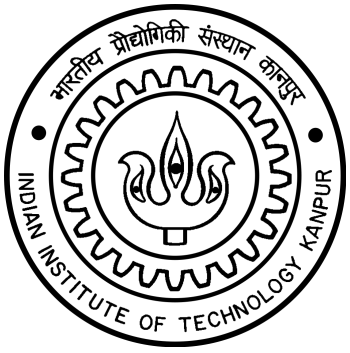
Automated Feedback and Grading for Programs in Introductory Programming Courses

A thesis submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Ziyaan Dadachanji
Roll Number: **14111048**



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

May, 2016

POST GRADUATE OFFICE
THESIS SUBMITTED
ON 31.05.2016
I. I. T., KANPUR - 16

CERTIFICATE

It is certified that the work contained in the thesis titled **Automated Feedback and Grading for Programs in Introductory Programming Courses**, by **Ziyaan Dadachanji** (*Roll Number:* **14111048**), has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Amey Karkare

Dr. Amey Karkare
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Arnab Bhattacharya

Dr. Arnab Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

May, 2016

ABSTRACT

Teaching introductory programming has been a challenge at the undergraduate level. The introductory programming course conducted at IIT Kanpur uses a software system called Prutor. It is a system developed to facilitate effective problem solving and provide individual feedback to students. During the course we noticed that students struggle with simple compile-time errors and hence are not able to solve the problem. Good feedback is essential to help speed up the learning process. Also, grading is done with the help of teaching assistants (TAs). This is a time consuming task and prone to inconsistencies. It is important to maintain the quality of grading as it serves as a measure to track the performance of a student.

In this thesis we present a system that will automatically provide feedback to students and automatically grade their solutions. Our feedback is in the form of simple suggestions with examples to explain the compile-time errors made by students. We studied the grading policies commonly set by instructors and decided the features to be used by our automated grading system based on these findings. We show that the automated feedback system helps students more than the feedback given by the compiler. Additionally, we show that the automated grading system works comparable to human TAs. It can also be used to detect inconsistencies in TA grading.

Dedicated to my family

Acknowledgements

I would like to sincerely thank my thesis supervisor **Dr. Amey Karkare** for his continuous help and support in the completion of this thesis. He always provided prompt suggestions and guided us towards our goal. I would like to extend my gratitude to my thesis co-supervisor **Dr. Arnab Bhattacharya** for his help and support during the work on this thesis. This work would not have been completed without their guidance.

I would like to thank a few people who have been very helpful during the course of this thesis.

Praveen Kumar Singh with whom I worked throughout this thesis. Without him this thesis would not have been possible.

Rajdeep Das for helping me with any issues with Prutor and guiding me in the integration of our system with it.

Umair Z Ahmed for his help in collecting the data that was useful to us in our experiments.

Sagar Parihar whose work was helpful to understand the domain of automated grading.

I would like to thank my family for their motivation and support. Last but definitely not least, I would like to thank my friends especially **Milan** who provided valuable suggestions in many situations, **Vikrant**, **Rishabh**, **Vivek Anand**, **Awanish** and many others for their valuable input.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Introductory Programming is a compulsory course conducted for all branches of engineering in many colleges and universities. Teaching introductory programming has been a challenge at the undergraduate level. Some colleges use Intelligent Tutoring Systems for conducting such courses and develop tools to make the learning experience more user friendly. It is necessary that the students are provided with good feedback during such courses. The basic form of feedback that students receive is through compiler messages. This feedback is not very detailed and does not always help in correcting the student's mistakes. Grading is another important form of feedback provided to students indicating their progress during the course. Immediate grading can be useful to students.

1.1 Idea and Motivation

1.1.1 Automated Feedback

It was observed that at the beginning of an introductory programming course students find it difficult to understand compiler error messages and hence struggle to fix even simple syntax errors such as a missing semicolon or a missing parenthesis. It takes time for students to become familiar with the syntax of various statements. It takes some students longer than others. As a result a lot of their time is spent in

correcting these errors and they are not able to complete their assignments in time. Some of them try different and random possibilities before arriving at the correct statement while some remain stuck and need to seek the help of TAs.

Thus, many students do not learn from their mistakes as they blindly follow what the TA says without trying to understand the compile-time error. Most of the compiler error messages are unclear to them and there should be a simpler form of feedback. To this end we have created an Automated Feedback System to give suggestions for errors in simple statements along with valid and invalid examples to help the students better understand compile-time errors. The valid examples show students the correct way to solve the error while the invalid examples show them the other errors that may occur related to that error. This helps the student learn and reduces the chance of similar errors in the future.

1.1.2 Automated Grading

Grading is the basic level of feedback to the students in any programming course. It serves as a measure to track the performance of the students. It is a powerful motivational tool that drives the students. The maintenance of quality in grading of introductory programming course assignments requires a significant amount of time and effort. To this end most universities hire human teaching assistants (TAs). This policy has two major drawbacks.

1) Turnaround time for grading is very large. A large number of students are enrolled in introductory programming courses making prompt action impossible. For example, at IIT Kanpur, where all first-year students (irrespective of the branches of study) undertake the introductory programming course, the TAs are unable to provide immediate grading for the submissions.

2) The TAs are post-graduate students who have graduated from different universities. They may have taken programming courses in the pursuit of their degrees. It stands to reason that there is a large variation in their knowledge and experience

in programming. In spite of specific and detailed instructions and grading policies being provided to the TAs, it is not possible to avoid human bias thus causing variability in grading. Consistency in grading similar assignments is compromised.

An efficient and well developed autograding system can either replace TAs grading process or complement it by pointing out any inconsistencies and biases.

1.2 Integration with an Online Tutor

Prutor (PRogramming tUTOR) [Das15] is a software system for teaching and conducting introductory programming courses. It was designed and developed by Rajdeep Das under the guidance of Dr. Amey Karkare at the department of Computer Science and Engineering at IIT Kanpur. Prutor was developed to facilitate effective problem solving and to provide feedback to students individually. It is a valuable tool for educationists who can use this system to collect data and help them to understand various patterns in the learning process of students.

ESc101: Fundamentals of Computing is an introductory programming course conducted at IIT Kanpur for all branches of engineering. It uses Prutor to teach the students introductory programming. All the assignments and lab exams are conducted on Prutor. It logs a large amount of data related to students' assignments, exams, marks awarded, TA grading tasks, compilation errors, etc. This has given us a lot of real world data to work with. Prutor also provides us with a rich platform to test our system.

1.3 Thesis Overview

The outline of the thesis is as follows:

- Chapter 2: **Related Work**

  This chapter describes the earlier work that has been done related to our work.

- Chapter 3: **Automated Feedback**

  This chapter describes our feedback system. It consists of our ideas and motivation along with the methods used to create the system.

- Chapter 4: **Automated Grading**

  This chapter describes the various features that have been used in our auto-grading tool along with the reasons of inclusion and methods of calculation.

- Chapter 5: **System Overview**

  This chapter describes how our tool has been integrated with Prutor [Das15] and its overall functioning.

- Chapter 6: **Experiments and Results**

  This chapter describes the experiments we have performed to evaluate our system and the results we obtained.

- Chapter 7: **Conclusion and Future Scope**

  This chapter concludes our thesis and describes a few points that have strong potential for future work.

Chapter 2

Background and Related Work

This chapter consists of a brief description of the earlier work that has been done related to our work. We have done our research on the related work before beginning with implementing our system.

A Platform for Data Analysis and Tutoring For Introductory Programming [Das15] created by Rajdeep Das at IIT Kanpur, has provided us with a platform which allows us to integrate feedback tools to run in introductory programming courses. This intelligent tutoring system is called Prutor (PRogramming tUTOR) [Das15] .

Prutor was developed to facilitate effective problem solving and to provide feedback to students individually. It is very useful to students, instructors and developers. Without this system, it would have been very difficult to get such rich real world data on which we could test our system.

Automated Grading Tool for Introductory Programming [Par15] created by Sagar Parihar used different features to grade students submissions. The features used are number of test cases passed, time taken by the student and the fraction of successful compilations made by the student. This tool gave a real number from 0 to 1 which when multiplied by the maximum marks of the problem gave the score that should be awarded to students.

Brenda et al. [CKLO03] studied the implementation of an automated grading system called Online Judge. It is a simple application that evaluates the student's submissions on a set of test cases taking into consideration memory and time limits. A submission is correct if its output matches the pre-specified answers. The efficiency of a submission is determined by the ability of the program to produce its output within the memory and time limits.

Impact of auto-grading on an introductory computing course [SBL+13] presented a web-based framework called Bottlenose. On receiving students submissions, it provides immediate feedback. Students can use this feedback to improve their submission and re-submit their code unlimited times before the deadline. They observed that the number of submissions made by students increased with this system which shows that students used this feedback to improve their submissions.

Sumit Gulwani, Ivan Radicek and Florian Zuleger authored Feedback Generation for Performance Problems in Introductory Programming Assignments [GRZ14] where they proposed and implemented a system that would automatically provide a list of repairs to students submissions. Different algorithmic specifications for each problem are required at the beginning and using these it tries to compare the students submission to see which specification is similar and accordingly gives feedback. It is important that different methods of solutions require different feedback.

Rishabh, Sumit and Armando presented a system that automatically provided feedback for programs in introductory programming courses [SGSL13]. This system required a reference implementation of the problem and an error model which consisted of potential repairs to common errors. Using this, it would provide feedback to the students in the form of minimal repairs to their solutions that are incorrect.

Compiler Error Messages: What Can Help Novices? [NPM08] authored by Nienaltowski, Pedroni and Meyer consists of a study conducted involving two groups of students. They used three different styles of messages and found that messages

which are more detailed would not necessarily help in understanding errors but their location and structure was more important.

On Compiler Error Messages: What They Say and What They Mean [Tra10] offered an analysis of the problem of cryptic compiler error messages and how this makes it difficult for beginners to learn programming quickly.

