Toward Data-Driven Example Feedback for Novice Programming

Rui Zhi
North Carolina State Univ.
Raleigh, NC, USA
rzhi@ncsu.edu

Samiha Marwan
North Carolina State Univ.
Raleigh, NC, USA
samarwan@ncsu.edu

Yihuan Dong
North Carolina State Univ.
Raleigh, NC, USA
ydong2@ncsu.edu

Nicholas Lytle
North Carolina State Univ.
Raleigh, NC, USA
nalytle@ncsu.edu

Thomas W. Price
North Carolina State Univ.
Raleigh, NC, USA
twprice@ncsu.edu

Tiffany Barnes
North Carolina State Univ.
Raleigh, NC, USA
tmbarnes@ncsu.edu

ABSTRACT

Viewing worked examples before problem solving has been shown to improve learning efficiency in novice programming. Example-based feedback seeks to present smaller, adaptive worked example steps during problem solving. We present a method for automatically generating and selecting adaptive, example-based programming feedback using historical student data. Our data-driven feature-based (DDF) example generation method automatically learns program features from data and selects example pairs based on when students complete each feature. We performed an experiment to compare three example generation methods: Student trace data, Data-Driven Features (DDF), and Expert examples. Two experts rated the quality of feedback for each generator, and they rated both the Expert and DDF example feedback as significantly more relevant to students' goals than the Student example feedback. However, there were no significant differences between the DDF and Expert examples. We compared these approaches to one that combined DDF with an Interactive Selection step (DDF-IS), where the user (in this case, an expert) selects their preferred data-driven feature before an example is selected. DDF-IS produced significantly more relevant examples than all other approaches, with significantly higher overall example quality than DDF. This suggests that our DDF approach allows more relevant examples to be selected than existing approaches, and that we may be able to leverage interactivity with the student to further improve example quality.

1. INTRODUCTION & BACKGROUND

Prior studies show that worked examples are an effective instructional support to help novices learn complex tasks [27, 30, 28, 8]. Sweller argues that "...for novices, learning via worked examples should be superior to learning via problem solving."[27]. In the domain of programming, researchers also suggest using worked examples to teach novices [3, 31]. Empirical studies have shown that interleaving worked examples with similar practice problems is more effective than solving only equivalent programming problems by writing code, as students spent less time on training tasks and performed better on a posttest [30]. However, worked examples are traditionally only offered to students in between problem solving attempts [30, 16, 7, 6], and they do little to assist students when they have difficulty *during* problem solving. Based on the idea of worked examples, researchers have explored example-based feedback [9, 4, 13], which shows a correct piece of code to help students learn during problem solving, as a form of adaptive, on-demand support [9]. Similar to a high-level on-demand hint, example-based feedback demonstrates one step in a correct solution to the problem the student is working on, selected adaptively to match the student's code. Keuning et al. argue for the need for such feedback in their review on automated feedback generation for programming, saying "the very low percentage of tools that give code examples based on the student's actions is unfortunate, because studying examples has proven to be an effective way of learning" [15]. As Gross et al. argue, showing a relevant partial solution could impose less cognitive load and be easier to visually present to novices than showing a full solution [9]. Ichinco et al. found that novices have trouble transferring what they learned from similar problems to their own code [14]. Example-based feedback addresses this by providing example steps from the *same* problem the student is working on. Figure 1 presents a prototype of what example-based feedback might look like in a block-based novice programming environment. The example is presented with a "before" state, similar to the student's current code, and an "after" state that completes a desired feature, helping the student easily identify the purpose and outcome of a single solution step.

Existing example-based feedback systems may rely on a library of expert-authored examples [11, 14, 4], which can be costly to maintain, as instructors are unlikely to create new examples [12]. Additionally, the example code may show an example solution to a related problem [11] that requires students to transfer knowledge to solve their current problem, which may be challenging for weaker students who need more help [13]. To address these limitations, we present a

Rui Zhi, Samiha Marwan, Yihuan Dong, Nicholas Lytle, Thomas Price and Tiffany Barnes "Toward Data-Driven Example Feedback for Novice Programming" In: *The 12th International Conference on Educational Data Mining*, Michel Desmarais, Collin F. Lynch, Agathe Merceron, & Roger Nkambou (eds.) 2019, pp. 218 - 227

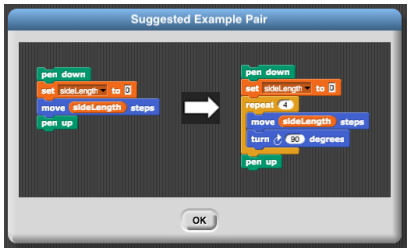

Figure 1: Example-based feedback prototype in iSnap.

data-driven method to create example-based feedback using historical student data. We focus on the domain of programming, where many problems have a vast space of possible solutions [24, 19], so a static set of examples is unlikely to be relevant to all students. To our knowledge, only Gross et al. have previously employed a data-driven method to derive examples adaptively to help students [9]. They found that examples derived from expert solutions are perceived by students as more helpful and help students make more solution improvements than those derived from student solutions. However, they simply presented the complete student solution which was most similar to the current student's code (according to a distance metric), rather than trying to systematically identify and address students' current programming goals. This suggests that existing evaluations of data-driven methods for example-based feedback have not explored the full potential of the approach. We hypothesize that with the innovations in this work, data-driven examples can be more adaptive than expert-authored ones, while still presenting correct and interpretable solution steps.

In this work, we present an approach to automatically detect students' progress towards a solution, and suggest example-based feedback from historical student data. Our data-driven method first processes prior correct student solutions to discover meaningful features, labels the parts of code that contribute to each feature, and removes code that does not contribute to the solution, to create simple data-driven code examples that contain only the code needed for each feature. We then automatically label the student's current code with a "feature state" representing the presence or absence of each data-driven feature needed for a correct solution. We then adaptively select example feedback that contains the same features as the current student code, and adds a new feature that is relevant to solving the current problem. In contrast, other example feedback systems show code from a related similar problem, requiring students to study the example and transfer what they learn to the current context [11].

