The Good, the Bad, and the Ugly: Mining for Patterns in Student Source Code

Kim Mens
Siegfried Nijssen
kim.mens@uclouvain.be
siegfried.nijssen@uclouvain.be
ICTEAM institute, UCLouvain
Louvain-la-Neuve, Belgium

Hoang-Son Pham
hoangson.pham@uhasselt.be
Data science institute, Hasselt University
Hasselt, Belgium

ABSTRACT

Research on source code mining has been explored to discover interesting structural regularities, API usage patterns, refactoring opportunities, bugs, crosscutting concerns, code clones and systematic changes. In this paper we present a pattern mining algorithm that uses frequent tree mining to mine for interesting good, bad or ugly coding idioms made by undergraduate students taking an introductory programming course. We do so by looking for patterns that distinguish positive examples, corresponding to the more correct answers to a question, from negative examples, corresponding to solutions that failed the question. We report promising initial results of this algorithm applied to the source code of over 500 students. Even though more work is needed to fine-tune and validate the algorithm further, we hope that it can lead to interesting insights that can eventually be integrated into an intelligent recommendation system to help students learn from their errors.

CCS CONCEPTS

• **Software and its engineering → General programming languages**; • **Computing methodologies → Machine learning**; • **Applied computing → Education**; • **Social and professional topics → CS1**.

KEYWORDS

Pattern mining, source code mining, CS education, programming

ACM Reference Format:
Kim Mens, Siegfried Nijssen, and Hoang-Son Pham. 2021. 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 (EASEAI '21), August 23, 2021, Athens, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3472673.3473958

EASEAI '21, August 23, 2021, Athens, Greece

ACM ISBN 978-1-4503-8624-1/21/08…$15.00
https://doi.org/10.1145/3472673.3473958

1 INTRODUCTION

University-level computer science courses, and undergraduate introductory programming courses in particular, have been exploring the use of a vast amount of different kinds of tools, such as simplified programming environments [7], visual programming languages [4], program visualization tools [5], programming assistants [13], automatic grading tools [3] and plagiarism detection tools [8, 12]. As software engineering research starts to embrace and make use of data science, machine learning, and artificial intelligence techniques[1], the same is true for tools such as the above. This paper focuses in particular on how pattern mining could be used to enhance such tools.

An obvious use case of mining for patterns in source code would be to enhance existing plagiarism or clone detection tools [2]. Another, which is the focus of this particular paper, is to discover what distinguishes good programming solutions from bad ones. We hope that the initial results presented in this paper eventually could be used to enhance existing programming assistants or automated graders to provide automated feedback on why a particular solution was wrong, as well as recommendations on particular misconceptions students may have and how these can be overcome. This places the paper at the intersection of the three domains of interest of this workshop: (computer science) education, advanced software engineering and artificial intelligence.

To mine for interesting good or bad coding idioms from a large set of source code fragments provided by students, we build on an algorithm that we proposed recently and that mines for frequent subtrees from a given dataset of programs represented as abstract syntax trees [11]. We adapt the algorithm to work with two datasets so that it can distinguish patterns that are more representative for one dataset than the other. We then apply it to the source code produced by students at the exam of an introductory programming course. Per question we create one dataset consisting of those solutions for which the students received a grade of 50% or more and another for which the students failed the question. The novelty of this approach with respect to the original algorithm is that rather than applying it to the codebase of a single large software system, we apply it to many small code fragments classified as either positive or negative, to see if good or bad coding idioms representative for either of those classes emerge.

The remainder of this paper is structured as follows. Section 2 presents the pattern mining algorithm and how we adapted it to find patterns that distinguish positive from negative examples.

[1] Exemplified by the Mining Software Repositories conference series: www.msrconf.org

Section 3 presents the evaluation of our adapted algorithm on the source code produced by over 500 undergraduate students participating in a programming exam. Section 4 concludes the paper and sketches avenues for future work since it is obvious that, even though the preliminary results of the mining approach presented in this paper are promising, further fine-tuning of the algorithm and its configuration are needed as well as a more elaborate validation and exploration of the results.

2 THE PATTERN MINING ALGORITHM

In this section we provide the details of the FREQTALS algorithm for mining patterns in source code [11], which itself is based on the FREQT tree mining algorithm [1], as well as how we adapted it to find patterns that discriminate positive from negative examples. Advantages of the FREQTALS algorithm include that, as a tree mining algorithm, it takes into account the structure of the programs, while as compared to other generic tree mining algorithms [6] it has been optimised already for mining patterns in source code. For conciseness, the explanation of the adapted algorithm will be intermixed with that of the original one on which it is based. In Subsection 2.1 we first specify formally the problem to be solved by the algorithm, before sketching its implementation in Subsection 2.2 and how we can group discovered patterns through clustering in Subsection 2.3. Section 3 will then illustrate the algorithm at work on a concrete use case.

