Flexible Program Alignment to Deliver Data-Driven Feedback to Novice Programmers*

Victor J. Marin, Maheen Riaz Contractor, and Carlos R. Rivero

Rochester Institute of Technology
vxm4964@rit.edu, mc1927@rit.edu, crr@cs.rit.edu

Abstract. Supporting novice programming learners at scale has become a necessity. Such a support generally consists of delivering automated feedback on what and why learners did incorrectly. Existing approaches cast the problem as automatically repairing learners' incorrect programs; specifically, data-driven approaches assume there exists a correct program provided by other learner that can be extrapolated to repair an incorrect program. Unfortunately, their repair potential, i.e., their capability of providing feedback, is hindered by how they compare programs. In this paper, we propose a flexible program alignment based on program dependence graphs, which we enrich with semantic information extracted from the programs, i.e., operations and calls. Having a correct and an incorrect graphs, we exploit approximate graph alignment to find correspondences at the statement level between them. Each correspondence has a similarity attached to it that reflects the matching affinity between two statements based on topology (control and data flow information) and semantics (operations and calls). Repair suggestions are discovered based on this similarity. We evaluate our flexible approach with respect to rigid schemes over correct and incorrect programs belonging to nine real-world introductory programming assignments. We show that our flexible program alignment is feasible in practice, achieves better performance than rigid program comparisons, and is more resilient when limiting the number of available correct programs.

Keywords: Automated Program Repair · Data-driven Feedback.

1 Introduction

The recent worldwide interest in computer science has originated an unprecedented growth in the number of novice programming learners in both traditional and online settings [2, 14, 18, 21]. A main challenge is supporting novice programming learners at scale [5, 10, 17], which typically consists of delivering feedback explaining what and why they did incorrectly in their programs [7]. Different than traditional settings, online programming settings often have a large proportion of novice learners with a variety of backgrounds, who usually tend to

* This material is based upon work supported by the National Science Foundation under Grant No. 1915404.

need a more direct level of feedback and assistance [1]. A common practice is to rely on functional tests; however, feedback generated based solely on test cases does not sufficiently support novice learners [3, 9, 17].

Current approaches cast the problem of delivering feedback to novices at scale as automatically repairing their incorrect programs [3, 12, 13, 15, 17, 18]. Note that, similar to existing approaches, we consider a program to be correct if it passes a number of predefined test cases [3, 18]; otherwise, it is incorrect. Once a repair is found, it can be used to determine pieces of feedback to deliver to learners [17]. Non-data-driven approaches aim to find repairs by mutating incorrect programs until they are correct, i.e., they pass all test cases [11]. Data-driven approaches exploit the fact that repairs can be found in existing correct programs and extrapolated to a given incorrect program [18]. This paper focuses on the latter since, in a given programming assignment, there is usually a variety of correct programs provided by other learners that can be exploited to repair incorrect programs [3, 13, 15, 18].

The "search, align and repair" [18] framework consists of the following steps: 1) Given an incorrect program p_i, search for a correct program p_c that may be useful to repair p_i; 2) Align p_i with respect to p_c to identify discrepancies and potential modifications in order to repair p_i; and 3) Apply those modifications to p_i until the resulting program p'_i passes all test cases. Current approaches instantiating the "search, align and repair" framework use rigid comparisons to align incorrect and correct programs, i.e., they require the programs to have the same or very similar control flows (conditions and loops), and they are affected by the order of program statements [3, 15, 18, 13]. As a result, such approaches may miss a potentially valuable set of correct programs that can repair incorrect programs using flexible program comparisons.

In this paper, we explore a new alignment step that relies on a flexible program comparison based on approximate graph alignment of program dependence graphs. On one hand, a program dependence graph combines information about the control and the data (use of variables) flows of a program [4]. On the other hand, approximate graph alignment finds a correspondence between the nodes of two graphs [8]. Such a correspondence takes both topology and semantics of the graphs into account. When applied over program dependence graphs, topology implies programs that approximately match with respect to their control and data flow information, while semantics are modeled as operations and calls performed in the programs. Each pair that belongs to an alignment has a node similarity associated to it. Replacement suggestions are computed as those pairs whose similarities substantially deviate from the average similarity of a given alignment. Furthermore, addition and removal suggestions are those non-aligned nodes that are connected to replacement suggestions.

