information

Article

Automatically Generated Visual Profiles of Code Solutions as Feedback for Students

Jakub Swacha

Department of IT in Management, University of Szczecin, 71-004 Szczecin, Poland; jakub.swacha@usz.edu.pl

Abstract: Providing students feedback on their exercise solutions is a crucial element of computer programming education. Such feedback can be generated automatically and can take various forms. This paper introduces and proposes the use of visual profiles of code solutions as a form of automatically generated feedback to programming students. The visual profiles are based on the frequency of code elements belonging to six distinct classes. The core idea is to visually compare a profile of a student-submitted solution code to the range of profiles of accepted solutions (including both reference solutions provided by instructors and solutions submitted by students who successfully passed the same exercise earlier). The advantages of the proposed approach are demonstrated on a number of examples based on real-world data.

Keywords: interactive learning environments; computer programming education; code visualization; automatic feedback

check for updates

Citation: Swacha, J. Automatically Generated Visual Profiles of Code Solutions as Feedback for Students. *Information* **2022**, *13*, 415. https://doi.org/10.3390/info13090415

Academic Editors: Petros Lameras, Panagiotis Petridis and Sylvester Arnab

Received: 1 August 2022
Accepted: 31 August 2022
Published: 1 September 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

1. Introduction

Feedback, along assessment, is one of the "key drivers for learning" [1]; it is "generally regarded as crucial to improving knowledge and skill acquisition" and as "a significant factor in motivating learning" [2]. This paper focuses particularly on the informative tutoring feedback, defined as "multiple-try feedback strategies providing elaborated feedback components that guide the learner toward successful task completion without offering immediately the correct response" [3].

Introductory programming courses, to much extent, consist in solving numerous programming exercises of increasing difficulty and complexity. It is not often for students to solve exercises at their first attempt—in fact, sometimes tens of attempts are made before a correct solution is produced [4]. Giving relevant feedback after each such attempt is a mundane task for a teacher, especially considering there are many students solving the same exercises at the same lesson. In a large class, it becomes very difficult for a teacher to provide meaningful feedback with sufficient speed; waiting for teacher's feedback, some students can figure out the solution themselves, yet others can get frustrated and lose interest in learning programming, contributing to the anecdotal high drop-out ratios of programming courses.

Help comes in the form of software supporting programming learning, capable of producing relevant feedback to students' attempted solutions automatically. Such tools started to appear in the 1960s and became popular in the 1990s [5]. Today, there are plenty of tools in this vein available (in 2018, Keuning et al. reported 101 tools identified using only two bibliographic databases and excluding tools providing only visual, non-textual feedback [5]), differing in covered programming languages, applied feedback generation methods, and feedback scope and form.

In the times of powerful code correction and completion tools such as OpenAI Codex [6], it becomes possible to automatically generate programming exercise solutions line by line. However, the students are not given programming exercises just to obtain their

solutions but to make the students learn something while solving the exercises. Additionally, one of the formative feedback guidelines says: "avoid using progressive hints that always terminate with the correct answer" [2].

Therefore, the problem addressed in this paper is how to make the feedback less unveiling while keeping it clear and useful for students. The intention is to motivate students to think (providing them with some clues) rather than to provide them with explicit suggestions on what to correct and how to do it.

Research in psychology and communication indicates a strong advantage of visual over text-based communication [7]. This motivates seeking graphical feedback forms. As a matter of fact, algorithm and program visualization (static or dynamic) is featured in many interactive learning environments [8]. Typically, they present the code as a kind of a diagram (often a flowchart), making all its components and transitions between them more approachable for students (in the case of dynamic visualization, the current state of an executing program is also presented, including used data structures and the results of the program attained so far).

In this paper, however, a different approach for visualizing students' code is taken: instead of presenting it visually with a fine grain of detail, only selected traits of the code are visualized and compared with accepted solutions (which could be both reference solutions provided by instructors and solutions submitted by students who successfully passed the same exercise earlier), striving to actively engage students in finding out the necessary improvements or additions to their submitted code.

The visual form of feedback proposed here is not intended to substitute other forms of feedback, as it is able to indicate only a subset of possible problems with the submitted code, but to complement the other forms of feedback, providing the students with easily comprehensible information sufficient to solve many typical errors in the code, which, at the same time, is not explicit and thus requires some thinking on behalf of the student, leading to better understanding of the correct solution.

In the next section, the related work on automatically generated feedback is reported. Then, the proposed approach is described, after which its proof-of-concept implementation and its validation results are discussed. The final section concludes.

2. Prior Research on Informative Tutoring Feedback

The research on informative tutoring feedback spans a wide range of topics, including the identification of feedback types, the techniques of implementing automatic feedback generators, and the measurement of feedback effectiveness.

2.1. Feedback Types

The most extensive list of informative tutoring feedback types (or, feedback components, as they are usually combined with each other) has been developed by Narciss [3]. She distinguishes three simple feedback components: knowledge of performance for a set of tasks, knowledge of result/response, and knowledge of the correct results, as well as five elaborated ones: knowledge about task constraints, knowledge about concepts, knowledge about mistakes, knowledge about how to proceed, and knowledge about metacognition.

As regards the simple feedback components, knowledge of performance provides a summative information on the achieved performance in a finished set of tasks (e.g., "5 of 20 correct"), knowledge of the correct results describes or indicates a correct solution, and knowledge of result/response communicates whether a solution is correct or incorrect—in the context of programming exercises, the following meanings of correctness are in use: (1) it passes all tests, (2) it is equal to a model program, (3) it satisfies one or more constraints, and (4) a combination of the above [3].

Regarding the five types of elaborated feedback for programming exercises—knowledge about task constraints includes, e.g., hints or explanations on the type of task (helping to put it in the right context among other, previously solved tasks), hints or explanations on task-processing rules (providing general information on how to approach the exercise),

hints or explanations on subtasks (helping to manage the task complexity), and hints or explanations on task requirements (e.g., forcing the use of a particular language construct or forbidding the use of a particular library function). Knowledge about concepts includes, e.g., hints or explanations on technical terms, examples illustrating the concept, hints or explanations on the conceptual context, hints or explanations on concept attributes, and attribute-isolation examples. Knowledge about mistakes includes, e.g., number of mistakes, location of mistakes, hints or explanations on type of errors, and hints or explanations on sources of errors. Knowledge about how to proceed includes, e.g., bug-related hints for error correction (i.e., what the student should do to fix the issue), hints or explanations on task-specific strategies, hints or explanations on task-processing steps (i.e., the next step a student should take to get closer to the solution), guiding questions, and worked-out examples. Knowledge about metacognition includes, e.g., hints or explanations on metacognitive strategies and metacognitive guiding questions [3].