We evaluated two data-driven methods to generate and select example-based feedback for historical student hint requests: Data-Driven Features (DDF), and Data-Driven Features with Interactive Selection (DDF-IS). We compared these methods against two baselines: (1) Expert-authored examples (Expert) and (2) examples generated naively from correct student solution traces, showing the code added between consecutive test runs (Student). The data-driven features (DDF) algorithm cleans prior students solutions and generates example pairs where the "start code" has the same features as the student's code, and the "end code" adds a new feature that is not yet present in the student's code. The DDF with Interactive Selection (DDF-IS) approach explored the potential to improve algorithmic DDF example feedback selection by having a user interactively select the data-driven feature with which they want help, before the algorithm selects an example. To simulate this experience, we used an expert to select this feature, representing a best-case scenario for interactive selection.

We adapted a multidimensional data-driven hint evaluation rubric from previous work to evaluate the example-based feedback quality based on Relevance, Progress, Interpretability, and Similarity. Our findings showed both the Expert and DDF feedback were significantly more relevant to students' goals than the Student feedback, but there were no significant differences between the DDF and Expert examples. This suggests that our DDF feedback can reasonably replace Expert-authored examples in situations where they are unavailable or difficult to scale. We also found that DDF-IS produced significantly more relevant examples than all other approaches, with the highest overall example quality, significantly higher than DDF. This suggests that in the best case, data-driven feedback may leverage interactivity with the student to further improve example quality.

The contributions of this paper are: 1) a data-driven algorithm capable of generating adaptive example feedback for students during programming, and 2) an initial evaluation showing that these adaptive examples can be more relevant than static, expert-authored examples.

2. METHOD

This work presents and evaluates a data-driven feature-based (DDF) method for generating and selecting example-based feedback (explained in Section 2.2). To evaluate our DDF approach, we generated example-based feedback for historical student help requests, and asked experts to evaluate the quality of each type of feedback, comparing against two baselines (explained in Section 2.3.1).

2.1 Dataset

Our dataset comes from iSnap [20], which extends the Snap*!* block-based programming environment with logging and on-demand, data-driven hint support. It logs student interactions with the system, including complete code snapshots after each edit. The data were collected during the Fall 2016 (F16), Spring 2017 (S17), and Fall 2017 (F17) semesters in an introductory computing course for non-majors, held at a research university[1]. In each semester, the students completed 3 in-lab assignments with access to help from teaching assistants and 3 homework assignments independently. In this paper, we selected one homework assignment *Squiral* for the example code generation and evaluation. In Squiral (shown in Figure 2), students program a "sprite" to draw a spiraling square-like shape using loops, variables, arithmetic operators and a custom block (function). Common solutions for Squiral contain 7-10 lines of code. The original dataset contains 57 (F16), 43 (S17), and 47 (F17) Squiral assignment submissions. Since iSnap offers students on-demand, data-driven hints which may alter students' problem-solving patterns, we exclude students who requests hints in the dataset

[1]All datasets are available at `https://pslcdatashop.web.cmu.edu/Project?id=321`

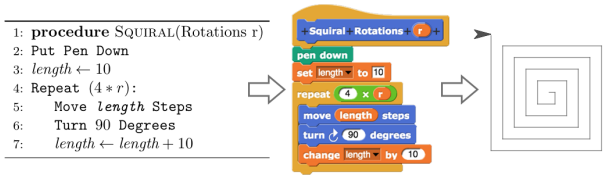

Figure 2: Squiral pseudocode, Snap*!* code and output [34].

used for the example-based feedback generation. Our remaining data contained 38, 29, 39 code traces for the F16, S17, and F17 semesters, respectively. Each code trace contains the set of timestamped snapshots that comprise all of a student's work on a problem.

2.2 Example-based Feedback Generation

We propose an algorithm to automatically generate data-driven, example-based feedback with the following high-level steps: 1) Extract a set of data-driven features from prior student code traces, which each describe a property of a correct solution. 2) Use the features to clean correct student code traces by removing extraneous code that does not contribute to a correct solution. 3) Generate pairs of example code snapshots that demonstrate how to complete a single feature. 4) Choose an appropriate, personalized example pair upon student request.

Our goal in this work is to perform a preliminary evaluation of our algorithm before implementing a user interface and evaluating its impact in practice. However, to contextualize our work, Figure 1 presents one way that such feedback could be presented to a student in iSnap when they request help. Our DDF example pairs consist of "start code," similar to the student's, and "end code" that demonstrates how to complete a single feature. Our design draws inspiration from a variety of theoretical and empirical sources, including cognitive load theory [29], Vygotsky's Zone of Proximal Development [32], worked examples [27], learning from subgoals [16], and compare/contrast tasks [17].

Research on worked examples suggests that seeing examples which break a problem down into sequential steps (e.g. features) can be a more efficient way of learning than problem solving [27, 30, 28]. According to cognitive load theory, worked examples are effective because they lower the extraneous cognitive load (mental effort) imposed by the instructional materials. We designed our examples to present steps as a pair of "start" and "end" code, as previous studies have shown that comparing and contrasting examples is an effective learning activity in many domains [17, 26]. The example is adaptively selected to keep students in the Zone of Proximal Development [32] by starting with code similar to the student's, which they can already understand, and scaffolding the completion of a new feature, which they cannot yet accomplish on their own. The work of Morrison et al. [16] suggests that programming examples that are broken into subgoals can help improve learning for novices. When examples are isomorphic to the problem solving task, as in our case, they found that it is most effective for students to label subgoals themselves, a feature we could easily incorporate into our example-based feedback.

2.2.1 Step 1: Data-driven Feature Generation

The goal of an example pair is to present how a meaningful self-contained portion of solution code, or *feature*, can be completed. An assignment may have students program multiple features, and the final correct solution should have all the correct features present. For example, in Squiral (shown in Figure 2), a feature could be to move the sprite in a square shape, draw some figure on the screen, or repeat the spiral the correct number of times. In our previous work [34], we manually defined expert-authored features (shown in Table 1) in a systematic way, and we also implemented a data-driven algorithm to automatically identify code features from student solutions. Our results showed that many of the data-driven features were easily interpretable and closely matched the expert-authored features. The two methods also had moderate agreement on whether a given student was in the same state or different states.

The full procedure for data-driven feature extraction is given in [34], but we outline its high-level steps as follows:

1) **Preprocess student solutions**: Some student solutions may contain extraneous code or procedures that were used for testing or resetting the environment, which we attempt to remove before extracting features. Specifically, we used the SourceCheck algorithm [21] to identify and remove whole scripts and procedures that do not match any element of an expert-authored solution in the correct student solutions.