We evaluate our alignment step using nine real-world introductory programming assignments from a popular online programming judge (CodeChef). We collected publicly-available correct and incorrect programs in these assignments from real-world learners. We compare our flexible alignment with respect to existing rigid alignments: CLARA [3], Refazer [15], and Sarfgen [18]. For a fair comparison, we evaluate program comparison schemes using a common repair framework. We use different scenarios in which we vary the number of correct programs available. We show that our flexible program comparison achieves a better repair performance than other rigid program comparisons, and that it is more resilient to provide repairs when the number of correct programs available is reduced.

The paper is organized as follows: Section 2 summarizes previous approaches and ours; Section 3 presents background information; Section 4 describes our flexible comparison; Section 5 discusses our experimental results; Section 6 presents the related work; and Section 7 recaps our conclusions and future work.

2 Overview

We consider CLARA [3], Refazer [15], Sarfgen [18], and sk_p [13] the state of the art in searching, aligning and repairing programs. CLARA and Sarfgen compare variable traces between an incorrect and a correct programs that share the same control statements like `if` or `while`. Refazer uses pairs of incorrect/correct program samples to learn transformation rules, which aid a program synthesizer to transform incorrect into correct programs. Finally, sk_p uses partial fragments of contiguous statements to train a neural network to predict possible repairs.

In the alignment step, these approaches compare an incorrect program with respect to a correct program based on rigid schemes, which limits their repair potential. To illustrate our claim, the Java programs presented in Figure 1 aim to compute the minimum value in an array and the sum of all its elements, and print both minimum and sum values to console. Note that the values of the input array are assumed to be always less or equal than 100. In Sarfgen, an incorrect program will be only repaired if its control statements match with the control statements of an existing correct program. This is a hard constraint since: a) It requires a correct program with the same control statements to exist, and b) Such a correct program may not "naturally" exist. For instance, the control statements of the correct program in Figure 1a do not match with the incorrect program in Figure 1b; in order to match, the correct program should "artificially" contain an `if` statement before or after line 7, and such a statement should not modify the final output of the program. CLARA relaxes these constraints such that, outside loop statements (`while` or `for`), both programs can have different control statements, but they need to be the same inside loops. This relaxation still forces a correct program with the same loop signature to exist.

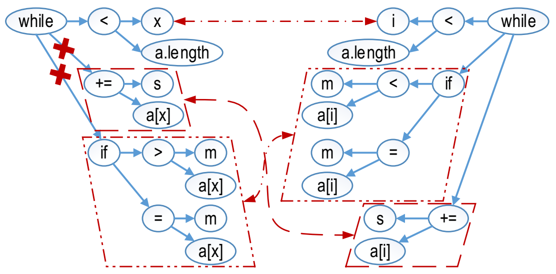
Refazer uses the tree edit distance to find discrepancies between two programs. The tree edit distance between two equivalent abstract syntax trees with different order of statements implies multiple edits. For example, Figure 1c shows an excerpt of the edits to transform the abstract syntax tree of the correct into the incorrect program in our example, which implies removing and adding full subtrees; however, only two edits would be necessary, i.e., changing "<" by ">" and removing the subtree formed by lines 8–9 in Figure 1b. In sk_p, different order of statements result in different partial fragments, so additional correct

```
 1 void f(int[] a){
 2   int x = 0, m = 101, s = 0;
 3   while (x < a.length) {
 4     s += a[x];
 5     if (m > a[x])
 6       m = a[x];
 7     x++;
 8   }
 9   print(s + "," + m);
10 }
```

(a) Correct program

```
 1 void g(int[] a){
 2   int i = 0, m = 101, s = 0;
 3   while (i < a.length) {
 4     if (m < a[i])
 5       m = a[i];
 6     s += a[i];
 7     i++;
 8     if (m == 0)
 9       i--;
10   }
11   print(s + "," + m);
12 }
```

(b) Incorrect program



(c) Excerpt of (simplified) abstract syntax tree edits

```
1 while (x < a.length) {
2   s += a[x];
3   if (m > a[x])
```

(d) Lines 3–5 in Figure 1a

```
1 while (x < a.length) {
2   if (m > a[x])
3     m = a[x];
```

(e) Fragment needed to fix Figure 1b

Fig. 1: Comparison of different existing methods

programs will be required to train the program repairer. For instance, Figure 1d shows a fragment extracted from the correct program; however, the incorrect program will only be fixed by a fragment like the one in Figure 1e.

