Richard Bornat. *Jape*. URL: http://www.cs.ox.ac.uk/people/bernard.sufrin/personal/%5Cnewline%20jape.org.

[32] Jon Barwise and John Etchemendy. *Hyperproof*. CSLI Publications, 1994.

[33] K. Broda, J. Ma, G. Sinnadurai, and A. Summers. “Pandora: A reasoning toolbox using natural deduction style”. In: *Logic Journal of IGPL* 15.4 (2007), pp. 293–304.

[34] W. Sieg. “The AProS project: Strategic thinking & computational logic”. In: *Logic Journal of IGPL* 15.4 (2007), pp. 359–368.

[35] C.K.F. Wiedijk and M.H.F. van Raamsdonk. “Teaching logic using a state-of-the-art proof assistant”. In: *PATE* (2007), p. 33.

[36] H. Van Ditmarsch. “User interfaces in natural deduction programs”. In: *User Interfaces* 98 (1998), p. 87.

[37] Coq Development Team. “The Coq proof assistant reference manual: Version 8.1”. In: (2006).

[38] Erik Andersen, Sumit Gulwani, and Zoran Popovic. “A trace-based framework for analyzing and synthesizing educational progressions”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2013, pp. 773–782.

[39] Nikolai Tillmann and Jonathan de Halleux. “Pex-White Box Test Generation for .NET”. In: *TAP*. 2008, pp. 134–153.

[40] Umair Z. Ahmed, Krishnendu Chatterjee, and Sumit Gulwani. “Automatic Generation of Alternative Starting Positions for Simple Traditional Board Games”. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI’15. Austin, Texas: AAAI Press, 2015, pp. 745–752. ISBN: 0-262-51129-0. URL: http://dl.acm.org/citation.cfm?id=2887007.2887111.

[41] G.B. Ramani and R.S. Siegler. “Promoting broad and stable improvements in low-income children’s numerical knowledge through playing number board games”. In: *Child development* 79.2 (2008), pp. 375–394.

[42] G.J. Duncan et al. “School readiness and later achievement.” In: *Developmental psychology* 43.6 (2007), p. 1428.

[43] Robyn Hromek and Sue Roffey. “Promoting Social and Emotional Learning With Games: “It’s Fun and We Learn Things””. In: *Simulation & Gaming* 40.5 (2009), pp. 626–644.

[44] John V Dempsey, Linda L Haynes, Barbara A Lucassen, and Maryann S Casey. “Forty simple computer games and what they could mean to educators”. In: *Simulation & Gaming* 33.2 (2002), pp. 157–168.

[45] S. Gottlieb. “Mental activity may help prevent dementia”. In: *BMJ* 326.7404 (2003), p. 1418.

[46] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Vol. 41. New York, USA: Harper & Row Pub. Inc., 1991.

[47] V. Allis. *A knowledge-based approach of connect-four*. Vrije Universiteit, Subfaculteit Wiskunde en Informatica, 1988.

[48] Wikipedia. *Chess960 – Wikipedia, The Free Encyclopedia*. [Online; accessed 9-September-2014]. 2014. URL: http://en.wikipedia.org/w/index.php?title=Chess960&oldid=623170305.

[49] D. Gale and F. M. Stewart. “Infinite games with perfect information”. In: *Annals of Math.* Studies No. 28 (1953), pp. 245–266.

[50] R.E. Bryant. “Graph-based algorithms for boolean function manipulation”. In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691.

[51] Fabio Somenzi. “CUDD: CU decision diagram package release”. In: *University of Colorado at Boulder* (1998).

[52] Wei Ji Ma. *Generalized Tic-Tac-Toe*. [Online; accessed 9-September-2014]. 2014.

[53] Frank Harary. *Generalized Tic-Tac-Toe*. 1977.

[54] M. Gardner. “Mathematical games in which players of ticktacktoe are taught to hunt bigger game”. In: *Scientific American* (Apr. 1979), pp. 18–26.

[55] M. Gardner. “Tic-tac-toe Games”. In: *Wheels, life, and other mathematical amusements*. Vol. 86. WH Freeman, 1983. Chap. 9.

[56] Eric W. Weisstein. *Tic-Tac-Toe*. From MathWorld–A Wolfram Web Resource; [Online; accessed 9-September-2014]. 2014. URL: http://mathworld.wolfram.com/Tic-Tac-Toe.html.

[57] Peter Kissmann and Stefan Edelkamp. “Gamer, a general game playing agent”. In: *KI-Künstliche Intelligenz* 25.1 (2011), pp. 49–52.

