Concept-Based Automated Grading of CS-1 Programming Assignments

Zhiyu Fan
National University of Singapore
Singapore
zhiyufan@comp.nus.edu.sg

Shin Hwei Tan
Concordia University
Canada
shinhwei.tan@concordia.ca

Abhik Roychoudhury
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Due to the increasing enrolments in Computer Science programs, teaching of introductory programming needs to be scaled up. This places significant strain on teaching resources for programming courses for tasks such as grading of submitted programming assignments. Conventional attempts at automated grading of programming assignment rely on test-based grading which assigns scores based on the number of passing tests in a given test-suite. Since test-based grading may not adequately capture the student's understanding of the programming concepts needed to solve a programming task, we propose the notion of a concept graph which is essentially an abstracted control flow graph. Given the concept graphs extracted from a student's solution and a reference solution, we define concept graph matching and comparing of differing concepts. Our experiments on 1540 student submissions from a publicly available dataset show the efficacy of concept-based grading vis-a-vis test-based grading. Specifically, the concept based grading is (experimentally) shown to be closer to the grade manually assigned by the tutor. Apart from grading, the concept graph used by our approach is also useful for providing feedback to struggling students, as confirmed by our user study among tutors.

CCS CONCEPTS

• **Applied computing** → **Computer-assisted instruction**; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

automated grading, programming education, concept graph

ACM Reference Format:
Zhiyu Fan, Shin Hwei Tan, and Abhik Roychoudhury. 2023. Concept-Based Automated Grading of CS-1 Programming Assignments. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3597926.3598049

1 INTRODUCTION

There has been a growing interest in computer science education in recent years. Several education initiatives (e.g., Coursera, EdX, and Udacity) provide online courses that are taken by thousands of students all around the world. These online courses are known as Massive Open Online Courses (MOOC), which include many computer science courses that use programming assignments for assessing students' learning outcomes. With the increasing number of student enrollments, the number of submitted programming assignments also grows extensively throughout the year. This motivates the need for an automated grading system that can save the time and effort spent in grading these assignments. In this paper, we study the problem of automated grading of introductory programming assignments, which is common in first-year programming courses. There exist certain inherent difficulties in grading introductory programming assignments written by a novice programmer. Part of the difficulty comes from the fact that these programming attempts are significantly incorrect, often barely passing any tests [31]. Yet manual inspection of the code can reveal some degree of understanding of the problem by the student which should ideally be taken into account. Overall, the test-based automated grading may be too harsh for introductory programming assignments. In the K-12 computing education domain, promising results have been shown by using rubrics for grading assignments written in a visual programming environment to evaluate whether assignments produced by students demonstrate that they have learned certain algorithms and programming concepts [4]. Although grading based on rubrics provides a reliable way of assessing students' learning, the current rubric-based grading approach in most universities still relies either on manual grading or semi-automated grading [2], which may be too labor intensive for the instructors and tutors.

Existing approaches in automated programming assignment grading [14, 20, 27, 28] have several limitations. These approaches either (1) generate a patch for the incorrect student's submission as feedback or (2) produce binary (Correct/Incorrect) results via test-based grading, (3) only compare syntactic differences between instructors' reference solution and student solution (CFG-based grading). Although feedback in the form of patches can be useful for experienced developers or graders, prior studies show that novice students may not know how to effectively utilize the generated patches as hints, causing the increase of problem-solving time when patches are given [31]. Meanwhile, despite the widespread adoption of test-based grading approaches for online judges, the binary results provided by the test-based grading approaches may be too coarse-grained and may underestimate students' effort. CFG-based grading approach cannot distinguish the syntactically different but semantic equivalent implementation, which gives inaccurate results if the student's solution is syntactically different from instructors' reference solution. In education literature, *convergent formative assessment* (this kind of assessment "determines if the

```python
def remove_extras(lst):
  newlist = []
  for i in lst:
    if i not in newlist:
      newlist.append(i)


  return newlist
```

(a) A reference program

```python
def remove_extras(lst):
  new_lst = [lst[0]]
  for i in lst:
    if i in new_lst:
       continue
    else:
       new_lst += [i]
  return new_lst
```

(b) An incorrect student program

Test Inputs	Expected Outputs	Actual Outputs
[1, 1, 1, 2, 3]	[1, 2, 3]	[1, 2] ✗
[1, 5, 1, 1, 3, 2]	[1, 5, 3, 2]	[1, 5] ✗
[]	[]	IndexError ✗
[3, 4, 5, 1, 3])	[3, 4, 5, 1]	[3, 4] ✗

(c) Test cases and actual output of incorrect student program

Figure 1: Examples from the *Duplicate Elimination* assignment

learners knew, understood or could do a predetermined thing") has been shown to enhance student learning by evaluating if a student knows a concept [5, 23]. In contrast to formative assessment, current test-based grading approaches may be more suitable for *summative assessment* (aims to evaluate student learning) instead of improving learning.