Sumit Gulwani, Ivan Radicek and Florian Zuleger [GRZ16] presented a method for generating feedback automatically. They first clustered the correct submissions of students and used these clusters to generate specifications. When presented with an incorrect submission, they automatically produced the minimal repair for the submission by running a repair procedure against all specifications.

Sahil and Rishabh [BS16] presented a method that would generate repair feedback for compile-time errors automatically. They used Recurrent Neural Networks to model token sequences that are syntactically valid. They used this model to predict token sequences that can repair the compile-time error. This sequence would then replace the token sequence or be inserted at the position of the error.

After this research we designed and developed our Automated Feedback and Grading System using different methods.

Chapter 3

Automated Feedback

3.1 What is Automated Feedback

Feedback is an essential part of teaching programming to students. A system which provides good feedback is definitely better than a system which does not. If a system gives good feedback it makes it easier for the student to understand his mistakes. The basic form of feedback in an introductory programming course is through compiler messages. The compiler will give a list of errors and warnings if there are any errors or warnings in the code. This can be used by the student to debug his code and arrive at the correct solution.

In Prutor [Das15], any standard C compiler can be used. For all our thesis results we have worked with the data received during the ESc101 course conducted in the 2015-16 odd semester at IIT Kanpur. During this time, the gcc [GCC] compiler was used to compile students' programs. The messages given by this compiler are at times difficult to understand for a student who has just begun programming and sometimes even for a student who knows basic programming.

In an introductory programming course, students are bound to make several compile-time errors in the beginning of the course. The compiler messages given by gcc did not seem sufficient and most students struggled to understand messages

for even basic errors like missing semicolon or missing & in scanf statement, etc. Thus, we found the need to improve the feedback provided to students during the course. Our goal is to help them understand their mistakes quickly and move on to solve the problem. The students find it very difficult to proceed with their program when faced with several compiler error messages. Some of them lose hope of solving the problem while some waste a lot of time in correcting these errors and still have trouble correcting them. Due to this, they request TAs for help and this hampers their learning as they become dependent on the TA to correct their compile-time errors. Our goal is to provide the student with suggestions / feedback in such a way that they will need little or no help from TAs in correcting compile-time errors. They can use this feedback to understand their mistakes and learn from them so that in the future they do not repeat the same mistakes.

This feedback would be shown to the students immediately when they compile their code. This would help them in debugging compile-time errors quickly and continue to solve the logic of the problem.

3.2 Improved Feedback

Our main goal is to improve the feedback given to students. We want them to spend less time correcting compile-time errors and spend that time in solving the logic to the problem. We have tried a few methods before arriving at our final Automated Feedback system.

1) **Changed Gcc Compiler to Clang** [Cla]

   The error messages given by clang were far better in many cases than gcc. gcc sometimes points to the line after the error and says that there is a mistake before this but clang points to the line on which the error is present so it makes it much clearer to students.

Example 3.1 Consider the following code with a missing semicolon on line 6.

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

When the above code is compiled with gcc, we get the following error message:

```
name.c:7:2: error: expected ‘;’ before ‘return’
   return 0;
  ^
```

It can be seen that the error is shown on line 7 and the error message confuses students who are just learning to program as they are not sure if they should put the semicolon before return or at the end of the line above return.
When the above code is compiled with clang, we get the following error message:

```
name.c:6:16: error: expected ‘;’ after expression
   scanf("%d",&a)
                 ^
                 ;
```

It clearly mentions that a semicolon is required after the statement on line 6.

This is a basic example to show the difference between the effectiveness of gcc messages vs clang messages. There are several other cases where clang messages are much better and simpler to understand than gcc messages.

2) **Rewritten Clang Error Messages**

Even though clang error messages are better than gcc error messages, it still is not enough of an improvement to the feedback given to students. Our goal is to help the students compile their programs easily.

Example 3.2 Consider the following code with a missing & in the scanf statement on line 6.

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

The default clang error message is as follows:

```
name.c:6:13: warning: format specifies type ‘int *’ but the
argument has type ‘int’ [-Wformat]
   scanf("%d",a);
          ~~  ^
```

This compiler error wants to tell the student that he has missed an & before the variable **a** on line 6. We observe that the error shown can be very difficult for a student to understand. Thus the student may waste a lot of time trying to understand this error before requesting a TA for help. We want to make the error message very simple for the student to understand and correct quickly.
Thus we have rewritten the message as follows:

```
You might be using a wrong format specifier (%d, %f, etc) OR have
missed an & OR put an & where not required.
   scanf("%d",a);
          ~~  ^
```

There are several other compile-time error messages that we have rewritten to help students correct their mistakes quickly.

3) **Suggestions**

From our experience, rewriting error messages is still not enough to help the students understand their mistakes. There are some cases in which it is easy for students to understand the rewritten messages but some still seem difficult to understand. Such cases are to be eliminated. Our aim is to enable the student to see the feedback from the system and immediately understand what the mistake is. We have provided examples as well in the feedback so that the student can understand similar situations where the error may occur and may understand before hand when faced with the similar error in the future.

This is the basis of our final Automated Feedback System. We give simple suggestions to the student when they make compile-time errors. Along with the suggestions, we give a few examples as well so that the student can understand the compile-time error better. This type of feedback is very useful to the students as it enables them to understand certain compile-time errors and learn from them so that in the future they will not repeat such mistakes. This feedback system has been integrated with Prutor [Das15] and students see the suggestions we have provided when they compile their program, in addition to the compile-time errors.

Example 3.3 If we look at the rewritten message in Example 3.2, it is still a little confusing for a student who has just begun programming. Our Automated Feedback System will give the following feedback for the same error:

```
Line 6: You have not put an & before ‘a’ in the scanf statement on
this line. Whenever you use scanf to input a value, you must put
an & before the variable (except for pointers and strings).
Examples:
Valid statements:
1. scanf("%d",&a);
2. char str[20];
   scanf("%s",str);
```

```
Invalid statements:
1. scanf("%d",a);
2. char str[20];
   scanf("%s",&str);
```

This suggestion is very simple for students to understand as it clearly says that the student has missed an & before his input variable 'a'. There are also a few examples of input statements with scanf given in the suggestion. This type of feedback proves to be much better than all of the above methods and thus has been used in our Automated Feedback System.

3.3 Automated Feedback System

3.3.1 Idea of Feedback System

Compile-time errors are the most basic form of errors made by every student in an introductory programming course. Once the student is confident of the syntax of statements in C, it is easier for him to code. However, it is very difficult for the students to understand the feedback given by the compiler due to which they get stuck and are not able to solve the problem. Without correcting compile-time errors a student cannot possibly complete his program. Once the compile-time errors are corrected, a student can comfortably think of the logic of the program. In case of compile-time errors we want to make it as easy as possible for students to correct them.

We want to make the students understand the errors quicker and learn from them so that they do not repeat them in the future. If the student sees simple feedback which they are able to understand it provides a motivation factor to learn. The simpler the feedback, the easier it is for students to understand. Our idea behind

the feedback system is that we provide easy to understand messages with examples so that the student reading them can proceed easily to correct his mistakes.

3.3.2 Selecting Errors

In the introductory programming course ESc101 there are a large amount of compilation errors made by students throughout the duration of the course. During the ESc101 course conducted at IIT Kanpur in the odd semester of 2015-16, there was a lot of data collected by Prutor [Das15]. A part of this data is summarized in Table 3.1 below:

Data	**Quantity**
Students	421
TAs	37
Events (Lab + Exam)	14
Problems	106
Assignments	12,374
Compilation Errors	5,14,585

Table 3.1: Data Collected

There were two types of events during the course including twelve labs and two exams. There were 106 different problem statements during these events. One assignment corresponds to one problem done by one student. There were 12,374 assignments which means that these many problems were solved by all the students during the lab and exam events. During these events, the number of compilation errors made by the students was 5,14,585. This shows us that there were a tremendous amount of errors made and we want to reduce these errors.

Our goal is to help the students learn from their mistakes so that they do not repeat them. It is not an easy task to give suggestions for every single compile-time error. We decided that we would pick the most frequently made errors and give suggestions for them. We combined the common errors among the 5,14,585 compile-time errors and sorted them in decreasing order of frequency. We started to give suggestions for errors which are most frequent. We finally gave suggestions for the top 55 most frequent errors.

The errors 1 to 55 are listed below along with their frequency.

Sr.No.	**Error Type**	**Count**
1	Undeclared variable	79,347
2	Unused variable	47,467
3	Missing & in scanf	38,086
4	Wrong format specifier in scanf	
5	Extra & in printf	
6	Wrong format specifier in printf	
7	Uninitialized variable OR Use of scanf after using the variable	35,900
8	Return statement missing	19,899
9	Missing semicolon	18,942
10	Expression result unused	13,971
11	Expected braces to match this (printf statement)	10,124
12	Expected braces to match this (if or for,etc)	
13	Missing header file	9,289
14	Integer to pointer conversion	9,243
15	Return datatype missing from function definition	8,340
16	Char type variable with multi character value	7,943
17	Expected }	7,432

Sr.No.	Error Type	Count
18	Implicit declaration of function ‘X’ is invalid	7,352
19	Use of ‘=’ instead of ‘==’	6,977
20	Pointer to integer conversion	5,465
21	Extra brace / parenthesis	5,364
22	More ‘%’ conversions than data arguments	4,341
23	Declared as variable, not array / pointer	4,181
24	Invalid format specifier in scanf	3,879
25	Expected semicolon in for statement	3,863
26	Array index out of bounds	3,774
27	Empty body of if, for, while, switch	3,632
28	More data arguments than ‘%’	3,535
29	Variable ‘X’ is used uninitialized whenever ‘Y’ loop exits	3,515
30	Incorrect spelling of declared variable	3,408
31	Invalid operands to binary expression	3,320
32	Use of . operator in place of → operator in reference type pointer variables	2,878
33	Redefinition of variable	2,430
34	Function definition is not allowed here	2,426
35	Use of ‘==’ instead of ‘=’	2,416
36	Expression is not assignable	2,248
37	Size missing in declaring array	2,188
38	Too few arguments to function call	1,932
39	Missing terminating character (’ or ”)	1,930
40	Void function ‘X’ should not return a value	1879

