Data-Driven Feedback Generator for Online Programing Courses

Ke Wang
University of California, Davis
Davis, USA
kbwang@ucdavis.edu

Benjamin Lin
Microsoft
Redmond, USA
benjlin@microsoft.com

Bjorn Rettig
Microsoft
Redmond, USA
bjoernr@microsoft.com

Paul Pardi
Microsoft
Redmond, USA
paul.pardi@microsoft.com

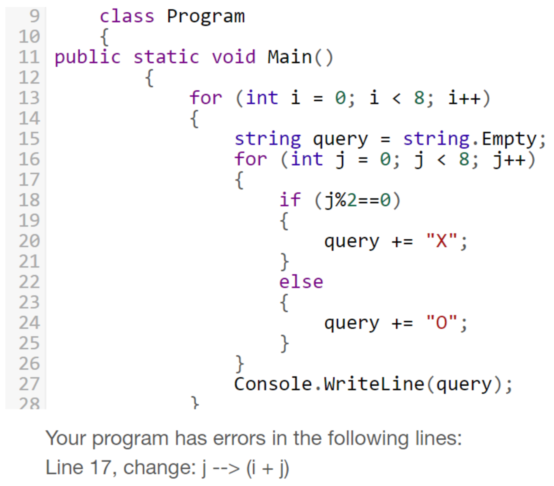
Rishabh Singh
Microsoft
Redmond, USA
risin@microsoft.com

ABSTRACT

Manually providing feedback for programming assignments is a tedious task in traditional classroom education. The challenge increases drastically in Massive open online courses (MOOCs), where the student-teacher ratio can reach thousands to one or even millions to one. Despite the necessity, the current automated feedback approaches suffer from significant weaknesses: inability to scale to larger programs, manual involvement of teacher effort, and lack of precision for pinpointing errors. We present a technique to tackle these challenges by developing a data-driven automated grader, iGrader, capable of generating instant and precise feedback for programming assignments.

INTRODUCTION

The current nascent paradigm shift in education towards Massive Open Online Courses (MOOCs) increases the accessibility of higher education substantially. Despite the profound impact, it presents several new challenges, particularly on the scalability aspect. In this paper, we target one specific challenge of providing personalized feedback for programming assignments in introductory programming language courses. Unfortunately, the traditional approach of providing manual instructor feedback for programming assignment is no longer feasible in MOOCs. As a counter-measure, online course vendors adopt the following approaches: 1) customize the style of questions to have multiple-choice or those of similar forms for which the hard-coded answer can be easily provided; 2) peer-review each other's answer to the same question [5]; 3) execute students' program submissions through a set of predefined tests. Even though the existing feedback generation mechanisms are valuable, they are far from ideal solutions.



```
 9      class Program
10      {
11  public static void Main()
12          {
13              for (int i = 0; i < 8; i++)
14              {
15                  string query = string.Empty;
16                  for (int j = 0; j < 8; j++)
17                  {
18                      if (j%2==0)
19                      {
20                          query += "X";
21                      }
22                      else
23                      {
24                          query += "O";
25                      }
26                  }
27                  Console.WriteLine(query);
28              }
```

Your program has errors in the following lines:
Line 17, change: j --> (i + j)

Figure 1. An example feedback generated by iGrader.

Background and Related Work

There has been two broad lines of research proposed for personalized feedback generation: program synthesis based [7] and machine learning based [4, 6]. The idea behind the former is to first generate a collection of candidate programs based on the error rules manually provided by the instructor, and then use SAT-based synthesis algorithm to find minimal repairs to correct the student programs. There are two major drawbacks of this approach. First, an instructor must manually provide error models for each problem. Second, scalability is a big issue especially with larger programs. There has also been some promising machine learning based approaches developed for personalized feedback generation [3,4], but their focus is to establish a mechanism that allows for fast searches into large dataset of student submissions. Thus, teachers can leverage

L@S 2017, April 20-21, 2017, Cambridge, MA, USA
© 2017 ACM. ISBN 978-1-4503-4450-0/17/04. . . $15.00
DOI: http://dx.doi.org/10.1145/3051457.3053999

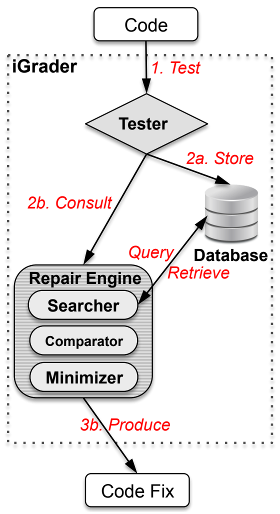


Figure 2. An example feedback generated by iGrader.

the redundancy of student homework to force-multiply their effort [3].

