Extracting Unit Tests from Patterns Mined in Student Code to Provide Improved Feedback in Autograders

Julien Lienard[1], Kim Mens[1] and Siegfried Nijssen[1]

[1]ICTEAM, UCLouvain, Belgium

Abstract

CS1 courses with large student numbers commonly use autograders to provide students automated feedback on basic programming exercises. Programming such feedback to integrate it into an autograder is a non-trivial and time-consuming task for teachers. Furthermore, such feedback is often based only on expected outputs for a given input, or on the teacher's perception of errors that students may make, rather than on the errors they actually make. We present an early implementation of a tool prototype and supporting methodology to address these problems. After mining the source code of earlier students' responses to exercises for frequent structural patterns, and classifying the found patterns according to these students' scores, our tool automatically generates unit tests that correspond to bad practices, errors or code smells observed in students' submissions. These unit tests can then be used or adapted by a teacher to integrate them into an autograder, in order to provide feedback of higher quality to future generations of students.

Keywords

CS education, pattern mining, unit testing, code generation, autograders, automated feedback

1. Introduction

Automated graders [1] are often used in the context of introductory programming courses to assist students by providing automated feedback on their programming exercises. This feedback should be of high quality and as detailed as possible, so that it can help the students to learn from and correct their errors autonomously.

Motivation

Creating such automated feedback to be integrated in an autograder is a non-trivial and time-consuming task for teachers. As a consequence, often they limit the feedback to what can be checked by comparing the output of students' programs on given inputs to the expected outputs. In this work, we argue that more qualitative feedback could be derived from structural regularities discovered in the source code of students' submissions.

Part of the challenge is to decide what regularities are most relevant for this purpose, and secondly how to transform these regularities into tests that could be run on students' submitted source code. Our proposal is to generate such tests after using a combination of a code mining algorithm and a manual analysis of the outcome of this code mining process by teachers. This data mining algorithm looks for frequent code patterns that are more representative for bad solutions than good solutions.

Reviewing earlier submissions can reveal a lot about how students approach problems and come up with answers. Teachers may be able to provide better feedback to students if they are aware of these patterns in typical good and bad solutions. Students who struggle with a particular exercise may benefit from seeing positive instances of 'good' solutions as well as from identifying 'poor' patterns in their code in order to understand why it is incorrect.

Research Questions

Our long-term goal is to aid teachers of introductory programming courses to include relevant feedback in autograders, in order for students to better understand and correct their mistakes. The main research questions of this paper are:

1. Can we discover recurrent error patterns in students' code and use them to create more advanced feedback?
2. Can we automatically generate test suites from such identified error patterns, so that instances of these errors can be detected and reported automatically to students?

To answer the first research question, we use a data mining algorithm called FREQTALS for finding frequent tree patterns in students' source code [2]. This technique was tested on a dataset of student code submissions and was shown previously to be able to find interesting trends representative of good and bad coding idioms. [3]

To tackle the second research question, we develop an automated tool to generate unit tests for the discovered patterns in previous student submissions. This tool

15th Seminar on Advanced Techniques & Tools for Software Evolution — SATToSE 2023

julien.lienard@uclouvain.be (J. Lienard); kim.mens@uclouvain.be (K. Mens); siegfried.nijssen@uclouvain.be (S. Nijssen)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3483/paper5.pdf

can help teachers incorporate these unit tests into an autograder, to provide more comprehensive feedback to students. We believe that our tool prototype has the potential to enhance instruction effectiveness and improve students' learning outcomes.

The focus of the current paper is mainly on the process of extracting unit tests from selected patterns. The mining algorithm itself was already reported upon before, and extending the unit tests with detailed feedback that helps students overcome their errors is mostly a manual effort still.

Validation

As initial ongoing validation of our tool prototype we compiled a sizable dataset of Python code taken from an instance of the INGInious [1] platform, an autograder that enables students to submit and receive feedback on programming exercises. The dataset corresponded to a final exam for an entry-level programming course with 569 students. We applied the FREQTALS code mining algorithm to this dataset to identify common patterns, and then generated unit tests to check which of these patterns are present in the source code of the student submissions. A teacher selected and augmented a few of these patterns to give more advanced feedback to students, based on the discovered structure in their submitted code. The generated unit tests for these patterns together with their additional feedback were then added to the autograder, to be used by students to prepare for future exam sessions. A more thorough analysis of how students use and appreciate this feedback will be the focus of a follow-up paper.

