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, USA.* ACM, New York, NY, USA, 12 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 learners knew, understood or could do a predetermined thing") has been shown to

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598049

```
1 def remove_extras(lst):
2   newlist = []
3   for i in lst:
4     if i not in newlist:
5       newlist.append(i)
6
7
8   return newlist
```

(a) A reference program

```
1 def remove_extras(lst):
2   new_lst = [lst[0]]
3   for i in lst:
4     if i in new_lst:
5       continue
6     else:
7       new_lst += [i]
8   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

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 **??**). 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.

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.

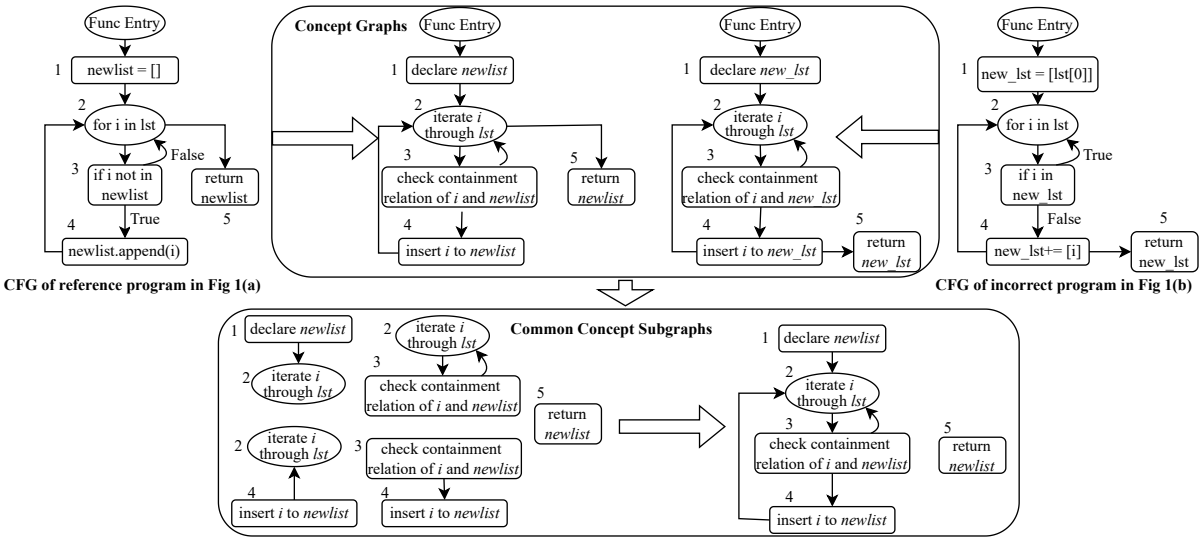
2 OVERVIEW

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 if-conditions represent the common idea of checking for an element in a list (represented by *containment relation of i and new_list*). At a high level, block 4 aims to insert an element into a list. One can perform the insertion in many ways, including invoke



(a) CFGs versus concept graphs (CGs) of the reference program and incorrect program listed in Fig 1.

(b) Score and feedback given by the three approaches for the incorrect solution in Fig 1.

Approach	Score	Feedback
Test-based	0/100	The solution passes 0/4 test cases
CFG-based	20/100	The solution makes mistakes in "new_lst =[lst[0]]", "i in new_lst", new_lst += [i]"
Concept-based	87/100	The solution makes mistakes in "declare new_lst" and "return new_lst"

Figure 2: Examples from the *Duplicate Elimination* assignment

built-in functions such as *append, extend, and insert* with different arguments, or directly use the list concatenation operator "+" to insert elements to end of a list. The student program in Figure 1 concatenates *new_lst* with *i*, while we abstract the statement as *insert i into new_list*. We construct the reference concept graph using a similar strategy. Compared to the student solution that uses *list concatenation* with augmented assignment "+=" at line 7, the reference program invokes the *append* method at line 5.

Concept Graph Matching and Grading. Before matching student concept graph and reference concept graph, we build a bijective variable naming relation of the two programs to avoid mismatch caused by different variable names (e.g., $\{newlist : new\_lst, i : i, lst : lst\}$) using dynamic execution approach proposed in [1, 16, 28]. Given reference concept graph CG_{ref} and student concept graph CG_{stu}, ConceptGrader searches for their common subgraphs. We first find a mapping for concept nodes in CG_{ref} and CG_{stu} if they represent the same concept. In the motivating example, the concept node matching result is $\{1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 3, 4 \rightarrow 4, 5 \rightarrow 5\}$. If there exists an edge $e_{ref} = (n_i, n_j)$ in CG_{ref}, and $e_{stu} = (n_k, n_l)$ in CG_{stu}, where n_i matches n_k and n_j matches n_l, we consider e_{ref} and e_{stu} as common edges. We derive all common edges in CG_{ref} and CG_{stu}, and construct common subgraphs.

The bottom subfigure in Figure 2 shows the common subgraphs of the reference program and the incorrect student program. To improve the accuracy of auto-grading based on concept graph matching, we employ an auto-folding and unfolding mechanism of concept graph to detect the differences between the reference program and the incorrect student program. As seen in Figure 2, although the student implements the *declaration* concept correctly, the way of *new_lst* initialization in concept node 1 is incorrect. Refer to Section 4.2 where we describe how we penalize this mistake via automated concept unfolding.

The concept node 5 in Figure 2 is disconnected from the common concept subgraph because the reference concept graph does not contain any edge from node 4 to node 5. The difference between the two concept graphs (the top subfigure in Figure 2) in the corresponding edges of concept node 5 helps ConceptGrader to identify the mistake at line 8 of Figure 1(b). Considering the mismatched edges of concept node 5 and the incorrect initialization of concept node 1 (we consider this as a partially matched node), ConceptGrader assigns 45 points for the matching concept nodes (4.5 nodes are matched out of 5 concept nodes in total), and 42 points for the matching concept edge (5 edges are matched out of 6 concept edges in total), leading to a total score of 87. Compared to the CFG-based approach where only the CFG nodes 2 and 5 are matched, ConceptGrader's score is more accurate.

3 PROGRAMMING CONCEPT ABSTRACTION

Programming topics and mini-Python. Our main insight to model the input Python programs is that although different institutions and online learning platforms offer a great variety of CS-1 introductory programming courses [8, 9, 22, 25], the programming topics taught in these courses are often the same. Specifically, the common topics covered in these courses include *Expressions* (e.g., arithmetic expressions), *Variables*, *Simple statement* (e.g., assignment), *Conditional* (*if*-statement), *Loops* (*for*-statement and *while*-statement),

Functions, *Lists*, and *Tuples*. Our goal is to design a concise representation of a Python program that models these programming topics. Our design is mainly based on: (1) the official Python 3 Abstract Syntax Grammar [11] that serves as a basis for our syntax rules, and (2) the key CS-1 programming topics that should be included in our concise representation. We select syntactic elements from the reference grammar that are also included in the common topics (i.e., we exclude advanced syntactic features such as lambda, yield, async and await expressions). Figure 3 shows the syntax for our Mini-Python grammar that supports programming topics studied in introductory Python courses. The grammar includes basic AST node types: Expression, Operators, and Statement. Each node type consists of multiple programming concepts.

