An Algorithm for Generating Explainable Corrections to Student Code

Yana Malysheva
yana.m@wustl.edu
Washington Univesity in St. Louis
St. Louis, Missouri, USA

Caitlin Kelleher
ckelleher@wustl.edu
Washington Univesity in St. Louis
St. Louis, Missouri, USA

ABSTRACT

Students in introductory computer science courses often need individualized help when they get stuck solving programming problems. But providing such help can be time-consuming and thought-intensive, and therefore difficult to scale as Computer Science classes grow larger in size. Automatically generated fixes with explanations have the potential to integrate into a variety of mechanisms for providing help to students who are stuck on a programming problem. In this paper, we present a data-driven algorithm for generating explainable fixes to student code. We evaluate a Python implementation of the algorithm by comparing its output at different stages of the algorithm to state-of-the-art systems with similar goals. Our algorithm outperforms existing systems that can analyze and fix beginner-written Python code. Further, fixes it generates conform very well to corrections written by human experts for an existing benchmark of code correction quality.

ACM Reference Format:
Yana Malysheva and Caitlin Kelleher. 2022. An Algorithm for Generating Explainable Corrections to Student Code. In *Koli Calling '22: 22nd Koli Calling International Conference on Computing Education Research (Koli 2022), November 17–20, 2022, Koli, Finland.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3564721.3564731

1 INTRODUCTION

Solving programming problems is an integral part of many introductory Computer Science (CS) courses. But beginner students often make errors in their code which they are unable to resolve without help. These impasses can actually become effective learning moments, if students can get timely help in resolving and understanding the issue[22, 24, 25]. But as computer science classes continue to grow in scale[28], it can be difficult to provide timely help and feedback in a scalable manner to all the students that need it.

Ideally, any assistance provided to the student would acknowledge and incorporate the student's original intended approach, and focus on resolving individual errors in the code. But trying to understand and debug an unfamiliar approach to a problem can be time-consuming and thought-intensive, even for experts. This is especially true when the code in question is esoteric, and tries to solve the problem in unexpected ways, as is often the case with beginner-written code.

To help address situations like these, we propose a data-driven algorithm which generates fixes to student code, alongside run-time illustrations of how each fix changes the way that the code behaves. The algorithm works in three stages: it first compares a student's incorrect solution to other students' correct solutions; it then finds a subset of the differences that, when applied to the incorrect solution, will make it perform correctly; finally, it groups these changes into independent fixes, and explains each fix by finding instances where running the code with the fix produces an expected runtime state, but without the fix does not.

The output of this algorithm could be used for several types of systems targeting different audiences. For example, the full set of fixes and explanations can be shown to a human assistant, who could be a teacher, peer, or near-peer. The assistant would use this information to quickly understand what is wrong with the student code, and use the their human judgement to decide how to filter and present that information to the student. Alternatively, the fixes and run-time explanations could be processed to generate student-facing hints that directly provide the student with some idea of where specifically their program may be going wrong.

We evaluated a Python implementation of the algorithm using a combination of comparisons to existing state-of-the-art systems with similar goals, and direct measurement of the effectiveness of parts of the algorithm. We found that our algorithm outperforms these state-of-the-art systems when the input data consists of code written by novices solving Python programming problems. We also found that our algorithm performs nearly as well as human tutors on an existing benchmark used for evaluating the quality of generated code corrections.

2 RELATED WORK AND BACKGROUND

Our work builds on data-driven analysis of student code in educational settings and leverages prior work in finding edit scripts between two abstract syntax trees (ASTs).

2.1 Data-driven analysis of student code

Researchers have created a variety of systems that analyze student code using comparisons to other student solutions to the same or similar problems. We discuss these systems first by their intended applications and then by the underlying methods used for code analysis.

2.1.1 Applications of code analysis. One of the main goals of automatically analyzing student-written code is to find a set of corrections for student code which is currently producing incorrect output.

Koli 2022, November 17–20, 2022, Koli, Finland

ACM ISBN 978-1-4503-9616-5/22/11.
https://doi.org/10.1145/3564721.3564731

Existing systems use code analysis to support student programmers in a variety of ways including 1) generating next-step hints, 2) assisting instructors by summarizing student solutions and streamlining feedback, 3) crowd-sourcing fix explanations.

Next-step hints. Many of the code-correction systems[7, 9–11, 18, 21, 26] seek to generate hints for the next step a student should take in order to progress towards solving a programming problem. These systems first generate a set of corrections for the erroneous student code, then filter and process these corrections to generate student-facing hints. These hints direct students' attention to a problem in their code, usually without giving away exactly what should be changed to make the code correct. Some systems[12] augment these generated hints with expert-created textual explanations of why a specific change is necessary.

Assisting Instructors. The growing size of computing courses has created a challenge for instructors to provide good feedback to students in a manageable amount of time. Systems that support instructors begin by clustering student solutions. OverCode[6] seeks to cluster correct solutions to help a lecturer in a large class maintain an accurate understanding of the types of solutions that students are writing in their class. MistakeBrowser and FixPropagator cluster incorrect student solutions that require similar corrections, and then enable instructors to propagate teacher-generated corrections and feedback to multiple students' solutions [10]. Both are powered by the Refazer algorithm [10].

Crowd-Sourcing Explanations. A final system tries to crowd-source human-readable explanations to commonly encountered fixes for syntax errors [9]. HelpMeOut shows clusters of related errors to experts and asks them to explain the required fixes [9].

Like many of these systems, our algorithm generates a set of fixes for incorrect student code that attempts to respect the approach in the existing code. This core algorithm could be used to improve a variety of systems that leverage code analysis. Further, we generate explanations that show how the suggested changes contribute to progress towards a correct solution. To the best of our knowledge, no other systems explicitly attempts to explain the effect of the fixes it generates.

2.1.2 Methods of code analysis. Existing systems focus on analyzing several aspects of the student code, often in combination: (1) the sequence of tokens in the code itself; (2) the AST of the code; (3) traces of how a particular student changes their code over time; and (4) outcomes of running the code with predetermined input.

Token-based code analysis. Several systems attempt to fix syntax errors in student code. Syntax errors, by definition, make it impossible to analyze the code by parsing or running it. Thus, systems that attempt to fix syntax errors must rely on analyzing the linear sequence of tokens in the code.

Some of these systems use Recurrent Neural Networks in order to attempt to learn valid sequences of tokens [1, 20], or directly learn corrections from pairs of correct and incorrect sequences[8].