In this paper, we present ConceptGrader, a novel automated grading approach that evaluates the correctness of students' conceptual understanding in their programming assignments to support convergent formative assessment.

Our key insight is that *introductory programming courses usually teach only a few concepts, and these concepts map well to the topics taught in the course syllabi.* To support convergent formative assessment, we introduce *concept graph*, a form of abstracted control-flow graph (CFG) where we (1) select some *important* (those that correspond to topics covered in the introductory programming course syllabi) nodes and edges of a CFG, and (2) translate the selected nodes/edges into natural-language like expressions (e.g., "insert *i* to *newlist*" in Figure 1 denotes the statement "newlist.append(i)"). ConceptGrader also introduces the idea of automated folding/unfolding of concept nodes for a more abstract level matching of concept graphs (Detail in Section 3.1). The proposed concept graph can be used for automated grading by calculating a score based on the differences between the concept graph for the reference solution and that for the incorrect solution. Such abstraction allows us to evaluate students' efforts from their comprehension to programming concepts.

Contributions of the paper. Overall, our contributions can be summarized as follows:

- We propose concept graph, an abstracted CFG that highlights programming concepts in submissions of introductory programming assignments. The concept graph contains expressions translated into natural language to enhance readability, and make it more suitable as hints to provide feedback to students. To allow more abstract matching of programs, we introduce concept node folding where we temporarily hide complex expressions in concept nodes for a fuzzy concept matching, and unfold (unhide) the expressions for precise concept matching whenever we detect a likely programming mistake within the folded concept node. Moreover, it can be used for automated grading to provide more accurate scores (with scores close to those given by manual grading) for introductory programming assignments.
- We present and implement ConceptGrader, a new automated grading approach that uses the differences between the student concept graph and reference concept graph to generate a score for a given incorrect student submission. The implementation is publicly available at *https://github.com/zhiyufan/conceptgrader*.
- We evaluate the effectiveness of ConceptGrader on 1540 student submissions from a publicly available dataset [16]. Our experiments show that compared to baselines (i.e. test-based approach and CFG-based approach), ConceptGrader performs better in terms of cosine similarity, root means squared error (RMSE), and mean absolute error (MAE) score.
- We also conduct a user study to assess the usefulness of the feedback produced by ConceptGrader. Our user study shows that ConceptGrader outperforms existing approaches by providing more useful feedback.

2 OVERVIEW

We give an overview of our concept-based automated grading approach by presenting a Python programming assignment for removing repeated elements in a list (*Duplicate Elimination*). Figure 1 shows the reference solution provided by the instructor, an incorrect solution submitted by a student, and a set of (input, output) pairs used to verify the correctness of each submission.

In the example in Figure 1, the student made two mistakes. First, instead of initializing an empty list, the student assumed that the input *lst* is not empty and initialized the new list with the given value at line 2. This incorrect assumption causes the third test case in Figure 1(c) to fail. Second, the student has an incorrect indentation of the return statement at line 8, which causes early termination of the program at the end of the second iteration and fails the other three test cases. As all test cases fail, a test-based grading approach will give the student a zero score for the submission. Compared to the tutor's manual inspection which gives 80% scores, the test-based grading approach underestimates the student's effort.

We now describe how we address the problem of inaccurate grading with a concept-based approach.

Concept Graph Abstraction. Given the reference and incorrect student program in Figure 1, we construct the control flow graph (CFG) for each program. For each CFG, we follow concept abstraction rules described in Section 3 to extract the programming concept represented by each basic block. Figure 2 shows the CFG and concept graph of the student program. The student program first declares a new list in block 1, we abstract it as *declare new_lst*. Then in block 2, the student uses a for-loop to iterate through elements in the input list *lst* in block 2, we use a concept *iterate i through lst* to show his/her understanding. In block 3, the reference program and student program use reverse conditions to check whether *i* exists in the previously declared *new_lst*. Although the operator is different and the exit edges point to the reverse direction, the two