2) **Generate code shapes**: To identify common code patterns in correct student solutions, we extract a set of *code shapes*, or syntactic structures, from the solution code by converting students' solution code into abstract syntax trees (ASTs) and then identify all pq-Gram subtrees [1] in each AST, to form our initial set of "code shapes." This includes all code shapes from all correct solutions.

3) **Remove duplicates**: The initial set of code shapes may include very similar shapes, including some AST patterns that are subsets of others. Therefore, we remove these duplicates by measuring the co-occurrence of code shapes in all student code traces and keeping only the more specific code shape and discard the other duplicate if two code shapes almost always appear together in the same code.

4) **Identify decision shapes**: Due to varied problem solving strategies, some code shapes may not appear in every correct solution. For example, a Squiral solution can either use nested repeat or a single repeat block to rotate the correct number of times (as shown in Figure 2), but not both. Therefore, we define a *decision shape* as a disjunction of code shapes, where almost all solutions contain exactly one of the component code shapes. A decision shape is present in a solution if any one of its code shapes exist in its code.

5) **Filter out uncommon code and decision shapes**: Since we are interested in using code and decision shapes to represent features of a correct solution, we keep only those shapes which appear in the vast majority of correct solutions, and filter out the rest.

6) **Form features**: Our goal is to define a small set of features that collectively represent a complete solution. However, the previous steps will generate tens to hundreds of code and decision shapes for a relatively simple problem like Squiral. We therefore combine these smaller shapes into larger features using a form of hierarchical clustering. We iteratively combine any two features that most frequently

Table 1: Expert Features and Corresponding Data-Driven Features, as derived in [34].

Feature Name	Brief Description	Data-driven Analogue
E1. Procedure	Primary code inside of a procedure.	D1: Create a procedure OR a variable.
E2. Draw Anything	Able to draw anything on screen.	D8: Use a 'repeat' AND create a variable OR a parameter.
E3. Move 'Square -like'	Able to move sprite in a square-like fashion.	D11: Have a 'move' AND a 'turn' in a 'repeat' AND have a 'pen down.'
E4. Correctly Use Parameter	Correctly uses parameter within custom block.	D4: Have a 'repeat' inside a procedure.
E5. Repeat Correct # of Times	Repeats square-like movement correct number of times.	D5: Have a 'multiply' block with a variable OR two nested 'repeats'.
E6. Move 'Variably'	Movement is based on a variable not literal amount.	D10: Have a 'move' with a variable argument inside of a 'repeat'.
E7. Move 'Squirally'	Increase length to move for each side.	D7: Change a variable inside a 'repeat'.

co-occur across student data until the size of the observed state-space defined by the features starts to decrease rapidly.

7) **Represent student code by feature vectors**: Once the features are formed, we can represent a student's current code as a vector indicating the presence or absence of each feature. A student starts with a feature state of all 0s, and a correct solution should have all features present, resulting in a vector of all 1s.

2.2.2 Step 2: Cleaning Student Code

To extract good example pairs from prior student traces, we need to first remove excess blocks which do not contribute to a correct solution, as these may distract students and make examples harder to interpret. Identifying the excess blocks can be difficult, especially for intermediate partial solutions, since students can construct solutions in a large variety of ways [21, 24]. We address this by leveraging the features we defined in step 1 to exclude *irrelevant code*, which does not belong to any feature. Specifically, our cleaning procedure removes one node from the abstract syntax tree at a time (including all its children), then checks whether removing this node causes a currently completed feature to become incomplete. If removing a node "breaks" a feature, we assume that it is necessary and add it back; otherwise, we remove it. We also check to make sure removing the node does not break any code dependencies, such as deleting a variable declaration when the variable is used elsewhere. We iterate over every node in a recursive, breadth-first manner, starting from the root node. Once this iteration stops, it produces a cleaned partial solution, where all irrelevant code has been removed. We apply this cleaning procedure to *all* snapshots in correct solution traces. With well-defined features, this process can effectively clean a large variety of both partial and complete solutions, ensuring that all remaining code is useful. We use this process both for cleaning code and for extracting example pairs, as described in the next step.

2.2.3 Step 3: Extract Example Pairs

Our goal is to create a database of correct, meaningful, and self-contained example pairs to offer as feedback to students. Naively, we could extract a single example pair for each feature from each correct code trace, since each student completed each feature at least once. However, we want to generate as many example pairs as possible, so that the algorithm can adaptively select one that is similar to another student's code. We therefore developed a method to generate many "synthetic" example pairs, each consisting of a pair of code states $(c_0, c_1)_i$, from any cleaned student code trace. Recall that each pair should cleanly demonstrate the completion of exactly one feature by contrasting a "start code" state (c_0) and an "end code" state (c_1). The algorithm first extract one example pair each time a student completes a feature f_i, with c_1 defined as the snapshot right after f_i was completed, and c_0 as the snapshot right after the *prior* feature f_{i-1} was completed. We generate additional example pairs from each cleaned snapshot in a student solution trace with the following procedure. For each snapshot, the algorithm labels it as an end state, c_1. It then removes exactly one data-driven feature from c_1 to create a c_0, and together these form the example pair. This feature removal is accomplished using the code cleaning procedure described above; however, instead of removing irrelevant nodes, we use it to remove whole features. The algorithm first tries to remove one leaf node, l_i, at a time. Since the snapshot has already been cleaned, removing this node will either create an invalid code state, or cause a feature to become incomplete. In the later case, the cleaning procedure was run to remove all other code associated with the removed feature. The resulting cleaned code becomes the c_0 for the example pair, and our cleaning procedure guarantees that c_0 will have exactly one less feature than c_1. The $(c_0, c_1)_i$ pair is added to a list which stores our example pairs. The algorithm then repeats this process recursively on c_0, which becomes the c_1 for new example pairs, until no new pairs can be generated. In this case, we generate many example pairs per *snapshot*in a solution trace. While some are redundant, many are unique.