Other systems, such as HelpMeOut[9] and BlueFix[26], directly compare the sequence of tokens that caused an error with similar sequences that also caused the same error, but were subsequently corrected by the author. They then show the most similar corrected examples directly to the student trying to resolve the syntax error. The student may then be able to use this example to gain a better understanding of why their own error is occurring.

AST analysis. While systems that focus on syntax errors must rely on the tokens in the code, systems that try to address semantic correctness - whether the code does what it should be doing - often use the Abstract Syntax Tree of the code as the primary data structure in analyzing the code.

Many existing systems [7, 10, 11, 13, 18, 21, 27] compare the AST of incorrect student code to solutions written by other students, in order to find code edits that might help fix the bugs when applied to the incorrect student code.

Analysis of student traces. Several systems for analyzing student code[13, 16] are based on the Hint Factory technique[23]. These systems look for changes that other, successful students have made to their code when they were in a similar code state to a student who may need help.

Refazer[10] also analyzes traces of how students changed their code to arrive at a correct solution. It only looks at the very last fix the student made before submitting correct code, but it is able to generalize that fix and attempt to apply it to many other students' solutions.

By contrast, CodeQ [11] is able to find and extract edits that students made throughout their code-writing process, as long as those edits seemed to make things better and not worse. Similar to Refazer, this system is able to apply these edits to other students' incorrect code. But it is also able to search for a **sequence** of applicable edits which may fix several independent bugs in the student code.

HelpMeOut[9] and BlueFix[26] also find pairs of states where the student had an error and then fixed it, but they focus on syntax errors and runtime errors instead of semantic bugs in the code.

Execution-based analysis. Many code-correction systems [10, 11, 21] run candidate corrected versions of student code against a set of unit tests for the problem the code was trying to solve. By doing so, they are able to evaluate novel solutions that were not present in the original dataset, but may be closer to the student's original intent than any of the solutions seen directly in the data.

Additionally, some systems go beyond looking at the final output of running a unit test, and analyze sequences of intermediate runtime states as the program is executed.

HelpMeOut[9] finds examples of runtime error fixes using a similar strategy to the way it finds fixes to syntax errors: It finds instances where a student's code encountered a runtime error on a particular line of code during execution, but the next version of the same solution got further in the execution without encountering the error. It considers the changes between the two versions to be an effective fix - or at least partial fix - for the runtime error.

Overcode[6] captures the sequences of values that each variable takes on during execution, and clusters code with identical sequences of values. These clusters capture different strategies that students used to arrive at correct solutions.

Similarly, the Hint Factory-based system in the BOTS game[16] groups together code that results in the same sequence of actions in the game. It uses these action sequences as the solution space in

which it attempts to find the best next step for the student.

Our system utilizes a combination of AST-based and runtime comparisons between student code and some correct solution to the same problem. Similar to existing systems such as ITAP[21], it uses AST comparisons to existing student solutions and unit test output to find a potentially novel correct solution that is close to the student's incorrect code. But in contrast to other systems, it relies on a fine-grained runtime analysis of the code to both evaluate the fixes and find explanations for the effect of a fix. In particular, to our knowledge, this is the first system that tracks the runtime state of the code at a finer grained level than line-by-line. We find that this finer-grained tracking is necessary for analyzing beginner code, where problem solutions can often be 1-2 lines long and contain few or no variables.

2.2 Edit scripts between Abstract Syntax Trees

An important sub-problem in our approach to generating code corrections is finding an edit script between two ASTs representing two different programs. An edit script is a sequence of node-level edits which, when applied to the first tree, changes it to exactly match the second tree. In the context of generating corrections for student code, the edit script describes a way of changing the student's incorrect code to exactly match a given correct solution.

Existing algorithms seek to find an edit script with a minimal length or cost, in order to capture the largest amount of similarities between the two trees. These algorithms differ in the types of operations that are allowed in the edit script. In particular, some algorithms allow for "move" operations, which move a subtree of the AST to a different location in one edit, while others only allow for deleting, inserting, and renaming nodes.

Allowing for move operations creates edit scripts that more precisely capture similarities between two programs. For example, moving a line of code out of a loop could be represented as a single move operation. But if move operations were not allowed, this change would need to be represented by deleting all the AST nodes associated with that line of code, and recreating them elsewhere.

However, finding the shortest edit script is an NP-hard problem when move operations are allowed [2]. So algorithms that allow moves use heuristics to approximate the optimal edit script. On the other hand, if move operations are not allowed, polynomial-time algorithms exist for finding an optimal edit script.

Regardless of the set of allowed operations, edit script algorithms consist of two steps: (1) finding a mapping between the nodes of the two ASTs, and (2) using the mapping to derive an edit script. The edit script is always completely determined by the mapping, and can be generated in polynomial-time from a valid mapping. So the problem of finding a minimal edit script can essentially be reduced to finding a suitable maximal valid mapping between the two trees.

2.2.1 Without move operations. APTED [14, 15] is a state-of-the-art general-purpose algorithm for finding a minimum-cost tree edit script which uses only insert, delete, and rename operations. The algorithm allows for assigning different costs to different operations, and has a worst-case runtime complexity of $O(n^3)$.

Notably, this algorithm allows for inserting and deleting nodes anywhere on the tree, not just leaf nodes. If a non-leaf node is deleted, then its parent node inherits all of the children of the deleted node. Conversely, a node can be inserted between some existing node and a subset of its children, taking over as parent to those children nodes. For example, in the context of an AST, inserting an if statement **around** some block of code could be represented using non-leaf insertions into the AST.

Because of this, and because it generates an optimal mapping and edit script, the mapping generated by the algorithm is the largest possible mapping which preserves ancestor relationships between nodes.

Since the set of edits allowed in APTED does not include move operations, it does not always allow us to precisely capture the similarities between two programs, as discussed above. However, the guarantee of optimality, together with the property of preserving ancestor relationships, make APTED a useful starting point for our algorithm.

2.2.2 With move operations. Since the problem of finding an optimal edit script is NP-hard when move operations are allowed, algorithms that choose to use move operations tend to use domain-specific heuristics based on assumptions about the underlying data in the trees being compared.

Chawathe et al. [3] first described an edit script algorithm which allowed for move, insert, delete, and rename operations. In contrast to algorithms like APTED, the insert and delete operations are only allowed to operate on leaf nodes. Before a non-leaf node can be deleted, all of its child nodes need to be deleted or moved.