2. Related Work

The development of automated methods to generate feedback on programming exercises is an important field when it comes to helping students who learn how to program.

Importance of feedback

A systematic literature review on automated feedback for programming exercises was conducted by Keuning et al. [4]. Their survey analysed and categorised 69 papers according to 4 factors:

1. **Kind** of feedback;
2. **Technique** used for the feedback;
3. **Adaptability** by the teachers;
4. **Quality** of the feedback.

According to this survey, the majority of current methods to generate automatic feedback for programming exercises look for coding mistakes by using static analysis methods such as parsing and type checking. Depending on the programming language used and the kind of feedback given, various methods can be helpful in different situations. The need to manage multiple programming languages, the complexity of natural language creation, and the issue of assessing the quality of the generated feedback were some of the challenges and restrictions the authors mentioned.

Our proposed technique tries to improve the kind of feedback given to students on programming exercises by helping teachers identify recurrent errors and to automatically generate unit tests that can detect such errors in students' code. We do not yet automatically produce a description of the feedback in natural language for those unit tests.

Solution errors occur in programs when they do not perform as expected. These errors can manifest themselves in the form of logic errors, when a program does not correctly accomplish a desired task. Our technique aims to transform detected *knowledge about mistakes* into automated feedback capturing *solution errors* to be included into an autograder.

Narciss [5] argues that feedback is very important when it comes to online learning. Having a feedback loop helps students to really understand and complete their exercises successfully. She also showed that offering answer-until-correct (AUC) exercises with feedback is generally more effective than feedback on *'one-shot'* exercises. AUC feedback alone may not be sufficient for some learning activities however, according to her study, which suggests that it should be coupled with rich elaborated feedback. AUC feedback is advantageous because it provides students with numerous chances to apply what they have learned and fix their mistakes, improving their memory of the material. In this work, we try to make feedback more rich by making it more personalised, by comparing the student's submission to recurring errors that have been detected in submissions by earlier generations of students.

Feedback generation

The *Concepts and Skills based Feedback Generation Framework (CSF2)* was proposed by Haldeman et al. [6] for designing programming assignments, analyzing student submissions, and generating hints for assisting students in correcting their errors. In this method, the set of concepts and skills, called knowledge map, that students are required to master to complete the assignment is taken into consideration while creating the task, creating the test suite, gathering and organizing submissions, and refining the test suite and knowledge map. Errors are then mapped to concepts and skills into what they call a *bucket*. New tests are then created by hand to assess the common errors found in each *bucket*. The described framework

presents a useful structure for improving classroom instruction and identifying students' errors and misconceptions. The framework is described as a sequence of steps. Our proposed approach is complementary in the sense that it could automate some of these steps, such as the creation of test suites.

One of the state of the art tools to create personalized feedback is AutoGrader [7]. Teachers first provide a reference solution and a list of possible mistakes before students can use AutoGrader. This information is then used by AutoGrader to generate potential code modifications that students could apply to reach the right answer. The authors start from the hypothesis that student mistakes can be predicted by teachers. In our approach, we do not make that assumption and instead attempt to identify frequent mistakes made by students by studying assignments submitted by prior student generations.

Another interesting automated tool, closer to what we propose, is CARLA [8]. It uses existing *correct* student solutions in order to try and repair the code of students who made mistakes. In order to do so, it first groups accurate student answers using the concept of dynamic equivalence. The clustering, which serves as basis for the repair process, groups together similar correct solutions. The repair algorithm then uses expressions from several correct answers that are located in a same cluster to build minimum corrections for an incorrect student attempt. The strategy reduces the adjustments required to make a student reach a proper solution by using the most comparable accurate solutions, written by other students. Our technique instead uses all students' code (not only correct solutions) and, rather than using dynamic equivalence, uses frequent tree pattern mining to group them. From those groups we create tests that correspond to the patterns found and let teachers pick the most interesting ones and add meaningful feedback to those tests. Our tool thus doesn't focus on code repair but rather on the automatic creation of unit tests.