2.2.4 Step 4: Select an Example Code Pair

When a student requests help, we aim to provide them with the most appropriate example pair in our database as feedback. We define two ways that we can identify this example pair: 1) a **Data-Driven Features (DDF)** approach, using an algorithm to select the best example pair, or 2) an **Data-Driven Features with Interactive Selection (DDF-IS)** approach, giving the student the information needed to select an example pair. We first consider the DDF approach, in which we attempt to select the example pair which is most similar to the current student's code. Based on this selection criteria, the selected example code pair should be very similar to the student's code, with the goal of minimizing

the effort needed to process the "start code" and allowing the student to focus on the feature demonstrated by the example pair. In this study, for the DDF feedback, we selected an appropriate example code pair as follows:

Generate proper example pair candidates: To filter out inappropriate pair candidates based on the student's completed features, we select only those example pairs whose "start code" has the same features as the current student code, from the example pair lists (generated in Section 2.2.3). If we cannot find any, we sort the example pairs by the Hamming distance between the feature states of "start code" and student's current code. Then we select the example pairs with the closest start-state to student's current code. If there are multiple example pairs available (they all have the same start state), we use the SourceCheck algorithm to sort the example pairs based on the similarity between the "start code" and student's current code.

Select one example pair from candidates: We iterate over the example pair candidates and select exactly one pair, which accomplishes a feature that the student has not finished yet, preferring features that the majority of students in a state similar to the current student will take next.

To generate *interactive* example-based feedback, we need a way to communicate to the student which features have an available example that they can request. By default, our features are unlabeled (having been generated automatically from data), but with a small amount of instructor effort (about 3 minutes), they can be labeled. By viewing the code shapes required by each feature, an instructor familiar with the problem can generate a short, human-readable description, such as "move the sprite using a variable", or "make the sprite move further each time". These options can then be shown to a student. Once the student has selected a feature for an example, we select an appropriate example pair that accomplishes that feature with start code matching the student's current feature state. If there are multiple options, we use the same criteria as in DDF: select the example pair with the most similar start state to the student's code, which contains a proper subset of the student's features. Note that unlike in DDF, a student *could* request an example for a feature that they have already completed. Students might decide to do this when they are unsure if they have completed a feature correctly and want to see an example for confirmation.

2.3 Expert Evaluation

To test the feasibility of our method before building the whole system and conducting a user study, we did a preliminary expert evaluation of our algorithm. We generated example pairs to support student code snapshots from our historical dataset and evaluated their quality. To simulate real student help requests where an example might be needed, we selected snapshots that corresponded to times when historical students requested hints from iSnap. As in prior work on evaluating feedback [22], we sampled up to two hint requests (and their corresponding student code snapshots) from each student. We sampled 50 hint requests in total including 20 in F16, 20 in S17, and 10 in F17. For each hint request we generated four example code pairs using different techniques: DDF, DDF-IS, and our two baselines, Student and Expert (explained in the next section). For the examples derived from student data (DDF, DDF-IS, Student), we generated examples using semester-based 3-fold cross-validation. For each semester, we used the other two semesters' data as training data to generate the examples. We derived 9, 9, and 10 data-driven features for F16, S17, and F17, respectively.

Two co-authors, who neither authored examples nor worked on the algorithm itself, served as experts to evaluate the generated example pairs. Both experts have extensive experience in Snap*!* and the Squiral assignment. We built an interface in iSnap to present each expert with the student's original code and the example pair. Then we asked the experts to assess the example pair based on a detailed example code rating rubric[2], adapted from [22]. Our rubric has 4 attributes, each rated 1, 2 or 3, with higher scores being better. These 4 attributes measured: 1) **Relevance**: how relevant the suggested example code pair is to the student's current goals, 2) **Progress**: how well the example code pair helps students make progress towards the final correct solution, 3) **Appropriateness & Interpretability**: how likely a tutor will be to suggest this example pair to a student and how easily a novice could understand the intention of the suggested example pair and 4) **Similarity**, how similar is the "start code" to the student's code. The first 3 attributes are meant to assess the quality of the example. The 4th is meant to help us understand the relationship between example similarity and quality, since all examples were selected based on their similarity to student code. During evaluation, experts had access to students' code history, and based their ratings on the student's individual context.

To ensure that the two experts had a similar understanding of the rubric, they rated 10 examples together, which were not used in this study. They then rated the 200 examples pairs used in this study in 2 rounds of 100 each[3]. In Round 1, they independently rated 40 example pairs and then discussed their ratings to resolve any conflicts and reach consensus. Their inter-agreement reliability across the 40 example pairs achieved squared-weighted Cohen's kappas of 0.94, 0.91, 0.82, 0.90 for Relevance, Progress, Appropriateness & Interpretability, and Similarity, respectively, indicating very strong agreement. They then split the remaining 60 pairs and rated them individually[4]. In Round 2, one expert rated the remaining 100 pairs individually, and the other expert rated 75 of these, which were discussed until consensus was reached. Across the 115 example pairs that were rated by both experts, the total squared-weighted Cohen's kappa was 0.77.

2.3.1 Baselines

We compare both the interactive and non-interactive data-driven example-based feedback against two baselines: Expert-authored examples and examples generated naively from Student data. For data-driven example-based feedback, we use the two strategies (DDF and DDF-IS) described in section 2.2.4. Our goal for the **Expert baseline** was to reflect

[2]Available at: `http://go.ncsu.edu/edm2019-rubric`

[3]Due to the order in which examples were generated, Round 1 included DDF and Expert examples, and Round 2 contained Student and DDF-IS examples. However, raters were blind to the condition of the example.

[4]The rated examples were in DDF and Expert conditions.

a straightforward way of generating example-based feedback using a small number of expert-authored example pairs, which an instructor might reasonably create. This corresponds to the Next Step of the Nearest Sample Solution (NSNSS) strategy introduced by Gross et al. [9], which they found to be optimal. This strategy selects the next step of the nearest expert solution. To generate expert example pairs, two experts (specifically, two co-authors who did not rate the example pairs during evaluation) manually authored example pairs for *all* steps in the most common solution paths, which at least 10% of students took to solve the problem. Each of the two co-authors created the example code separately, based on the understanding that the example code should be useful to students and no extra explanations will be used to help students understand the suggested example code. During this process, the experts could review as many students' code (including all the history) as they needed. Afterwards, they met and discussed all the example pairs that they authored and came to consensus on the example pairs. In the Squiral assignment, the common solution graph has 14 nodes and 13 edges, of which 7 were on a primary solution path and 6 were on two alternative paths. To select an Expert example-pair for a student, we first used the SoureCheck algorithm to sort the Expert example pairs based on the size of the "start code" and the similarity between their "start code" and the student's current code. We select the example pair whose "start code" accomplishes fewer features than the student's code and is very similar to the student's code.