Abstraction rules. Based on the mini-Python grammar in Figure 3, we derive a set of abstraction rules to translate the syntactic elements in the grammar. Table 1 shows our set of abstraction rules.

We follow two design principles for designing abstraction rules:
Human readable: We translate each syntactic element to natural language to generate human-readable feedback that can be used for explaining incorrect programming concepts. Figure 2(b) shows an example feedback generated by ConceptGrader.
Mitigating the program aliasing problem: We mitigate the program aliasing problem (i.e., semantically equivalent programs having several syntactically different forms [32]) by mapping several semantically equivalent syntactic elements to the same translation. For example, we translate $x = x + [i]$ and $x+ = [i]$ into "insert i to x". Mitigating the program aliasing problem helps in increasing the accuracy in matching two semantically equivalent concept nodes.

```
Expression    e ::= e boolop e | e op e | uop e | e cop e
                  | e (e, ..., e) | const | id
                  | {e : e, ..., e : e} | {e, ..., e} | [e, ..., e] | (e, ..., e)
                  | e[e] | e : e : e
Statement     s ::= e | s;s | e = e | e op = e
                  | for e in e : s | while e : s | if e : else : s
                  | continue | break | return e
BoolOp boolop ::= and | or
BinaryOp      op ::= + | - | * | / | % | **
UnaryOp       uop ::= ~ | not | UAdd | USub
CompareOp     cop ::= == | != | < | <= | > | >=
                  | is | is not | in | not in
```

Figure 3: Syntax of mini-Python based on the abstract Python grammar [11]

Concept graph construction. We first define the notion of a concept graph for a program. The nodes of the concept graph are called *concept nodes* which are defined as follows:

Definition 1 (Concept Node). *A concept node cn=$(c_1, c_2, \ldots, c_i)$ in a program p is a set of programming concepts, where each concept node cn is abstracted from a basic block $b = (s_1, s_2, \ldots, s_j)$ in control-flow graph of p that $\forall c \in cn, \exists s \in b(c \equiv f(s))$, where f is one of the abstraction rules in Table 1.*

Definition 2 (Concept Edge). *A concept edge ce=(cn_1, cn_2) in a program p is a transfer of control flow from a concept node cn_1 to cn_2.*

Table 1: Human-Readable Abstraction Rules

Rule Category	Sub-category	Example
Expression	BoolOp	x and y → logical relation of x and y
	BinOp	$x + y$ → arithmetic relation of x and y
	UnaryOp	not x → not x
	Compare	$x == y$ → equivalence relation of x and y
		$x > y$ → relational relation of x and y
		x in y → containment relation of x and y
	Call	$len(i)$ → call of len
		$x.append(i)$ → insert i to x
	Subscript	$x[2]$ → element of x
	Slice	$x[1:3]$ → subrange of x
Simple Statement	Assign	$x = y$ → declare x
		$x = []$ → reset x
		$x = x + 2$ → add x with constant
		$x = y + z$ → update x
		$x = x + [i]$ → insert i to x
	AugAssign	$x+ = 2$ → add x with constant
		$x+ = [i]$ → insert i to x
	Return	*return* expr → return abstract(expr)
Control Statement	If	*if* expr → check abstract(expr)
	For	*for* i *in* lst → iterate i through lst
	While	*while* x < y → iterate compare relation

Each concept edge ce is abstracted from the corresponding edge e in the control-flow graph of p. Specifically, our abstraction preserves the control flow transitions of e but removes the true/false label from the conditional edges of e.

Definition 3 (Concept Graph). *Let $CG(p) = (N, E)$ be the concept graph of p, $CG(p)$ is an abstracted graph of $CFG(p)$, where each node $n \in N$ represents a concept node, and each concept edge $e = (n_i, n_j) \in E$ corresponds to a possible transfer of control from concept n_i to concept n_j.*

Given a control-flow graph CFG of program p, we construct a concept graph CG of p by following Algorithm 1. For each edge $e = (b_{src}, b_{tgt}) \in CFG$, we abstract the source and target basic blocks separately, and re-construct a concept edge $ce = (c_{src}, c_{tgt})$. Given a basic block b as input, the *abstract(b)* procedure produces the concept node for b by traversing the AST of each statement to convert each statement to a programming concept. Starting from the parent node of each leaf node, ConceptGrader uses the corresponding abstraction rules from Table 1 (line 19 in Algorithm 1) for all non-leaf nodes via a bottom-up traversal until the AST node is the root node or it has been previously abstracted.

Abstracting concept edges. For each edge $e = (b_{src}, b_{tgt}) \in CFG$, we abstract the source and target basic blocks separately, and re-construct a concept edge $ce = (c_{src}, c_{tgt})$. In a traditional control flow graph, the edges are usually annotated with a label representing the conditional branches (e.g., "True" and "False" in the CFGs in Figure 2). In contrast, in a concept graph CG, we abstract away the true/false label but we still keep the actual predicates in a control flow edge e if the source node of e includes a conditional statement. This abstraction is based on our observation that students often implement conditional statements in various syntactically different but semantically equivalent ways.

Algorithm 1 Concept Graph Construction

Input: Control-flow graph of program *p CFG*
Output: Concept graph of program *p CG*

```
1:  procedure ConstructConceptGraph(CFG)
2:      Let CG be the concept graph
3:      for basic blocks (b_src, b_tgt) in CFG.edges() do
4:          c_src = abstract(b_src)
5:          c_tgt = abstract(b_tgt)
6:          CG.addEdge(c_src, c_tgt)
7:      return CG
8:  procedure abstract(b)
9:      Let cn be a concept node, visited be a list of abstracted AST nodes
10:     for stmt in basic block b do
11:         for l in getLeafASTNodes(stmt) do
            ▷ abstract non-leaf node
12:             pnode = getParent(l)
13:             c = abstractNode(pnode, visited)
14:             visited.add(pnode)
15:             cn.addConcept(c)
16:     return cn
17: procedure abstractNode(n, visited)
18:     Let c be a programming concept
19:     c = abstractRules(n)                    ▷ abstract using rules in Table 1
20:     if foldable(c) then
21:         c = fold(c)                         ▷ fold by hiding the content of node
22:     if isRootNode(n) or n ∈ visited then
23:         return c
24:     return abstractNode(getParent(n),visited)
```

Concept node folding. Automated source code folding is a technique that automatically creates a code summary by hiding unimportant code elements in a program that are not useful and helps developers to get an overview idea of the program on first viewing. It has shown promising results in the context of source code summarization to optimize the similarity between the code summary and the source code [12]. Inspired by the idea of source code folding in code summarization, we introduce the idea of *concept node folding* where we temporarily ignore part of the complex expressions in a concept node. The folded concept nodes are unfolded until an automated repair engine detects that patches exist for the folded nodes when matching reference and student concept graphs (refer to Section 4.2 for the details). Given the AST of the expressions in a statement, we define *concept depth* as the number of times that abstraction rules are applied such that the AST tree depth is compressed to 1. Given a $CG(p)$, we say a concept node to be *foldable* if the concept node it represents has *concept node depth>2*. Specifically, ConceptGrader hides the content of a node if it is foldable (lines 20–21 in Algorithm 1). By hiding the content of a concept node with complex expressions, our approach is essentially excluding parts of a complex expression during concept graph matching. In this case, concept folding helps us to abstract away the irrelevant differences (i.e., different but correct implementation) between the reference program and the student program. The folded concept node will be unfolded only when our approach detects that a difference in the reference program and student program is related to a fix in the corresponding concept node in the student program (the fix is generated by an automated program repair engine [16]).