Our work is comparable to a tool called Codewebs proposed by Nguyen et al. [9]. Their strategy uses the solutions of prior submissions to create customized feedback for large MOOCs. They created a queryable index for quick searches into a sizable dataset of student assignment entries by breaking down online homework submissions into a vocabulary of "code phrases". They argue that by force-multiplying teacher effort, tools like Codewebs can help improve the quality of free education. They use shared structure among submissions to create specific feedback on student errors. Both our work and Codewebs use the data from student submissions from a MOOC, but our approach uses a different method to identify shared structure in student submissions.

Other sources of inspiration

As stated in the introduction, our paper can be regarded as an extension the experiment described by Mens et al. [3]. They used pattern mining on students' code submissions to detect recurring errors made by students participating in a first-year programming course. Our main contribution is the addition of test generation for matching the discovered patterns, in order to apply it to new students' code submissions. The goal of this extension is to, after having discovered frequently occurring mistakes using the technique described by Mens et al., generate tests that can make students aware of such mistakes and as such help them correct and avoid them in the future. We created a first tool prototype, used it to create a test task for students using this generation technique, and are currently analysing initial results to see if the feedback received helps students correct their mistakes.

3. Methodology

Figure 1 summarises the different steps of our approach and tools to help teachers integrate more advanced feedback for students in an autograder, based on patterns observed in previous students' submissions. The first step consist of gathering the source code of these prior submissions (cf. 3.1). The second step extracts abstract syntax trees from this source code to be given as input to the tree mining algorithm, which then finds frequently occurring patterns in those ASTs (cf. 3.2). From the list of discovered patterns, in step 3 teachers select the most interesting ones that they would like to convert into python unit tests (cf. 3.3). They can then adapt and integrate these generated unit tests into the autograder to provide additional feedback to the students based on the structure of their code to help them understand and correct their mistakes (cf. 3.4).

3.1. Data gathering

To gather a sufficiently large data sample, we had access to the source code submitted by students on small programming exercises and exam questions in the context of an entry-level programming course at university. They submitted their code on and received feedback from the INGInious platform [1], an open source and online teaching assistant used by several universities around the world. For the experiment described in this paper, we collected the code submitted by students in response to an exam question, which asked students to write a program that computes the prime factors of a given number.

The following signature and specification of the function they had to write was given to the students:

[1] https://inginious.org/

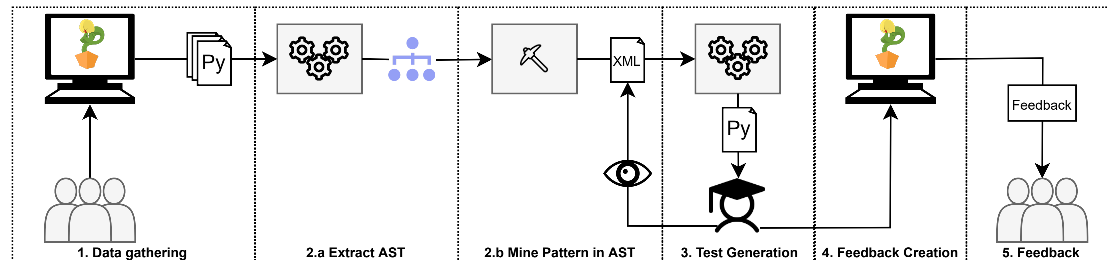

Figure 1: The 5 steps of our technique to provide improved feedback to students based on unit tests generated from patterns discovered in submissions made by previous generations of students.

```
def factors(n):
  """
  Pre:  n a strictly positive integer
  Post: returns integer representing
        number of prime factors of n
  """
  # code to complete
```

together with an example of what results the function should return for some given cases:

```
factors(33)  # returns 2 (33 = 3x11)
factors(12)  # returns 3 (12 = 2x2x3)
factors(8)   # returns 3 (8 = 2x2x2)
factors(127) # returns 1 (127 prime)
```

The exam context guarantees that students responded to the questions individually and under similar conditions, on university computers in a classroom setting. Furthermore, whereas students received automated feedback throughout the semester and when preparing for the exam, during the exam itself the autograder's feedback option was turned off. This implies that the patterns which we discovered after analysing the student's responses to the exam questions were not biased by any feedback they received on those questions.