In this paper, we introduce a data-driven approach for automatically generating feedback for programming assignments. Our key insight is given the large amount of student submissions; the sub-parts of vast majority of incorrect attempts should have correct counterparts that can be used for correcting incorrect submissions.Given an incorrect submission, iGrader first finds closely related submission (both syntactically and semantically) to compute corresponding expression discrepancies and then finds a minimal set of repairs from the discrepancies to correct the student program. We have implemented our technique in a new feedback generator system, iGrader, and preliminarily evaluated it on student submissions for Microsoft C# online course on Edx [2] and CodeHunt [1]. The results show iGrader is very effective — repairing more than 70% of the submission with correct control flow and requiring less than three fixes, within few seconds.

OVERVIEW OF THE APPROACH

In this section, we give a high-level algorithm of how iGrader takes student submissions and produces automatic fixes. Figure 2 depicts the architecture of iGrader and Figure 1 depicts an example feedback generated by iGrader. We discuss the key components in this section, and defer the discussion of the core componen — the "Repair Engine" — to later section. We next give a description of iGrader's workflow:

1. **Testing**: iGrader first tests programs for functional correctness using a set of predefined test cases.
2. **Test Outcome**: Two cases:
   (a) **Storing**: If the program passed all the test cases, iGrader stores it in an internal database.
   (b) **Consulting**: Otherwise, iGrader calls Repair Engine into action, which
      i. **Querying and Retrieving**: automatically queries the database and retrieves a set of correct programs that are most similar to the student program.
      ii. **Comparing**: compare the programs in a pair-wise way to collect potential fixes.
      iii. **Minimizing**: minimize the set of fixes that the student program needed for repair.
3. **Feedback Generation**: Finally, iGrader produces the minimal set of fixes back to students

REPAIR ENGINE

In this section, we give a detailed presentation on the repair engine — the centerpiece of iGrader that consists of three components: *Searcher*, *Comparator* and *Minimizer*.

Searcher

Given a database of correct solutions, iGrader performs a hierarchical search to find the programs that are similar to the student submission. The first level targets the control-flow structure. Specifically, iGrader only looks for programs that have the same control flow graph as the student program. iGrader then proceeds to the next level of non-control statement/expression. At this level, iGrader defines the similarity to be the maximum number of matching nodes in the two ASTs of the respective non-control statement/expression over the total number of nodes in the two ASTs. iGrader considers two non-leaf nodes to be a match if they represent identical type of expression such as parenthesis, binary op, method invocation, etc., and leaf nodes to be a match if they are the same identifier, operator, literals, etc. Furthermore, matching two nodes in the context of trees necessitates two additional constraints: 1). Any node in one tree can match one and only one node in the other and vice versa. 2) Any two nodes in an ancestral relationship in one tree must match two nodes in the same relationship in the other and vice versa. Now we introduce an algorithm that computes the maximum number of matching nodes in two ASTs under the two structural constraints.

Given the two ASTs (T_1 and T_2 rooted at α and β) to be matched, we split the problem into three separate scenarios:

i α is directly matched with β in which case the maximum number of matching nodes will be equal to total maximum number of matching nodes from each of the subtrees rooted at the each of the children nodes of α and α in order plus 1 if α is indeed matched to β.

ii α is matched with γ, one descendent nodes of β, in which case the maximum number of matching nodes will be equal to that between T_1 and the tree rooted at γ.

iii α's descendent node, ε, is matched with β in which case the maximum number of matching nodes will be equal to that T_2 and the tree rooted at ε.

We use a dynamic programming algorithm to compute the maximum number of matching nodes between the two trees.

Step	Subtree in T_1 (Root)	Subtree in T_2 (Root)	Best Score	Best Score Scenario
1	a	a	1	(i)
2	a	+	0	(i)
3	a	c	0	(i)
4	a	a+c	1	(ii)
5	a	(a+c)	1	(ii)
6	a	(a+c)*b	1	(ii)
	...	...	...	...
24	a*b	a	1	(iii)
...	...	...	...	
32	a*b	(a+c)*b	3	(i)

Table 1. The table shows the matching procedure between the subtree in T_1 and T_2 in order.