Algorithm 2 Concept Graph Matching

Input: Reference concept graph G_r, Student concept graph G_s, Variable mapping of reference program and student program vM
Output: The matched subgraphs *subgraphs*

```
1:  procedure GraphMatching(G_r, G_s, vM)
    ▷ dict of matched concept nodes and list of edges
2:      nodeDict, edges = {}, []
3:      for n_s in G_s.nodes() do
4:          n_r = findNodeInGraph(G_r, n_s, vM)
            ▷ update dictionary for newly matched nodes
5:          if n_r ∉ nodeDict.val() then
6:              nodeDict[n_s] = n_r
7:      for (n_s, n_r) in nodeDict do
8:          N_s, N_r = listOfNeighbors(n_s), listOfNeighbors(n_r)
9:          N' = findMatchedNodes(N_s, N_r, nodeDict)
10:         edges.addEdges(n_s, N')
11:     subgraphs = merge(edges)
12:     return subgraphs
```

4 GRAPH MATCHING AND GRADING

Given concept graphs of reference solution and student solution CG_{ref} and CG_{stu}, we perform graph matching to assess how the intention of a student solution matches the reference solution at the concept level. Finding the maximum common subgraphs between two graphs is a NP-complete problem [6]. However, students' programs in CS-1 education context are often small. We find common subgraphs by iterating all common edges and then connect edges together via connected components to get all subgraphs. The graph matching algorithm consists of two phases. First, we construct a set of subgraphs to represent the common concepts of CG_{ref} and CG_{stu} based on their common concept nodes and edges. Second, we introduce the idea of an automated concept unfolding approach to distinguish the minor difference between two matched concept nodes to improve match accuracy further.

4.1 Concept Graph Matching

The goal of concept graph matching phase is to find an initial concept-matching relation of reference and student solution at a high level. For each concept node in the student concept graph, we find a concept node from the reference concept graph that (1) represents the same programming concepts category and (2) involves mapped variables of the student concept node in the abstraction.

Then, ConceptGrader identifies the neighbor nodes N_s and N_r for matched concept node pairs $(n_s : n_r)$ in *nodeDict* and finds matched nodes of N_s and N_r by checking nodes $n_s^{'} \in N_s$, whether $nodeDict[n_s^{'}] == n_r^{'}$ and $n_r^{'} \in N_r$. For all nodes $n_s^{'} \in N_s$ that satisfy the condition, we consider $e = (n_s, n_s^{'})$ as a matched edge and add it into list of matched edges *edges* (lines 7–10 in Algorithm 2). For each pair of edges (i.e., e_1=(src_1, dst_1) and e_2=(src_2, dst_2)) in *edges*, ConceptGrader then merges e_1 and e_2 into a subgraph if $src_1 == dst_2$ or $dst_1 == src_2$ (line 11 in Algorithm 2). Note that standalone concept nodes (e.g., concept node 5 in Figure 2) and edges might exist, which eventually lead to a set of subgraphs.

4.2 Automated Concept Unfolding

As mentioned in Algorithm 1, we construct the concept graph at a high level of abstraction and fold concept nodes to avoid exposing details, which allows more flexible matching. However, when we match the student concept graph with the reference concept graph, we may need to unfold certain concept nodes on-the-fly during the matching. This is because for the concept nodes in student program which contain mistakes, the folding process may mask those mistakes by showing only high-level concept.

In the example of Figure 2, the reference and incorrect student programs have the same programming concept *declare newlist* in folded student concept node cn_{s1}, but the specific values assigned to the two *newlist* variables are different. In this case, the folded concept node fails to capture the differences in terms of the declared values, so ConceptGrader assigns an overestimated score to the student program.

To assess student programs more precisely, the details of concept nodes with mistakes need to be explored by unfolding. We leverage program repair engine to get patches for each incorrect student program. Our intuition is that if a patch of the incorrect student program exists within a folded concept node *cn*, then this indicates that *cn* contains a programming mistake that needs to be fixed, but the mistake was hidden because of folding.

When a program repair engine detects a patch exists for a student concept node, the unfolding mechanism is triggered to expand the previously folded content of both the student concept node and the matching reference concept node. Then, ConceptGrader deducts scores by performing detailed matching of each mistake made in the student concept node.

Consider the example in Figure 1 where the program repair engine generates a patch $new\_lst = [lst[0]] \rightarrow new\_lst = []$ for the concept node *declare new_list*. As a patch exists within the concept node of the incorrect student program, ConceptGrader unfolds the concept node of incorrect program into {"declare new_list", "element of lst"}, while the corresponding concept node of the reference program still remains unchanged. Consider another example with incorrect control-flow transition in our motivating example (Line 8 in Figure 1(b)). Unfolding is not triggered in this example because the patch is meant for fixing a concept edge, and our approach in Algorithm 2 is able to detect this discrepancy by producing a separate concept subgraph with only one concept node ("*return newlist*" in Figure 2).

4.3 Concept Based Grading

Our goal is to compute score for each matching graph $G_{matched}$ between the student concept graph G_s and the reference concept graph G_r, so as to compute the total score for the student program. We calculate the score of a matching graph $G_{matched}$ by comparing the concept node similarity and concept edge similarity between G_s and G_r. Algorithm 3 shows the overall grading workflow. ConceptGrader first constructs concept graphs G_s for student program P_s and G_r for reference program P_r, then it matches G_s and G_r with the help of a variable mapping relation of P_s and P_r to get the list of matched subgraphs *matchedList* by following Section 4.1 (Lines 2–5). *runAPR* invokes program repair engines to generate *patches* for the incorrect student program, which involves automated concept unfolding as described in Section 4.2.