As in the mining experiment described by Mens et al. [3], we separated the dataset in 2 groups: students who obtained a score of 50% or more on the question and those who obtained a score of less than 50%. This score was calculated automatically by running a set of unit tests that checked different input-output pairs that a correct solution should respect. Taking these two subsets as input, the mining algorithm would try and find patterns that are more representative for one set than for the other.

3.2. Pattern mining

FREQTALS [2] is a tree mining algorithm designed to identify frequent patterns in the abstract syntax trees (ASTs) of programs. Each pattern is a tree structure that represents a part of an AST. FREQTALS is based on the FREQT algorithm [10] and identifies frequent patterns in a dataset by iteratively generating candidate sets of patterns and pruning those that do not meet a minimum support threshold. FREQTALS requires the ASTs it takes as input to be in a particular XML tree format. Our tool makes sure this format is followed, by using a Python parsing library [2] to convert the Python programs written by students into the format required by the miner. Code that could not be parsed was not considered for the test generation.

FREQTALS can be configured with various constraints, such as the minimum number of files in which the pattern can be found, the minimum or maximum size of the patterns, and the list of allowed root nodes (i.e. what types of syntactic constructs, such as Python function bodies, the miner should focus on). All these constraints were configured as described in the original paper.

As the ASTs of the students' code are tree-like representations, FREQTALS searches for patterns in tree structures. It takes into account the structural characteristics of the patterns, such as the placement of the child nodes and the frequency with which particular labels appear in the tree. The pattern miner discovers different types of patterns, including control flow patterns, data structure patterns, and algorithm pattern types. Common programming constructs like loops, conditionals and function calls can be included in these patterns, as well as more specialized language constructs like list comprehension or generator expressions.

We used the FREQTALS tree miner to find common patterns in the ASTs of the programs submitted by both the group of students who obtained 50% or more and by the group who scored less than 50%, in order to assess the efficacy of our tool. Whereas we kept the same configuration settings as in the original paper, we did experiment with different values to set the threshold for a pattern to be considered as 'frequent'. We experimentally deter-

[2] https://docs.python.org/3/library/ast.html#ast.parse

Listing 1: Example of an augmented unit test written using our technique.

```
1 class TestQ(unittest.TestCase):
2   def __init__(self,*args,**kwargs):
3     super().__init__(*args, **kwargs)
4     with open(str(q.__name__)+'.py','r') as f:
5       self.ast = ast.parse(f.read())
6   def test_hard_coded_list(self):
7     if match_pattern_2(self.ast):
8       self.fail("Feedback: Avoid using hard coded lists of precalculated numbers.
           Try to generalise your solution.")
```

mined that four occurrences seemed to strike a decent compromise between too broad and too narrow patterns.

3.3. Unit test generation

The pattern miner's output is an XML file containing the tree structure of all discovered patterns. We transform this output into a set of Python unit tests. We want each resulting unit test to check whether a code submission matches a pattern found by the mining algorithm. To do so, for a given pattern, our unit test generation tool traverses its structure. For each node of the pattern's tree, the tool generates a Python code block that searches for corresponding AST nodes in the code to be checked. If the pattern node has children, the tool searches the children of the found AST node for matches in its children. This search is repeated until all nodes in the pattern's XML structure have been checked, but if no match is discovered, the tool backtracks to the next matching AST node. A fragment of such a generated unit test can be found in Listing 4 in Subsection 4.2.

3.4. Feedback creation

Our longer-term research goal is to assist teachers in providing students with more insightful feedback on their exercises. The generated tests are to be used to evaluate students' code automatically and to be augmented with more specific feedback on the errors they make.

After having run the pattern miner on previous students' submissions to existing assignments, teachers can select any pattern that catches their attention, and generate a Python unit test for it. They can then add additional feedback to that unit test and include it into the autograder for that assignment, or they can use it to design a new assignment around that pattern.

The generated test will fail and reveal problems in students' code if it does not adhere to the pattern. Students can then rely on this augmented feedback provided by the teacher in order to help them fix their mistakes (see Listing 1). Compared to just receiving a mark or general feedback on the overall quality of their code, this may be more useful.

The augmented unit test of Listing 1 would match a pattern (`match_pattern_2`, shown in Listing 4) that verifies whether a student uses a hardcoded a list of numbers in their solution (typically a list of prime numbers to be used for the factorisation) and suggest the student to avoid using such a hardcoded list in order to obtain a more generic solution.