Sr.No.	Error Type	Count
41	Implicit conversion from ‘X’ to ‘Y’ changes value from A to B	1,731
42	Array subscript is not an integer	1,680
43	No member named ‘X’ in ‘Y’	1,580
44	Expected ‘(’ after if/for/while/switch	1,354
45	‘&&’ within ‘||’	1,089
46	Cannot dereference non pointer variable	1,064
47	Use of → operator in place of . operator in non-pointer variables	1,042
48	Array type ‘X’ is not assignable	1,005
49	Use of logical ‘||’ with constant operand	953
50	Invalid suffix ‘X’ on integer constant OR Invalid digit ‘X’ in decimal constant	910
51	Assigning to ‘X’ from incompatible type ‘Y’	871
52	Multiple unsequenced modifications to ‘X’	818
53	Too many arguments to function call	574
54	Header file not found	553
55	Non-void function ‘X’ should return a value	341
56	Others	< 300

Table 3.2: List of Selected Errors

X,Y,A,B are the actual variable name, function name, etc in the error statements. The total count of errors for which we are giving suggestions is 4,04,848. This means that we are giving suggestions for 78.67% of the total compilation errors made by students.

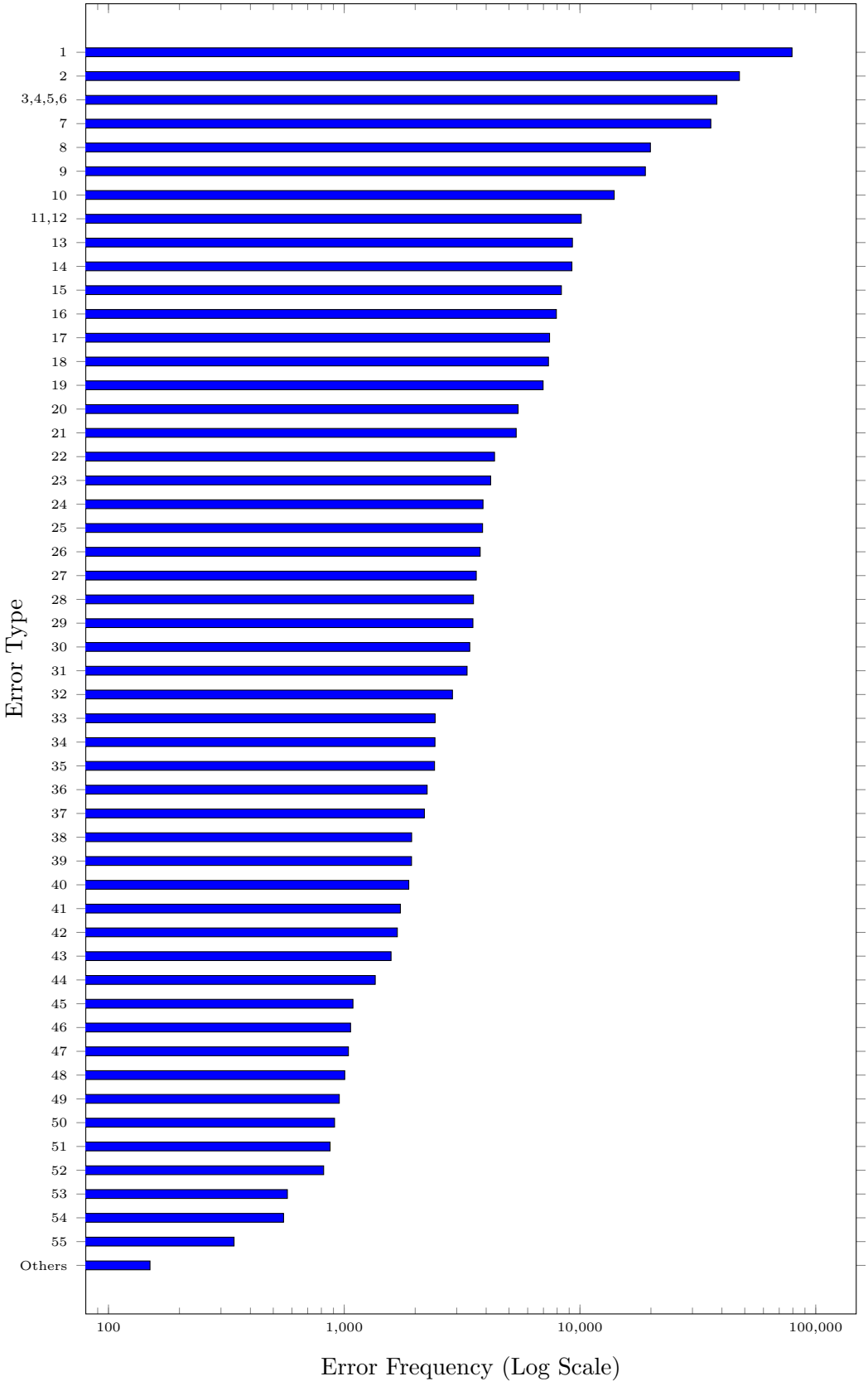


Figure 3.1: Frequency of Different Types of Errors

The error numbers mentioned in Figure 3.1 correspond to the errors mentioned in Table 3.2. We can see that these errors are more frequently occurring than the others. Thus, we have chosen to give suggestions for these errors. It makes sense to choose these errors as students can learn from these most common mistakes. Our suggestions are simple statements which are easy to understand.

3.3.3 Feedback

Our tool gives simple suggestions for the most common compile-time errors. Our suggestions are of the following format: First we rewrite the compiler error messages into simple statements. Along with this we give examples of two types.

- Valid: to explain the error so the student can understand how to correct it.
- Invalid: to show what must not be done in cases similar to the error.

Giving both valid and invalid examples makes it very easy for the student to understand the error and correct it quickly. The idea of including invalid examples is that the student can understand what are the other possible errors that can be made related to the current error. Our feedback to the students is much simpler than the default feedback from the compiler. It is displayed in Prutor [Das15] beside the code in a very ideal position so that students can immediately see it without having to navigate anywhere. They find it much easier to correct their compile-time errors looking at the detailed suggestions provided by our Automated Feedback System as compared to the compiler feedback.

Example 3.4 A few samples of our suggestions for the most frequent errors messages are presented in Table 3.3.

Error Type: **Undeclared Variable**
Compiler Error Message: use of undeclared identifier ‘a’ printf(“%d”,a); ∧

Suggestion:
You have not declared the variable ‘a’ that you have used on this line. Please declare it before using it. All variables must be declared before you can use them.
Eg:
Valid:
int a=3,b=6,count=0;
scanf(“%d”,&count);
count=a+b;
printf(“%d”,count);
 	
Invalid:
int a=3,b=6;
scanf(“%d”,&count);
count=a+b;
printf(“%d”,count);
Here variable ‘count’ has not been declared. This is not allowed.

Error Type: **Missing & in scanf**
Compiler Error Message: format specifies type ‘int *’ but the argument has type ‘int’ [-Wformat] scanf(“%d”,b); ~~ ∧
Suggestion: You have not put an & before ‘b’ in the scanf statement on this line. Whenever you use scanf to input a value, you must put an & before the variable (except for pointers and strings). Eg: Valid statements: 1. scanf(“%d”,&a); 2. char str[20]; scanf(“%s”,str); Invalid statements: 1. scanf(“%d”,a); 2. char str[20]; scanf(“%s”,&str);

Error Type: **Wrong format specifier in printf**
Compiler Error Message: format specifies type ‘double’ but the argument has type ‘int’ [-Wformat] printf(“%f”,b); ~~ ∧ %d
Suggestion: You have used an incorrect format specifier while printing the value of ‘b’. Integer variables require %d Float variables require %f Char variables require %c, etc. Eg: Valid statements: 1. int a; printf(“%d”,a); 2. float b; printf(“%f”,b); Invalid statements: 1. int a; printf(“%f”,a); 2. float b; printf(“%c”,b);

Error Type: **Missing semicolon**
Compiler Error Message: expected ‘;’ after expression printf(“%d”,a) ∧
Suggestion: You are missing a semicolon(;) at the end of the statement on this line. All statements in C must end with a semicolon. Eg: Valid statements: 1. int a; 2. scanf(“%d”,&a); Invalid statements: 1. int a 2. scanf(“%d”,&a)

Error Type: **Uninitialized variable / Use of scanf after the variable**
Compiler Error Message: variable ‘b’ is uninitialized when used here [-Wuninitialized] printf(“%d”,b); ∧

Suggestion:
You have not initialized the variable ‘b’ before you are using it on this line. The variable ‘b’ has been declared on Line 3. Please go and initialize the variable there with something like ‘b=0;’. All variables must be initialized before they can be used.
Eg:
Valid:
int n=0,sum;
sum=n+3;
 	
Invalid:
int n,sum;
sum=n+3;
 	
It is possible that you may have taken input from the user using scanf for this variable after using it by mistake.
For example you are calculating the area of a rectangle as area=length * breadth and after this you are asking the user to input the values of length and breath.
int l,b,area;
area=l*b;
scanf(“%d%d”,&l,&b);
 	
The correct code would be:
int l,b,area;
scanf(“%d%d”,&l,&b);
area=l*b;

Table 3.3: Some Frequent Errors

Chapter 4

Grading Model

Grading is the key aspect of evaluating a student's performance in any course. In an introductory programming course like ESc101, there are around 40 TAs who are given the job of manually grading students' weekly programming assignments. This may bring about some inconsistencies in the grading as some TAs may deduct one mark for a certain mistake while some may deduct two to three marks for the same mistake.