This algorithm was developed for the application of describing the set of changes between two sequential versions of hierarchically structured data, such as two versions of an HTML page. In order to find a mapping between the two trees, it uses a set of heuristics which are based on the assumption that the two trees being compared are different versions of the same structured text-based document.

Given the mapping, the edit script is then generated using a five-phase algorithm: Update, Align, Insert, Move and Delete. This algorithm does not make any assumptions about the structure of the mapping, and so can be used with any mapping between two trees to generate a valid edit script. For this reason, subsequent work re-uses the edit script generation stage described by Chawathe et al., and focuses on alternative heuristics for generating the mapping.

In particular, GumTree[5] is an algorithm which builds on the set of edits and edit script generation algorithm described by Chawathe et al. GumTree is a state-of-the-art algorithm for finding the difference between two versions of the same code, for example two consecutive commits in a version control repository. Its mapping heuristics start by greedily finding and matching the biggest possible identical sub-trees. It then tries to match up as many additional nodes as possible around these large matching chunks. The MTDIFF set of optimizations [4] can then be applied to repair and augment this mapping in order to find a smaller set of edits, but fundamentally, the algorithm relies on anchoring the mapping in large chunks of identical code.

Most existing algorithms that use move operations are optimized for the use case of highlighting differences in two sequential versions of the same data, whether the trees being compared represent ASTs or structured text. But our use case is different: the fix generation algorithm must find similarities between two different solutions written by different people, both of whom may be beginners who write code that is esoteric in different ways. This means that the heuristics in existing algorithms are often not applicable in our case.

3 IMPLEMENTATION

Our algorithm for generating a set of explainable fixes for an incorrect solution to a programming problem can be divided into three stages, as illustrated in Figure 1:

(1) Compare the incorrect solution to all available correct solutions to the same problem, and **calculate the edit script** from the incorrect solution to each correct version.
(2) **Simplify each edit script** by finding and removing edits that don't affect the correctness of the final code
(3) **Generate the sequence of explainable fixes**: Choose the best (shortest) edit script, and group the node-level edits into a sequence of fixes with runtime explanations of what is fixed by each one.

3.1 Calculating the edit script

Given the incorrect solution to a programming problem, for each available correct solution to that same problem, we calculate the edit script that would change the AST of the incorrect solution to exactly match the AST of the target correct solution.

The resulting edit script uses the set of edit operations first described in Chawathe et al. [3], namely:

- **Insert**: Insert a node as a child to some existing node in the AST.
- **Delete**: Delete a leaf node in the AST.
- **Move**: Move a subtree rooted at some node to a different location in the AST.
- **Rename** (sometimes also called update): Change the value of a particular node, for example change a + binary operator to a -.

These operations closely resemble the types of actions a person might take when editing code or structured data: inserting and deleting parts of the program, moving code around, and changing individual values or tokens in the code.

As described in the background subsection on edit scripts, the problem of finding an edit script that uses these operations can be reduced to the problem of finding a mapping between the two ASTs (step 1a in Figure 1). We can then use the algorithm described by Chawathe et al. to derive the edit script directly from the mapping (step 1b in Figure 1).

In order to find a mapping which will result in a concise edit script, our algorithm starts with the mapping generated by the APTED algorithm. Even though the APTED mapping is optimized for a different set of edit operations, it is a good first step because it finds the maximum possible mapping which preserves ancestor relationship between nodes: for any two nodes, A and B, in the original tree that are mapped to nodes A' and B' in the target tree, A is an ancestor of B if and only if A' is an ancestor of B'.

This property means that the mapping will be able to capture similarities in the structure of the two programs. However, it also means that it may not be able to match similar or identical code that is in a different location in the two versions of the code. For example, if the two versions both define two identical variables, but do so in different order, the APTED algorithm would only be able to map one of the variable definitions. The second variable definition would remain unmapped between the two versions of the code, as if the two lines were completely different from each other.

In order to capture these types of similarities, we simply repeat the process on the remaining unmapped nodes: we delete all of the already-mapped nodes from each AST, connecting the children of the deleted node to its parent. We then run the APTED mapping algorithm on the two resulting trees. We repeat this step until it stops producing any new mappings. This strategy allows for mappings between similar sub-trees even when they are located in different places in the original tree. The edit script defined by such a mapping can then use a single move operation to place the sub-tree in the correct position.

The current implementation augments this general algorithm with three heuristic optimizations. The first two of these optimizations change the cost of the "rename" operation when calculating the optimal APTED mapping for different pairs of trees. The third optimization is a post-processing step which removes some of the mappings that would have resulted in an esoteric edit script.

3.1.1 Optimization 1: Adjusting the base cost of renaming. We apply a different cost to the renaming operation for the initial APTED mapping and the subsequent mappings of the remaining unmapped nodes.

For the initial mapping, we ensure that the cost of renaming (changing the value of) an individual node is less than the cost of deleting that node and creating a new node in the same place with the new value. In doing this, we encourage the mapping of nodes with different values, but similar subtree structures.

For subsequent passes, we raise the cost of renaming a node so that deleting and inserting would have the same cost. Since much of the structure of the original code is missing in these subsequent iterations, this discourages the algorithm from mapping unrelated nodes to each other simply because they are in roughly the same place in the remaining unmapped trees.

3.1.2 Optimization 2: Trying to match variable assignments that have similar overall results. Consider the following two equivalent pieces of code:

(1)
```
x = -b+math.sqrt(b**2-4*a*c)
y = 2*a
z = x/y
```
(2)
```
z = (-b+math.sqrt(b**2-4*a*c))/(2*a)
```

The variable z represents the same value in both of these cases. But a naive AST mapping algorithm is likely to map the line calculating x in the first version to the line calculating z in the second version, because the ASTs of the two lines are very similar.