Among the 101 tools analyzed by Keuning et al., the most popular elaborated feedback type was knowledge about mistakes, implemented in 96% of the tools [5]. Knowledge about how to proceed came second (featured in 45% tools) and knowledge about concepts and knowledge about task constraints, respectively, third (17% tools) and fourth (15% tools). Only one of the analyzed tools implemented knowledge about metacognition [5].

There are also other feedback classifications in use. Le, who considered only adaptive feedback (i.e., one which provides different information for different answers), identified five types of feedback that were supported by systems supporting programming education: Yes/No feedback (e.g., "Passed" or "Failed"), syntax feedback (similar to compiler messages, yet more comprehensible for students), semantic feedback (which can be divided into intention-based, striving to determine the intention of the student, and code-based feedback, which ignores the intention of the student), layout feedback (on whether the submitted code follows a specific coding convention), and quality feedback (on whether the student's solution is efficient in terms of time and memory resources) [9]. Among the 20 tools he analyzed, the most (15) implemented code-based semantic feedback; 6 of them have also implemented intention-based semantic feedback. Syntax feedback has been featured in four tools, Yes/No feedback in three tools, layout feedback in two tools, and quality feedback in just one tool [9].

2.2. Techniques for Automatic Feedback Generation

There are various techniques used to implement automatic feedback generators. According to the results of Keuning et al., the most widespread is automatic testing (59% of the analyzed tools), followed by program transformations (38%), basic static analysis (37%), intention-based diagnosis (21%), external tools (12%), model tracing (10%), data analysis (9%), and constraint-based modeling (4%) [5].

Automatic testing consists in running the student-submitted code and comparing its output to the output of the correct solution. As running students' code is dangerous, safety mechanisms have to be implemented, such as jailed sandboxes, which restrict the tested process to a specified subset of system resources, virtual machines or containers [10]. Another option made possible by the vast progress in the capabilities of web browser script engines is using them as execution environments. While students' code written in JavaScript can be run natively, this approach can also be used with other programming languages, provided its interpreter or compiler has been written in or transpiled to JavaScript (see, e.g., [11]).

The aim of program transformations is to help match students' submissions with model solutions. The difficulty of this matching is due to the fact that a specific algorithm can be correctly implemented using many programs, differing in their used statements, statement order, logical structures, or even variable names. The work-around is to transform all semantically equivalent forms into a canonical one. For instance, Xu and Chee identify 13 types of semantics-preserving variations and propose a rule-based transformation of each program to the form of an Augmented Object-oriented Program Dependence Graph,

which eliminates many of these variations while supporting matching programs with each other [12].

Basic static analysis is performed on the code without running it and can detect code which is syntactically correct but shows misunderstood concepts (e.g., in JavaScript, using IsLooselyEqual operator where IsStrictlyEqual should have been used), or check for code elements which are correct in general terms but not allowed in the context of a certain exercise or execution environment (e.g., if an exercise asks students to implement from scratch a function which is already available in the standard library), or for code elements which are required in any correct solution of an exercise (e.g., if an exercise asks students to solve a problem by implementing a recursive algorithm) [13]. The basic static analysis is implemented using rules whose definition is very time-consuming, but there are dedicated domain-specific languages making it less tedious (see, e.g., [14]).

Intention-based diagnosis tries to infer what the program was intended to do, how it was intended to work, and identify errors in these intentions or their realization [15]. Well-known tools featuring this technique are PROUST and CHIRON [16].

External tools are usually static analysis tools developed rather for software development purposes than programming education, which are yet capable of producing feedback which could be valuable for students. The exemplary external tools in use are: Check-Style (generating feedback on preserving coding convention), FindBugs (reporting suspicious code that may result in bugs), and PMD (reporting bad coding practices) [5].

Model tracing tools strive to trace and analyze the process that the student is following solving a problem by applying rules which classify student steps as correct or wrong and suggest the right direction whenever the student wanders off the path. The classic example of such tools is the LISP tutor [17].

The common trait of data analysis tools is generating hints using data acquired from students themselves. These tools may have very diverse characters, as there are various aspects of solution development which could be addressed with this approach. While Keuning et al. described such tools as "using large sets of historical data" [5], this is not truly necessary, as data analysis can be performed on single student submissions as well: for instance, various software metrics could be calculated for it and presented to students [18], giving the additional benefit of students learning how to interpret software metrics. Similarly, tools based on sample program sets provided by teachers do not need any historical data; a good example of such tool is described by Coenen et al., which uses machine learning techniques to match the student code to the most similar sample solution, striving to facilitate the learner in making further progress [19]. Moreover, ITAP (Intelligent Teaching Assistant for Programming), which uses state abstraction, path construction, and state reification to automatically generate personalized hints for students, even for given states that have not occurred in the data before, indicates teachers as the source of two required pieces of input information: at least one reference solution to the problem and a test method that can automatically score code [20].

Using students' historical data has, however, an obvious advantage of avoiding the preparation of sample programs by teachers. Lazar and Bratko proposed to exploit the history of students' attempts, containing several incorrect submissions and at least one correct submission, to identify sequences of line edits that fix a buggy program, which can be used as a basis for providing hints [21]. SourceCheck uses a set of correct student solutions of an assignment to annotate the student's code with suggested edits, including code that should be moved or reordered [22]. Zhi et al. propose a method for automatically selecting most appropriate program examples using the automatic extraction of program features from historical student data [23].

Constraint-based modeling is based on the assumption that "correct solutions are similar to each other in that they satisfy all the general principles of the domain" [24]. The students are thus allowed to perform whatever actions they please which do not lead to a state that is known to be wrong. As constraints define equivalence classes of problem states, and an equivalence class triggers the same instructional action, it is therefore

possible to attach feedback messages directly to constraints [24]. The best-known example of this approach is J-LATTE (Java Language Acquisition Tile Tutoring Environment), which supports two modes: concept mode, in which the student designs the program without having to specify contents of statements, and coding mode, in which the student completes the code [25].