Further, there may be some biased grading as well. Biased grading means that some TA may tend to give a particular student extra marks in spite of his code being incorrect or lesser marks in spite of his code being correct. To overcome this, we have created an auto grading tool that will evaluate the students assignments according to certain common features that follow the instructor's grading policies. According to the grading policy of the instructor, we have chosen features that will enable our tool to perform like an actual human TA.
The features are enlisted and explained below.

1) **Number of test cases passed**

   During the introductory programming course, we noticed that the instructor gives a lot of weightage to passing of test cases while grading a program. Thus, a test case is one of the most important features of any programming grading task. If a program passes few or all test cases it means that the student is doing

something correct while a program which does not pass any test cases means that the student is making a mistake. Passing test cases is the most basic way of deciding if a particular problem is correct or not.

In Prutor [Das15] there are two types of test cases in every programming assignment namely Hidden and Visible. The instructor normally gives extra importance to passing of hidden test cases as compared to passing of visible test cases. We have followed this grading policy and our tool has different weightage for hidden test cases and visible test cases.

i) **Visible Test Cases**

These normally cover the basic and simple cases of any program. We have given these slightly less weightage than the hidden test cases.

ii) **Hidden Test Cases**

These are mostly the boundary cases of a program. They are kept hidden from students so that they can think about solving the problem as a whole without looking at the test cases. Passing these test cases shows that the student has understood the problem and thus we have given higher weightage to these type of test cases.

2) **Comments**

When a student writes comments in his program it becomes easier for the TA grading it to understand what the student has done in his code. Comments are used to explain non trivial parts of code. On revisiting the code later in the course, comments will help the student remember what he has done in his code. All instructors insist on the students writing comments.

3) **Indentation**

The third feature is the indentation of the code. Every student is expected to indent his code. Indentation makes a program readable and easier to understand. It also makes it easier for TAs to read the students' submissions and also makes it

easier for a student to read his code later during the course. Also, when preparing for exams a properly indented code will help the student to go through it quickly.

4) **Time to Solve**

The time taken by the student to solve the problem is inversely proportional to the marks the student is awarded. If a student spends less time he may be given more marks than a student spending more time to solve the problem.

5) **Successful Compilations**

The number of successful compilations out of the total compilation attempts is directly proportional to the marks given to the student. If a student makes less compilation errors it would mean that he has understood concepts better and he may be rewarded for it as opposed to a student making many compilation errors.

6) **Program Run-time**

The time taken by a program to evaluate is also something that can be considered while awarding marks to students. If a student has written an optimal code taking less execution time he may be rewarded for it accordingly.

7) **Memory Requirement**

The memory requirement of a program similarly can be considered where a student's program which takes less memory will be awarded higher marks than a program which requires more memory.

We have chosen the following three features as the basis of our grading tool:

- Number of Test Cases Passed.
- Number of Comments
- Indentation

The reason for choosing these features and not others is that they are directly related to the instructor's grading policy. The other features are appropriate in programming competitions but in an introductory course, the instructors do not grade according to the time taken by a student, successful compilations, memory requirement, program run-time, etc.

We have chosen weights according to the most common grading policy chosen by the instructor. We also give the instructor full freedom to change these weights according to the problem statement. This makes the tool more efficient and flexible to use.

4.1 Features for Grading

4.1.1 Number of Test Cases Passed

We evaluate the program on the test cases given by the instructor. The test cases consist of input and expected output. On evaluating the program on these test cases, we determine if the output generated by the student's program matches the expected output. If yes then we say that the program has passed the corresponding test case and if not, then we say that the program has not passed the test case. We find the total number of test cases that are passing out of the total number of test cases provided by the instructor. We calculate two values here:

- T_v indicating the number of visible test cases passing out of the total number of visible test cases available.

$$T_v = \frac{\text{Total no.of visible test cases passing}}{\text{Total no.of visible test cases}}$$

- T_h indicating the number of hidden test cases passing out of the total number

of hidden test cases available.

$$T_h = \frac{\text{Total no.of hidden test cases passing}}{\text{Total no.of hidden test cases}}$$

Values of T_v and T_h lie between 0 and 1 both inclusive.

4.1.2 Number of Comments

We captured the number of words used in comments by a student relative to the total size of the program. We did not check the content of the comments and their meaning. However, this captured the main idea behind expecting a student to write comments in his code. We calculated the comment score (CS) as follows

$$CS = \frac{\text{No.of words in comments}}{\text{No.of words in program including comments}}$$

Value of CS lies between 0 and 1 both inclusive.

We used crowdsourcing to find out different thresholds and assigned marks accordingly. We had a few TAs manually grade problems for comments and calculated CS for each of these problems. This activity was very helpful as it allowed us to find very accurate thresholds. According to the marks given by TAs we found out the final marks for comments (C) that our auto grading tool should award.

$$C = \begin{cases} 0 & \text{if } 0 <= CS < 0.14 \\ 0.5 & \text{if } 0.14 <= CS < 0.22 \\ 1 & \text{otherwise} \end{cases}$$

C=1 means that the student should be awarded full marks for comments, C=0.5 means half the marks should be awarded and C=0 means no marks for comments should be awarded to the student.

4.1.3 Indentation

To check if a student has properly indented his code we checked a few parameters. Firstly, we counted the number of blocks (functions, loops, conditions, etc) that are present in the program. Secondly, we calculated the number of mistakes in indentation made by the student (IS). The mistakes consist of any cases where the student is deviating from the normal indentation guidelines followed.

$$IS = \frac{\text{Mistakes made}}{\text{No.of blocks}}$$

If No.of blocks = 0 it implies that there is no code, so we are setting $IS = 1$.
Using the value IS calculated above and a similar crowdsourcing activity as in the case of comments, we decided the marks that the student should get for indentation (I) as follows:

$$I = \begin{cases} 1 & \text{if } 0 <= IS < 0.5 \\ 0.5 & \text{if } 0.5 <= IS < 0.75 \\ 0 & \text{otherwise} \end{cases}$$

I=1 means that the student should be awarded full marks for indentation, I=0.5 means half the marks should be awarded and I=0 means no marks for indentation should be awarded to the student.
The four values T_v, T_h, C, I are used in calculating the final grade.

4.2 Additional Checks by Auto Grading Tool

We have carried out a few additional checks on the students programming assignments to improve the results of our auto grading tool.

4.2.1 Hardcoded Solutions

Many times some students hard code certain programs which have binary output i.e. the output of the program is either YES or NO. In such cases, if a student prints

YES for all inputs, he may pass around 50% of the test cases and similarly if he prints NO for all inputs, he may pass around 50% of the test cases. This will lead to him getting 50% of the marks that are awarded for passing of test cases.

To prevent this, we have performed an additional check on the output of the student's program in which we are checking if the output is the same for all inputs. If the student is always printing the same output (say YES) for all the inputs, then we are flagging this as a hardcoded case. In such a situation, the student will be awarded zero marks. This check prevents some students from being given marks when it is not deserved. A person hardcoding his solution should not be given more marks than a person who has made an honest attempt but failed to pass any test cases.

4.2.2 Relaxed Output

It is also quite common for some students to solve the problem correctly but while printing the output they may print some extra characters along with the expected output. The actual output may be close to the output expected by the instructor. For example, the expected output for a particular input is "25.65" and the student prints "The average is 25.65". Such cases can be considered and the student should be awarded some marks for this.

We have given the instructor the option to choose whether he wants to give students marks in such cases. There are a few cases of relaxed output which we have taken into consideration.

- **Case Insensitive**

  If the output of the student's program and the output expected by the instructor are same on ignoring case then we can award him marks for the corresponding test case.

Example 4.1 If expected output is "YES" but student is printing "yes" or "Yes" or something which is only different in the case then we are ignoring case and indicating that the student has passed the corresponding test case.

- **Order Independent**

  If the student has to print a list of values but the order in which he has printed them is different from the order expected by the instructor.

  Example 4.2 i) Expected output is a set of all prime numbers from 1 to 100 in sorted order but the actual output of the student is a set of prime numbers from 1 to 100 in a random order, we are ignoring this and the student is awarded marks for the corresponding test case.

  ii) Expected output is a set of all substrings of a string that are palindromes and they should be in a particular order (say increasing length) but the student has printed them in any order, the student will be awarded marks for the corresponding test case.

- **Intermediate Values**

  It may happen that the student is asked to print the nth term of a series but he is printing all terms leading up to the nth term as well. This is something we are considering and awarding corresponding marks to the student.

  Example 4.3 Expected output is the 5th term of the Fibonacci series, i.e. "3" but the student prints "0 1 1 2 3".

- **Extra Characters**

  In case the student prints extra characters in his output in addition to the expected output, we are considering this as a simple mistake and awarding marks for the corresponding test case.

  Example 4.4 If the output is "23" and the actual output of the student is "The result is 23".

- **Duplicate Values**

  Sometimes the student prints the same value twice in his output. It may be that he has called a function to calculate a value and he is printing it on returning from the function in the calling function as well as in the function itself leading to the output being printed twice. This is also a mistake that can be overlooked and the student can be awarded marks.

  Example 4.5 Output expected is "2" but the student prints "2 2"

4.3 Idea for Grading System

The main idea behind choosing these features for grading students programming submissions is that in an introductory programming course the main goal is for the student to learn basic programming. The course does not try to make students expert programmers. At the end of the course it is expected that when given a problem statement of easy/moderate difficulty, the student can derive the logic and code it. The time taken and complexity are not as important. As long as the student finishes the assignment/exam during the lab/exam hours, it is considered as correct. We are not giving marks to a student if he finishes faster than another student. Similarly, the number of successful compilations, program run time, memory requirement, etc have not been considered. However, these are excellent features for programming competitions.

Thus, number of test cases passed is our most important feature. Passing of test cases is an indication that the student is heading towards the correct answer. If a student passes all test cases it means that his code is correct and if he passes no test cases, it means that his code is incorrect. Partial passing of test cases could mean that he is heading towards the right solution and thus, he will be awarded partial marks.