It may seem unfair to compare the Interactive Data-driven examples to (non-interactive) Expert examples. However, we note that it would not be reasonable to create an *Interactive*-Expert baseline. Since the Expert examples consist of a small number of hand-authored examples, which completed features in a specific order (e.g. Feature 1 is *always* completed before Feature 2), it is not reasonable to give a student the option of selecting a desired example. If the student has only completed 2 features, they could simply select the example for the final feature and see a full solution. Since the goal of example-based feedback is to show only a single, incomplete feature, we always selected the closest Expert example-pair to the student's current code. By contrast, our data-driven approach generates enough example-pairs that we are able to provide many choices of features to complete, without revealing any other features in the process.

Our goal for the **Student baseline** was to reflect a naive approach to extracting examples from student code, without the feature-based cleaning and selection of our own algorithm. Gross et al. [9] defined their baseline of student-derived examples to show only students' submitted solutions, essentially giving away the whole answer. Instead, our baseline extracts multiple examples from students, based on when they ran their code. We hypothesized that students often run their code when they have completed a meaningful feature, making this a meaningful way to demarcate examples. We extracted examples from all correct student solution traces that did not request help. We selected student code snapshots based on when they ran their code. We treated consecutive run events within 15 seconds of one another as a single run event, and we took the last of these events as the boundary between example pairs. For each consecutive pair of run events (more than 15 seconds apart), we extract an example pair consisting of the two corresponding snapshots. The code snapshot that happened earlier serves as the "start code" of the example pair, and the one happened later serves as the "end code". When selecting an example pair to show as feedback for a given student, we used the SourceCheck algorithm to find the nearest "start code" and present that example pair as feedback.

3. RESULTS

We structured our analysis around the following research questions: **RQ1:** Can we create useful example-based feedback naively from student data, without a data-driven algorithm? **RQ2:** How does the quality of data-driven example-based feedback compare with that of expert-authored example-based feedback? **RQ3:** Can the quality of data-driven example-based feedback be improved if an example is selected interactively, rather than automatically?

We address each RQ by comparing the quality of example-based feedback generated by four feedback approaches explained above: 1) Expert-authored (Expert), 2) Naive Student Data (Student), 3) Data-Driven Features (DDF), and 4) Data-Driven Features with Interactive Selection (DDF-IS). We evaluated quality in terms of Relevance, Progress, and Interpretability, as explained above. For RQ1, we hypothesized that our results would be consistent with Gross et al. [9] that naively extracted student examples would not lead to high-quality feedback. For RQ2, we hypothesized that data-driven example-based feedback would be more relevant to the student's code than expert feedback and just as useful otherwise. For RQ3, we hypothesized that interactive selection would improve the quality of data-driven example-based feedback.

3.1 Feature Coverage

From 106 student solution traces, the DDF algorithm was able to generate 13,927 unique data-driven examples, as shown in Table 2. Of these, only 728 (5%) were derived directly from a student's trace, and the rest were generated with the recursive algorithm explained in Section 2.2.3. It took around 10 minutes (645 seconds) to generate all the data-driven example pairs. Table 2 shows the number of examples generated by each algorithm. It also gives the "snapshot coverage" and "hint request coverage" of each algorithm. The former refers to the percent of all observed snapshots which had an available example in the same feature-state (meaning the example started with the same set of features completed). For this calculation, we used the expert-authored features defined in [34]. Hint request coverage considers only the 50 hint request snapshots we evaluated. These numbers are averaged across the 3 semesters.

Table 2: Total number of generated example pairs, corresponding average snapshot coverage and hint coverage in the expert-defined feature space.

Algorithm	**DDF & DDF-IS**	**Stud.**	**Exp.**
# of examples generated	13,927	242	13
Snapshot coverage	0.843	0.670	0.709
Hint request coverage	0.821	0.709	0.687

3.2 Expert Ratings

Our dataset consists of 50 hint requests and 200 example pairs (one example pair per feedback approach for each hint request). All the example pairs were rated on four attributes on a scale of 1-3: Relevance, Progress, Appropriateness & Interpretability, and Similarity, as described in Section 2.3. We found the first three attribute ratings showed significant positive pairwise Spearman correlations ranging from 0.51 to 0.81 (all $p < 0.001$). Similarity also had a lower, positive correlation with the other attributes, ranging from 0.24 to 0.34 (all $p < 0.001$). Due to the high positive correlation of the Relevance, Progress, and Appropriateness & Interpretability attributes, we also compute a Quality attribute, which sums all three attributes, with scores ranging from 3 to 9. Because all 4 example-based feedback approaches selected examples using SourceCheck's code similarity function, we also investigated the relationship between SourceCheck's calculated similarity and the expert-rated Similarity. We found a significant, positive correlation ($\rho = 0.29; p < 0.001$), suggesting that the similarity function is reasonable but could be improved. Table 3 reports mean values of each attribute for each example-based feedback approach[5].

Table 3: Mean attribute ratings (with standard deviation) for example pairs in from each approach.

N = 50	**Relev.**	**Prog.**	**A.&I.**	**Qual.**	**Simil.**
DDF-IS	2.62 (0.73)	2.46 (0.76)	2.22 (0.82)	7.30 (2.05)	2.22 (0.86)
Expert	2.24 (0.89)	2.36 (0.80)	2.14 (0.90)	6.74 (2.24)	2.18 (0.80)
DDF	2.12 (0.92)	2.06 (0.89)	1.84 (0.82)	6.02 (2.38)	2.28 (0.88)
Student	1.72 (0.90)	2.12 (0.90)	1.80 (0.86)	5.64 (2.20)	2.24 (0.87)

To address our research questions, for each attribute we used Kruskal-Wallis test to determine if there was a significant difference in ratings across feedback generation approaches. For the overall Quality attribute, we found a significant difference among conditions ($\chi^2(3) = 16.06, p < 0.001$). We performed a post hoc Dunn's test with Benjamini-Hochberg correction for multiple comparisons[6] [2, 5] to identify pairwise significant differences between approaches. This showed a significant difference between Expert and Student examples ($z = 2.48, p = 0.026, r = 0.25$[7]), DDF and DDF-IS ($z = 2.76, p = 0.017, r = 0.28$), DDF-IS and Student ($z = 3.69, p = 0.0013, r = 0.37$), suggesting that for overall quality, Student < DDF < DDF-IS, and Student < Expert.