We propose an alignment step based on approximate alignment of program dependence graphs. Figure 2 shows an excerpt of the program dependence graphs derived from the programs in Figure 1. Each node corresponds to a statement in such programs, e.g., u_3 corresponds to line 3 in Figure 1a. The first step consists of transforming programs into program dependence graphs that are further annotated with semantic labels. For example, *Ctrl* in u_3 summarizes that the corresponding statement is a control statement. In addition, u_3 is also annotated with *Lt* that represents the "less than" operation of the statement. We apply approximate graph alignment over two (correct and incorrect) program dependence graphs G_1 and G_2. For each pair of nodes (u_i, v_j) such that u_i and v_j belong to G_1 and G_2, respectively, we compute a node similarity based on topology and semantic labels. Having all pairwise node similarities, we compute

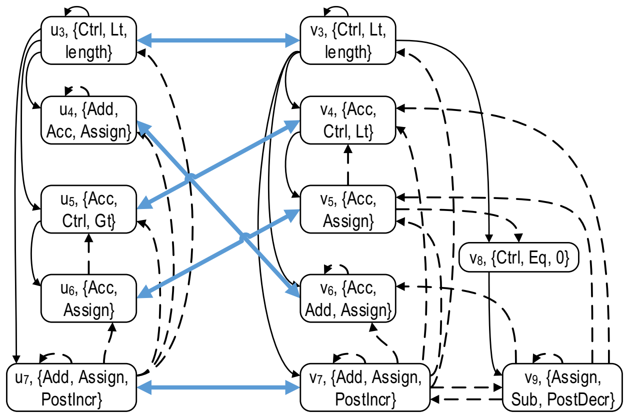


Fig. 2: Alignment of program dependence graphs derived from Figures 1a and 1b

an alignment from the nodes in G_1 to the nodes in G_2. We cast this problem as finding a matching with maximum similarity in bipartite graphs [16]. Bold, double-headed edges in Figure 2 represent a sample alignment with maximum similarity. Finally, we discover individual pairs in a given alignment that are useful for repairing an incorrect program. We rely on the node similarities in a given alignment to make this decision, i.e., each pair of nodes whose similarity deviates k standard deviations from the mean of the node similarities in the alignment are selected as repair suggestions. The intuition behind this is that the similarity of such pairs is smaller than the rest of the similarities in the alignment, i.e., they are less similar than others. In our example, we suggest (u_5, v_4) as a repair to fix the incorrect program in Figure 1b. There may be nodes in the larger program dependence graph that are not present in the alignment. These nodes are suggested to be added or removed depending on whether they belong to a correct or an incorrect program, respectively. In our example, both v_8 and v_9 belong to an incorrect program, so they are suggested to be removed.

3 Background

A program dependence graph $G = (V, E, l_s, l_e)$ of a program p is a directed, labeled multigraph, where V is a set of nodes representing statements in p, $E : V \times V$ is a set of directed edges, $l_s : V \to (L_s, m)$ is a node labeling function, and $l_e : E \to \{Ctrl, Data\}$ is an edge labeling function. Let $(v_s, v_t) \in E$, $l_e((v_s, v_t)) = Ctrl$ indicates that the execution of node v_t depends on node v_s evaluating to true; furthermore, $l_e((v_s, v_t)) = Data$ represents that v_t uses a variable declared or re-assigned by v_s. L_s contains labels that summarize the semantics of a program statement, such as *Assign*, *Call*, and *Ctrl* to denote variable assignments, calls to other methods, and condition or loop statements, respectively. In addition, L_s includes labels to represent operation semantics and constants of a program statement, which include *Acc*, *Add*, and *Sub* to encode array access, addition, and subtraction, respectively. Since a program statement may contain multiple operations that are the same, e.g., multiple array accesses, $m : L_s \to \mathbb{N}$ is a function to support multisets. Note that, for the sake of simplicity, we omit the details of multi-method program dependence graphs, i.e., programs that contain multiple methods. In such cases, we extend L_s and $\{Ctrl, Data\}$ allowing nodes denoting method entry points, method calls, and parameters and result of a method call [4].