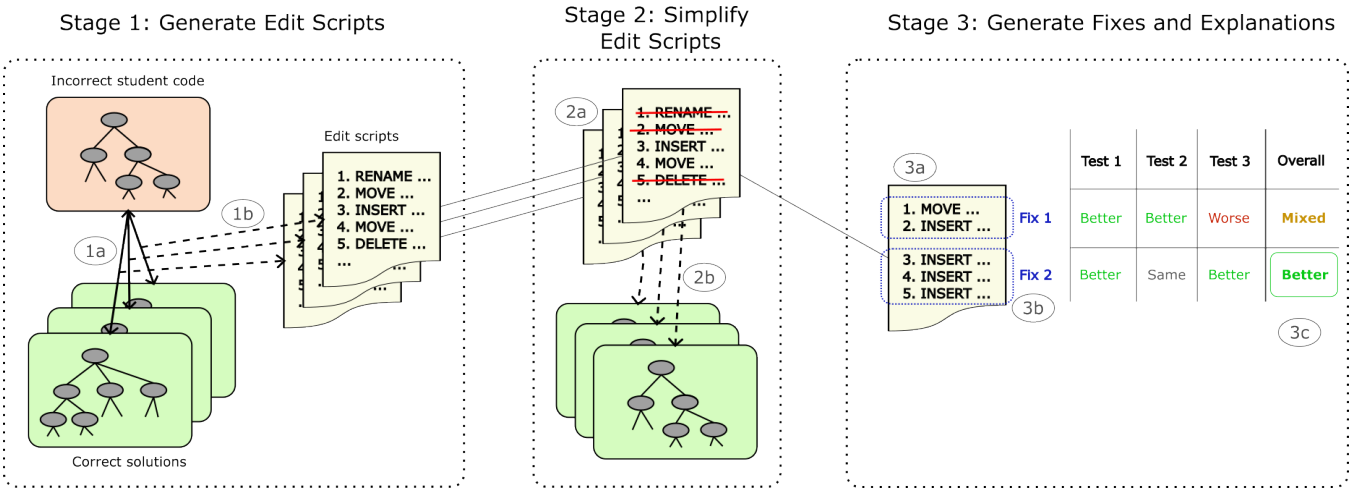

Figure 1: An overview of the fix generation algorithm. Stage 1: Generate a mapping from the incorrect student solution to each available correct solution (1a). From each mapping, derive an edit script which would transform the incorrect solution to the correct one (1b). Stage 2: Simplify each edit script by removing sets of edits that don't affect correctness (2a). Applying each simplified edit script results in a novel correct solution (2b). Stage 3: Select the shortest simplified edit script (3a). Group the edits into fixes (3b). Analyze whether applying each fix results in better performance on each unit test (3c). In the example in the figure, applying Fix 2 directly to the student code results in better overall performance than Fix 1. Therefore, the algorithm would apply Fix 2 first, and then re-evaluate the effect of Fix 1 on the new resulting code.

To try to avoid this issue, in the first pass which creates the initial APTED mapping, we penalize mapping nodes that are:

(1) child nodes of the right-hand side of an assignment operator
(2) are at different depths in the tree, relative to the ancestor assignment operator.

We apply this penalty by giving a mapping between such nodes a "renaming" cost, even if the two nodes are actually identical and do not need to be renamed

In the above example, the initial pass of the APTED mapping would avoid mapping the two copies of `-b + math.sqrt( b**2-4*a*c )` to each other, because this would incur a high renaming cost. It may, however, map the two partial subtrees representing `z = __/__` to each other, thus creating a correct top-level mapping between the two variable assignments. Subsequent passes would not have to deal with this penalty, and would create the correct mappings between the two copies of `-b+math.sqrt(b**2-4*a*c)`. The edit script derived from the overall mapping would then move that calculation into the correct place.

3.1.3 Optimization 3: Removing unnecessary mappings of certain leaf nodes. Because the APTED algorithm is optimizing for a different set of operations, it can sometimes map pairs of nodes that have all three of the following properties:

(1) leaf nodes in the original trees
(2) have parents that are not directly mapped to each other
(3) have different values from each other

This means that the resulting edit script would need to include both a move operation and a rename operation for this node. But this does not create a shorter or simpler edit script than simply deleting the old node and inserting a new leaf node with the new value. Moreover, these nodes are usually not actually semantically related to each other, but instead are located in roughly similar places in some iteration of the APTED mappings. Thus, we remove any such mappings in a post-processing step.

3.2 Simplifying edit script

The result of the previous step of the algorithm is a concise edit script which changes the incorrect program to exactly match some other correct solution to the same problem. But our goal is to find changes that bring the code into any correct state at all, not necessarily into a state that is an exact copy of someone else's correct code. Therefore, some of the edits may be unnecessary to achieve correctness.

In this stage of the algorithm, we try to find and remove unnecessary edits that do not affect the correctness of the final version of the code (step 2a in Figure 1). In the current implementation, we measure correctness by checking whether the program passes a set of unit tests. However, in principle, any measure of correctness could be used, as it is independent of the simplification logic.

The simplification stage of the algorithm has two independent steps: a general algorithm for finding sets of edits that can be removed, and a special case for undoing any variable renaming that happens as part of the original edit script. The variable renaming step happens first, because the general simplification algorithm is more computation-intensive. Undoing some of the edits with the simpler variable renaming logic may save processing time in the general step.

3.2.1 Undoing variable renames. Since the original incorrect code and the correct version are usually written by different people at different times, the two versions of the code are likely to use

different variable names. But edits that change the variable names are not going to affect correctness of the code. To undo these edits, we first use the mapping between the two ASTs to find variable initializations that have been mapped to each other, but initialize differently-named variables in the two programs. For each such pair of variable names, we find and remove any edits in the edit script which rename the variable. Additionally, we find all edits that insert the new variable name (e.g. uses of the variable which weren't present in the original code), and change them to use the original variable name.

3.2.2 Removing other sets of edits. The more general simplification step repeatedly tries to remove subsets of edits from the edit script, and then runs a correctness check on the program that results from applying the rest of the edits to the original code. If the resulting code is still correct, then that subset of edits was not necessary.

There are two non-trivial problems that must be solved when using this general strategy:

(1) Choosing the candidate subsets of edits to remove
(2) Choosing the order in which to remove them

3.2.3 Choosing subsets of edits. In many cases, it is possible to remove a subset of edits without affecting the correctness of the code, even though removing just one of those edits would in fact break the program. For example, the Python expressions `x**2` and `math.pow(x,2)` are equivalent, and edits that change one to the other are unnecessary. But this change would be represented by several edits in the edit script. Undoing or removing just one of those edits may result in an in-between expression that is incorrect, e.g. `math.pow(x)`. Thus, it is important to identify sets of edits that together, create an "undoable" change.