2.3. Automatic Feedback Effectiveness

Keuning et al. report that over 71% of the works they analyzed included an empirical assessment of some kind [5]. In 38% of cases, it was based on technical analysis (e.g., measuring the number of correctly recognized solutions, the time needed for the analysis, and a comparison to the analysis of a human tutor, e.g., [12]), in 33%, student and/or teacher survey results were provided, and in 30%, the learning outcome was evaluated (usually by comparing the outcomes of a control group that used an alternative learning strategy and the group that used the proposed solution, e.g., [26]) [5]. Strangely, Keuning et al. have not aggregated the assessment results; such data can yet be found in Le's review, where "pedagogical evaluation studies" have been reported for 16 out of 20 analyzed tools (80%), and all of them were positive [9]. Similar results can be found in another review, by Cavalcanti et al., which, however, has a somewhat different scope—as it covers all online learning environments providing automatic feedback, not only supporting programming education (although those supporting programming comprise the largest, a 30% share of the software covered in the review) [27]. Among the 63 works analyzed therein, 41 (65%), including 32 (51%) supported by empirical evaluation, provided results related to student performance—and all of them were positive. This should not be interpreted, though, that automatic feedback always improves learning performance, as we are aware of works from outside the sets analyzed by Le or Cavalcanti et al. that report negative evaluation results (e.g., [19]). There are also other negative consequences linked to some types of feedback, e.g., Kyrilov and Noelle reported reduced student engagement and increased cheating after providing instant Yes/No feedback on programming exercises [28].

3. Visual Profiles of Programming Exercise Solutions

3.1. Concept of Visual Code Profiles

The core idea of visual profiles of programming exercise solutions can be described as striving to generate an image depicting core traits of a program, so that similar images are generated for similar programs, and different images are generated for different programs. Such visual profiles can be, consequently, used to generate feedback to programming students by visually comparing the submitted solution with the correct solutions (the plural is intended, as the latter may differ in form).

The problem of generating such visual profiles is related to the problem of finding a canonical representation of a program, which is central to the technique of program transformations (and is relevant to some of the other techniques as well, as, e.g., using such program representation instead of source code as input may greatly improve the efficiency of data analysis). There are, however, important differences. Obviously, a canonic representation of a program does not have to be visualized, and usually is not, though there are solutions that exploit such an opportunity (e.g., [29]). The key difference, actually, is that the canonical representation of a program, to be useful, has to convey all semantically relevant traits of a program, particularly its logic, whereas its visual profile may convey any subset of its traits, as there is no intention to reconstruct the logic of the program based on its visual profile. In fact, in the context of feedback generation, the lack of possibility of easily converting a visual profile into a working program is considered as its advantage, as it motivates the students to figure out their own solution based on the provided hints rather than to copy one from a correct example.

3.2. Graphic Form of Visual Code Profiles

The following design requirements relevant to generating visual profiles of programming exercise solutions have been defined:

1. It must be possible to automatically construct a visual profile from any code which conforms to the syntax of the taught programming language.
2. The visual profile must have a graphic form which is both capable of conveying all intended information and be readable for students.
3. It must be possible to render at least two visual profiles in the same drawing space for the ease of comparing them by students.
4. It must be possible to generate visual profiles of sets of programming exercises, i.e., not only of individual programming exercises.

Typically, programs are visualized using diagrams capable of displaying the flow of the algorithm [8,29]. While such a form is essential in explaining the program logic, for all but trivial examples, it results in images far too complex to make fast visual comparisons. For this reason, it was decided to use a different form as a base for visual profiles of programming exercise solutions, one that would be much easier to interpret and compare. Several alternative solutions were considered for this purpose. Although the visual attractiveness of word clouds was appreciated, along with their capability of conveying various types of information at the same time (the rendered text, its size, color, and orientation and position in both axes), they were found to be hard to interpret in detail and very difficult to compare. Japanese Candlestick charts, conversely, could be easily adapted to compare the submitted solution traits with the correct solutions, but they are not easy to interpret (except for students having long experience with stock market data analysis). Histograms are readable and can be easily adapted for presenting comparisons in an effective way, yet their proportions change with the number of considered variables, and when it is high, either more horizontal space is needed, or their readability decreases.

Consequently, a radar chart has been chosen as a form of visual profiles, as it:

- Can show multiple traits at the same time in a compact space without sacrificing readability.
- It has fixed proportions that do not change with the number of presented variables (as is the case with, e.g., bar charts).
- It allows for easy comparison of profiles of various exercises and exercise sets by rendering them in the same chart yet using different colors.

3.3. Set of Traits Considered in Visual Code Profiles

In order to define a set of traits to be considered in visual code profiles, an archive of over 9000 student-submitted solutions from an introductory Python course (whose contents are described in detail in [4]) was analyzed in a search for most obvious differences between accepted and failed submissions. It resulted in the list of types of frequent students' errors, consisting of either using a wrong program element (that no accepted solution contained) or not using a needed program element (that most accepted solutions contained). The list included (the order is meaningless):

1. Not using an instruction essential for the solution (e.g., *break* in a program requiring complex loop control).
2. Not using an operator essential for the solution (e.g., ** in a program requiring power calculation).
3. Not using a built-in function essential for the solution (e.g., the *int* function in a program requiring conversion of strings to integers).
4. Not using a method of a built-in type essential for the solution (e.g., the *sort* method of lists in a program requiring ordering of data).
5. Not importing a module providing functions essential for the solution (e.g., *random* in a program requiring randomization).
6. Producing output not meeting the exact requirements of an exercise (e.g., printing "Hello" instead of "Hello!").

Based on that, the following list of program traits has been defined:

1. Instructions.
2. Operators.
3. Built-ins.
4. References to components of built-in classes.
5. Imported modules.
6. String and numeric constants.

3.4. Generating Visual Code Profiles

While it is possible to render multiple traits in one chart, it would hurt readability strongly. Therefore, each trait is rendered in its own dedicated chart. Depending on the available drawing space, the charts of all traits can be rendered one next to another, or one chart is provided at one time, and the student is able to switch between the rendered traits.

The profiled code is parsed in order to identify the tokens relevant to any of the traits, which are then retrieved and counted. For each trait, its statistical profile can be represented as a list of pairs (*token*, *number of occurrences*) containing all tokens relevant to that trait that were found in the profiled code.

For each respective trait and each exercise, the statistical profiles of all student solutions of that exercise which were accepted as correct are aggregated into a list of triples (*token*, *minimum number of occurrences*, *maximum number of occurrences*) containing all tokens relevant to that trait that were found in at least one accepted solution of that exercise.