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two directed graphs such that $|V_1| \leq |V_2|$. Subgraph isomorphism consists of finding all non-induced solutions $\phi : V_1 \to V_2$ such that $\forall (u_i, u_j) \in E_1 \Rightarrow (\phi(u_i), \phi(u_j)) \in E_2$. The problem of approximate graph alignment consists of finding an injective function $\varphi : V_1 \to V_2$ such that $|V_1| \leq |V_2|$. This problem is a relaxation of the subgraph isomorphism problem in which we assume that $|V_1| \approx |V_2|$, and G_1 is approximately contained in G_2.

4 Flexible program alignment

We wish to compute an alignment between the statements in a correct program with respect to the statements in an incorrect program. The computation of such an alignment takes topological and semantic information into account. On one hand, topological information encodes the context of each statement regarding its control and data dependencies, i.e., what are the statements that must be fulfilled in order for a given statement to be executed, and what are the variable uses of such a statement. On the other hand, semantic information allows us to distinguish statements that are performing different operations, such as addition or an API call. Let $G_1 = (V_1, E_1, l_s^1, l_e^1)$ and $G_2 = (V_2, E_2, l_s^2, l_e^2)$ be two program dependence graphs. At this stage, we are agnostic to correctness and incorrectness of the programs evaluated, so G_1 can be either correct (and G_2 is then incorrect), or incorrect (and G_2 is then correct). Our first goal consists of computing all the pairwise similarities between nodes in V_1 and V_2. The similarity of nodes $u_i \in V_1$ and $v_j \in V_2$ is measured as follows [6, 8, 20]: $Sim(u_i, v_j) = \alpha\, Top(u_i, v_j) + (1 - \alpha)\, Sem(u_i, v_j)$, where Top and Sem are topological and semantic similarities, respectively, and $\alpha \in [0, 1]$ is a parameter to balance the contribution of each type of similarity. $Sim(u_i, v_j) = 1$ entails that both nodes are identical.

We compute similarities $Sim(u_i, v_j)$ for every pair of nodes $u_i \in V_1$ and $v_j \in V_2$. The next step consists of computing an alignment between both graphs, i.e., $\varphi : V_1 \to V_2$ ($|V_1| < |V_2|$). We cast the problem of finding an alignment as finding a maximum weight matching in bipartite graphs. Let $B = (V_1, V_2, E, \omega)$ be a bipartite graph where V_1 and V_2 are the sets of nodes of G_1 and G_2 ($V_1 \cap V_2 = \emptyset$), respectively, $E : V_1 \times V_2$ is a set of undirected edges, and $\omega : E \to \mathbb{R}$ is a function that assigns weights to the edges as follows: $\omega(u_i, v_j) = Sim(u_i, v_j)$. There are several algorithms in the literature to compute maximum weighted matchings that find augmenting paths that alternatively connect edges in V_1 and V_2, ensuring that the final similarity weight is maximized and, thus, producing the alignment φ with maximum similarity [16].

Once we have computed an alignment $\varphi : V_1 \rightarrow V_2$ between two program dependence graphs G_1 and G_2, our goal is to discover repair suggestions, i.e., statements in the correct program that can be used to fix the incorrect program. To discover these statements, we rely on the node similarities of the pairs available in the approximate graph alignment φ. Recall that approximate graph alignment assumes that $|V_1| \leq |V_2|$, which leads to two different situations: 1) If G_1 is correct and G_2 is incorrect, non-aligned nodes belong to the incorrect program and we aim to remove them. This is the case of superfluous/inadequate statements. 2) Otherwise, non-aligned nodes belong to the correct program and we aim to add them to the incorrect program. This is the case of missing statements. Programs in Figures 1a and 1b are an example of the former situation since the first one is correct and smaller than the second one, which is incorrect. In such a case, we aim to remove lines 8 and 9 from the incorrect program.

First, we address the problem of finding replacement suggestions, i.e., which pairs of nodes in a given alignment φ are appealing to repair incorrect statements replacing them by correct statements. Intuitively, we analyze which node similarities in φ significantly deviate from the rest of the node similarities, for which we rely on mean and standard deviation. Let μ_φ and σ_φ be the mean and standard deviation of the node similarities in φ, respectively. We establish a similarity threshold that, for all pairs whose node similarities are below such threshold, we will consider them as replacement suggestions, i.e., they significantly deviate from the rest. Therefore, we consider a pair $(u_i, \varphi(u_i))$ to be a replacement suggestion if $Sim(u_i, \varphi(u_i)) < \mu_\varphi - k\,\sigma_\varphi$, where $k \in \mathbb{R} \geq 0$.