Our heuristic is to select sets of edits that are connected by *dependency* relationships within the edit script. We say that edit A is *dependent* on edit B if the edit script would be invalid when B is removed from the edit script, but A remains. For example, an edit deleting a non-leaf node in the original AST depends on all of the edits that delete or move the children of that node, because any edit script which deletes that non-leaf node without first deleting or moving all child nodes is an invalid edit script.

This heuristic captures edits that, together, modify the same section of the AST and change that section to match the target AST. If the change to that particular section is unnecessary, then that group of edits can be undone. However, this is still a heuristic, and doesn't always capture all "undoable" groups of changes.

In particular, it can sometimes create groups that are too large. For example, it always groups all edits that insert or delete an entire sub-tree into one set. But the inserted code could still be a mix of necessary and unnecessary changes. In that case, the heuristic would not be able to remove this group of edits, and all of the changes - both necessary and unnecessary - would remain inserted.

Conversely, there are some groups of changes that are connected semantically, but are not connected in the AST. Renaming variables is one such group of changes, which is why we handle that as a special case, as described in the previous section.

3.2.4 Choosing removal order. As mentioned above, some groups of changes are dependent on each other semantically, but are entirely disconnected in the AST. In some cases, these groups of changes can still be removed one at a time, but only in the right order. For example, an edit script may call for inserting two separate lines of code, one of which initializes a variable and the other updates it. However, this variable does not actually affect the output of the final code. These edits would be grouped into two separate subsets of edits, one for initializing the variable and one for updating it. Both of these subsets of edits are unnecessary. The simplification step should remove both of them, but it will attempt to remove them one at a time. If we remove the variable update first but leave in the initialization, the code will still be correct. But if we try to remove the variable initialization first, and leave in the update, the code will fail. Thus, the order in which we remove the groups of edits is important.

Instead of trying to analyze the code semantically, and account for all possible semantic dependencies, we chose a removal strategy which effectively tries all removal orders.

At each iteration, we attempt to remove each identified group of edits from the edit script. If any of the groups were successfully removed, we start a new iteration of removals, to see if any of the previously-necessary group of edits can now be removed because we've removed some semantic dependency. We continue this iteration until no more groups of edits can be removed.

This heuristic allows us to remove many different groups of semantically dependent edits, but it does not work when the groups have mutual or circular dependencies. In particular, this heuristic is not able to undo variable renaming, since all uses of the same variable name depend on each other, and there is no order in which we could undo the individual renamings one at a time while keeping the code correct.

3.3 Generating explainable fixes

After the previous two stages of the algorithm, we have some number of candidate edit scripts, each of which changes the incorrect code to a version of the code which passes correctness checks. Each of these candidate edit scripts resulted from comparing the original incorrect code to some existing correct solution, but after the simplification step, each edit script may produce a never-before-seen correct solution that is closer to the original incorrect code (2b in Figure 1).

Out of this set of candidate solutions, we choose the one with the shortest edit script (3a in Figure 1). A short edit script should result in code that is close to the original incorrect code, but fixes all the issues that made the original code incorrect.

To organize the edit script into explainable fixes, we reuse the edit grouping logic from the previous step. The resulting groups represent self-contained modifications to subsections of the code (3b in Figure 1). We call each of these groups a *fix*, and analyze the runtime effect of each fix: how the fix changes what happens when the program runs.

The precise effect of each fix can depend on the order in which the fixes are applied, so we also attempt to find an optimal fix order where for each fix, we can find a runtime illustration of how the fix improves the program.

To analyze the runtime effect of a particular fix being applied to a particular intermediate state of the code, we compare three versions of the code: before the fix is applied, after the fix is applied,

and after all fixes are applied. In this last case, the code is known to be correct, so we use this version as the canonical version of what should happen during the execution of the program.

We compile each version of the code into bytecode, retaining the connection between each bytecode instruction and the node in the AST that produced that instruction. Using these connections, and the AST mappings between the different versions of the code, we can also create mappings between the different bytecode versions.

We run each of the three versions using each available unit test (or other way of specifying input to the program, if applicable) and record the sequence of executed bytecode instructions and the output value of each one. We compare these runtime sequences to find where each of the intermediate versions (before and after the fix) deviates from the canonical version.

To find this deviation point, we first find the longest common subsequence(LCS) of the two execution sequences. For the purpose of this LCS, two bytecode instructions in the execution sequence are considered the same if and only if they originated from AST nodes that were mapped to each other in the mapping between the two versions of the code. This way, the LCS captures the longest possible matching slices of execution, even in cases where the overall execution sequence is drastically different, such as when a small code change affects the number of iterations through a loop.

We define the *deviation point* as the last instruction in the LCS where the output value of the bytecode instruction was the same in both versions. This means that after the deviation point, the two programs are no longer calculating the same thing. There may also be differences before the deviation point, but they are likely to represent variations that do not affect the final outcome, since we know that the values converge again after that.

Once we find the point where each intermediate version starts deviating from the canonical version, we can compare them to see which version of the code got further before deviating from the expected execution sequence. Specifically, we can categorize the runtime effect of applying the fix into one of four types of effect (3c in Figure 1). We say that applying the fix makes the code:

- *The same* if for each available unit test, the versions with and without the fix deviate from the canonical code **in the same exact place**.
- *Better* if for each available unit test, the version without the fix deviates from the canonical code **before or at the same time** as the version with the fix does (but strictly before for at least one unit test).
- *Worse* if for each available unit test, the version without the fix deviates from the canonical code **after or at the same time** as the version with the fix does (but strictly after for at least one unit test).
- *Mixed* if the version without the fix deviates **before** the version with the fix for some unit tests, and **after** for others.

The order of applying the fixes can change the outcome of applying any given fix. For example, adding in code that uses a variable before we add code that initializes that variable would make the program crash, even if that variable use is a necessary part of the final solution.

We try to find an order of the fixes where each fix demonstrates the best possible runtime effect. To achieve a reasonable (partial) order, we repeatedly greedily apply all fixes that would, in the current context, have the best possible runtime effect. If, in the current code state, any of the fixes make the code **better**, we apply that set of fixes. If not, we look for fixes with **mixed** effect, then fixes with the **same** effect, and finally resort to applying fixes that make the resulting code perform **worse**. Then we recalculate the new intermediate state that results from applying these fixes, and re-run the runtime analysis to calculate the effect of the remaining fixes on this new intermediate state. We continue until there are no more fixes to apply.