4. Initial Experiment

In this section we will describe the results of an initial experiment we conducted to assess the pattern mining tool and a first prototype implementation of our unit test generation tool.

4.1. Patterns

For the exam question mentioned in subsection 3.1, we gathered the code submitted by a total of 569 students. Of these, 560 were parsed successfully and included in our analysis. Most of the code that didn't parse was unfinished code or code containing syntax errors. The code was then split into two groups: 149 submissions for which the students obtained a score of less than 50%, and 411 submissions with a score of 50% or higher. (These scores were calculated automatically with an autograder, after completion of the exam, using a set of graded unit tests that were programmed by the teacher beforehand.)

Next, playing the role of a teacher, we analysed all 147 patterns mined by the FREQTALS algorithm to identify which of them could be interesting or potentially useful to transform into unit tests. This analysis included a review of the number of times each pattern appeared in the code of the high-scoring and low-scoring groups, as well as a manual review of the patterns themselves to identify if they captured a common mistake made by several students. Other criteria we used in our selection were whether the pattern's absence or presence affected the code's quality, the total number of occurrences of the

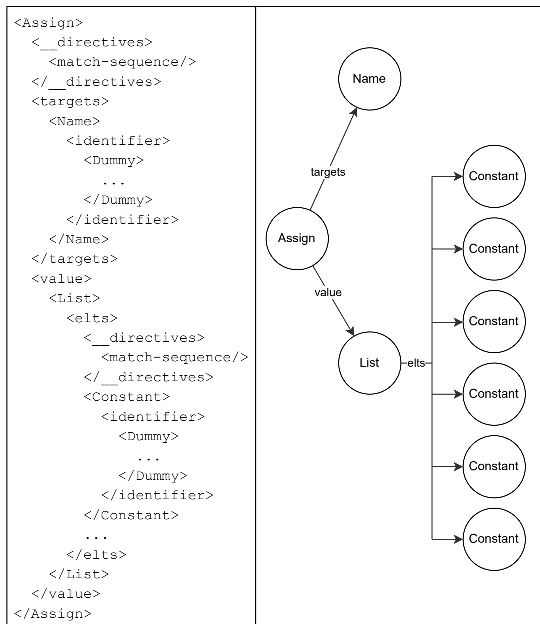

Figure 2: Example of a fragment of the XML and AST for the pattern 2

pattern and the difference between matching code in the high-scoring and low-scoring groups.

We discovered several patterns that could reveal potential bad practices or errors in students' code. An example of such a bad practice is illustrated in Listing 2. This code corresponds to a match of the second pattern that was found. The corresponding XML pattern and AST representation are shown in Figure 2.

The part of the code that matches the pattern is highlighted in red in Listing 2. The pattern matches code fragments where students hard coded the list of prime factors to check. We found 6 students' code that matched this pattern. A high-level feedback that the teacher may want to add for this pattern is that computing those numbers is probably a more generic solution than hard-coding them in a list.

Listing 3 is an occurrence of another pattern showing a student's code that is close to a good solution. Unfortunately the student should have used a while-statement instead of an if-statement at line 9. This pattern was found in the code of 8 students. A teacher may want to provide as feedback to these students that they may have confused an 'if' for a 'while'.

In addition to analysing the discovered patterns themselves, the mere fact of having to walk through code fragments that match certain patterns, sometimes leads to the discovery of other recurrent errors that were not

Listing 2: Example of a student's bad practice: hard coding the list of prime numbers to check.

```
def factors(n):                              1
    l = [2,3,5,7,11,13,17,19]  #list of primes   2
    i = 0                                    3
    t=[]                                     4
    for i in range(len(l)) :                 5
        if n%l[i] == 0:                      6
            t.append(l[i])                   7
            i -= 1                           8
    nbr = len(t)                             9
    return (nbr)                             10
```

Listing 3: Example of a student's code close to the correct solution, but confounding 'if' and 'while'.

```
def factors(n):                              1
    count=0                                  2
    if n==1:                                 3
        return 0                             4
     if n<0:                                 5
        return False                         6
    s=n                                      7
    for i in range (2,n):                    8
        if n%i==0:                           9
            s=s/i                            10
            print(s)                         11
        else:                                12
            count+=1                         13
    return count                             14
```

found by the miner. One such error is the overuse of nested loops. Even if we found some cases where students achieved to create correct code using more than 2 nested loops, we did observe that such code becomes quite hard to read and is to be avoided.