For each $G_{matched}$ in *matchedList*, we traverse all concept node *cn* and extract the folded concept node pair (cn_s, cn_r) representing student concept node cn_s and reference concept node cn_r. Then, ConceptGrader unfolds cn_s, cn_r to get detailed content if the automated program repair engine has produced patches for the corresponding cn_s (Lines 13–15). By comparing all concepts in cn_s and cn_r, we collect a list of concepts *matchC* that exist both in cn_s and cn_r, and $totalC$ that represents a list of all concepts in cn_r (Line 16).

Specifically, the score of a matching graph $G_{matched}$ consists of two parts: (1) average concept node similarity and (2) average concept edge similarity. Given $G_{matched}$, we define *average concept node similarity* of $G_{matched}$ as the average number of matching concept nodes in the reference concept graph G_r, calculated using the equation below:

$$conceptNodeSim(G_{matched}) = \frac{1}{nodeSize(G_r)} \sum_{i=1}^{n} \frac{matchC(cn_i)}{totalC(cn_i)}$$

where n denotes the number of matching concept nodes in $G_{matched}$, *matchC* represents concepts that exist in student concept node cn_s and reference concept node cn_r, $totalC$ denotes all concepts in cn_r, and nodeSize(G_r) denotes the number of nodes in G_r.

We define *average concept edge similarity* of $G_{matched}$ as the number of matching concept edge in the reference concept graph G_r, calculated using the equation below:

$$conceptEdgeSim(G_{matched}) = \frac{edgeSize(G_{matched})}{edgeSize(G_r)}$$

where $edgeSize(G)$ returns the number of edges in a graph G (e.g., edgeSize($G_{matched}$) denotes the number of edges in $G_{matched}$).

Finally, we compute the final score of the student program P_s as the sum of scores for all matched concept subgraphs $G_{matched}$ in between G_r and G_s. The equation is shown below:

$$score(P_s) = \frac{\alpha}{2} \times \sum_{i=1}^{m} (conceptNodeSim(G_i) + conceptEdgeSim(G_i))$$

In this equation, α represents the total score of the programming problem (usually determined by the instructor), m is the number of graphs in the list of matched subgraphs *matchedList* and G_i is a matched concept subgraph in *matchedList*.

Feedback generation. ConceptGrader generates feedback by pointing out (1) missing concepts, and (2) problematic concepts ("...makes mistakes..." in Figure 2). Specifically, ConceptGrader identifies *missing concepts* by checking if (1) the concept nodes exist in reference concept graph, but (2) a matching node cannot be found in student concept graph. ConceptGrader considers the matched student concept nodes as *problematic concepts* if (1) concept nodes exist in reference concept graph and have matching concept nodes in student concept graph, but the unfolding mechanism indicates that a programming mistake exists, or the transfer relations of the matched concept nodes are different (e.g., concept node 5 in Figure 2 is shown as a mistake in the generated feedback). Instead of providing feedback via patches (prior study show that novice students may not know how to effectively utilize the generated patches as hints [31]), our feedback highlights the wrong concepts

to promote active learning by asking "how can you fix the code here?".

Algorithm 3 Overall Grading Workflow

Input: Student program P_s, Reference program P_r, Test suite T, Total score α
Output: The final *score*

procedure GRADE(P_s, P_r, T, α)
  G_s = constructConceptGraph(P_s) ▷ Algorithm 1
  G_r = constructConceptGraph(P_r) ▷ Algorithm 1
  vM = variableMapping(P_s, P_r, T)
  $matchedList$ = graphMatching(G_r, G_s, vM) ▷ Algorithm 2
  $patches$ = runAPR(P_s, P_r, T)
  $score = 0$
  for $G_{matched}$ in $matchedList$ **do**
    $score$ += computeScore($G_{matched}, G_r$)
  return $\alpha \times score$
procedure COMPUTESCORE($G_{matched}, G_r$)
  for concept node cn in $G_{matched}.nodes$ **do**
    cn_s, cn_r = findConceptNode(cn)
    if hasPatches($patches, cn_s$) **then**
    ▷ unfold by expanding the content of node
      cn_s = unfold(cn_s)
      cn_r = unfold(cn_r)
    $matchC, totalC$ = compareConceptNode(cn_s, cn_r)
    $conceptNodeSim$ += $\frac{matchC}{nodeSize(G_r) \times totalC}$
  $conceptEdgeSim = edgeSize(G_{matched})/edgeSize(G_r)$
  return ($conceptNodeSim$ + $conceptEdgeSim$) / 2

5 EVALUATION

We evaluate ConceptGrader by addressing the following research questions:

RQ1: How does ConceptGrader perform in terms of grading accuracy, as compared to baseline approaches?
RQ2: How does test failure rate affect performance of ConceptGrader and baseline tools?
RQ3: What are the reasons for ConceptGrader's incorrect grading?

Implementation. We implemented the proposed approach in the tool ConceptGrader. We choose Refactory [16] as the automated program repair tool invoked during unfolding because it has shown promising results in fixing introductory assignments written in Python, particularly with respect to a reference correct solution. Similar to prior evaluations of approaches designed for programming assignments that sample additional reference solutions from correct students' submissions [1, 14, 16, 28], ConceptGrader follows the procedure of prior work [28] that selects five programs from correct students' submissions as additional reference solutions to mitigate the problem when students' implementation and reference's implementation use a different solving approach. To select five additional reference programs as representatives of most student programs, we (1) run Clara [14] to cluster correct student submissions, and (2) select one representative program from the top-5 clusters with most student programs. If an instructor's reference solution is the same as one of the five additional reference solutions, we select the instructor's reference solution and four other reference solutions. Then, ConceptGrader compares a student program against all reference solutions and selects the highest score as the final score. ConceptGrader currently supports programs with Python 3.10. We construct CFG using staticfg [7] and further customize it to build a concept graph.

Dataset. We evaluate ConceptGrader on five assignments from a CS-1 Python dataset used in the prior evaluation of introductory programming assignment [16]. For each programming assignment, the instructor prepared a reference solution and a test suite that evaluates students' correctness. Other datasets used in previous work [14, 28, 31] are either not publicly available or use different programming languages (e.g., C). Our concept abstraction rules currently do not support all programming language constructs (e.g., lambda expression). We exclude submissions with unsupported features, and trivial student programs without any real implementation (i.e., programs with less than three lines of code) from our evaluation. In total, we have 1540 incorrect submissions remaining.

Ground truth construction. The Refactory dataset [16] uses execution results of tests as feedback to students, and it does not have the *ground truth score* (the correct score to be assigned for an assignment) for each submission. We invited eight senior Computer Science undergraduate students who have experience working as teaching assistants to be the annotators for grading those incorrect submissions. We provide the annotator with the problem description, instructors' reference solution, and instructors' test suite. We asked annotators to run test cases and grade the submissions by functionality. To provide freedom in grading, we did not mention any other steps (e.g., using the execution results of tests).