This gives us a series of fixes such that, for each fix, we can show an illustration of how the fix changes what happens when the code runs. Moreover, for most fixes, this illustration shows an **improvement** resulting from the fix: for some unit test, and for some moment in time in the execution of the three versions of the code, the version with the fix produces a value that is correct, while the version without the fix produces a value which deviates from the expected canonical value.

4 EVALUATION

Because there is no direct comparison for our algorithm, we elected to evaluate the output of each stage of the algorithm separately. For the first and third stage, we compare the results to existing state-of-the-art algorithms that have similar goals to the respective stage of our algorithm. For the simplification stage, we analyze how effectively the edit scripts are simplified by calculating the reduction in edit script length.

As input for evaluating the first two stages, we used a publicly-available set of data of novices solving Python programming problems [17]. This dataset was originally published as part of the CSEDM 2019 Data Challenge, and consists of traces of student attempts to solve a set of python programming problems. We used 90% of the data as training data (to find correct student solutions to these problems) and 10% of the data as the test input (to find incorrect solutions to find fixes for). For the third stage, we use a benchmark for correction quality which is based on a similar but distinct dataset [19]. This dataset also consists of novice programmers' attempts to solve a similar set of Python programming problems, but it was generated specifically for the QualityScore benchmark for hint generators[19].

4.1 Calculating edit script

We compared the edit scripts generated by our algorithm to the output of GumTree [5], a state-of-the art algorithm for comparing ASTs, augmented by MTDIFF[4], a set of post-processing optimizations that "repair" mappings between two ASTs.

To test our algorithm and GumTree+MTDIFF, we used pairs of novice-created programs that attempt to solve the same problem. The first program in the pair is always an incorrect solution, and the second program is a correct solution written by a different student. In finding solutions, we want to stay as close as possible to the student's original code. Thus, shorter edit scripts that make fewer changes are preferable. Our algorithm generated edit scripts that were strictly shorter than those of GumTree+MTDIFF in 68% of the cases; the same length in 26.6% of the cases; and strictly longer in 5.6% of the cases.

Since both GumTree and, to an extent, our algorithm optimize for short edit scripts, we can conclude that our algorithm tends to perform better on this type of input data. This makes sense, since GumTree and MTDIFF are optimized with the assumption that we are comparing two sequential versions of the same code. This is reflected in some of GumTree's underlying heuristics. But in our use case, the two programs being compared are most often written by different people independently of each other, and therefore benefit from a different approach to AST matching.

4.2 Simplifying edit script

To analyze the effectiveness of the simplification step, we compared the length of the edit scripts with and without simplification. As in the previous step, we target shorter edit scripts to minimize the changes to repair broken student code while respecting their approach.

We first ran the simplification process for each possible pair of incorrect student scripts from the test data and correct solutions to the same problem from the training data. Out of 1768 such AST comparisons, the simplification step was able to find a shorter edit script in 1162 cases (65.7%). In those cases, the length of the simplified edit script was, on average, 70% of the original length.

However, we note that only the pairs of incorrect-correct solutions with the shortest edit script length would be selected for use. Thus, we are also interested in determining the effect of simplification on the final chosen edit script for a given incorrect solution. Specifically, if we were to choose the shortest available edit script as input for the fix generation stage of the algorithm, how would the results differ with and without the simplification step? Out of 113 incorrect student solutions in the test data, the simplification step resulted in a shorter final edit script in 75 of the cases (66.3%). In those cases, the length of the simplified edit script was, on average, 57% of the original length. Moreover, the cases where the simplification step did not result in a shorter edit script tended to come from code where the edit script was already short: 6.05 edits on average, compared to 14.5 edits for those programs where some simplification was possible.

This shows that the simplification step can be effective in finding and removing unnecessary edits.

4.3 Generating fixes

Finally, we evaluated the educational quality of fixes generated by our algorithm using the QualityScore benchmark [19]. QualityScore was designed to evaluate the quality of generated next-step hints by comparing them to a set of gold standard hints created by a group of human experts. The hints being evaluated need to be expressed as the AST of the student code, with a minimal correction applied to it.

Since our algorithm seeks to generate explainable fixes rather than next-step hints, we had to make two adjustments to the final output in order to conform to the expectations of QualityScore:

(1) Because QualityScore only considers next-step hints, we only select those fixes that our algorithm would choose to apply directly to the original incorrect student code - namely, the fixes that show the best runtime improvement when applied to the original code. We omit those fixes that only showed improvement when applied at a later stage.
(2) QualityScore requires "minimal" hints. It only considers hints valid or partially valid if they exactly match or are a subset of the edits in a "gold standard" hint. On the other hand, if a correction makes the same changes as a "gold standard" hint but also makes some additional changes, it is considered completely invalid. This minimal hint constraint is not well aligned to our problem space, since the goal of our algorithm is to generate complete and explainable fixes, rather than next-step hints. Therefore, we post-processed each fix to extract a "minimal" subset of edits from the fix. ITAP[21], the algorithm that scored highest in the original QualityScore evaluation, also needed to process their generated corrections to find minimal hints. ITAP's strategy was to take just the top-level token edit to the original code. We modeled our approach for selecting minimal corrections on ITAP's approach.

We chose to compare performance on the "partial match" metric in QualityScore, which counts some hints as a valid match even if they only make a subset of the edits that the gold standard hint makes. Using the "partial match" metric allows us to de-emphasize the importance of choosing the correct "minimal" subset of edits in a fix. If we had used the "full match" metric, which requires the proposed hint to match the gold standard exactly, we would see many false negatives where the post-processing step for selecting a "minimal" set of edits chose too few of the edits present in the fix

Table 1: mean overall QualityScore

	Our algorithm	ITAP	Human Tutors
Gold Standard	0.789	0.706	0.842
All expert-generated hints	0.897	0.745	0.928

Table 2: mean per-problem QualityScore (evaluated against gold standard hints)

	Our algorithm	ITAP	Human Tutors
firstAndLast	0.714	0.857	0.865
isPunctuation	0.750	0.846	0.875
kthDigit	0.786	0.5	0.868
oneToN	0.75	0.5	0.752
helloWorld	1.0	1.0	0.832

Table 3: mean per-problem QualityScore (evaluated against all expert-generated hints)

	Our algorithm	ITAP	Human Tutors
firstAndLast	0.714	0.857	0.921
isPunctuation	0.788	0.923	0.937
kthDigit	1.0	0.5	0.949
oneToN	0.95	0.6	0.959
helloWorld	1.0	1.0	0.832

that the algorithm originally generated. With the "partial match" metric, as long as both the fix generated by our algorithm and the gold standard hint written by human experts are pointing toward the same bug or needed correction, the fix is likely to register as a match for that gold standard hint.