We then inspected the difference for each individual attribute. A Kruskal-Wallis test showed a significant difference among approaches for the Relevance ($\chi^2(3) = 24.45, p < 0.001$). A post-hoc test using Dunn's test with Benjamini-Hochberg correction showed a significant difference between DDF and Student ($z = 2.14, p = 0.049, r = 0.21$), Expert and Student ($z = 2.79, p = 0.016, r = 0.28$), DDF-IS and DDF ($z = 2.76, p = 0.011, r = 0.28$), DDF-IS and Student ($z = 4.90, p < 0.001, r = 0.48$), DDF-IS and Expert ($z = 2.11, p = 0.04, r = 0.21$), but we did not find a significant difference between DDF and Expert ($z = 0.65, p = 0.515, r = 0.07$). This suggests that for Relevance, Student < DDF = Expert < DDF-IS.

We also found a significant difference among conditions for the Appropriateness & Interpretability ($\chi^2(3) = 8.98, p = 0.030$). However, a post-hoc test using Dunn's test with Benjamini-Hochberg correction did not find any pairwise significant differences between conditions. For the other two attributes, similarly, we did not find any significant difference between conditions: Progress ($\chi^2(3) = 7.14, p = 0.068$); Similarity ($\chi^2(3) = 0.62, p = 0.89$). The results suggest that DDF-IS example pairs are more relevant to student code and are equally helpful and interpretable compared with expert-authored examples.

3.3 Inspection of Examples

To better understand our quantitative results, we manually inspected examples generated by each approach to understand how they differed and how those differences impacted the expert ratings. We investigated the following questions and present situations that highlight possible answers:

Why does the naive Student baseline have low quality? Our quantitative results show that our naive Student baseline does not produce useful example pairs. We manually investigated some Student example pairs to better understand why. Recall that the Student baseline extracts examples showing the changes between consecutive run of student code. We derived 242 of these example pairs across 3 semesters from 39 correct student submissions that did not request hints. Upon inspection, it is clear that this segmentation approach did not always produce meaningful example pairs. For example, a Student example pair might simply rearrange the order of some code blocks. This is a common debugging behavior, but it does not usually yield a useful example. We also found that even when Student example pairs demonstrated a meaningful step, they were often selected for students who had already completed that feature. This is not surprising, since the Student baseline selected the example with the most similar start code, but this does not guarantee the student will not have additional features completed. Lastly, many Student examples contained extraneous code that made them harder to interpret. These three problems – lack of meaningful steps, repeating completed steps, and extraneous code – are all addressed in our DDF-based example selection. With good features, we can identify meaningful example steps, ensure that all examples complete new features, and remove extraneous code.

When were data-driven examples more adaptive than Expert examples? Our quantitative results suggest that the DDF-IS approach was able to generate examples that were significantly more Relevant than those of the Expert approach. We hypothesized that this was enabled by the large number of unique example pairs generated by DDF (13,927). By contrast the Expert approach used only 13 examples pairs, which covered the common solution paths

[5]Although attribute ratings are ordinal, we report the mean and SD, since the median values for each attribute are generally the same (2 or 3)

[6]We report p-values corrected with the Benjamini-Hochberg procedure to control the false discovery rate at 0.05, keeping the α significance threshold at 0.05.

[7]The effect size r is calculated as described in [25]

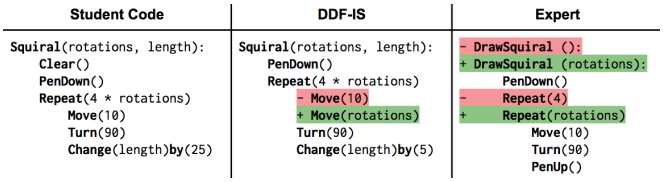

Figure 3: DDF-IS and Expert examples for one snapshot (red/green indicate changed code in the example).

that at least 10% of students took to solve the problem. While some of DDF-IS's improvement may have come from its Interactive Selection step (explored below), our results in Section 3.1 also show DDF examples have larger coverage than the expert-authored examples for both the hint requests and student snapshots. Our manual inspection uncovered a number of situations when a relevant expert Example was not available, but a DDF example was. For example, a strong minority of students solved the Squiral problem by using a second input (parameter) to store the side length of shape. Figure 3 shows that for the same hint request, the DDF-IS can suggest examples not only similar to the student code, but also relevant to what the student is working on. However, the expert-authored example only has one parameter in the custom block and suggests something that the student has already done. The Expert example becomes less helpful and irrelevant when a student has solution that deviates from the Expert examples. However, our data-driven examples are more adaptive in this scenario and can provide examples both similar and relevant to the student. We note that this adaptivity was enabled in large part by the recursive example generation algorithm outlined in Section 2.2.3. Only 2 of the 50 examples selected by DDF-IS were extracted directly from student code traces (the "naive approach"); the other 48 were generated synthetically.

When did DDF-IS select better examples than DDF? Since DDF and DDF-IS were selecting examples from the same pool of examples, but DDF-IS has significantly higher overall quality, we wanted to understand where the selection algorithms differed. We investigated some pairs and focused on when our algorithm failed to select relevant examples. In those scenarios, we found that the DDF would suggest the student to add a 'pen down', or add a custom block in the main script area. Those features are necessary for a correct solution, but they may have lower priority compared with other features such as "control the sprite to draw a square" or "repeat the spiral the correct number of times." Additionally, we also found that the data-driven feature detector sometimes counted a feature as complete when it still had a bug or a missing block. Thus, it would suggest example pairs that accomplish new features without fixing the broken one. In the opposite case, the student may have already finished a feature, but the feature detector sometimes failed to detect it, showing a redundant example. This occurred frequently when students accomplished a feature in a unique way, not captured by our data-driven feature definition. For example, a few students attempted to use a recursive approach to solve Squiral, but since this was quite uncommon, our data-driven features failed to generate meaningful example pairs for this solution. The DDF-IS can help resolve the first two cases, since students can choose whether they need help on a feature that the DDF failed to detect.