Second, we address the problem of finding suggestions of statements to be added or removed. Recall that we suggest statements to be added when the size of the incorrect program is less than the size of the correct program (total number of nodes). Otherwise, if the correct program is smaller than the incorrect, we suggest statements to be removed. Note that, in practice, the number of addition or removal suggestions can be large if the core of both programs are similar but they have differences in implementation. A common example in our experiments is learners who reutilize their own implementation of a console manager for reading from and writing to console. Other learners exploit utility classes to achieve the same behavior, e.g., `java.util.Scanner`. In such cases, even when the core of both programs is similar, there are a large number of non-aligned nodes that correspond to the ad-hoc console manager. If we remove such nodes, it is very unlikely that the resulting program would be correct. As a result, we only suggest nodes to be added or removed if they are directly or indirectly (one hop) connected with nodes suggested as replacements without taking direction into account. More formally, $v \in V_2 | v \notin ran\,\varphi$ is suggested as an addition or removal if $\exists u \in V_1 | (u, \varphi(u)) \in P \wedge |Path_U(\varphi(u), v, G_2)| \leq 1$, where P is the set of replacement suggestions and $Path_U(u, v, G)$ is the shortest path between nodes u and v in the undirected version of graph G.

We adapt edge correctness [8] (EC) to compute a global similarity between program dependence graphs that measures the number of edges that are preserved in a approximate alignment φ, which is defined as follows: $EC(\varphi) =$

Table 1: Summary statistics of CodeChef assignments

	Id	#C	#I	LOC	#V	#E
BUYING2	BU	861	741	43.4 ± 29.9	45.2 ± 25.7	108.7 ± 62.4
CARVANS	CA	719	1,122	36.6 ± 28.0	37.0 ± 23.7	91.0 ± 57.7
CLEANUP	CL	1,650	889	55.4 ± 29.0	57.4 ± 23.1	154.6 ± 66.9
CONFLIP	CO	1,203	450	41.8 ± 30.7	39.3 ± 25.1	81.4 ± 62.3
JOHNY	JO	1,534	454	39.3 ± 28.3	40.3 ± 24.4	99.3 ± 65.0
LAPIN	LA	561	288	49.6 ± 32.3	53.6 ± 28.3	125.4 ± 78.3
MUFFINS3	MU	2,394	527	23.6 ± 27.4	20.5 ± 24.0	40.2 ± 63.0
PERMUT2	PE	1,890	1,083	41.7 ± 28.4	38.3 ± 22.4	89.1 ± 55.8
SUMTRIAN	SU	1,883	1,032	49.3 ± 28.4	52.5 ± 23.2	147.5 ± 60.7

$\sum_{(u_i,u_j)\in E_1} IP(u_i,u_j,\varphi,E_2)/|E_1|$, where $IP(u_i,u_j,\varphi,E_2) = 1$ iff $(\varphi(u_i),\varphi(u_j)) \in E_2 \wedge l_e^1((u_i,u_j)) = l_e^2((\varphi(u_i),\varphi(u_j)))$; otherwise, $IP(u_i,u_j,\varphi,E_2) = 0$.

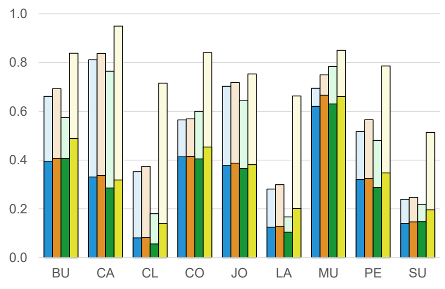
5 Evaluation

We focus on nine assignments from CodeChef (https://codechef.com) shown in Table 1, where #C and #I entail the total number of correct and incorrect programs, respectively; LOC, #V and #E stand for average and standard deviation of lines of code, and nodes and edges in the program dependence graphs, respectively. Programs were collected in Nov, 2017. All CodeChef assignments follow the same structure: each test case must be read from console by a given program, and such test case consists of a single block of text that requires parsing and, usually, involves more than one loop before performing any computations to solve the assignment at hand.