The visual profile of the code is rendered as a radar chart using three parameters:

- l denoting the maximum total number of tokens shown in one chart; it should be set in accordance with the space available for drawing the profile to ensure that the chart is not overloaded with data, which would make it unreadable.
- m denoting the minimum number of variables shown in the chart; by default, it should be set to 3 to ensure a two-dimensional form of the drawn radar chart contents (with less than 3 variables, the resulting radar chart would be difficult to interpret visually); if the number of tokens is fewer than m, the data for existing tokens are duplicated to generate m variables.
- n denoting the minimum number of tokens of each category shown in one chart; this is explained below.

Regarding the n parameter, the considered categories are:

(I) The tokens which are the most characteristic for the profiled code but were not found in accepted solutions of the programming exercise that the profiled code attempts to solve.
(II) The tokens which are most characteristic for the accepted solutions of the programming exercise that the profiled code attempts to solve and were also found in the profiled code.
(III) The tokens which are the most characteristic for the accepted solutions of the programming exercise that the profiled code attempts to solve but were not found in the profiled code.

This parameter matters only if l is smaller than the number of identified tokens. As long as this remains true, iteratively, the least characteristic token is discarded from the list of category I tokens (considered as the least important, as they do not indicate what the proper solution should have) if their count is larger than n; if that is not true, the least characteristic token is discarded from the list of category II tokens also if their count is larger than n; if that is not true, the least characteristic token is discarded from the list of category III tokens (considered as the most important, as they indicate what the proper solution should have but the profiled code has not). Obviously, the value of n has to meet the requirement: $n \leq l/3$.

The tokens most characteristic of the profiled code are the tokens most frequently occurring in that code. The tokens most characteristic of the programming exercise are the

tokens having the highest value of the tuple: (*minimum number of occurrences in any of the accepted solutions*, *maximum number of occurrences in any of the accepted solutions*).

By including the tokens most characteristic of the programming exercise in the visual profile, the user may easily notice elements which are typical for accepted solutions but missing in the profiled code. By including the traits most characteristic of the profiled code in the visual profile, the user may easily notice elements which are untypical for accepted solutions but present in the profiled code.

While the values of parameters l, m, and n are implementation-specific, in interactive learning environments, they could be allowed to be adjusted by the user at runtime.

For each token, its level in the profiled code is shown in color A as a point denoting the number of its occurrences in the profiled code, connected with a line with the respective points of the tokens neighboring it in the chart, and its level in the accepted programming exercise solutions is shown as a pair of points in colors B and B', respectively, denoting the minimum and maximum number of its occurrences in the accepted solutions' code, each connected with a line with the respective minimum and maximum level points of the tokens neighboring it in the chart, with the area between the lines connecting minimum- and maximum-level points filled with color B'', whereas the area between the lines connecting minimum-level points and the center of the chart is filled with color C.

Regarding interpretation, if the point or line in color A is within the lines in color B or B' (or within the shape in color B''), the element of the profiled code it represents was also similarly used in the accepted solutions (so, it is probably correct); if the points or lines in color A are closer to the chart center than the line in color B', the element it represents was used more times in any of the accepted solutions than in the profiled code (so, it should probably be used more in it); otherwise, if the points or lines in color A are further from the chart center than the line in color B, the element of the profiled code it represents was used in the accepted solutions, yet to a lesser degree (it could be considered wrong or unnecessary if the level of the token represented by the point in color B is 0).

The colors A, B, B', B'', and C are implementation-specific, with the requirements that colors A and B should be contrasting, and colors B, B', and B'' should be similar (it is recommended to use two pale shades of color B as colors B' and B''). In interactive learning environments, they could be allowed to be adjusted by the user at runtime.

The visual profiles can only be drawn for traits present in either the profiled or the reference code, e.g., an attempt to draw a chart of imported modules for a program which does not import any module and no accepted solution of the same exercise does it would result in an empty chart.

4. Proof-of-Concept Implementation and Tests

4.1. Proof-of-Concept Implementation

In order to validate the feasibility of the proposed concept and its usability on real-world data, its proof-of-concept implementation has been developed in Python 3.8.2. The main reason for choosing this particular language was the ease of implementation due to the availability of the *ast* standard module providing a source code parser along with the full grammar of the language.

The detailed list of the tokens (under names defined in the Python grammar used by *ast*) assigned to respective program traits is provided in Appendix A. For Instructions, Operators, Built-ins, and Built-in classes' components, it is the number of occurrences of the respective tokens themselves that is counted as a parameter of a given trait (e.g., programs can be compared by the number of *for* statements they contain), whereas for Modules and Constants, it is the number of occurrences of respective token values that is counted (e.g., programs can be compared by the names of modules they import).

For real-world applications in programming learning environments, producing charts of high visual attractiveness may be important to attain a positive user experience. However, in the proof-of-concept implementation, which is not subject to the user experience evaluation, visual attractiveness of the generated charts is irrelevant, for which reason, the

default form of radar charts as implemented in the *matplotlib.pyplot* Python module was chosen for their drawing.

The parameters l, m, and n were set, respectively, to 18, 3, and 6. Red was chosen as color A and three shades of blue were used as colors B, B', and B'', whereas light gray was chosen as color C.

4.2. Test Dataset and Procedure

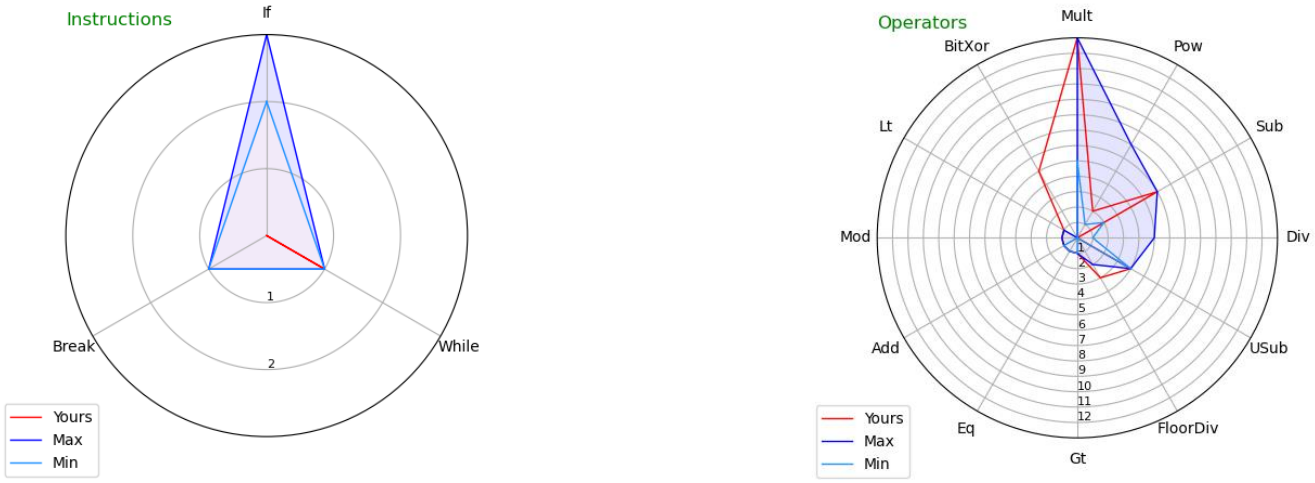
As the test dataset, the already-mentioned archive of over 9000 student-submitted solutions from an introductory Python course was used. The archive comprises solutions of 94 exercises arranged in 12 thematic lessons (details can be found in [4]) provided by 39 students. For each solution, information is included whether it was accepted (i.e., passed all tests) or not (i.e., failed to pass at least one of the tests). The archive was exported from the University of Szczecin's instance of the FGPE PLE platform [30] in the form of a CSV file.