It should also be said that not all patterns that were mined were of interest. Some of them do capture recurring Python code fragments but do not necessarily provide any information on the student's code quality. For example, pattern number 60 described a code fragment containing an `if` with a `return` inside followed by a `return` at the end of the function. That by itself doesn't say anything about whether the code is good or bad. It is the teacher's job to filter, from all patterns discovered by the miner, the most relevant ones worth checking.

Listing 4: Example of test block generated for the pattern 2 that will search a function definition inside all blocks in the AST

```
1  def match_pattern_2(myast):
2      blocks1 = [val for val in ast.walk(myast) if hasattr(val, 'body')]
3      if len(blocks1) < 1:
4          return False
5      for block1 in blocks1:
6          functiondefs1 = [val for val in ast.iter_child_nodes(block1) if isinstance(
               val, ast.FunctionDef)]
7          if len(functiondefs1) < 1:
8              continue
9          for functiondef1 in functiondefs1:
10             # more test blocks ...
```

4.2. Test generation

The FREQTALS algorithm identified a total of 147 frequent patterns in the code submitted by both groups of students. The patterns can occur in one or both the high-scoring and low-scoring groups. When a same pattern occurs in the high-scoring group or in both groups at the same time, it often means that the pattern is either a part of or close to a good solution (such as the pattern depicted in Listing 3) or an uninteresting pattern. When a pattern can only be found in the low-scoring group, it is often a pattern representing an error or bad practice (such as the pattern depicted in Listing 2) but sometimes also a pattern of no interest. For each of the discovered patterns, our tool could produce a unit test.

It should be noted that out of the 147 frequent code patterns found, about two thirds were patterns too generic to be useful (for example a pattern that just matches the presence of two assignment statements in the code). Of the remaining patterns, about half were representative of code fragments occurring in good solutions and the other half were patterns occurring mostly in the bad solutions. Some patterns often tend to be quite close to other patterns as well, so in the end the teacher kept only a handful of relevant patterns to be transformed into unit tests.

Apart from this problem of having to wade manually through the many patterns found to retain only a few relevant ones, we also faced another issue which was to ensure that the Python code generated for unit tests remained understandable to teachers. This is desirable as it allows teachers to understand the pattern just by looking at the Python code. It also allows them to adapt the tests afterwards if need be. This is not easily achieved, however, since we want our generated unit tests to match exactly the same code fragments as the patterns found by the miner and since this matching is non-trivial, amongst others due to the need to for backtracking.

To keep the tests understandable, we construct them using test blocks. An example of two combined test blocks is shown in Listing 4. In the first block (lines 2-5), we search for all instances of a 'body' node in the code. If no 'body' nodes are found, the pattern cannot be matched (lines 3-4). Then, we iterate through each identified 'body' node (line 5). The second block is visible from lines 6 to 9.

The first block employs the `ast.walk()`[3] function to search for the root node of the pattern anywhere in the code, while the second block uses `ast.iter_child_nodes()`[4] to search only within the direct child nodes of the currently matched node. A `return` statement only appears in the first block, while the `continue` statement on line 8 allows us to fall back to the next matching node. This same structure is followed by all of the other test blocks as well.

We believe that, to some extent, this block structure allows us to reach an acceptable trade-off between readability and correctness of the test. We do think that, once a teacher understands the block structure of the generated unit tests, the tests remain sufficiently understandable. Adapting the tests slightly to capture slight variations of the pattern is possible, although not trivial.

4.3. Feedback creation

For our dataset consisting of student submissions for an exam question, we created 4 unit tests generated from or inspired by the mined patterns. We offered this question, with the additional unit tests adorned with the dedicated feedback we added, to new students as a revision exercise to help them prepare for an upcoming exam session. Of the 4 unit tests that we added, 3 of them were generated directly from the patterns found by FREQTALS, including the two patterns shown in red in Listings 2 and 3. The third pattern that we used was one which represents the usage of `return` inside of loops. The last one did not correspond to a mined pattern but rather to a recurrent

[3] https://docs.python.org/3/library/ast.html#ast.walk
[4] https://docs.python.org/3/library/ast.html#ast.iter_child_nodes

error that we discovered manually in the students' code while analysing the dataset, as explained in Section 4.1.