Comments are very essential especially in introductory programming courses as they

allow the student to explain his code. This helps the TAs grading the solution and also can be helpful to the students when they revisit their code before exams. Indentation is a must for any programmer. An indented code is nice to look at, simple to read and easier to understand.

TAs normally look at these features when grading any solution. Also, instructors specify these features in the grading policy. However, the weight for each of these features can be different according to the problem difficulty or whether the program is part of a lab or exam. We have made provision for the instructor to set these weights for our auto grading tool so that it can work just like a human TA. Marks for comments and indentation are only awarded if the student's code passes a certain number of test cases because the student shouldn't be given marks for comments or indentation unless his code is at least partially correct. This value can be set by the instructor.

4.4 Final Score Calculation

After we calculated the feature scores on which we are grading the student, we used a simple weighted formula to calculate the final score (FS).

$$FS = T_v * w_v + T_h * w_h + C * w_c + I * w_i$$

where,
T_v, T_h, C and I are the feature scores for number of visible test cases passed, number of hidden test cases passed, number of comments and indentation respectively.
w_v, w_h, w_c and w_i are the weights for the parameters: number of visible test cases passed, number of hidden test cases passed, number of comments and indentation respectively.

Marks for comments and indentation are only awarded if the student's code passes a certain number of test cases (threshold). Thus, our new final score function becomes:
If Total no.of test cases passed $> \theta$

$$FS = T_v * w_v + T_h * w_h + C * w_c + I * w_i$$

else

$$FS = T_v * w_v + T_h * w_h$$

where,
θ is the threshold mentioned above.

The value of FS lies between 0 and 1. When multiplied with the maximum marks of the problem statement, we will get the final marks that is awarded to the student.

The weights mentioned above (w_v, w_h, w_c and w_i) can be set by the instructor according to the problem difficulty or whether the problem is a lab or exam question.

Chapter 5

System Overview

5.1 Automated Feedback System Overview

The automated feedback system is fully integrated with Prutor [Das15]. When students are programming in Prutor and they hit Compile, they are presented with a message. This says that the compilation is either successful or that there are errors/warnings in the program. If there are compilation errors it is up to the student to correct his program and continue. The normal form of feedback from Prutor is through compiler messages.
The compiler messages consist of the following details:

- Line number

  This is the line number on which the error has occurred.

- Column number

  This is the position of the error on the line number mentioned above.

- Type

  This states whether the message is an error or warning, etc.

- Message

  This is the actual error message given by the compiler.

If the compilation of a student's code is not successful, a list of errors is presented in the above form. These are not very easy to understand for beginners.

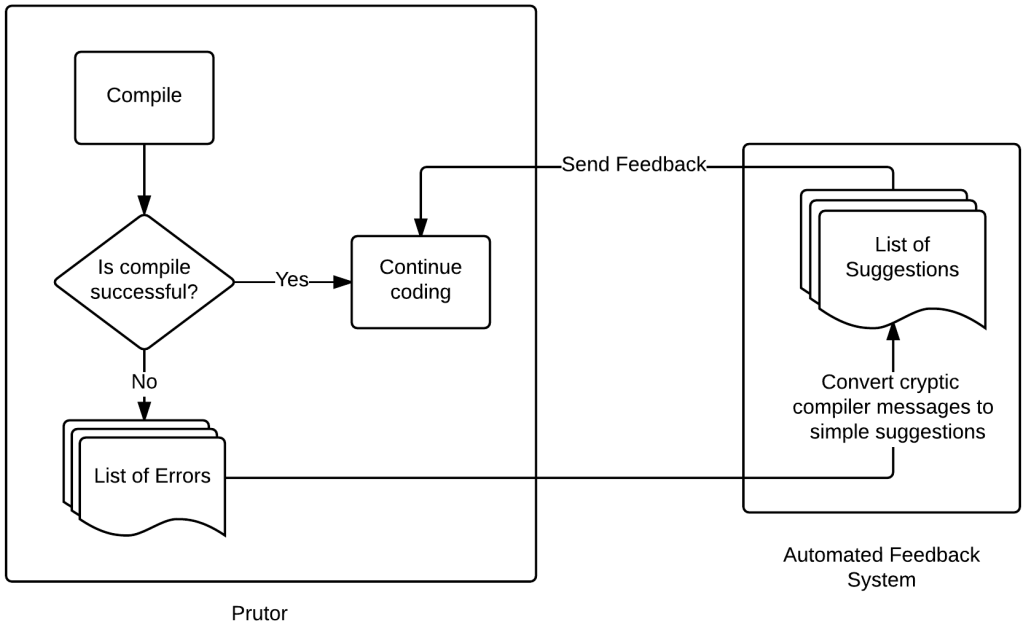


Figure 5.1: Automated Feedback System

Our feedback system comes into play at this time. The list of errors serves as the input to our system. It is then traversed and simple detailed suggestions with examples are created for each error in the list. Our suggestions consist of the following:

- Line number

  This is the line number on which the error has occurred.

- Explanation of Error

  This is a simple explanation of the error made by the student.

- Examples

  There are two types of examples given.

  - Valid statements:

    These are examples consisting of the correct way to write the statement

in which the error was made and some other correct ways to write similar statements.

- Invalid statements:

  These are examples of similar errors that a student may make related to the error made.

After the list of errors is exhausted and we have our suggestions for all the errors, these suggestions are put into a list. This list is the output of our system. The output is sent to Prutor where it is displayed to the students as feedback. The suggestions are displayed beside the code and the students don't need to click any button to access them.

5.2 Automated Grading System Overview

Our Automated Grading System has been fully integrated with Prutor [Das15]. When students finish their program in Prutor, they are expected to submit the code. Once the code is submitted only then it is graded by the TA. If a student does not submit his code, he will be given zero marks for the corresponding problem. Every problem has a list of test cases associated with it. These consist of input to the program and corresponding expected output.

Once the student submits his code, we take this submitted code along with the list of test cases from Prutor. Our first task is to compile the code. We then get a message saying if the compilation was successful or not. If it failed, our automated grading system exits and zero marks are awarded to the student or marks for comments and indentation are given depending on the instructor. If the compilation is successful, we evaluate the code on the given test cases. This means that we run the code on the inputs mentioned in the test cases and see if the output of the code matches the expected output mentioned in the test cases.

After evaluating the code, we calculate the test case score, i.e. the total number of test cases passing out of the total number of test cases given by the instructor as explained in Section 4.1.1. We then calculate the comment score, i.e. the number of comments in the program as explained in Section 4.1.2. Finally we calculate the indentation score as explained in Section 4.1.3.

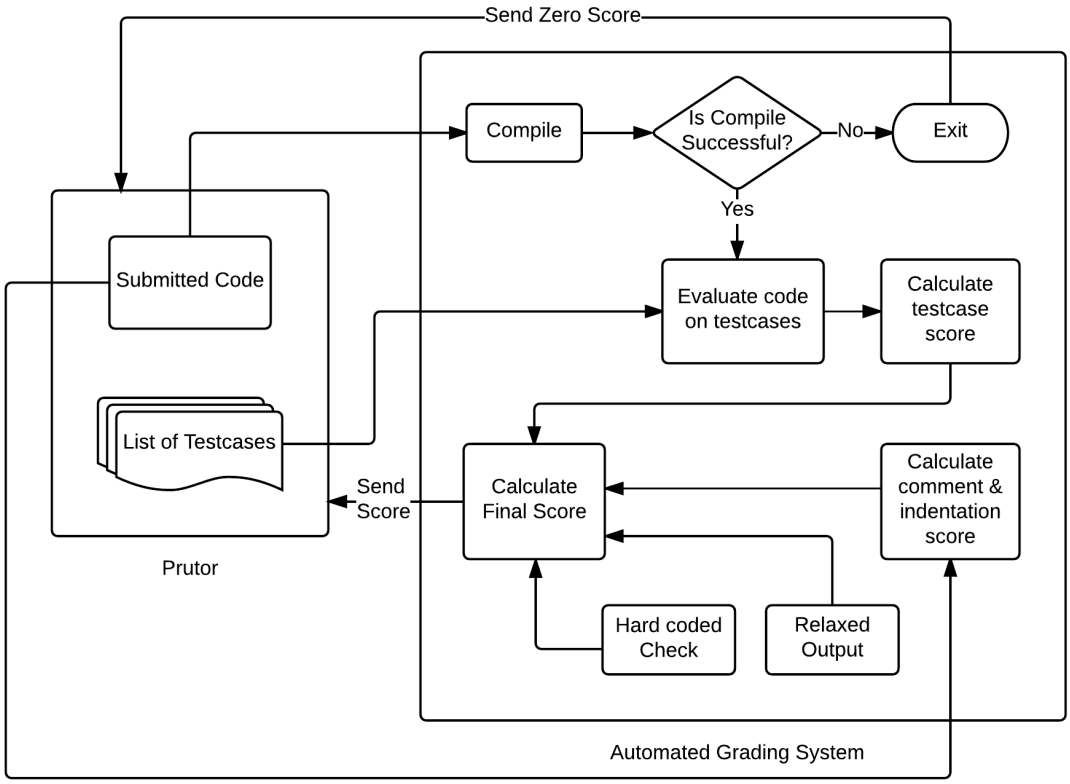


Figure 5.2: Automated Grading System

We now perform a few additional checks on the code. First, we check if the student has hardcoded his solution as explained in Section 4.2.1. Then we also allow for some relaxed output for the student's code as explained in Section 4.2.2.

Once we get the above scores, we calculate the final marks that is to be awarded to the student as explained in Section 4.4. Once this score is calculated it is sent to Prutor where the instructor can access it. It can also be shown to the student immediately after he submits his code if the instructor wishes to allow it.

5.3 Configuration Parameters

The page in Figure 5.3 has been made for the instructor where he can set certain key values that correspond to the parameters used by our system.