First, for each exercise, all its accepted solutions were parsed to retrieve all occurrences of relevant tokens, which were then used to establish the minimum and maximum number of occurrences of each token of each trait.

Then, the failed submissions were screened to find examples of evident differences between the submitted solution code and the accepted solutions code pertaining to each of the considered traits. Each such example supports the rationale for including the relevant trait in the set of considered traits. Then, the visual profiles of the exemplary submissions were drawn and then analyzed to verify whether the rendered chart contains sufficient information presented in a form noticeable enough for the user to spot the difference. Each such positively verified example demonstrates a use case in which the visual profile proved useful.

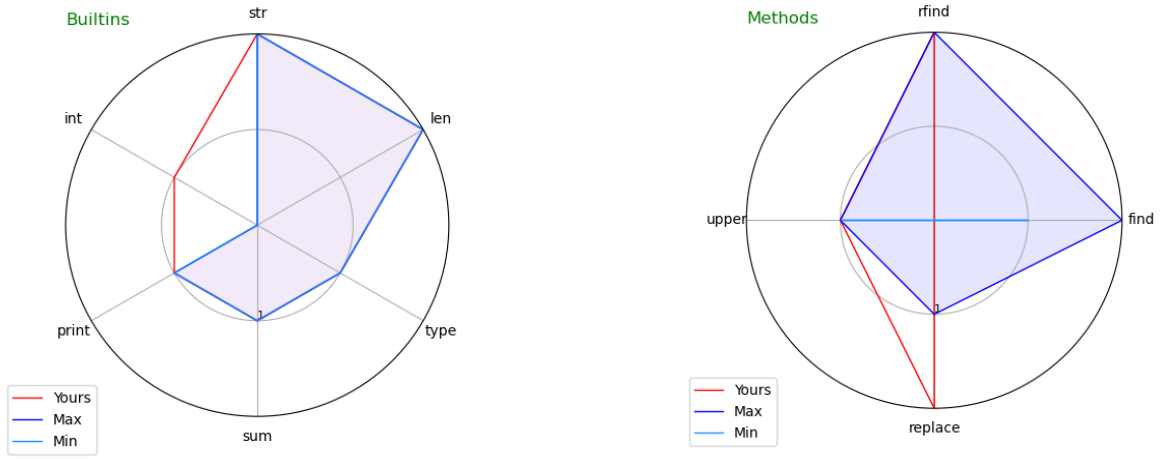
4.3. Test Results

Figures 1–3 present visual profiles of exemplary students' programs showing respective traits. Note that each figure features a different student-submitted exercise solution.



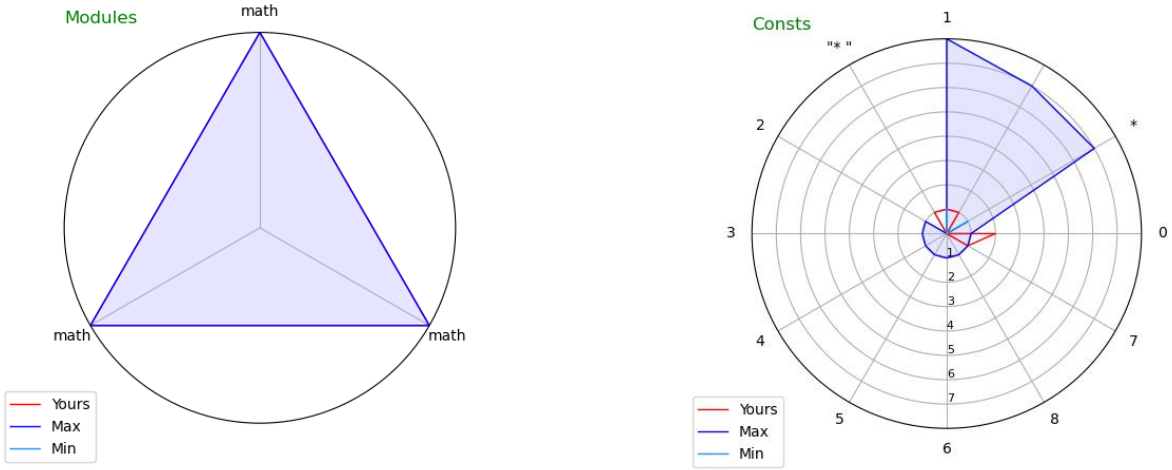
(**a**) (**b**)

Figure 1. Visual profiles of exemplary students' programs showing: (**a**) Instructions and (**b**) Operators.



(**a**) (**b**)

Figure 2. Visual profiles of exemplary students' programs showing: (**a**) Built-ins and (**b**) Methods.



(**a**) (**b**)

Figure 3. Visual profiles of exemplary students' programs showing: (**a**) Modules and (**b**) Constants.

Figure 1a covers Instructions. The exercise the student attempted required complicated while loop control. As we can see, all accepted solutions used *if* and *break* instructions (as indicated by the pale blue line), whereas the student's submission does not contain them (no red line is shown for these elements, indicating level 0). The visual profile clearly indicates the required instructions that the student's submission is lacking.

Figure 1b covers Operators. The exercise the student attempted required calculating a power, denoted as ** in Python (here, represented by the *Pow* token). The student made the mistake of using another operator, ^ (here, represented by the *BitXor* token), which represents *bitwise exclusive or* in Python. As we can see, all accepted solutions used *Pow* and did not use *BitXor*, which the students' submission contains. Interestingly, the student used *Pow* in one instance, implying either he/she actually knew this operator (bringing the question of why it was not used in the other instances) or copied the code fragment containing it from another student without understanding its meaning. Note that token class names are used for displaying operators in the chart in the proof-of-concept implementation for the sake of its simplicity, whereas for real-world implementations, using the actual form of operators is suggested, as it would be more readable for students.