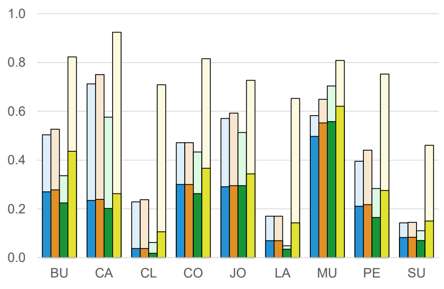
We aim to compare our flexible alignment approach with respect to rigid alignments used in state-of-the-art approaches: CLARA [3], Refazer [15] and Sarfgen [18]. We implemented a common repair framework with a number of variations in the search step as follows:

– In SameCDG (SC), a correct and an incorrect programs are considered only if there exists a graph isomorphism between their control nodes (Sarfgen).
– SameLoop (SL) is a relaxation of SameCDG such that any combination of control statements are allowed unless they are included in a loop (CLARA).
– Flexible (FL) ranks correct programs based on edge correctness with respect to an incorrect program and selects top-t correct programs.
– Rigid (RI) is more restrictive than FL since only edge correctness that belongs to the interval $[1, .85)$ are considered (Refazer).

We evaluate the power set of suggestions in an incremental way starting from the empty set with an upper limit l [19]. The repair process takes as input the lines of the incorrect program that are impacted, and adds, removes, and/or replaces them by lines in the correct program. When adding or replacing lines,



(a) Using C_{100}, $\alpha = .5$ and $k = 1$



(b) Using C_{25}, $\alpha = .5$ and $k = 1$

Fig. 3: Repairs achieved by the different approaches. From left to right, bars correspond to SC, SL, RI and FL, respectively. The Y axis presents the percentage of incorrect programs fully (darker color) and partially (lighter color) repaired.

variables can be different since we may compare programs coming from different learners. As a result, the repair process also evaluates all possible combinations of variables in the original incorrect program [19]. We perform this repair step for all possible combinations of correct-incorrect programs resulting from the search step depending on each approach (SC, SL, RI and FL). We evaluate all possible pairs of correct-incorrect programs; however, CLARA and Sarfgen use variable traces to select a single correct program.

We consider two scenarios in which we limit the number of correct programs available to repair incorrect programs sorted by submission date, ascending: 100% (C_{100}) and 25% (C_{25}). These sets simulate different stages in time of a given assignment in which we have only collected a partial number of correct programs. We deem $\alpha = .5$ and $k = 1$ as proper parameter values. Comparing C_{100} in Figure 3a with respect to C_{25} in Figure 3b, we observe a performance drop for all approaches except for FL, which keeps competitive performance in both full and partial repairs. These results match our hypothesis that our flexible program alignment is appealing when there exist fewer correct programs for a given assignment. In C_{100}, we observe that FL is able to achieve more full repairs than SC and SL except in the CA, JO and MU assignments, where SC and SL perform better. In these assignments, there exist correct programs with the same loop structure that are suitable to repair incorrect programs; however, our ranking based on edge correctness does not promote these in favor of other correct programs with a different loop structure but similar semantics.

6 Related work

CLARA [3] compares variable traces to cluster correct programs based on test cases. A single correct program is selected as a cluster representative. Each incorrect program is compared to every cluster representative based on variable traces to find the minimal repairs to transform from incorrect to correct. Sarfgen [18] searches for similar correct programs that share the same control flow structure as the incorrect program. To identify the best correct program to repair an incorrect program, it summarizes variable traces into vectors that are compared using Euclidean distance, so the correct program with the smallest distance is selected. Incorrect and correct programs are fragmented based on their control flows, and, for each fragment pair that is matched, potential repairs are computed using abstract syntax tree edits. CLARA and Sarfgen only consider pairs of programs whose control flow match, which is a hard constraint since such a pair may not currently be present in the set of correct programs, or such a control flow may not even be possible.

Refazer [15] proposes "if-then" rules to match and transform abstract syntax subtrees of a program. Such rules are synthesized from sample pairs of correct/incorrect programs, in which tree edit distance comparisons between correct and incorrect programs help identify individual transformations. A clustering algorithm finds transformations that can be abstracted away into the same rule. sk_p [13] relies on neural networks to repair incorrect programs. First, all variables in each program are renamed to tokens, and sk_p constructs partial fragments of three consecutive statements using these renamed tokens. The middle statements in fragments are removed and fed to the repairer for training, i.e., each training pair consists of the partial fragment without the middle statement and the full fragment. Given an incorrect program, sk_p computes all candidate statements to be fixed, which form a search space that needs to be explored to find all the necessary repairs. The order of statements is one of the main drawbacks of Refazer, Sarfgen, and sk_p: Refazer and Sarfgen rely on edit distances of abstract syntax trees, while sk_p treats programs as documents. Our approximate alignment allows to account for more implementation variability and flexible comparison of programs.