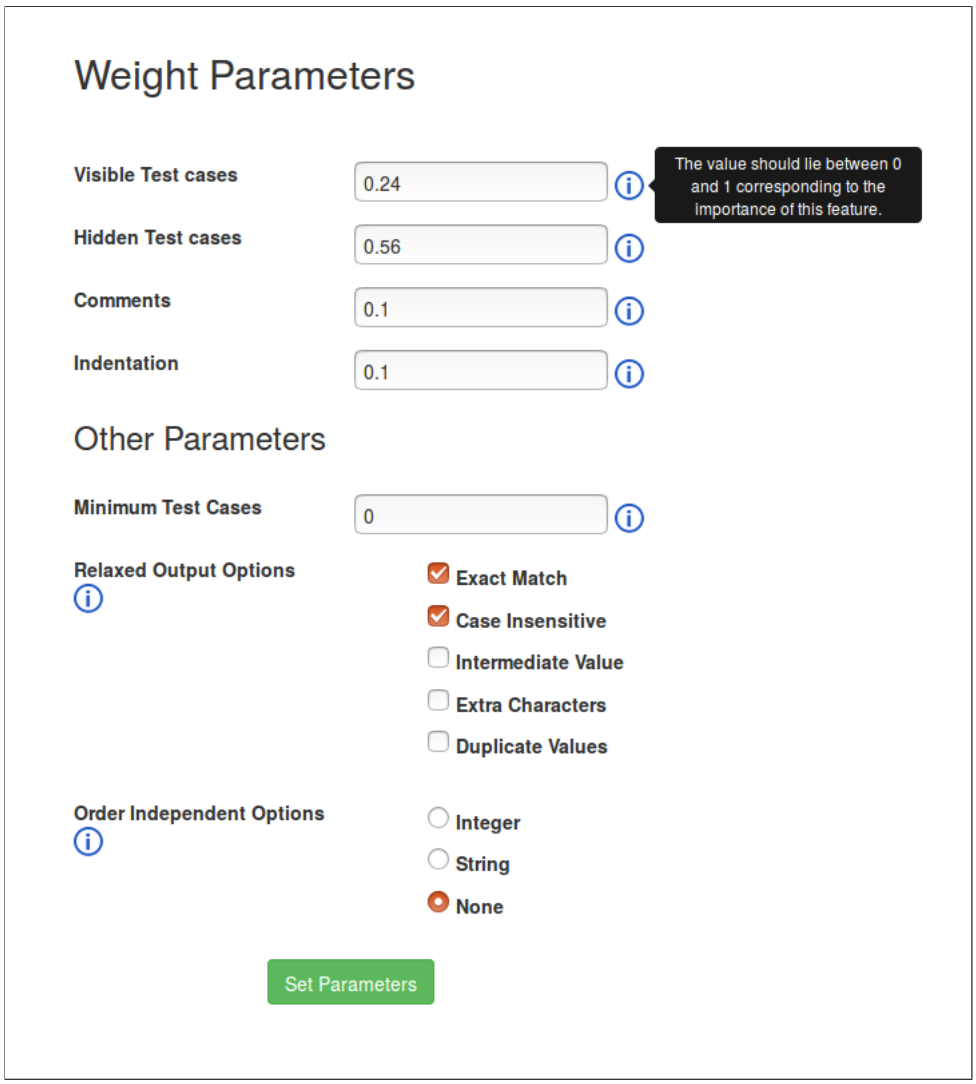


Figure 5.3: Configuration Parameters Web Page

The parameters are:

1) Weight Parameters

   Here the instructor can set the different weightages he wants to give for each of the below features. They must all sum to 1.

   i) Number of Visible Test Cases Passed

ii) Number of Hidden Test Cases Passed

iii) Number of Comments

iv) Indentation

2) Minimum Test Cases Required

The marks for comments and indentation can be awarded to a student if his code passes a certain number of test cases only. The instructor can set this value accordingly. If this value is zero, then the student will be awarded marks for comments and indentation irrespective of how many test cases have passed.

3) Relaxed Output

The following cases of relaxed output can be allowed by the instructor. They are enlisted below and explained in detail in Section 4.2.2.

i) Exact Match

ii) Case Insensitive

iii) Order Independent (int or string)

iv) Intermediate Values

v) Extra Characters

vi) Duplicate Values

Chapter 6

Experiments and Results

All our experiments have been carried out on real world data that we received from the introductory programming course ESc101 conducted in the odd semester of 2015-16 at IIT Kanpur. C programming language is used for the whole course. Labs of around 3 hours are conducted every week by ESc101 for enrolled students who are assigned programming problems. These students are given few problem statements to code and their task is to submit solutions to their assignments within the specified period. The students are allowed to take help from the TAs during the labs regarding issues with the system and sometimes with trivial errors in the code. The submitted solutions are then tested against a set of test cases. Finally the Teaching Assistants (TAs) manually inspect the codes and after considering the number of test cases passed, judge and grade the submitted solutions by awarding them marks. Exams carrying much more weightage than the lab assignments are conducted. The students are not allowed any help from TAs during the lab exams.

Prutor [Das15] gave us access to a large amount of data related to the course including assignments, submissions, marks, problem statements, test cases, compilation errors, grading tasks, etc.

6.1 Rating of Suggestions

We did not have access to a live lab to test our feedback system. Thus, we conducted a crowd sourcing activity with some TAs who had a lot of experience with helping students during the labs. These TAs have seen students struggling with simple compiler errors. Most of the times students need to resort to asking the TAs for help in correcting the errors. Due to this we thought that in the absence of a live lab, they would be the most suitable to rate the suggestions given by our system.

We created a web page for this purpose. It consisted of the students code, our system feedback and compiler feedback. The TA can see the code along with the corresponding compiler errors and our feedback and rate our suggestions on a scale of 0 to 3 depending on how much easier it is to understand the suggestions compared to the compiler feedback. There is a text field for the TA to enter his rating. Each code was rated by only one TA. We selected student's codes that have some compilation errors as successfully compiled code will not help us in rating our system suggestions.

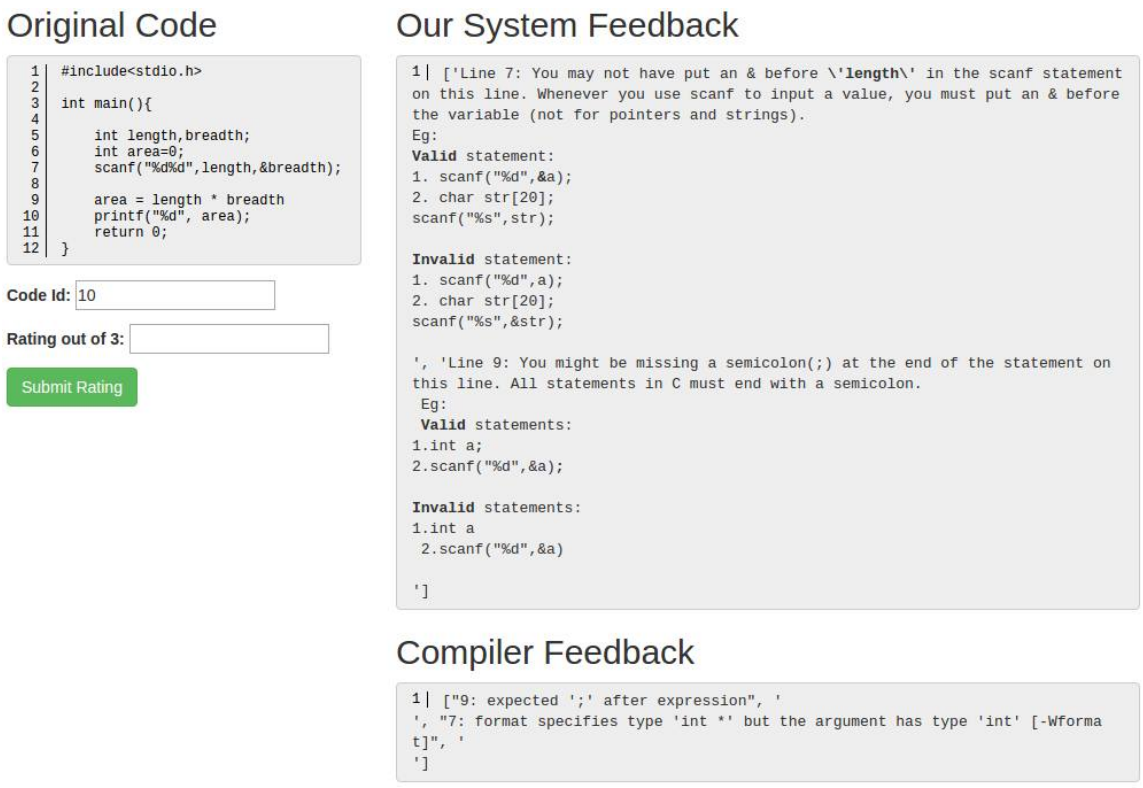


Figure 6.1: A Web Page to Rate Suggestions

We had 8 TAs performing this task and a total of 1,000 submissions which did not compile successfully were inspected. The TAs rated our suggestions comparing them to compiler feedback as either 0,1,2 or 3 where

- 0 → wrong/bad suggestions
- 1 → no suggestions
- 2 → some good suggestions and no suggestions for some
- 3 → very good suggestions.