Figure 2a covers Built-ins. The exercise the student attempted required some calculations; as indicated by the red line outside of the blue space, the student made the mistake

of converting intermediate results at one point to the integer type (using the *int* built-in function), which none of the accepted solutions used. The visual profile clearly indicates the unnecessary built-in function that the student used which led to the error in the result.

Figure 2b covers References to components of built-in classes—in particular, it indicates the methods of built-in classes used in the code. The exercise the student attempted to solve required obtaining a part of a string delimited by specific signs. As indicated by the blue shape, all accepted solutions used the *find* method of the *str* class, whereas the student's submission (as indicated by the red vertical line) does not contain it. Note that it is possible to solve the exercise without using the *find* method, although it is much more complicated, which explains why no student managed to accomplish it this way. The visual profile thus clearly indicates the method that should have been used.

Figure 3a covers Imported modules. The exercise the student attempted required a certain mathematical function. As we can see, all accepted solutions imported the *math* module containing that function, whereas the students' submission does not contain it. The visual profile thus clearly indicates the module that the student's submission is lacking. This very simple example was included also to illustrate the role of parameter *m* (set to its default value of 3): there was only one module imported in all accepted solutions (*math*), but it was repeated three times in the chart to meet the requested minimum number of variables shown in each chart. We would see only a single straight blue line if the limit was not enforced, which would hamper the readability of the chart.

Figure 3b covers String and numeric constants. The exercise the student attempted required printing eight asterisks in a loop. The student used the *for* loop over a *range* (0,7), which actually results in 7 repetitions, not 8. Note that the constant 7 can also be found in some of the accepted solution (which used, e.g., the *while* $n \leq 7$ loop). There is yet another visible difference between the students' submission and the accepted ones: the former has "*" as a constant, the latter just "*". Superfluous spaces are often a reason for which the students' submissions are not accepted, as their output is verified to strictly match the expected results, so this indicates a valid use case for visual profiling of Constants, although in the particular case of the profiled submission, the student made no mistake, as he/she used the *end* = "*" parameter of the *print* function disabling the automatic generation of whitespaces between displayed values (the other students used "*" and *end* = "*" to achieve the same effect). This example was included also to illustrate the weak sides of the visual profiles (as the first mistake was not clearly indicated, and the cue for the second mistake was not leading to the actual reason of not accepting the submission). These are discussed in the following section.

5. Discussion and Future Work

The presented examples show that for all six considered traits, the visual profiles may bring valuable cues by visually indicating key differences between the submitted code and the already-accepted solutions of that exercise in terms of used Instructions, Operators, Built-ins, Built-in class components, Modules, and Constants. Figures 1a, 2b and 3a clearly indicate elements missing in the submitted code; Figures 1b, 2a and 3bclearly indicate elements which should not have been used as they are wrong or, at least, an unnecessary complication of the solution.

Of course, we selected the examples that contained such differences; however, while for the bulk of submissions, the differences will not be visible for all considered traits, for most wrong submissions, there is at least one trait where the difference becomes apparent.

There are three main limitations of the presented visual code profiles. The first is when the diversity of accepted solutions results in hiding the difference, as exemplified in the case of constant 7 in Figure 3b. This is a problem specific to blank sheet exercises, for which the student is allowed to use any known approach; it is much less characteristic for exercises whose author provides a solution skeleton that the students are supposed to extend, thus limiting the diversity of solutions. A possible workaround would be to consider token pairs in addition to single tokens when drawing the visual profile, which in the discussed case

could indicate the difference between the possibly good pair (*while*, 7) and the most probably wrong pair (*for*, 7). The problem is how to select the pairs most adequate for visualization, considering that for more complicated programs there could be hundreds of them, and there is a limit on the number of variables a radar chart can render while staying readable. Finding a solution to this problem is the first direction of our future work.

The second limitation is when there is actually no difference between two programs with regard to any trait, as the difference lies in the program structure and/or the flow of statements. The visual code profiles are not an adequate tool to handle the differences of this kind. While there is no technical problem in identifying and counting the occurrences of specific fragments of program structure, presenting them in readable form would be far from easy, as for all but the most trivial exercises, there will be many alternative statement combinations found in their solutions. As the errors in program logic are already well-addressed by existing solutions (see Section 2.2), we do not consider them as a topic which has to be addressed by visual profiles. The two types of feedback (visual profiles and program logic analysis) can be easily combined in a user interface of a programming learning environment.

The third limitation is typical for all feedback techniques based on the prior student submissions. Visual profiles use all submissions accepted so far as their reference, therefore indicating notable differences for unorthodox solutions, which may be unrelated to the actual error, which caused the student's submission to not be accepted—as was the case of the space–asterisk string in Figure 3b. Suggesting the student to check a possibly (but not truly) problem-causing program element is, however, not all wrong, as it may increase his/her awareness of the role of that element (and in the described exemplary case, following the suggestion would lead to a simpler code). This limitation has also another aspect—that there is no reference available before the first submission is accepted; it, however, can be addressed by the instructor solving all exercises before the students are allowed to do that.

We do not consider the scope of the considered traits to be an actual limitation. As explained in Section 3.3, the selection of traits was based on the observation of simple errors students often make. There is no technical barrier in extending the list of the considered traits to include other classes of tokens, e.g., expression types or named function parameters. Experimenting with other traits indicates the second direction of our future work.

In spite of the limitations described above, the proposed visual profiles have a large potential for implementation in interactive programming learning environments, primarily to provide non-obvious feedback to students learning with them. There is yet another possible point of implementation in interactive programming learning environments: in their teacher console, where the visual profiles could support teachers in diagnosing and possibly correcting students' submissions.

The main threat to validity is the form of testing which did not involve students. Therefore, the third and most important direction of our future work will be to implement the visual profiles in an interactive programming learning environment by embedding them in the feedback section of its user interface, so that their real-world effectiveness could then be measured in such dimensions as:

- Inclination of students to have a look at provided visual profiles.
- Ability of students to grasp cues from the visual profiles.
- Share of submissions improved thanks to cues from the visual profiles.
- Effect of using the visual profiles on the students' progress in the course.