[58] David Williams-King, Jörg Denzinger, John Aycock, and Ben Stephenson. “The Gold Standard: Automatically Generating Puzzle Game Levels”. In: *AIIDE*. 2012.

[59] M. Hunt, C. Pong, and G. Tucker. “Difficulty-driven Sudoku puzzle generation”. In: *UMAPJournal* (2007), p. 343.

[60] Y. XUE, B. JIANG, Y.A. LI, G. YAN, and H. SUN. “Sudoku Puzzles Generating: From Easy to Evil”. In: *Mathematics in Practice and Theory* 21 (2009), p. 000.

[61] Adam M. Smith, Erik Andersen, Michael Mateas, and Zoran Popović. “A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game”. In: *FDG*. 2012.

[62] Gillian Smith, Mike Treanor, Jim Whitehead, and Michael Mateas. “Rhythm-based level generation for 2D platformers”. In: *FDG*. 2009.

[63] Jonathan Liu. “DragonBox: Algebra Beats Angry Birds”. In: *Wired* (June 2012).

[64] Alex Mabanta, Chloe Hunt, and Shannon Najmadadi. *How big is UC Berkeley's biggest class?* Accessed: 2017-12-26. URL: https://web.archive.org/web/20171226115213/http://www.dailycal.org/2013/09/03/how-big-is-uc-berkeleys-biggest-class/.

[65] Katherine Mangan. *A First for Udacity*. Accessed: 2017-12-02. URL: https://web.archive.org/web/20171202001557/http://www.chronicle.com/article/A-First-for-Udacity-Transfer/134162/.

[66] Stuart Zweben and Betsy Bizot. “2017 CRA Taulbee Survey”. In: *Computing Research News* 30.5 (2018), pp. 1–47.

[67] Tracy Camp, Stu Zweben, Ellen Walker, and Lecia Barker. “Booming Enrollments: Good Times?” In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM. 2015, pp. 80–81.

[68] *List of Automated Program Repair Publications*. 2018. URL: http://program-repair.org/bibliography.html (visited on 10/29/2018).

[69] Martin Monperrus. “Automatic Software Repair: a Bibliography”. In: *University of Lille, Tech. Rep. hal-01206501* (2015).

[70] V Javier Traver. “On Compiler Error Messages: What They Say and What They Mean”. In: *Advances in Human-Computer Interaction* 2010 (2010).

[71] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. “All syntax errors are not equal”. In: *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM. 2012, pp. 75–80.

[72] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[73] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. “DeepFix: Fixing Common C Language Errors by Deep Learning”. In: *AAAI*. 2017, pp. 1345–1351.

[74] Edward R Sykes and Franya Franek. “Presenting JECA: a Java error correcting algorithm for the Java Intelligent Tutoring System”. In: *IASTED International Conference on Advances in Computer Science and Technology, St. Thomas, Virgin Islands, USA*. 2004.

[75] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. “Compiler error messages: What can help novices?” In: *ACM SIGCSE Bulletin*. Vol. 40. 1. ACM. 2008, pp. 168–172.

[76] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. “Compilation Error Repair: For the Student Programs, from the Student Programs”. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET ’18. Gothenburg, Sweden: ACM, 2018, pp. 78–87. ISBN: 978-1-4503-5660-2. DOI: 10.1145/3183377.3183383. URL: http://doi.acm.org/10.1145/3183377.3183383.

[77] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. “sk_p: a neural program corrector for MOOCs”. In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM. 2016, pp. 39–40.

[78] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. “A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 740–751. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106262. URL: http://doi.acm.org/10.1145/3106237.3106262.

[79] Rajdeep Das, Umair Z. Ahmed, Amey Karkare, and Sumit Gulwani. “Prutor: A System for Tutoring CS1 and Collecting Student Programs for Analysis”. arXiv:1608.03828 [cs.CY]. 2016. URL: https://www.cse.iitk.ac.in/users/karkare/prutor/.

[80] Martin Šošić and Mile Šikić. “Edlib: a C/C++ library for fast, exact sequence alignment using edit distance”. In: *Bioinformatics* 33.9 (2017), pp. 1394–1395.

[81] Andrej Karpathy. “The unreasonable effectiveness of recurrent neural networks”. In: *Andrej Karpathy blog* (2015). URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[82] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. “Efficient estimation of word representations in vector space”. In: *arXiv:1301.3781* (2013).