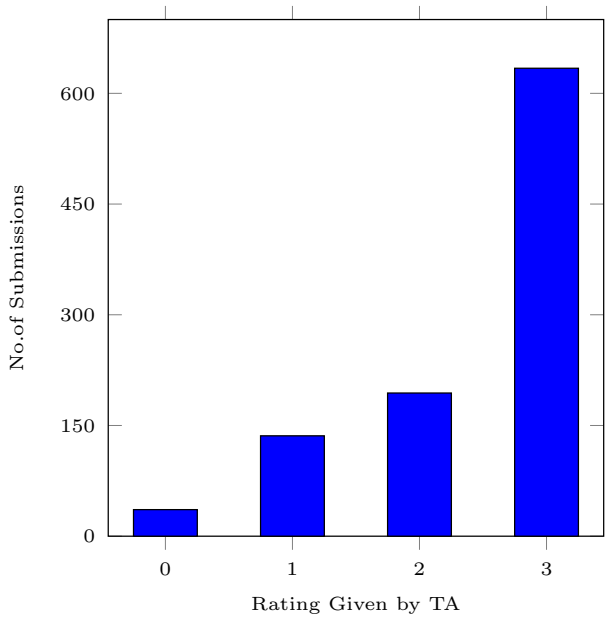


Figure 6.2: Ratings Given to Submissions

The plot in Figure 6.2 shows the number of submissions where the suggestions were given a rating of 0,1,2 and 3 respectively. We took the average of the rating given by all TAs and found that the average rating given to our feedback system was 2.426. This shows that students can benefit a lot from our feedback. This will help them save time in correcting compiler errors and enable them to correct their mistakes with little or no help from TAs.

6.2 Response Time of Grading System

The response time [Wikb] of a system is the time taken by it to respond / react to a certain input or request. It should be as fast as possible so that people will want to use the system. We conducted a few experiments to determine the time taken by our grading system to grade assignments. During the course, there were twelve lab events and two exams conducted.

Our system can grade the following events:

1) A single submission

   Our grading system takes anything between 56 milliseconds and 309 milliseconds to grade a single lab submission. This range is due to the fact that some programs require more computational resources [Wika] than others. Computational resources include computation time, memory space required, number of steps required to solve the problem, source lines of code [Wikc], etc. If the source lines of code are more, then the computational resources may increase thus increasing the total response time of the grading system. For a single exam submission the average response time is between 282 milliseconds and 455 milliseconds. This is slightly higher because exams usually consist of problems which require more computational resources.

2) All submissions of a particular lab

   The time taken to grade each lab differs from each other due to the above mentioned reasons. We found that it takes 52 seconds to grade lab 2 which consists of 927 submissions and 4 minutes 7 seconds to grade lab 7 which consists of 798 submissions. Even though lab 7 has lesser number of submissions, the time taken to grade is more because the problem statements in lab 7 require more computational resources than problem statements in lab 2.

3) All submissions of all labs

   When run on all labs, the grading system takes 25 minutes 25 seconds to grade

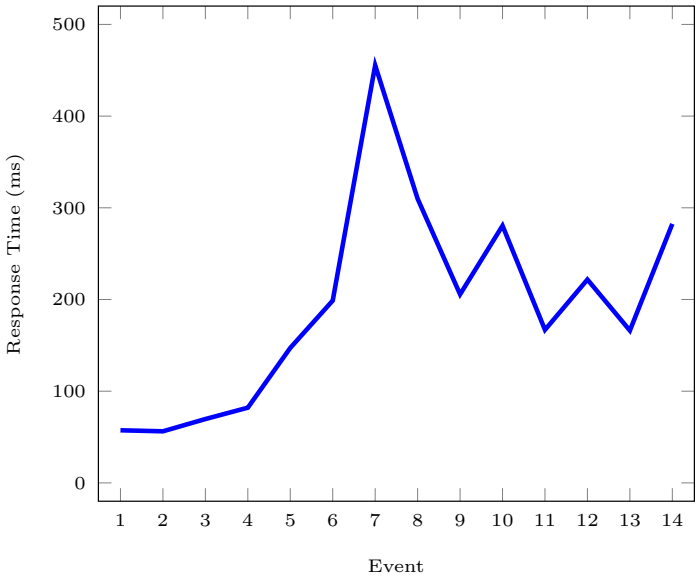
submissions of all labs.

4) All submissions of all exams

When run on the exams, the grading system takes 17 minutes 35 seconds to grade all exam submissions.

Sr.No.	**Event**	**Time Taken**
1	One Submission (Lab)	56 ms - 309 ms
2	One Submission (Exam)	282 ms - 455 ms
3	A complete Lab	52 sec - 4 min 7 sec
4	All Labs	25 min 25 sec
5	Exam 1	12 min 3 sec
6	Exam 2	5 min 32 sec

Table 6.1: Response Time of Grading System



Event 1-6 : Lab Events
Event 7 : Exam 1 (Mid Semester)
Event 8-13 : Lab Events
Event 14 : Exam 2 (End Semester)

Figure 6.3: Performance of Autograder Over Events in ESc101

Figure 6.3 shows how the response time varies from event to event. Some events in which the problems require students to use data structures like pointers, structures, etc. require more computational resources and may take longer to grade than others which require less computational resources. The graph rises substantially during Exam 1. This could be due to many reasons. The number of test cases during Exam 1 are more than the normal labs so testing on them will take longer time. Also, due to the lack of help from TAs and friends during the exam, students tend to make more mistakes / compile-time errors. Thus, a submission which has errors or which requires a long time to execute, will take a longer time to grade.

6.3 Automated Grading vs Human TA

The main goal of our automated grading system is to make it perform like a human TA. We want the marks given by our grading system to be comparable to that of a TA. We conducted the following experiment to compare our results with a TA.
We graded all submissions that compiled from all lab events and recorded some useful data. There were 6,021 such submissions. The reason for choosing only the compiled submissions is that our tool does not work well with submissions that do not compile. Some of the data recorded included:

- Assignment ID
- Max Marks
- TA Score
- Autograder Tool Score
- Difference between TA and Tool Score

The difference ranges from negative of the maximum marks to the maximum marks for the problem.

We found the number of submissions / assignments for each of the values in this range of differences and created a plot as shown in Figure 6.4. We noticed that the

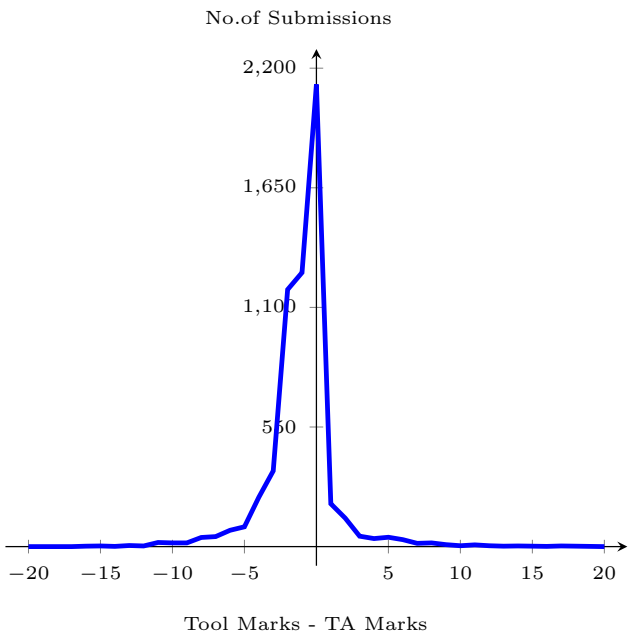


Figure 6.4: Difference Between TA Marks and Tool Marks

number of submissions where the difference is zero is the most (2,125). The average marks given by TA over these 2,125 submissions is 18.5 and the average marks given by our tool is also 18.5. This shows that our tool gives marks very close to the TA in such cases where the submission is passing almost all test cases. This decreases as the difference increases. The plot shows that most of the submissions are in the difference range -5 to 3 and very few submissions are in the range -20 to -6 and 4 to 20. The ideal case would be to have all submissions in the difference 0 category but this is very difficult to achieve. There can be many reasons for this difference in marks between TA and Tool. We analysed this and found two main causes for the difference.

- Inconsistency in TA Grading
- Mismatch in Output

6.3.1 Inconsistency in TA Grading

Inconsistent grading is very common due to a large number of TAs grading student's assignments. Different students having similar solutions will be awarded

similar marks by our system but this cannot be guaranteed by human TAs. One TA may deduct more marks than another for the same error, thus bringing about inconsistencies. This is not fair to the student who gets lesser marks. In spite of the instructor's grading policy being given, this inconsistency occurs. It is very difficult for the instructor to catch such inconsistencies as he will need to manually inspect each problem which defeats the purpose of having TAs grading assignments. Our tool can help the instructor to catch these inconsistencies.

Example 6.1 Consider the following two submissions made by different students for the same problem. Both are similar and deserve the same amount of marks.
Problem Statement: Write a program to determine if the year given as input is a leap year or not.

Student 1

```c
#include<stdio.h>

int main()
{
    int a,b;
    scanf ("%d",&a);

    b=(a%4) ;

    if (b==0) {
    printf ("Leap Year");
    }
    else {
        printf ("Not Leap Year");
    }
    return 0;
}
```

Student 2

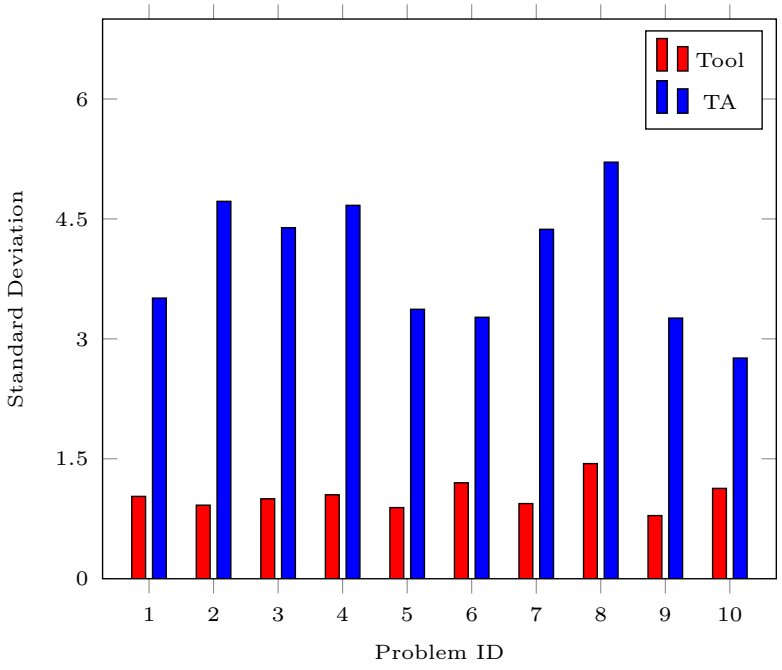
```c
#include<stdio.h>

int main()
{
    int y;
    scanf("%d",&y);
    if(y%4==0)
    {
        printf("Leap Year");
    }
    else{
        printf("Not Leap Year");
    }
    return 0;
}
```

[Both submissions were commented well but we have not shown the comments.]
The maximum marks for the problem is 20. When graded by a human TA, student 1 was given 19 marks and student 2 was given 5 marks. On looking at the submissions in Example 6.1 above we can see that the submissions are exactly same and both

students should be given the same marks. This inconsistency is very unfair to student 2 and biased towards student 1. This sort of grading is not acceptable. Our grading system gives 12 marks to both students. Not only does our system grade fairly but also can help in catching such inconsistencies made by human TAs.