To mitigate potential grading bias, annotators are unaware of the existence of ConceptGrader, and each submission is graded by two annotators, and we asked each annotator to grade each submission out of the same total score of 100 (α=100 in the last equation in Section 4.3). If the scores given by the two annotators differ less than 10, we take their average as the ground truth score. Otherwise, a third annotator participates in the discussion until they reach a consensus. We use the same ground truth for evaluating RQ1–RQ3.

Baselines. We compare ConceptGrader against two baselines: (1) test-based approach [10, 15, 17, 18, 26, 29], and (2) CFG-based approach [27]. We compare with the test-based approach because it is the most widely used approach. To ensure fair comparison, we provide the same set of test cases to the test-based approach and the program repair engine used in our unfolding (described in Section 4.2). We do not compare ConceptGrader against the recent CFG-based automated grading technique [27] because (1) its implementation targets C programming assignments where ConceptGrader focuses and is evaluated on Python programming assignments, and (2) their approach can only be used to grade correct programs (passing all tests) as we confirmed with the authors, where ConceptGrader focuses more on evaluating incorrect programs. To ensure a fair comparison, we implemented a CFG-based baseline by removing concept abstraction, concept graph construction, and concept folding/unfolding from ConceptGrader (i.e., we

keep the variable mapping to allow CFG-based baseline to handle different naming styles). Moreover, we do not compare with AutoGrader because it only returns a binary correct/incorrect based on path deviation [20].

The goal of automated grading is to automatically assign a score for a student program such that tutors can directly accept it or minimally adjust it. We use three metrics: (1) Cos-sim (cosine similarity), (2) RMSE (root means squared error), and (3) MAE (mean absolute error) to evaluate the distance between auto-generated scores and tutors' ground truth scores. Given the auto-generated and ground-truth scores for all incorrect student programs, Cos-sim evaluates their cosine similarity in the normalized vector space.

Given the ground truth score y_i, the auto-generated score $\hat{y_i}$, and N samples, their equations are: $Cos{-}sim(y, \hat{y}) = \frac{y \cdot \hat{y}}{|y||\hat{y}|}$, $RMSE(y, \hat{y}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y_i} - y_i)^2}$, and $MAE(y, \hat{y}) = \sum_{i=1}^{N}|y_i - \hat{y_i}|$

RMSE and MAE are often used to evaluate the differences between values predicted by a model [30]. They represent the absolute closeness of auto-generated scores and ground-truth scores by computing their standard deviation and absolute distance. Lower RMSE and MAE values indicate better performance.

5.1 RQ1: Overall Grading Accuracy

Table 2 shows the results of all incorrect student programs from the five selected assignments [16]. On average, ConceptGrader outperforms all baseline approaches by achieving Cos-sim at 0.92, whereas test-based approach and CFG-based approach achieve 0.81 and 0.79 for Cos-sim, respectively. Compared to the two baseline approaches, ConceptGrader produces the lowest RMSE (30.41) and MAE (22.93) values, which improves the result of test-based approach by 30% and 29% and the result of CFG-based approach by 41% and 49%, in terms of RMSE and MAE. The low average RMSE and MAE values indicate that ConceptGrader is effective in predicting the ground truth scores for programming assignments.

It is worthwhile to mention that the grading accuracy of test-based approach in *Sequential search* is better than the other four assignments. We analyzed the reason for the higher accuracy of test-based approach in "Sequential search" task. Table 3 shows that around 59.3% (323/544) student programs pass more than 75% of test cases in "Sequential search", whereas the ratio on the other four assignments is 19.0% on average. Passing more test cases often indicates better quality of a program. When a program passes majority of test cases, tutors also tend to assign relatively high scores. However, if a program fails majority of test cases, it does not necessarily mean the program is completely incorrect because even a subtle mistake can cause different behavior.

Average time taken. In terms of the average time taken to generate a score for a student program, test-based grading is the fastest as it only requires running the student program against all test cases. ConceptGrader is slower than other approaches because it may need to invoke program repair engine several times to generate patches for concept unfolding in the final grading process. Overall, the average time taken 35.49s is acceptable as prior study shows that human tutors often take 100 seconds to grade one student submission [31].

Effectiveness of concept abstraction and concept unfolding. Although ConceptGrader shows better grading accuracy compared to the two baseline approaches, it is worthwhile to investigate the effectiveness of each component in ConceptGrader. We implemented another version of ConceptGrader denoted as *CG-wo-f* by removing concepts unfolding (described in Section 4.2). We first compare *CFG* and *CG-wo-f* to show the impact of automated concept abstraction. The difference between *CFG* and *CG-wo-f* is that *CFG* matches CFG of student's program and CFG of reference program by comparing the source code in basic blocks, whereas *CG-wo-f* first applies the abstraction rules in Table 1 for basic blocks in CFG of student's program and CFG of reference program to construct corresponding concept graphs, then matches nodes in student concept graph and reference concept graph. Table 2 shows that with concept abstraction and concept graphs, *CG-wo-f* improves Cos-sim over CFG-based approach by 0.12, and reduces RMSE and MAE over CFG-based approach by 19.59 (38.9%) and 18.96 (42.3%) respectively. In addition, *CG-wo-f* has almost no overhead regarding time taken to grade a student program (average time taken is 5.94s).

Based on *CG-wo-f*, *CG* takes advantage of patches generated by automated program repair engine (Refactory) as hints to identify students' mistakes that have been abstracted in automated concept folding process, and unfolds those students' concepts to compare in detail to capture those minor mistakes. The comparison between *CG* and *CG-wo-f* shows the impact of automated concept unfolding. Overall, the result of Cos-sim does not change much. This is because ConceptGrader with folding and unfolding is a fine-tuning procedure. When a folded concept node in student concept graph finds a matching in reference concept graph but Refactory reveals that a patch is required for the concept node, ConceptGrader still assigns partial scores based on the coverage of matching unfolded concepts in the concept node. The usefulness of folding and unfolding is shown by the lower RMSE and MAE values (i.e., the values improved by 5.4% and 11.3%, respectively). This means that *CG*'s grading has less discrepancy with respect to the tutors' ground truth compared to *CG-wo-f*.