Table 1 shows the results of evaluating our algorithm against two versions of the QualityScore benchmark: (1) comparison to "gold standard" hints, which are hints that at least two experts agreed on; and (2) comparison to all hints that at least one expert thought were valid. Each score represents the average fraction of hints per student solution which matched a human-generated hint in the comparison dataset. So the scores can range from 0 (no valid hints generated) to 1 (all generated hints were considered valid). As Table 1 shows, our algorithm outperformed ITAP in both versions of the benchmark, although hints generated by human tutors still did better overall.

The QualityScore dataset consists of several incorrect student solutions for each of five different programming problems. Table 2 and Table 3 show the per-problem breakdown of the results. We can see that ITAP did outperform our algorithm on two of the problems, though the significance of this difference is hard to determine due to the small size of the dataset. For example, for the firstAndLast problem, there were only 7 student solutions, and ITAP algorithm scored better than our algorithm on exactly one of them.

From Table 3, we can also observe that on three out of the five problems, our algorithm got a perfect or nearly-perfect score when compared to all expert-generated hints. This means that for those problems, all fixes generated by our algorithm matched hints that at least one human expert thought was valid.

4.4 Analyzing Non-gold-standard-matching Corrections

It was difficult to systematically analyze the instances where our algorithm's output did not match any gold-standard and expert-generated hints, and understand why those particular corrections were considered invalid, because the data format used in QualityScore is not very human readable. Each hint or correction is encoded as a custom JSON representation of the AST of the corrected code, with some embedded metadata, and there does not seem to be a way to convert it back to a valid Python AST representation.

Nevertheless, we have investigated the situations in which our algorithm did not generate any gold-standard corrections. We organize this discussion based on how the corrections fell short of the gold standard. Specifically, we consider the following groups:

- Corrections that did not match any gold-standard hints, but did match expert-generated hints.
- Corrections that were conceptually similar to an expert-generated hint, but were not matches due to the match policies used.
- Corrections that did not match any gold-standard or expert-generated hints.

4.4.1 Matching Only Non-gold-standard hints. Our algorithm generated seven corrections that were considered invalid when compared to the "gold standard" dataset, but valid when compared to all hints generated by all experts. This means that each of these corrections matched some expert-generated hint, but that expert-generated hint did not make it into the "gold standard", which has a higher standard of two out of three experts agreeing that the hint is valid.

We spot-checked two of these corrections to try to understand why they did not meet the higher bar, since the corrections seemed to suggest sensible edits.

One of these instances dealt with a case where the student did not have any code written at all, but requested a hint. Our algorithm generated two suggestions: start writing a necessary import line (import string), and start writing the function definition. These matched the two expert-generated hints in the QualityScore dataset, but one of these hints was not marked as a "gold standard" hint.

The other instance was in a program attempting to find the kth digit of a number. The student wrote a mathematical expression for doing so, but made two mistakes: using `%` (modulo) instead of `//` (integer division), and dividing by `k` instead of `k-1`. Our algorithm suggested corrections for both of these mistakes, and again, both corrections matched expert-generated hints, but one of them - the `k-1` correction - was not considered to be part of the gold standard. Interestingly, other identical corrections to off-by-one errors elsewhere in the dataset **are** included in the "gold standard" set of hints.

Without being able to thoroughly examine the entire set of expert-generated hints, it is difficult to hypothesize why these particular hints were excluded from the gold standard. Nevertheless, they seem to represent sensible corrections to the student code. Therefore, we consider the more permissive standard - hints that were considered valid by at least one expert - to be a better comparison point for determining whether the corrections generated by our algorithm are valid and sensible.

4.4.2 Conceptually matching corrections. We examined each of the corrections generated by our algorithm for the first two problems (firstAndLast and isPunctuation) that were judged to not match any expert-generated hints. We manually compared these corrections to the corresponding hints in the QualityScore dataset, and found that four of these corrections did have a conceptually matching hint. However, they were not recognized as a valid match by QualityScore.

Out of these four corrections, three were single-node edits which just updated the value of one node in the AST. For example, in one problem, the student tried to use the keyword `string` as a variable name, but this confused the Python interpreter. Our algorithm generated one correction changing `string` to `character` for each incorrect use of this variable. The corresponding expert-generated hints in these three instances seemed to update those same nodes, but also make additional changes. So conceptually, the minimal edits generated by our algorithm were partial matches to the expert-generated hints. But in the QualityScore algorithm, a partial match is only possible if it includes a matching insertion operation. So these three single-node edits were not considered valid partial matches.

The fourth instance involved a case where the student both made errors in calculating the intended return value, and forgot to return it. Our algorithm generated a fix which inserted a return line, added necessary code, and moved the valid part of the student's

code to the new return line. The "minimal" subset of edits for this fix was to insert a return statement on a new line. However, the corresponding hint or hints in the QualityScore dataset seem to add the return statement directly to the line with the student's incorrect calculation. In fact, QualityScore's heuristic for deriving edit scripts from corrected ASTs does not seem to account for move operations at all.

4.4.3 Corrections with no matches. Three of the corrections generated by our algorithm for firstAndLast and isPunctuation did not correspond to any expert-generated hints in the QualityScore dataset.

One of these corrections addressed a bug where the student tried to access the last letter in a string using the expression `s[n]`, where `n` was the length of the string. Our algorithm changed the token `n` to the literal `-1`, which produced a correct Python expression for accessing the last element, `s[-1]`. However, this expression doesn't quite match the student's original intent, since it does not utilize the length `n` which the student explicitly calculated earlier in the code. So in this case, it makes sense that this correction was not present in the QualityScore dataset.

The second correction addressed a bug where instead of concatenating two strings with a `+`, the student created a tuple by putting a `,` between them. The set of expert-generated hints for that problem did not seem to include any hint which actually concatenated two strings. Instead, there was one hint which deleted the second string and the comma entirely. It is not clear whether the expert tutors generating these hints thought it would be incorrect to directly replace the `,` with a `+`, or simply did not consider this option.