Due to the above mentioned inconsistency, we began to analyse further. We selected 10 problem statements from all labs and created 10 sets corresponding to the problem statements. Within each problem set we selected a group of solutions which are similar and deserve similar marks. The number of submissions selected in each set was between 10 and 20. We then found the standard deviation of our tool and standard deviation of TA and plotted the results in Figure 6.5 below.



Problem 1-10 : Problem Sets Chosen from Different Labs

Figure 6.5: Inconsistencies in TA Grading Compared to Autograding

Problem IDs 1 to 10 correspond to the different problem statements that were selected from all labs. The plot in Figure 6.5 above captures the inconsistencies in TA grading. It can be seen that in each of the problem sets, the standard deviation of TA grading is much more than that of our grading system. As the solutions in each set are similar, the standard deviation should be lower. This shows that the

inconsistency in TA grading is high. The instructor can use this data to identify such TAs and thus make the grading process fairer.

6.3.2 Mismatch in Output

Our system is allowing for certain relaxation in the output as explained in Section 4.2.2. However, there are still a few cases of relaxed output which we have not considered. These are little difficult to identify for an automated system and they may lead to identifying an incorrect solution as correct. However, human TAs can identify such cases and give marks after deducting a small penalty for the mistake.

Example 6.2 Consider the following program in which the student has to calculate the distance between two points.

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

int main(){
    float x1,x2,y1,y2,distance;
    scanf("%f",&x1);
    scanf("%f",&x2);
    scanf("%f",&y1);
    scanf("%f",&y2);
    distance=sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
    printf("The distance between (%.3f,%.3f) and
            (%.3f,%.3f) is\n%.3f.",x1,x2,y1,y2,distance);
    return 0;
}
```

Consider the input to the program as "-1 -2 -1.2222 -3.2222". The output of this program is expected to be in the format "The distance between (-1.000,-1.222) and (-2.000,-3.222) is 2.236." but the output of this code is "The distance between (-1.000,-2.000) and (-1.222,-3.222) is 2.236.". It can be seen that the student has interchanged the values -1.222 and -2.000. He has also put an unnecessary new line character before printing the final value. Such a difference in the outputs is difficult to capture. In these cases, our tool will award much lower marks than the TA leading to an increase in the absolute difference of marks between TA and Tool.

Chapter 7

Conclusions and Future Scope

7.1 Conclusions

This is the final chapter of our thesis. We have achieved our aim of designing and developing an Automated Feedback and Grading System that is fully integrated with Prutor [Das15] and can be used during introductory programming courses like ESc101 conducted at IIT Kanpur.

After performing our experiments the results show that our grading system works comparable to TAs. It helps to catch inconsistencies in TA grading as well. In situations where the actual and expected output do not match or when the submission does not compile, our grading system does not work very well. From Section 6.1, we see that our suggestions, given as feedback, when a student tries to compile his program are very helpful as they clearly explain any compilation error that the student may have made.

Prutor is also used to conduct programming examinations for PhD and M.Tech. admissions at IIT Kanpur. During these exams the grading policy is normally completely based on passing of test cases. Our system would be perfect for grading such exams and there would be no need for human TAs. We can set the weights for comments and indentation to be zero so that the system grades only based on passing of test cases.

7.1.1 Applications

There are several ways in which our Automated Feedback and Grading System can be useful and add functionality to Prutor [Das15].

1) The instructor can grade a particular assignment by simply clicking a button on the grading page.

2) There is a web page for the instructor as shown in Figure 5.3 where he may change parameters for different labs / exams according to the difficulty of the problems or according to his needs. There is another option to grade all labs or all exams together or grade all submissions for a particular lab.

3) The details of the grading can be used by the instructor to inspect the grading of TAs and find inconsistencies therein.

4) The marks given by the tool can be used as suggestions to the TA while grading.

5) The marks can be shown to the students during the labs, as immediate feedback once the code is submitted. As opposed to giving the actual marks, we can show a performance rating, i.e. a score on a scale of 1 to 5 where 1 indicates a wrong solution and 5 indicates a correct solution.

We have integrated all the above features with Prutor. They can easily be turned on or off according to the instructor's need.

7.2 Future Scope

We will discuss the points that have strong potential for future work.

7.2.1 Improvements to Grading

At the moment our grading model is looking at test cases, comments and indentation to decide the marks to be awarded to the student. These are the most

appropriate features to consider while grading programs in introductory programming courses as most instructors specify these parameters in their grading policies. We also check for hardcoded outputs and allow for a certain amount of relaxation in the actual output of the student.

There is still scope to improve the performance and accuracy of the automated grading system.

1) In addition to the number of comments written by a student, we can check for meaningful comments while awarding marks for comments.

2) Sometimes the instructor may specify in the problem statement that it is compulsory to use a certain data structure or program construct. If the student has not followed this instruction, few or zero marks will be awarded. There can be a check for this so that if a student has not used the required data structure or construct, he would be penalized.

3) There are certain times when the instructor specifies that certain header files must not be included in the program. A check for this can be done and a penalty can be imposed according to the instructor.

4) The number of test cases per problem can by increased to catch students attempting to hardcode their solutions. If the student is passing 3 out of 6 test cases by hardcoding the output he will get 50% of the marks for passing test cases feature. But if we increase the test cases to around 50 - 100, then the fraction of the passing test cases will reduce tremendously and thus the marks awarded to the student will be significantly reduced.

5) There are certain situations in which the student's output is correct but it is not exactly matching with the output expected by the instructor. These relaxations in output can be added.

6) If a student's submission does not compile, our system gives zero marks or a few marks for comments and indentation depending on the instructor. We can

implement a tool that corrects these compilation errors and then tries to grade the submission.

7.2.2 Automated Repair

Our automated grading system faces a challenge when programs do not compile successfully. It awards zero marks to the student or a few marks for comments and indentation in such cases depending on the instructor. In an introductory programming course this might not be a good method as a student might be solving the logic to the problem correctly but it is not compiling and if so zero marks is not a fair grade. A student should get better marks for a submission that does not compile but is close to the correct solution as opposed to a student's submission which is far from the correct solution but compiles.

Consider the situation where a student has made one or more compilation errors in his code, which on correcting results in the passing of some or all test cases. In such a case, the TA grading would deduct a small penalty for the mistake and grade on the logic of the program. If the system took care of this, the marks given would be more appropriate and closer to a human TA.

We have implemented such an Automated Repair System [Sin16] which corrects the student's compilation errors and then tries to automatically grade their submissions. It will give a small penalty for these corrections according to the instructor's needs. This is a complementary system to our Automated Grading System.

References

[BS16] Sahil Bhatia and Rishabh Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *CoRR*, abs/1603.06129, 2016.

[CKLO03] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers and Education*, 41(2):121 – 131, 2003.

[Cla] Clang. A c language family frontend for llvm.
URL: `http://clang.llvm.org/`.

[Das15] Rajdeep Das. A platform for data analysis and tutoring for introductory programming. M.tech. thesis, Indian Institute of Technology Kanpur, India, 2015.

[GCC] GCC. The gnu compiler collection.
URL: `https://gcc.gnu.org/`.

[GRZ14] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Feedback generation for performance problems in introductory programming assignments. *CoRR*, abs/1403.4064, 2014.

[GRZ16] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *CoRR*, abs/1603.03165, 2016.

[NPM08] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? *SIGCSE Bull.*, 40(1):168–172, March 2008.

[Par15] Sagar Parihar. Automated grading tool for introductory programming. M.tech. thesis, Indian Institute of Technology Kanpur, India, 2015.

[SBL+13] Mark Sherman, Sarita Bassil, Derrell Lipman, Nat Tuck, and Fred Martin. Impact of auto-grading on an introductory computing course. *J. Comput. Sci. Coll.*, 28(6):69–75, June 2013.

[SGSL13] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *SIGPLAN Not.*, 48(6):15–26, June 2013.

[Sin16] Praveen Kumar Singh. Automated repair of programs in introductory programming courses. M.tech. thesis, Indian Institute of Technology Kanpur, India, 2016.

[Tra10] V. Javier Traver. On compiler error messages: What they say and what they mean. *Adv. in Hum.-Comp. Int.*, 2010:3:1–3:26, January 2010.

[Wika] Wikipedia. Computational resource.
URL: `https://en.wikipedia.org/w/index.php?title=Computational_resource&oldid=702911944`.

[Wikb] Wikipedia. Response time (technology).
URL: `https://en.wikipedia.org/w/index.php?title=Response_time_(technology)&oldid=694600918`.

[Wikc] Wikipedia. Source lines of code.
URL: `https://en.wikipedia.org/w/index.php?title=Source_lines_of_code&oldid=721803500`.