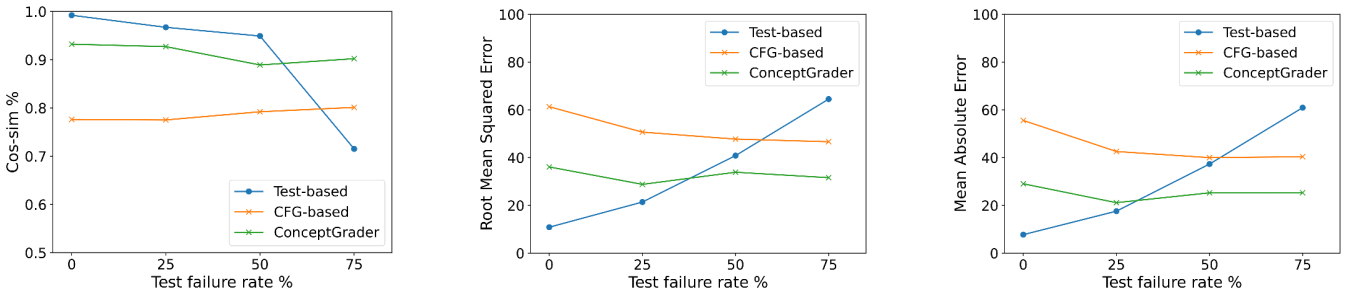
5.2 RQ2: Relation with Test Failure Rate

Our intuition of designing ConceptGrader is that in introductory programming assignments, even a simple mistake could fail many tests within the test suite, which leads to a test-based grading approach that may underestimate students' understanding and effort. As Table 2 shows the overall grading accuracy of each approach on all incorrect student programs, we are also interested in investigating the effect of *test failure rate* (percentage of failing tests in the entire test suite for an assignment) on grading accuracy. We divide all incorrect student submissions (we consider a student submission as incorrect if test failure rate > 0) into four groups based on their test failure rate (0%–25%, 25%–50%, 50%–75%, and 75%–100%). Figure 4 shows the grading accuracy of all four approaches regarding different test failure rates.

For all the evaluated metrics (Figures 4(a), 4(b), and 4(c)), our results show that when test failure rate is low (0–50%), test-based grading tends to be more effective. However, as test failure rate increases, the performance of test-based grading downgrades. In contrast, the performance of ConceptGrader tends to be stable as the

Table 2: Automated grading results of four approaches for incorrect student submissions on five assignments from the Refactory dataset[16]. The columns *Test* and *CFG* denote test-based grading and CFG-based grading, and *CG* and *CG-wo-f* show ConceptGrader and ConceptGrader without concept unfolding. The column "# of Inc. Sub." shows the number of incorrect submissions for each assignment, whereas the column "# of TC" denotes the number of test cases for each assignment, the column LoC represents the line of code. The columns "Cos-sim", "RMSE", and "MAE" represent the cosine similarity, root means squared error, and mean absolute error between automatically generated and ground truth scores. The columns "Average Time Taken (s)" denotes the average time taken in seconds to produce the score and feedback for a student submission in each assignment. We highlight the best result in bold.

Assignment	# of Inc. Sub.	# of TC	LoC	Cos-sim				RMSE				MAE				Average Time Taken (s)			
				Test	CFG	CG-wo-f	CG	Test	CFG	CG-wo-f	CG	Test	CFG	CG-wo-f	CG	Test	CFG	CG-wo-f	CG
Sequential search	544	11	10	**0.95**	0.79	0.93	0.94	**26.70**	56.91	32.19	30.26	**17.41**	51.03	25.03	23.62	4.18	**1.22**	1.23	24.84
Unique dates/months	353	17	28	0.84	0.80	0.85	**0.87**	25.21	27.54	23.25	**22.53**	17.59	21.91	18.59	**16.84**	**6.91**	8.37	9.28	35.19
Duplicate elimination	272	4	7	0.69	0.80	0.85	**0.88**	64.41	50.91	38.73	**35.18**	60.39	43.33	30.97	**28.05**	**4.07**	7.85	7.87	37.52
Sorting tuples	263	6	9	0.68	0.81	0.92	**0.93**	49.58	41.00	26.38	**23.12**	43.77	34.62	20.82	**18.35**	**3.97**	10.35	9.00	44.28
Top-k elements	108	5	11	0.62	0.74	**0.90**	**0.90**	63.41	53.69	34.92	**32.31**	58.70	46.66	28.96	**26.11**	**5.07**	18.58	17.69	63.70
Total/Average	1540	9	14	0.81	0.79	0.91	**0.92**	42.96	51.73	32.14	**30.41**	32.56	44.82	25.86	**22.93**	**4.89**	6.31	5.94	35.49



(a) Average Cos-sim **(b) Average RMSE** **(c) Average MAE**

Figure 4: Average grading performance of all incorrect student submissions across different test failure rates.

Table 3: The test failure rate distribution for evaluated submissions.

Assignment	# Incorrect Submissions	# of TC	Test failure rate (%)			
			0–25	25–50	50–75	75–100
Sequential search	544	11	323	91	59	71
Unique dates/months	353	17	159	54	29	111
Duplicate elimination	272	4	14	9	29	220
Sorting tuples	263	6	10	23	82	148
Top-k elements	108	5	6	3	6	93
Total/Average	1540	9	512	180	205	643

test failure rate increase. This result indicates that ConceptGrader preserves the ability to capture students' misunderstanding, not affected by the changes in test failure rate. Considering the fact that most students' programs fail half of test cases, it illustrates the importance of a concept-based grading approach.

5.3 RQ3: Limitations of ConceptGrader

To understand the limitations of ConceptGrader, we manually analyzed (1) the cases where ConceptGrader performs worse than test-based approach, (2) the quality of reference solutions to ConceptGrader.

Analyzing Unreasonable Scores: To reduce the manual effort in analyzing cases where the differences between the scores given by a test-based s_{test} and those assigned by ConceptGrader $s_{concept}$ are minor, we only analyze cases where the scores given by ConceptGrader is *unreasonable* (i.e., the difference between $s_{concept}$ and s_{test} is greater than 5 points). In total, we observed 565/1540 (36.7%) scores to be unreasonable.

Our manual analysis of the 565 unreasonable scores shows that unreasonable scores occur due to: (1) syntactically different student implementations, and (2) inaccurate variable mapping. Specifically, although we design the abstraction rules to mitigate the program aliasing problem by translating different programs to the same representation, our rules are not exhaustive so ConceptGrader fails to match correctly when the incorrect student programs are substantially different from the reference solutions, especially for cases where the test failure rate is low. When the students' programs use sub-optimal algorithms or syntactically different implementations, ConceptGrader could not match the concept nodes and edges accurately, resulting in a lower score assigned to the incorrect student programs. Meanwhile, as ConceptGrader relies on the variable mapping mechanism of Refactory [16], we observe that ConceptGrader may produce inaccurate scores when the student programs use too many temporary variables which increase the number of un-mapped variables in the variable mapping.

In the future, a hybrid automated grading tool that combines ConceptGrader and test-based approach may be interesting to be explored. Using the AST edit distance between student program and reference program as estimator of the quality of student program, ConceptGrader suggests scores when AST edit distance is small but

test failure rate is high, while test-based approach can still be used when test failure rate is low, but AST edit distance is high (indicating there is no good reference solution for the student program).