4. DISCUSSION

RQ1: Can we create useful example-based feedback naively from student data, without a data-driven algorithm? Our results suggest that naively extracting examples from student code based on when students ran their code does not yield high-quality example-pairs. These naive Student examples were lowest or second-lowest, rated on every dimension, with significantly lower Relevance than all other approaches and significantly lower overall Quality than the DDF-IS approach and the Expert-authored examples. This is consistent with our hypothesis for RQ1. It is also in agreement with the work of Gross et al. [9]., who found that using student solutions as example-based feedback were rated lower by experts and led to less improvement in students than other, expert-authored examples. However, rather than using a student's submitted code as an example as Gross et al. did, we used the changes that students made between running their code. We believe that this represents a more reasonable student baseline, since it only shows a part of the answer like the other feedback approaches. It should also theoretically limit the cognitive load needed for students to learn from the examples. Our results show that even so, a naive approach does not create high-quality student examples, compared to other approaches. Our data-driven approach attempts to address this by creating examples based on when students completed features.

One of the goals of using student data to generate examples is that they can more closely match the student's current code. In fact, we were able to extract 242 unique Student examples, compared with 13 for the Experts. This did enable our system to identify Student examples that were more *Similar* to the help-requesting student's code than Expert examples (though not significantly more). However, it is also clear that Similarity alone did not translate into Relevance, since the Student examples had the lowest Relevance scores. We cannot guarantee that students make meaningful changes between two consecutive snapshots, and thus the generated example pairs may not accomplish a meaningful chunk of code. For example, the Student example pair may suggest something that a student has already done. In this case, we could use the data-driven features to identify a later snapshot with more relevant changes as the "end code." This suggests the need for a more deliberate, data-driven process that can still create a large number of examples to select from without sacrificing quality.

RQ2: How does the quality of data-driven example-based feedback compare with that of expert-authored example-based feedback? We hypothesized that our Data-Driven Features (DDF) examples would be more relevant to help-requesting student's code than the expert baseline, with similar levels of Progress and Interpretability. However, our results did not support this hypothesis. There were no significant differences between the two approaches, with the DDF algorithm performing similarly on Relevance and slightly worse on Progress and Interpretability. Our original hypothesis was based on the premise that data-driven examples would be more Relevant, since they are selected from a much larger database of examples (4,000 to 5,000) that represents a large variety of student code. While this was

not confirmed, our results do suggest that our current, *non-interactive* data-driven algorithm performs significantly better than naive Student examples and only marginally worse than Expert examples. Since it is often difficult to get instructors to author examples [12], this suggests our DDF examples should be a reasonable substitution.

There are a few possible explanations for the poorer performance of DDF. First, it is possible that Squiral problem we analyzed may not have been complex enough to necessitate generating a large variety of examples, which is one of the key proposed advantages of our algorithm. If most hint-requesting students performed the steps of the problem in the same order as the Expert examples, these Expert examples would generally be Relevant, leaving less room for the DDF examples to improve upon them. We have some support for this explanation, since across semesters 53.8% to 85.7% of the hint requests we analyzed had an expert feature state that matched one of our Expert examples exactly. However, we also note that the expert baseline was rated less than 3 for Relevance 46% of the time, so there was clearly room for improvement, as we discuss later.

It is also clear that the DDF approach did not perform as well as expected, scoring between the Expert and Student baselines for Progress and Interpretability. There are two possible explanations for this: either the algorithm is failing to *generate* good example pairs, or it is generating useful example pairs, but failing to *select* the best pair to show a given student. The higher scores of the DDF-IS algorithm, which generated the same example pairs as DDF, but allowed a human to select the best one, suggest that the problem lies in the DDF algorithm's selection of example pairs. This selection process (explained in Section 2.2.4) primarily uses the SourceCheck algorithm to identify the example with the most similar start state to the student's code. While this approach did lead to DDF having the highest Similarity ratings, there was only a $\rho = 0.238$ correlation between Relevance and Similarity. This suggests that Similarity alone is not a good proxy for example Relevance or overall quality. The challenge of automatically selecting an appropriate example also reflects prior work on data-driven feedback [23], where low-quality hints arose because the hint generation algorithm was unable to identify the most useful hints and filter out less useful hints.

RQ3: Can the quality of data-driven example-based feedback be improved if an example is selected interactively, rather than automatically? Our results show that the quality of data-driven example-based feedback can be improved if the examples are selected interactively. The DDF-IS algorithm scored significantly higher than the DDF algorithm on Quality and Relevance, even significantly outperforming the expert baseline on Relevance, with the best overall scores. Recall that for the DDF-IS examples, we manually tagged the data-driven features with natural language labels, which a student could reasonably select from when making an example request. In this preliminary evaluation, we simulated this selection process by manually selecting the best *data-driven feature* to show to each student. Importantly, we only selected the feature (e.g. "move the sprite using a variable"), not the example, and the algorithm still selected the best possible example for a given feature (from hundreds of possible choices). Still, it is likely that an expert familiar with the system is more likely to choose an appropriate feature than a student, so this represents an *upper bound* for how well an interactive, data-driven example selection algorithm could reasonably perform. These results show that, when this proper selection is applied, the larger database of example pairs generated by our algorithm can lead to more Relevant examples than a static set of expert-authored examples. This represents a growing trend among data-driven systems that support programming to leverage human (i.e. student and instructor) knowledge in conjunction with data-driven algorithms [10, 18, 33]. However, from our preliminary expert evaluation, we can not make claims about how students would actually react to such a choice. Our results also suggest ways that we can improve the *automated* example selection procedure. The majority of low-quality DDF example pairs would select a feature that the student needs but could be less relevant than other features. For example, one example pair suggests adding the student's custom block into the main script so they can test it. However, in Snap*!*, students can click on the custom block to test it directly. This feature should therefore have lower priority than others. For the example shown in the Results section, the data-driven feature detector failed to detect correct variations in student code, and suggested a feature that the student had already completed. These examples suggest that it may be more important to select the most important feature to demonstrate, before considering the similarity of the example to the student's current code.