The final correction inserted a `return False` statement into a function which was supposed to return a Boolean. Our algorithm's correction inserted it into an `else` clause of a student-created `if` statement. The set of expert-generated hints did contain a hint which added `return False`, but this hint added the line *after* the user-created if statement as a fall-through option. Both corrections produce code that returns `False` at the right time, assuming other mistakes are corrected in the code first. Again, it is unclear whether the experts creating these hints explicitly considered only one of these options to be pedagogically sound, or did not evaluate any other possible locations for inserting `return False`.

These results show that the fixes generated by our algorithm represent the types of fixes that humans consider valid at least comparably well to the state of the art. For more complex types of problems, the algorithm seems to perform significantly better than the state-of-the art, and closely matches the types of fixes that are generated by human experts. This is very promising in the context of generating interpretable, explainable hints.

Unlike ITAP, our algorithm achieves this while remaining quite general, without relying on domain-specific heuristics such as common esoteric beginner coding patterns. This may make it more robust and generalizeable.

5 LIMITATIONS AND FUTURE WORK

A major limitation of this paper is that we did not directly evaluate the quality and effectiveness of the runtime explanations to generated fixes, though the same runtime logic did contribute to the fixes evaluated using the QualityScore metric.

It would be difficult or impossible to evaluate the effectiveness of these explanations separately from any interface which presents them to a user, because the raw explanation data is not very human-readable, even for experts: it consists of two versions of a program, together with the entire memory state of a specific moment during the execution of each one, including all intermediate values of expressions (not just the ones stored in variables).

Therefore, we plan on evaluating the explanations, together with an interface that presents them to users, in the future. Nevertheless, the feature of finding these important moments during the execution of the code is an integral part of the algorithm, as it is used to make sequencing decisions for the generated fixes.

6 CONCLUSION

We presented a novel algorithm for generating fixes for incorrect student code, and explanations to these fixes which illustrate how the effect of running the code changes after each fix. We evaluated a Python implementation of the algorithm and showed that it generates concise fixes, and outperforms state-of-the-art algorithms with similar goals. We believe that this algorithm can be an important resource in being able to provide individualized help to beginner students who get stuck when trying to solve programming problems.

ACKNOWLEDGMENTS

This material is partially based upon work supported by the National Science Foundation under Grant No. iis-2128128

REFERENCES

[1] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *arXiv:1603.06129 [cs]* (March 2016). http://arxiv.org/abs/1603.06129 arXiv: 1603.06129.

[2] Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337, 1-3 (June 2005), 217–239. https://doi.org/10.1016/j.tcs.2004.12.030

[3] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change detection in hierarchically structured information. In *Acm Sigmod Record*, Vol. 25. ACM, 493–504.

[4] Georg Dotzler and Michael Philippsen. 2016. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 660–671.

[5] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[6] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Transactions on Computer-Human Interaction* 22, 2 (April 2015), 1–35. https://doi.org/10.1145/2699751

[7] Sebastian Gross, Bassam Mokbel, Benjamin Paaßen, Barbara Hammer, and Niels Pinkwart. 2014. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 10* 9, 3 (2014), 248–280. Publisher: Inderscience Publishers Ltd.

[8] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2018. Deep Reinforcement Learning for Programming Language Correction. *arXiv:1801.10467 [cs]* (Jan. 2018). http://arxiv.org/abs/1801.10467 arXiv: 1801.10467.

[9] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In

Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. 1019–1028.

[10] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale - L@S '17*. ACM Press, Cambridge, Massachusetts, USA, 89–98. https://doi.org/10.1145/3051457.3051467

[11] Timotej Lazar, Aleksander Sadikov, and Ivan Bratko. 2017. Rewrite Rules for Debugging Student Programs in Programming Tutors. *IEEE Transactions on Learning Technologies* 11, 4 (2017), 429–440.

[12] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. The Impact of Adding Textual Explanations to Next-step Hints in a Novice Programming Environment. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, Aberdeen Scotland Uk, 520–526. https://doi.org/10.1145/3304221.3319759

[13] Benjamin Paaßen, Barbara Hammer, Thomas William Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2018. The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *arXiv:1708.06564 [cs]* (June 2018). http://arxiv.org/abs/1708.06564 arXiv: 1708.06564.

[14] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015), 3.

[15] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157–173.

[16] Barry Peddycord III, Andrew Hicks, and Tiffany Barnes. 2014. Generating hints for programming problems using intermediate output. In *Educational Data Mining 2014*. Citeseer.

[17] Thomas Price. 2021. thomaswp/CSEDM2019-Data-Challenge. https://github.com/thomaswp/CSEDM2019-Data-Challenge original-date: 2018-12-30T21:05:59Z.

[18] Thomas Price, Rui Zhi, and Tiffany Barnes. 2017. Evaluation of a Data-Driven Feedback Algorithm for Open-Ended Programming. *International Educational Data Mining Society* (2017).

[19] Thomas W. Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A Comparison of the Quality of Data-driven Programming Hint Generation Algorithms. *International Journal of Artificial Intelligence in Education* 29, 3 (2019), 368–395. Publisher: Springer.

[20] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. 39–40.

[21] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (March 2017), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[22] Caroline P. Rosé, Dumisizwe Bhembe, Stephanie Siler, Ramesh Srivastava, and Kurt VanLehn. 2003. The role of why questions in effective human tutoring. In *Proceedings of the 11th International Conference on AI in Education*. 55–62.

[23] John Stamper, Tiffany Barnes, Lorrie Lehmann, and Marvin Croy. 2008. The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track*. 71–78.

[24] Kurt VanLehn, Stephanie Siler, Charles Murray, and William B Baggett. 1998. What Makes a Tutorial Event Effective? (1998), 6. 39.

[25] Kurt VanLehn, Stephanie Siler, Charles Murray, Takashi Yamauchi, and William B. Baggett. 2003. Why do only some events cause learning during human tutoring? *Cognition and Instruction* 21, 3 (2003), 209–249. 475.

[26] Christopher Watson, Frederick WB Li, and Jamie L. Godwin. 2012. Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *International Conference on Web-Based Learning*. Springer, 228–239.

[27] Kurtis Zimmerman and Chandan R. Rupakheti. 2015. An automated framework for recommending program elements to novices (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 283–288.

[28] Stuart Zweben and Betsy Bizot. 2018. 2017 CRA Taulbee survey. *Computing Research News* 30, 5 (2018), 1–47.