Impact of Different Numbers of Reference Solutions: In previous sections, we conducted experiments for ConceptGrader using multiple reference solutions. Although using multiple reference solutions from correct student solutions is a recent trend in other relevant work [1, 14, 16, 28], there may not be sufficient high-quality reference solutions available for each programming assignment in practice. To address this concern, we analyze the impact of different numbers of reference solutions to ConceptGrader by grading with fewer reference solutions. Given the five reference solutions crafted in Section 5, we gradually remove the reference solutions being used by ConceptGrader, starting from the most less popular reference solution, until there is only instructors' provided reference solution. Table 4 shows the average results for the five programming assignments as we reduce the number of reference solutions used in ConceptGrader. From Table 2 and Table 4, we can observe that using only two reference solutions, ConceptGrader already performs better than test-based approach. Compared to only one reference solution, the performance of ConceptGrader with three reference solutions increases by 7% for Cos-sim, 21.1% for RMSE, and 23.6% for MAE, which reaches a comparative level of the default configuration, using all reference solutions (# of Ref. Solutions=5).

Table 4: The impact of different number of reference solutions in ConceptGrader (CG) and ConceptGrader without unfolding (CG-wo-f).

# of Ref.	Cos-sim		RMSE		MAE	
Solutions	CG	CG-wo-f	CG	CG-wo-f	CG	CG-wo-f
5	0.92	0.91	30.41	32.14	22.93	25.86
4	0.91	0.90	32.05	33.82	23.76	27.05
3	0.91	0.88	33.28	34.97	26.82	28.53
2	0.89	0.85	36.42	37.51	30.46	31.68
1	0.85	0.82	42.23	45.83	35.11	37.32

6 USER SURVEY

User Survey Setup. To obtain qualitative data for demonstrating the effectiveness of ConceptGrader, we conducted a survey among 29 tutors from two semesters of a large CS-1 introductory programming course. The tutors include both lab instructors who taught lab sessions and graders who grade programming assignments. All tutors are undergraduates who have taken the course in previous semesters from the Computer Science department. Among the 29 tutors with whom we have shared the survey, we received 16 replies. Participation in the survey is voluntary, and the authors do not have any personal connection with the participants. To reduce bias due to personal preference towards a particular approach, we anonymize the name of each approach. The survey aims to collect tutors' opinions on the grade and feedback generated by three automated approaches. In total, it contains ten incorrect student submissions drawn from five assignments.

User Study Questions. Each tutor answers a question about prior teaching experience. On average, the 16 tutors have served as tutors 2.1 times. We randomly sampled two incorrect submissions for each assignment from the evaluated dataset (in Section 5). For each incorrect submission, we provide (1) the instructor's reference solution, (2) an assignment description, (3) test cases, and (4) the questions below:

Q1. Rate the quality of the automated mark in terms of assessing students' understanding and effort.
Q2. Rate the usefulness of automated feedback to students in terms of improving their learning outcome, based on your previous learning experience.
Q3. To what extent will the automated feedback be preferable or as good as the feedback that you would manually give?

The first question (Q1) aims to assess the quality of the generated score, whereas Q2 and Q3 are designed to assess the quality of the generated feedback (Figure 2(b) shows an example of the generated feedback). For each incorrect student submission, participants need to rate each item based on a five-point Likert scale (with 1 being very low and 5 being very high). We allocate 30 minutes for each tutor to complete the survey.

User Study Results. Figure 5 presents the results of the 16 tutors' ratings for all user study questions. Overall, we observe that the tutors show a positive attitude of ConceptGrader for all questions (Q1 – Q3) with a mean rating of 3.8. Tutors rate highest (average rating of 3.7) for the quality of ConceptGrader's generated scores (Q1), compared to test-based and CFG-based approaches (average rating of 1.3 and 3.2, respectively). As those who are invited for the ground truth constructions are different from tutors for the user study, this further confirms our grading accuracy experiment in Section 5. For the usefulness of automated feedback in terms of improving students' learning outcomes (Q2), tutors think that ConceptGrader is the most useful among all approaches (average rating of 3.9). This indicates that our approach provides better support for convergent formative assessment. Compared to the baseline approaches, tutors prefer the quality of the feedback generated by ConceptGrader (average rating for Q3 is 3.8). This shows that the feedback constructed via our human-readable abstraction rules may benefit tutors in designing personalized feedback for students.

Significance of study result. To validate the significance of our study result, we performed a two-tailed T-test for the difference between the results for CFG-based approach and ConceptGrader. The result shows that our study has a p-value < 0.001 for Q1 to Q3, indicating that the difference between CFG-based approach and ConceptGrader in Figure 5 is statistically significant. Moreover, the standard deviation of CFG-based approach and ConceptGrader for Q1 to Q3 is (0.84, 0.77, 0.89) and (0.93, 0.76, 0.86) respectively.

7 RELATED WORK

Automated Grading. Many approaches have been proposed for automatic assessment of programming assignments [10, 15, 18, 20, 26, 29]. These systems rely either on (1) test cases [10, 15, 17, 18, 26, 29], (2) formal semantics [20], or (3) syntactic differences (e.g., in the form of CFG) between reference solution and student solution [3, 21, 27] to grade introductory programming assignments. Test-based grading approaches (e.g., AutoGrader [10]) assign scores to programming assignments by relying on program's outputs on

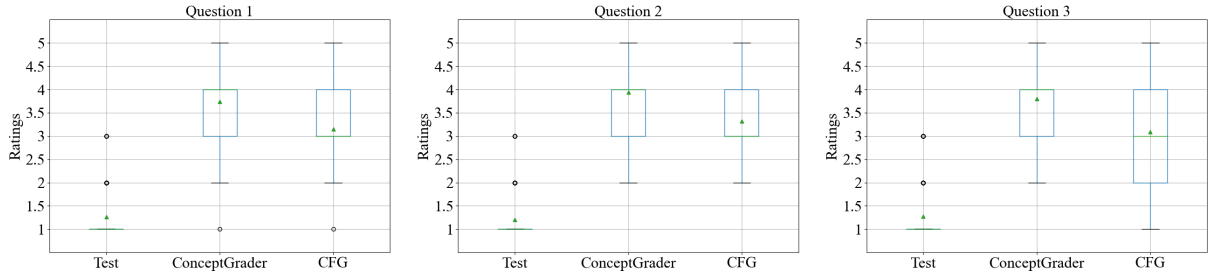


Figure 5: The boxplot of average rating of all user study questions. Green triangle represents the mean value. The whiskers denote the minimum/maximum value, and the rectangle denotes the first/third quartile.

a set of test inputs [18, 26, 29]. These test inputs can either be manually designed by course instructors or automatically generated [13, 20]. These test-based approaches have several limitations, including: (1) they cannot reflect the students' effort and mastery of knowledge because a minor mistake could fail many test cases which lead to a large portion of marks being deducted, (2) students may struggle to identify their mistake using only the failing test cases as feedback. Different from these approaches, ConceptGrader grades a student's submission using programming concepts, and generate intuitive feedback which points out the missing or wrongly used programming concepts. Another popular approach to automated grading is calculating the similarity between different program representations (e.g., control flow graph) of a student's submission and corresponding reference implementation [3, 21, 27]. However, these approaches do not support convergent formative assessment as they only grade student's submissions without providing feedback. Liu et al. [20] proposed an approach based on formal semantics for automated grading of programming assignments. They use symbolic execution techniques to explore the semantic difference between instructor's reference solution and students solution in the form of path deviations. However, their approach only produces a binary correct or incorrect result, while our approach gives a quantitative evaluation of student submissions.