6. Conclusions

The paper introduced visual profiles of programs as a form of automatically generated feedback to programming students, alternatively to text-based feedback. A visual profile depicts the frequency of code elements belonging to one of six distinct classes in both student-submitted solution code and the already accepted solutions, allowing a fast visual comparison of the two. The visual profiles were shown to provide useful cues for

improvement for a number of exemplary programming exercise solutions selected from a real-world data set.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data are available from the author on request.

Conflicts of Interest: The author declares no conflict of interest.

Appendix A

The list of Python tokens considered in program trait visualization:

1. Instructions
   `Assert, AsyncFor, AsyncFunctionDef, AsyncWith, Await, Break, ClassDef, Continue, Delete, ExceptHandler, For, FunctionDef, Global, If, IfExp, Import, ImportFrom, Lambda, Nonlocal, Pass, Raise, Return, Try, While, With, Yield, YieldFrom`
2. Operators
   `Add, And, BitAnd, BitOr, BitXor, Div, Eq, FloorDiv, Gt, GtE, In, Invert, Is, IsNot, LShift, Lt, LtE, MatMult, Mod, Mult, Not, NotEq, NotIn, Or, Pow, RShift, Sub, UAdd, Usub`
3. Built-ins
   `_, __build_class__, __debug__, __doc__, __import__, __loader__, __name__, __package__, __spec__, abs, all, any, ArithmeticError, ascii, AssertionError, AttributeError, BaseException, bin, BlockingIOError, bool, breakpoint, BrokenPipeError, BufferError, bytearray, bytes, BytesWarning, callable, ChildProcessError, chr, classmethod, compile, complex, ConnectionAbortedError, ConnectionError, ConnectionRefusedError, ConnectionResetError, copyright, credits, delattr, DeprecationWarning, dict, dir, divmod, Ellipsis, enumerate, EnvironmentError, EOFError, eval, Exception, exec, exit, False, FileExistsError, FileNotFoundError, filter, float, FloatingPointError, format, frozenset, FutureWarning, GeneratorExit, getattr, globals, hasattr, hash, help, hex, id, ImportError, ImportWarning, IndentationError, IndexError, input, int, InterruptedError, IOError, IsADirectoryError, isinstance, issubclass, iter, KeyboardInterrupt, KeyError, len, license, list, locals, LookupError, map, max, MemoryError, memoryview, min, ModuleNotFoundError, NameError, next, None, NotADirectoryError, NotImplemented, NotImplementedError, object, oct, open, ord, OSError, OverflowError, PendingDeprecationWarning, PermissionError, pow, print, ProcessLookupError, property, quit, range, RecursionError, ReferenceError, repr, ResourceWarning, reversed, round, RuntimeError, RuntimeWarning, set, setattr, slice, sorted, staticmethod, StopAsyncIteration, StopIteration, str, sum, super, SyntaxError, SyntaxWarning, SystemError, SystemExit, TabError, TimeoutError, True, tuple, type, TypeError, UnboundLocalError, UnicodeDecodeError, UnicodeEncodeError, UnicodeError, UnicodeTranslateError, UnicodeWarning, UserWarning, ValueError, vars, Warning, WindowsError, ZeroDivisionError, zip`
4. Built-in classes' components
   `__add__, __and__, __class__, __contains__, __del__, __delattr__, __delitem__, __dict__, __dir__, __doc__, __enter__, __eq__, __exit__, __format__, __ge__, __getattribute__, __getitem__, __getnewargs__, __gt__, __hash__, __iadd__, __iand__, __imul__, __init__, __init_subclass__, __ior__, __isub__, __iter__, __ixor__, __le__, __len__, __lt__, __mod__,`

`__mul__, __ne__, __new__, __next__, __or__, __rand__, __reduce__, __reduce_ex__, __repr__, __reversed__, __rmod__, __rmul__, __ror__, __rsub__, __rxor__, __setattr__, __setitem__, __sizeof__, __str__, __sub__, __subclasshook__, __xor__, _checkClosed, _checkReadable, _checkSeekable, _checkWritable, _CHUNK_SIZE, _finalizing, add, append, buffer, capitalize, casefold, center, clear, close, closed, copy, count, detach, difference, difference_update, discard, encode, encoding, endswith, errors, expandtabs, extend, fileno, find, flush, format, format_map, fromkeys, get, index, insert, intersection, intersection_update, isalnum, isalpha, isascii, isatty, isdecimal, isdigit, isdisjoint, isidentifier, islower, isnumeric, isprintable, isspace, issubset, issuperset, istitle, isupper, items, join, keys, line_buffering, ljust, lower, lstrip, maketrans, mode, name, newlines, partition, pop, popitem, read, readable, readline, readlines, reconfigure, remove, replace, reverse, rfind, rindex, rjust, rpartition, rsplit, rstrip, seek, seekable, setdefault, sort, split, splitlines, startswith, strip, swapcase, symmetric_difference, symmetric_difference_update, tell, title, translate, truncate, union, update, upper, values, writable, write, write_through, writelines, zfill`

5. Modules
   `Import, ImportFrom`
6. Constants
   `bool, bytes, complex, constant, float, int, NoneType, string`

References

1. Lameras, P.; Arnab, S. Power to the Teachers: An Exploratory Review on Artificial Intelligence in Education. *Information* **2021**, *13*, 14. [CrossRef]
2. Shute, V.J. Focus on Formative Feedback. *Rev. Educ. Res.* **2008**, *78*, 153–189. [CrossRef]
3. Narciss, S. Feedback Strategies for Interactive Learning Tasks. In *Handbook of Research on Educational Communications and Technology*; Routledge: London, UK, 2008; pp. 125–143.
4. Szydłowska, J.; Miernik, F.; Ignasiak, M.S.; Swacha, J. Python Programming Topics That Pose a Challenge for Students. In Proceedings of the Third International Computer Programming Education Conference (ICPEC 2022), Barcelos, Portugal, 2–3 June 2022; Simões, A., Silva, J.C., Eds.; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Dagstuhl, Germany, 2022; Volume 102, pp. 7:1–7:9.
5. Keuning, H.; Jeuring, J.; Heeren, B. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* **2019**, *19*, 1–43. [CrossRef]
6. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H.P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374. [CrossRef]
7. Dansereau, D.F.; Simpson, D.D. A Picture Is Worth a Thousand Words: The Case for Graphic Representations. *Prof. Psychol. Res. Pract.* **2009**, *40*, 104. [CrossRef]
8. Sorva, J.; Karavirta, V.; Malmi, L. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Trans. Comput. Educ.* **2013**, *13*, 1–64. [CrossRef]
9. Le, N.-T. A Classification of Adaptive Feedback in Educational Systems for Programming. *Systems* **2016**, *4*, 22. [CrossRef]
10. Peveler, M.; Maicus, E.; Cutler, B. Comparing Jailed Sandboxes vs Containers within an Autograding System. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA, 22 February–2 March 2019; ACM: New York, NY, USA; pp. 139–145.
11. Edwards, S.H.; Tilden, D.S.; Allevato, A. Pythy: Improving the Introductory Python Programming Experience. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education, Atlanta, GA, USA, 5 March 2014; ACM: New York, NY, USA; pp. 641–646.
12. Xu, S.; Chee, Y.S. Transformation-Based Diagnosis of Student Programs for Programming Tutoring Systems. *IEEE Trans. Softw. Eng.* **2003**, *29*, 360–384. [CrossRef]
13. Striewe, M.; Goedicke, M. A Review of Static Analysis Approaches for Programming Exercises. In Proceedings of the International Computer Assisted Assessment Conference, Zeist, The Netherlands, 30 June–1 July 2014; Springer: Berlin/Heidelberg, Germany; pp. 100–113.