At the time of writing, about 66 students have given this revision exercise at least one try. Of those, 23 students received at least one of the 4 dedicated feedbacks that we added to this exercise. We analysed the submission of those 23 students and saw that the given feedback was useful for at least 6 of them. Indeed, we could observe that, after they took the feedback into account, the students obtained a better score for the question and didn't match the pattern anymore after they corrected their code.

5. Conclusion

In this paper we explored whether we could identify recurrent patterns highlighting errors or bad practices in students' code and use them to create more advanced feedback to be included in an autograder. We did so by first mining the solutions of a large amount of students for frequently occurring patterns using the FREQTALS tree mining algorithm. We then manually analysed each of these patterns to find those that match typical mistakes that many students seem to make in their solution. We also found that for some of these patterns, creating advanced feedback when they are detected could help student produce code of better quality.

Our second step was then, for the selected patterns, to automatically extract unit tests that match exactly the same source code as the mined patterns. These unit tests, which would thus check for typical coding errors made by students, could then be extended by a teacher with a more personalized feedback on that particular kind of error. These unit tests can then be included in an autograder so that students making these mistakes get more accurate feedback on the errors they make.

As both the FREQTALS miner and the test generation tool in essence only require an AST to mine for patterns and generate the tests, these tools can easily be applied to other programming languages than Python as well.

In this work in progress paper, we mainly presented our vision and the current implementation of our tool prototype. Although we have started exploring the usage of this tool on real data, for now the purpose of the validation was more to ensure that the tool has potential and is working correctly. Obviously a more thorough and in-depth validation on more data and with real students and teachers is still needed. On the one hand we want to study and understand how teachers use the tool and how the tool can be improved further to satisfy their needs. On the other hand we want to validate that the students effectively benefit from the improved feedback provided by the generated test suites (i.e., better scores and better code quality).

References

[1] G. Derval, A. Gego, P. Reinbold, B. Frantzen, P. Van Roy, Automatic grading of programming exercises in a MOOC using the INGInious platform, European Stakeholder Summit on experiences and best practices in and around MOOCs (EMOOCS'15) (2015) 86–91.

[2] H. S. Pham, S. Nijssen, K. Mens, D. D. Nucci, T. Molderez, C. D. Roover, J. Fabry, V. Zaytsev, Mining patterns in source code using tree mining algorithms, in: International Conference on Discovery Science, Springer, 2019, pp. 471–480.

[3] K. Mens, S. Nijssen, H.-S. Pham, The good, the bad, and the ugly: mining for patterns in student source code, in: Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence, 2021, pp. 1–8.

[4] H. Keuning, J. Jeuring, B. Heeren, Towards a systematic review of automated feedback generation for programming exercises, Association for Computing Machinery, New York, NY, USA, 2016.

[5] S. Narciss, Feedback Strategies for Interactive Learning Tasks, 2008, pp. 125–144.

[6] G. Haldeman, A. Tjang, M. Babeş-Vroman, S. Bartos, J. Shah, D. Yucht, T. D. Nguyen, Providing meaningful feedback for autograding of programming assignments, Association for Computing Machinery, New York, NY, USA, 2018.

[7] R. Singh, S. Gulwani, A. Solar-Lezama, Automated feedback generation for introductory programming assignments, in: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, 2013, pp. 15–26.

[8] S. Gulwani, I. Radiček, F. Zuleger, Automated clustering and program repair for introductory programming assignments, ACM SIGPLAN Notices 53 (2018) 465–480.

[9] A. Nguyen, C. Piech, J. Huang, L. Guibas, Codewebs: Scalable homework search for massive open online programming courses, in: Proceedings of the 23rd International Conference on World Wide Web, ACM, 2014, pp. 491–502. doi:10.1145/2566486.2568023.

[10] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, S. Arikawa, Efficient substructure discovery from large semi-structured data, IEICE TRANSACTIONS on Information and Systems 87 (2004) 2754–2763.