Because in any of the three scenarios, the maximum number of matching nodes between the two trees will depend on that of the subtrees. Hence we can start by finding the maximum matching nodes for the atomic trees rooted at the leaf nodes and then propagate in a bottom-up fashion until we reach the root. Any two subtrees in the process for which we compute the maximum matching nodes, we compare the three scenarios and find the one yielding the best score. For instance, given the two ASTs depicted in Figure 3, we show a truncated version of computation procedure in Table 1.

Comparator
After the *Searcher* finds a set of most similar programs to what student submitted, it's up to *Comparator* to produce the operations (edit/delete/insert) that can transform the student program to different counterparts. In other words, *Comparator* outputs the set of all fixes that can repair the student program. Given the matching between the two ASTs from *Searcher*, *Comparator* recursively traces the roots of the subtrees on the optimal path and produces the operations based on the matching scenarios in a top-down manner. For example, in T_1 and T_2, the matching between $a*b$ and $(a+c)*b$ suggests an edit operation, specifically changing $a*b$ to $(a+c)*b$. In addition, the matching scenario also entails that a in T_1 and $(a+c)$ in T_2 will become the subsequent roots for consideration. This time their matching indicates an insertion operation — adding $(a+c)$ in between $a*b$ and a in T_1. Finally, a in T_1 will be matched with a in T_2, meaning $a+c$, $+$ and c will all be added into T_1.

Minimizer
We can fix the student program using the set of fixes obtained from *Comparator*, but that will not result in good repairs, because: 1) the number of fixes *Comparator* produces will typically be large; 2) not all of them are necessary to repair the program. The *Minimizer* module discovers the minimum set of fixes a program needs. We use an enumerative algorithm to apply the set of fixes in increasing number of fixes to output the minimum set of fixes.

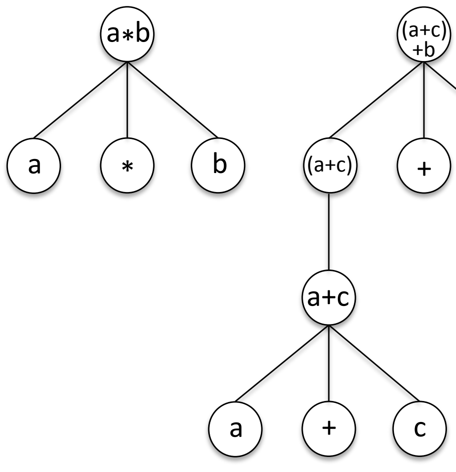


Figure 3. AST T_1 (left) and T_2 (right).

PRELIMINARY EVALUATION
We collected submissions from the Microsoft C# online course Module Two programming assignment on *Edx* (3,361 student submissions). We used Microsoft's Roslyn compiler framework for static parsing and dynamic execution of the C# programs. The current results are as follow:

- 46% of student submissions failed to match with any reference solution at the control flow level.
- Among the submission that succeeded at the control flow matching, 72% of them can be corrected with less then three fixes.
- iGrader generates feedback within 2-5 seconds for each program (depending on the number of fixes).

To further demonstrate the generality of iGrader's, we have also used two other programming problems[1] from CodeHunt [1]. The iGrader system is already capable of repairing 95%, 61% of the student submissions with less than three fixes.

FUTURE WORK
There are several directions we want to pursue. 1) Feedback generation for performance issues. 2) A sophisticated user interface on the instructor side which control how the feedback will be provided to the students. 3) We would also like to extend iGrader to other online courses to benefit more students.

REFERENCES
1. 2013. Code Hunt. `https://www.codehunt.com/`. (2013). Accessed: 2017-01-10.
2. 2014. Programming with C# | edx. `https://www.edx.org/course/programming-c-microsoft-dev204x-2`. (2014). Accessed: 2017-01-10.

[1]One problem was to find the difference between the minimum and maximum values from an array and the other was to write a parenthesis-matching algorithm for a given string.

3. Elena L Glassman, Rishabh Singh, and Robert C Miller. 2014. Feature engineering for clustering student solutions. In *Proceedings of the first ACM conference on Learning@ scale conference*. ACM, 171–172.

4. Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*. Citeseer, 25.

5. Chinmay E Kulkarni, Michael S Bernstein, and Scott R Klemmer. 2015. PeerStudio: rapid peer feedback emphasizes revision and improves performance. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. ACM, 75–84.

6. Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*. ACM, 491–502.

7. Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48, 6 (2013), 15–26.