7 Conclusions

Nowadays, programming is perceived as a must-have skill. It is thus not surprising that the number of learners have scaled to millions, especially in online settings. Delivering feedback is addressed by repairing learners' incorrect programs. The trend in data-driven approaches is to perform a rigid matching between correct and incorrect programs to discover snippets of code with mending capabilities. The downside is that potential repairs that could be captured by looser alignments may be missed. This paper explores using a flexible alignment between statements in pairs of programs to discover potential repairs. We compare flexible with respect to rigid program comparisons. The former is capable of repairing more programs than rigid schemes, which supports our hypothesis that rigid approaches might be missing valuable code snippets for reparation that could be discovered by an approximate method otherwise. As a result, we claim that "search, align and repair" approaches should rely on flexible alignments to improve their repair capabilities.

References

1. Coetzee, D., Fox, A., Hearst, M.A., Hartmann, B.: Should your MOOC forum use a reputation system? In: CSCW. pp. 1176–1187 (2014)
2. Garcia, D.D., Campbell, J., DeNero, J., Dorf, M.L., Reges, S.: CS10K teachers by 2017?: Try CS1K+ students now! coping with the largest CS1 courses in history. In: SIGCSE. pp. 396–397 (2016)
3. Gulwani, S., Radicek, I., Zuleger, F.: Automated clustering and program repair for introductory programming assignments. In: PLDI. pp. 465–480 (2018)
4. Horwitz, S., Reps, T.W.: The use of program dependence graphs in software engineering. In: ICSE. pp. 392–411 (1992)
5. Jawalkar, M.S., Hosseini, H., Rivero, C.R.: Learning to recognize semantically similar program statements in introductory programming assignments. In: SIGCSE. p. 1264 (2021)
6. Khan, A., Wu, Y., Aggarwal, C.C., Yan, X.: NeMa: Fast graph search with label similarity. PVLDB **6**(3), 181–192 (2013)
7. Kirschner, P.A., Sweller, J., Clark, R.E.: Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. Educational Psychologist **41**(2), 75–86 (2006)
8. Kuchaiev, O., Milenković, T., Memišević, V., Hayes, W., Pržulj, N.: Topological network alignment uncovers biological function and phylogeny. RSIF **7**(50), 1341–1354 (2010)
9. Marin, V.J., Pereira, T., Sridharan, S., Rivero, C.R.: Automated personalized feedback in introductory Java programming MOOCs. In: ICDE. pp. 1259–1270 (2017)
10. Marin, V.J., Rivero, C.R.: Clustering recurrent and semantically cohesive program statements in introductory programming assignments. In: CIKM. pp. 911–920 (2019)
11. Monperrus, M.: Automatic software repair: A bibliography. CSUR **51**(1), 17:1–17:24 (2018)
12. Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M., Guibas, L.J.: Learning program embeddings to propagate feedback on student code. In: ICML. pp. 1093–1102 (2015)
13. Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: sk_p: A neural program corrector for MOOCs. In: SPLASH. pp. 39–40 (2016)
14. Rodriguez, C.O.: MOOCs and the AI-Stanford like courses: Two successful and distinct course formats for massive open online courses. EURODL **15**(2) (2012)
15. Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., Hartmann, B.: Learning syntactic program transformations from examples. In: ICSE. pp. 404–415 (2017)
16. Sankowski, P.: Maximum weight bipartite matching in matrix multiplication time. TCS **410**(44), 4480–4488 (2009)
17. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: PLDI. pp. 15–26 (2013)
18. Wang, K., Singh, R., Su, Z.: Search, align, and repair: data-driven feedback generation for introductory programming exercises. In: PLDI. pp. 481–495 (2018)
19. Xin, Q., Reiss, S.P.: Leveraging syntax-related code for automated program repair. In: ASE. pp. 660–670 (2017)
20. Zhang, S., Tong, H.: FINAL: Fast attributed network alignment. In: KDD. pp. 1345–1354 (2016)

21. Zweben, S., Bizot, B.: 2015 Taulbee Survey. Tech. rep., Computing Research Association (2016)