14. Swacha, J. Exercise Solution Check Specification Language for Interactive Programming Learning Environments. In Proceedings of the 6th Symposium on Languages, Applications and Technologies (SLATE 2017), Vila do Conde, Portugal, 26–27 June 2017; Queirós, R., Pinto, M., Simões, A., Leal, J.P., Varanda, M.J., Eds.; Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2017; Volume 56, pp. 6:1–6:8.
15. Johnson, W.L. *Intention-Based Diagnosis of Novice Programing Errors*; Research Notes in Artificial Intelligence; Morgan Kaufmann: London, UK, 1986; ISBN 978-0-934613-19-4.
16. Sack, W.; Soloway, E. *From PROUST to CHIRON: ITS Design as Iterative Engineering*; Intermediate Results Are Important! In Computer-Assisted Instruction and Intelligent Tutoring Systems; Routledge: London, UK, 1992; pp. 239–274.
17. Anderson, J.R.; Skwarecki, E. The Automated Tutoring of Introductory Computer Programming. *Commun. ACM* **1986**, *29*, 842–849. [CrossRef]
18. Koyya, P.; Lee, Y.; Yang, J. Feedback for Programming Assignments Using Software-Metrics and Reference Code. *ISRN Softw. Eng.* **2013**, *2013*, 805963. [CrossRef]
19. Coenen, J.; Gross, S.; Pinkwart, N. Comparison of Feedback Strategies for Supporting Programming Learning in Integrated Development Environments (IDEs). In Proceedings of the Advanced Computational Methods for Knowledge Engineering, International Conference on Computer Science, Applied Mathematics and Applications, Berlin, Germany, 30 June–1 July 2017; Le, N.-T., van Do, T., Nguyen, N.T., Thi, H.A.L., Eds.; Springer International Publishing: Cham, Switzerland, 2018; pp. 72–83.
20. Rivers, K.; Koedinger, K.R. Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor. *Int. J. Artif. Intell. Educ.* **2017**, *27*, 37–64. [CrossRef]
21. Lazar, T.; Bratko, I. Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In Proceedings of the Intelligent Tutoring Systems, Honolulu, HI, USA, 5–9 June 2014; Trausan-Matu, S., Boyer, K.E., Crosby, M., Panourgia, K., Eds.; Springer International Publishing: Cham, Switzerland, 2014; pp. 306–311.
22. Price, T.W.; Zhi, R.; Barnes, T. Evaluation of a Data-Driven Feedback Algorithm for Open-Ended Programming. In Proceedings of the 10th International Conference on Educational Data Mining, Wuhan, China, 25–28 June 2017; International Educational Data Mining Society: Montréal, QC, Canada.
23. Zhi, R.; Marwan, S.; Dong, Y.; Lytle, N.; Price, T.W.; Barnes, T. Toward Data-Driven Example Feedback for Novice Programming. In Proceedings of the 12th International Conference on Educational Data Mining (EDM 2019), Montréal, QC, Canada, 2–5 July 2019; International Educational Data Mining Society: Montréal, QC, Canada, 2019; pp. 218–227.
24. Mitrovic, A.; Koedinger, K.R.; Martin, B. A Comparative Analysis of Cognitive Tutoring and Constraint-Based Modeling. In Proceedings of the User Modeling 2003, Johnstown, PA, USA, 22–26 June 2003; Brusilovsky, P., Corbett, A., de Rosis, F., Eds.; Springer: Berlin, Heidelberg, 2003; pp. 313–322.
25. Holland, J.; Mitrovic, A.; Martin, B. J-Latte: A Constraint-Based Tutor for Java. In Proceedings of the 17th International on Conference Computers in Education (ICCE 2009), Hong Kong, China, 30 November–4 December 2009; pp. 142–146.
26. Lane, H.C.; VanLehn, K. Teaching the Tacit Knowledge of Programming to Novices with Natural Language Tutoring. *Comput. Sci. Educ.* **2005**, *15*, 183–201. [CrossRef]
27. Cavalcanti, A.P.; Barbosa, A.; Carvalho, R.; Freitas, F.; Tsai, Y.-S.; Gašević, D.; Mello, R.F. Automatic Feedback in Online Learning Environments: A Systematic Literature Review. *Comput. Educ. Artif. Intell.* **2021**, *2*, 100027. [CrossRef]
28. Kyrilov, A.; Noelle, D.C. Binary Instant Feedback on Programming Exercises Can Reduce Student Engagement and Promote Cheating. In Proceedings of the 15th Koli Calling Conference on Computing Education Research, Koli, Finland, 19 November 2015; ACM: New York, NY, USA; pp. 122–126.
29. Jiang, L.; Rewcastle, R.; Denny, P.; Tempero, E. CompareCFG: Providing Visual Feedback on Code Quality Using Control Flow Graphs. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, Trondheim, Norway, 15 June 2020; ACM: New York, NY, USA; pp. 493–499.
30. Paiva, J.C.; Queirós, R.; Leal, J.P.; Swacha, J.; Miernik, F. An Open-Source Gamified Programming Learning Environment. In Proceedings of the Second International Computer Programming Education Conference (ICPEC 2021), Braga, Portugal, 27–28 May 2021; Henriques, P.R., Portela, F., Queirós, R., Simões, A., Eds.; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Dagstuhl, Germany, 2021; Volume 91, pp. 5:1–5:8. [CrossRef]