5. CONCLUSION

This study presents a method to generate data-driven examples automatically from historical student data. We evaluated two versions of data-driven generated examples using the student code when they need help, compared with two baselines: examples authored by experts and examples derived naively from student solution traces. Experts evaluated all of the example pairs based on a multidimensional rubric. Our preliminary results demonstrate that by selecting appropriate features, our data-driven examples can be more relevant than both baselines while retaining the usefulness and interpretability. These promising results suggest that our method can generate adaptive, data-driven examples automatically with quality similar to that of expert-authored examples. Even though our method is based on a block-based programming environment, it is language-agnostic and may also be generalized to textual programming languages; though further studies are needed to verify this proposed generalizability.

Limitations: This preliminary study relied on experts ratings to assess example quality, yielding valuable insight, but future work is needed to determine whether these results translate into improved student performance and learning. For example, even a well-rated example pair may still give away too much of the answer and impede learning. We also do not know how our results will generalize beyond the single assignment evaluated here, since we only applied this method to a short, block-based problem with a solution that is usually 7-10 lines of code. In our DDF-IS condition, we had an expert interactively select the feature to be presented, rather than a student. As noted earlier, this is likely an optimistic implementation, as we do not know

whether students can effectively select examples. Finally, all conditions used the SourceCheck algorithm to select similar examples to the student's code, but other approaches may better capture example relevance (e.g. [11]). We are currently planning a student evaluation of the DDF algorithm to address these limitations.

6. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under grant 1623470.

7. REFERENCES

[1] N. Augsten, M. Böhlen, and J. Gamper. The pq-gram distance between ordered labeled trees. *TODS*, 2010.

[2] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society*, 57(1):289–300, 1995.

[3] M. E. Caspersen and J. Bennedsen. Instructional design of a programming course: a learning theoretic approach. In *ICER (Workshop)*, pages 111–122, 2007.

[4] J. Coenen, S. Gross, and N. Pinkwart. Comparison of feedback strategies for supporting programming learning in integrated development environments (ides). In *ICCSAMA*, pages 72–83. Springer, 2017.

[5] O. J. Dunn. Multiple comparisons using rank sums. *Technometrics*, 6(3):241–252, 1964.

[6] B. J. Ericson, J. D. Foley, and J. Rick. Evaluating the efficiency and effectiveness of adaptive parsons problems. In *ICER*, pages 60–68. ACM, 2018.

[7] B. J. Ericson, L. E. Margulieux, and J. Rick. Solving parsons problems versus fixing and writing code. In *Koli Calling*, pages 20–29, 2017.

[8] P. Gerjets, K. Scheiter, and R. Catrambone. Designing instructional examples to reduce intrinsic cognitive load: Molar versus modular presentation of solution procedures. *Instructional Science*, 32(1-2):33–58, 2004.

[9] S. Gross, B. Mokbel, B. Hammer, and N. Pinkwart. How to select an example? a comparison of selection strategies in example-based learning. In *ITS*, pages 340–347. Springer, 2014.

[10] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *L@S*, 2017.

[11] R. Hosseini and P. Brusilovsky. A study of concept-based similarity approaches for recommending program examples. *New Review of Hypermedia and Multimedia*, 23(3):161–188, 2017.

[12] I.-H. Hsiao and P. Brusilovsky. The role of community feedback in the student example authoring process: An evaluation of annotex. *British Journal of Educational Technology*, 42(3):482–499, 2011.

[13] M. Ichinco, K. J. Harms, and C. Kelleher. Towards understanding successful novice example user in blocks-based programming. *Journal of Visual Languages and Sentient Systems*, 3:101–118, 2017.

[14] M. Ichinco and C. Kelleher. Exploring novice programmer example use. In *VL/HCC*, 2015.

[15] H. Keuning, J. Jeuring, and B. Heeren. A systematic literature review of automated feedback generation for programming exercises. *TOCE*, 19(1):3, 2018.

[16] B. B. Morrison, L. E. Margulieux, and M. Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In *ICER*, 2015.

[17] E. Patitsas, M. Craig, and S. Easterbrook. Comparing and contrasting different algorithms leads to increased student learning. In *ICER*, pages 145–152, 2013.

[18] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas. Learning program embeddings to propagate feedback on student code. In *ICML*, pages 1093–1102, 2015.

[19] T. W. Price, Y. Dong, and T. Barnes. Generating data-driven hints for open-ended programming. *EDM*, 16:191–198, 2016.

[20] T. W. Price, Y. Dong, and D. Lipovac. isnap: towards intelligent tutoring in novice programming environments. In *SIGCSE*, pages 483–488. ACM, 2017.

[21] T. W. Price, R. Zhi, and T. Barnes. Evaluation of a data-driven feedback algorithm for open-ended programming. In *EDM*, 2017.

[22] T. W. Price, R. Zhi, and T. Barnes. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *AIED*, 2017.

[23] T. W. Price, R. Zhi, Y. Dong, N. Lytle, and T. Barnes. The impact of data quantity and source on the quality of data-driven hints for programming. In *AIED*, pages 476–490. Springer, 2018.

[24] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *IJAIED*, 2017.

[25] J. Robertson and M. Kaptein. *Modern statistical methods for HCI*. Springer, 2016.

[26] D. L. Schwartz, C. C. Chase, M. A. Oppezzo, and D. B. Chin. Practicing versus inventing with contrasting cases: The effects of telling first on learning and transfer. *Journal of Educational Psychology*, 103(4):759, 2011.

[27] J. Sweller. The worked example effect and human cognition. *Learning and instruction*, 2006.

[28] J. Sweller and G. A. Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and instruction*, 1985.

[29] J. Sweller, J. J. Van Merrienboer, and F. G. Paas. Cognitive architecture and instructional design. *Educational psychology review*, 10(3), 1998.

[30] J. G. Trafton and B. J. Reiser. Studying examples and solving problems: Contributions to skill acquisition. In *CSS*, pages 1017–1022, 1993.

[31] A. Vihavainen, M. Paksula, and M. Luukkainen. Extreme apprenticeship method in teaching programming for beginners. In *SIGCSE*, 2011.

[32] L. Vygotsky. Interaction between learning and development. *Readings on the development of children*, 23(3):34–41, 1978.

[33] M. Wu, M. Mosse, N. Goodman, and C. Piech. Zero shot learning for code education: Rubric sampling with deep learning inference. *arXiv preprint arXiv:1809.01357*, 2018.

[34] R. Zhi, T. W. Price, N. Lytle, Y. Dong, and T. Barnes. Reducing the state space of programming problems through data-driven feature detection. In *EDM (Workshops)*, 2018.