[83] Denny Britz. “Recurrent neural networks tutorial, part 1 introduction to RNNs”. In: *Denny Britz blog* (2015). Accessed: 2017-12-23. URL: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns.

[84] Nal Kalchbrenner and Phil Blunsom. “Recurrent Continuous Translation Models”. In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Vol. 3. 39. 2013, p. 413.

[85] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. “Sequence to Sequence Learning with Neural Networks”. In: *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NIPS)*. 2014, pp. 3104–3112.

[86] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. “On the Properties of Neural Machine Translation: Encoder-Decoder Approaches”. arXiv:1409.1259 [cs.CL]. 2014.

[87] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training Recurrent Neural Networks”. In: *Proceedings of the 30th International Conference on Machine Learning (ICML)* (2013).

[88] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[89] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. arXiv:1409.0473 [cs.CL]. 2014.

[90] Christopher Olah. “Understanding LSTM Networks”. In: *Christopher Olah blog* (2015). URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[91] Leonard Blier and Charles Ollion. “Attention Mechanism”. In: *Heuritech blog* (2016). URL: https://blog.heuritech.com/2016/01/20/attention-mechanism/.

[92] Fan Long and Martin Rinard. “Automatic Patch Generation by Learning Correct Code”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2016, pp. 298–312.

[93] Umair Z. Ahmed, Renuka Sindhgatta, Nisheeth Srivastava, and Amey Karkare. “Targeted Example Generation for Compilation Errors”. In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2019. IEEE/ACM, 2019.

[94] Brett A Becker, Kyle Goslin, and Graham Glanville. “The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test”. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM. 2018, pp. 640–645.

[95] Alexander Renkl. "Toward an instructionally oriented theory of example-based learning". In: *Cognitive science* 38.1 (2014), pp. 1–37.

[96] François Chollet et al. *Keras*. 2015. URL: https://keras.io.

[97] Anand Rajaraman and Jeffrey David Ullman. "Data Mining". In: *Mining of Massive Datasets*. Cambridge University Press, 2011, pp. 1–17. DOI: 10.1017/CBO9781139058452.002.

[98] Jason Brownlee. "How to Prepare Text Data for Deep Learning with Keras". In: *Jason Brownlee blog* (2017). URL: https://machinelearningmastery.com/prepare-text-data-deep-learning-keras/.

[99] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[100] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. "A survey of machine learning for big code and naturalness". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 81.

[101] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[102] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[103] Andrej Karpathy. *Convolutional Neural Networks for Visual Recognition*. Accessed: 2019-01-17. URL: http://cs231n.github.io/neural-networks-1/.

[104] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[105] Min-Ling Zhang and Kun Zhang. "Multi-label learning by exploiting label dependency". In: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2010, pp. 999–1008.

[106] Yuhong Guo and Suicheng Gu. "Multi-label classification using conditional dependency networks". In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011.

[107] Andreas Stefik and Susanna Siebert. "An empirical investigation into programming language syntax". In: *ACM Transactions on Computing Education (TOCE)* 13.4 (2013), p. 19.

[108] Andrew Luxton-Reilly et al. "Introductory programming: a systematic literature review". In: *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM. 2018, pp. 55–106.

[109] Raymond S Pettit, John Homer, and Roger Gee. "Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive." In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM. 2017, pp. 465–470.

[110] Brett A Becker. "An effective approach to enhancing compiler error messages". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM. 2016, pp. 126–131.

[111] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. "Enhancing syntax error messages appears ineffectual". In: *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM. 2014, pp. 273–278.

[112] Thomas Flowers, Curtis A Carver, and James Jackson. "Empowering students and building confidence in novice programmers through Gauntlet". In: (2004).

[113] Rupert G Miller Jr. *Survival analysis*. Vol. 66. John Wiley & Sons, 2011.

[114] Edward L Kaplan and Paul Meier. "Nonparametric estimation from incomplete observations". In: *Journal of the American statistical association* 53.282 (1958), pp. 457–481.

[115] Cameron Davidson-Pilon et al. *CamDavidsonPilon/lifelines: v0.14.6*. July 2018. DOI: 10.5281/zenodo.1303381. URL: https://doi.org/10.5281/zenodo.1303381.

[116] Cristina Sanz and Kara Morgan-Short. "Positive evidence versus explicit rule presentation and explicit negative feedback: A computer-assisted study". In: *Language Learning* 54.1 (2004), pp. 35–78.

[117] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. "OverCode: Visualizing variation in student solutions to programming problems at scale". In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 22.2 (2015), p. 7.