Repair of Introductory Programming Assignments. Prior work focuses on fixing introductory programming assignments using automated program repair techniques, and providing the automatically patches as feedback [14, 16, 19, 24, 28, 31]. Although ConceptGrader uses patches generated by a program repair engine (Refactory [16]) to trigger the unfolding mechanism to improve the accuracy of the score calculation for incorrect student submissions, ConceptGrader does not use the automatically generated patches directly as feedback. As we design the abstraction rules to be human readable, the feedback generated by ConceptGrader can provide explanation of student mistakes to support convergent formative assessment. Different from existing repair approaches, ConceptGrader is designed for automated grading of introductory programming assignments.

8 THREATS TO VALIDITY

External. Our findings of programming concepts focus on Python introductory programming courses. Hence, our experiments may not be exhaustive and generalize to other languages. We evaluate and implement ConceptGrader within the scope of mini-python, ConceptGrader may produce inaccurate scores if programming assignments include language features beyond mini-python. We left the extension of more advanced programming features in Python as future work. (i.e., it does not currently support advanced programming topics such as lambda expression). ConceptGrader may produce inaccurate concept matching if the student program and reference solution solve a programming problem with different algorithms. We mitigate this by following previous work [1, 14, 16, 28] to include correct students' programs (i.e., student submissions that pass all test cases) as additional reference solutions.

Internal. Our code and automated scripts may have bugs that can affect our reported results. To mitigate this threat, we have made our tool and data publicly available. Our implementation of the CFG-based approach [27] may not be as effective as the original implementation for C programs. Nevertheless, as our concept graph uses the same CFG as basis for abstraction, our evaluation that compares the CFG approach and the abstracted CFG (our concept graph) ensures fair comparison of the two approaches.

9 CONCLUSION

We propose ConceptGrader, an automated grading approach for programming assignments to assess students' understanding via programming concepts. We derive programming concepts from common programming topics in first-year programming courses, and design a concept graph that abstracts incorrect student program and reference solution. Such an abstract representation allows us to identify students' misunderstanding of a specific problem, so as to generate reasonable scores to reduce tutors' workload and improve students' learning outcomes through convergent formative assessment. Compared to test-based automated grading and CFG-based automated grading, our evaluation shows that the scores generated by ConceptGrader are more accurate in terms of cosine similarity, RMSE, and MAE. Our user study among tutors also shows that the automated generated scores and feedback can help tutors in constructing their manual feedback that eventually assists students in rectifying their mistakes. In the future, we plan to extend extend ConceptGrader to handle more advanced programming features. We also plan to integrate ConceptGrader into an intelligent tutoring system and deploy it for live interactive programming teaching.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001.

REFERENCES

[1] Umair Z Ahmed, Zhiyu Fan, Jooyong Yi, Omar I Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified repair of programming assignments. *ACM Transactions on Software Engineering and Methodology* (2022).

[2] Tuukka Ahoniemi and Tommi Reinikainen. 2006. ALOHA-a grading tool for semi-automatic assessment of mass programming courses. In *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*. 139–140.

[3] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. 2004. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research* 3, 1 (2004), 245–262.

[4] Nathalia da Cruz Alves, Christiane Gresse von Wangenheim, Jean Carlo Rossa Hauck, and Adriano Ferreti Borgatto. 2020. A large-scale evaluation of a rubric for the automatic assessment of algorithms and programming concepts. In *Proceedings of the 51st ACM technical symposium on computer science education*. 556–562.

[5] Beverley Bell, Nigel Bell, and B Cowie. 2001. *Formative assessment and science education*. Vol. 12. Springer Science & Business Media.

[6] Horst Bunke, Pasquale Foggia, Corrado Guidobaldi, Carlo Sansone, and Mario Vento. 2002. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, 123–132.

[7] coetaur0. 2019. Python3 control flow graph generator. https://github.com/coetaur0/staticfg. Accessed: 2022-10-09.

[8] Berkeley EECS. [n. d.]. CS 9H: Python for Programmers. https://selfpaced.bitbucket.io/#/python/calendar.

[9] Stanford Engineering. [n. d.]. CS106A - Programming Methodology. https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1206/schedule.html.

[10] Computer Science for ALL Students. 2022. AutoGradr. https://www.csforall.org/members/autogradr_automated_grading_for_programming_assignments/. Accessed: 2020-10-06.

[11] Python Software Foundation. 2022. Abstract Syntax Trees. https://docs.python.org/3/library/ast.html.

[12] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2017. Autofolding for source code summarization. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1095–1109.

[13] Liang Gong. 2014. Auto-grading dynamic programming language assignments. *University of California, Berkeley, Tech. Rep* (2014).

[14] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices* 53, 4 (2018), 465–480.

[15] Jan B Hext and JW Winings. 1969. An automatic grading scheme for simple programming exercises. *Commun. ACM* 12, 5 (1969), 272–275.

[16] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based Program Repair applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 388–398.

[17] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*. 86–93.

[18] Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The boss online submission and assessment system. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 2–es.

[19] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 739–750.

[20] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic grading of programming assignments: an approach based on formal semantics. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 126–137.

[21] Greg Michaelson. 1996. Automatic analysis of functional program style. In *Software Engineering Conference, Australian*. IEEE Computer Society, 38–38.

[22] MIT OpenCourseWare. [n. d.]. 6.0001 Introduction to Computer Science and Programming in Python. https://ocw.mit.edu/courses/.

[23] John Pryor and Barbara Crossouard. 2008. A socio-cultural theorisation of formative assessment. *Oxford review of Education* 34, 1 (2008), 1–20.

[24] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.

[25] Carnegie Mellon University. [n. d.]. CMU 15-122 Fundamentals of Programming and Computer Science. https://www.cs.cmu.edu/~112/schedule.html.

[26] Urs Von Matt. 1994. Kassandra: the automatic grading system. *ACM SIGCUE Outlook* 22, 1 (1994), 26–40.

[27] Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and Viktor Kuncak. 2013. Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology* 55, 6 (2013), 1004–1016.

[28] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 481–495.

[29] Michael Wick, Daniel Stevenson, and Paul Wagner. 2005. Using testing and JUnit across the curriculum. *ACM SIGCSE Bulletin* 37, 1 (2005), 236–240.

[30] Wikipedia contributors. 2022. Root-mean-square deviation — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Root-mean-square_deviation&oldid=1117272661 [Online; accessed 10-November-2022].

[31] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 740–751.

[32] Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. SemRegex: A semantics-based approach for generating regular expressions from natural language specifications. In *Proceedings of the 2018 conference on empirical methods in natural language processing*.

Received 2023-02-16; accepted 2023-05-03