2.1 Specification

Abstract syntax trees. Since FREQTALS and FREQT mine for frequent subtrees from a set of trees, we need to represent the source code to mine in the form of *abstract syntax trees* (ASTs). An example of such an AST for a small piece of Python code is given in Figure 1. An advantage of providing the input as ASTs is that, apart from the configuration of a few constraints, the mining algorithm itself is largely agnostic of the particular programming language in which the source code was written. We can regard these source code ASTs as a dataset of *labeled ordered trees* on which we can apply a data mining algorithm for mining *induced labeled ordered subtrees under constraints*.

Mining induced subtrees. An ordered tree $T = (V, E, \lambda, \Sigma)$ is defined by a set of consecutive integer node identifiers $V = \{1, 2, \ldots, n\}$, where the integers indicate the order, a set of edges $E \subseteq V \times V$, and a label function $\lambda : V \mapsto \Sigma$ that associates labels to nodes of V; Σ is the set of allowed labels. In our example AST, for instance, we see that $\lambda(1) = \textit{Function}$ and $\lambda(2) = \textit{name}$.

In the ordered trees that represent the ASTs, we are looking for patterns that themselves are also ordered trees. We use the concept of *induced subtrees* for this. Informally, one tree is an induced subtree of another if the smaller tree can be obtained from the larger one by repeatedly removing leafs, or the root if the root has only one child. Hence, the induced subtree maintains the same order of nodes, but may skip over certain children in the larger tree; in the case of ASTs, this for instance means that the pattern may skip over certain instructions in the code.

More formally, given two trees $T_1 = (V_1, E_1, \lambda_1, \Sigma)$ and $T_2 = (V_2, E_2, \lambda_2, \Sigma)$, T_2 is an *induced subtree* of T_1 ($T_1 \geq T_2$) if there exists an injective *occurrence* function $f : V_2 \mapsto V_1$ such that:
(1) edges are preserved: $\forall (v, v') \in E_2$: $(f(v), f(v')) \in E_1$;
(2) labels are preserved: $\forall v \in V_2$: $\lambda_2(v) = \lambda_1(f(v))$;
(3) order is preserved: $\forall v, v' \in V_2 : v < v' \implies f(v) < f(v')$.

An example of an induced subtree is also given in Figure 1. In this smaller tree T_2, $\lambda_2(1) = \textit{Block}$; in the one occurrence of T_2 in T_1 we have $f(1) = 11$, as the number of the node with label *Block* in the larger tree T_1 is 11. Intuitively, the smaller tree captures only the return statement of the full AST. Note that a small abstract syntax tree T_2 could be an induced subtree of many different ASTs $T_1, T_1', \ldots$ in the same input dataset.

Discriminating positive from negative examples. In the adapted algorithm used in this paper, we are interested in finding patterns that distinguish *positive* from *negative* examples. To this aim, for each pattern we determine its positive and negative *support*. The *total support* of a pattern is defined as the number of ASTs in the dataset D that have the pattern as induced subtree. We partition the dataset in good and bad ASTs.[2] The *positive* (resp. *negative*) *support* is the number of ASTs in the good (resp. bad) set that have the pattern as induced subtree.

Based on the positive and negative support we can calculate a χ^2 *score*. Suppose the number of occurrences of a pattern p in the dataset D is given by the 2×2 contingency table below:

Examples	Pattern present	Pattern absent	
Good set	n_{++}	n_{+-}	$n_{+*} = n_{++} + n_{+-}$
Bad set	n_{-+}	n_{--}	$n_{-*} = n_{-+} + n_{--}$
Full dataset	$n_{*+} = n_{++} + n_{-+}$	$n_{*-} = n_{+-} + n_{--}$	$n = n_{+*} + n_{-*}$ $= n_{*+} + n_{*-}$

Then the χ^2 score of a pattern p is defined as follows:

$$\chi^2(p, D) = \sum_{i \in \{+,-\}} \sum_{j \in \{+,-\}} \frac{(n_{ij} - E_{ij})^2}{E_{ij}}$$

where E_{ij} is the expected value

$$E_{ij} = \frac{n_{i*} \times n_{*j}}{n}$$

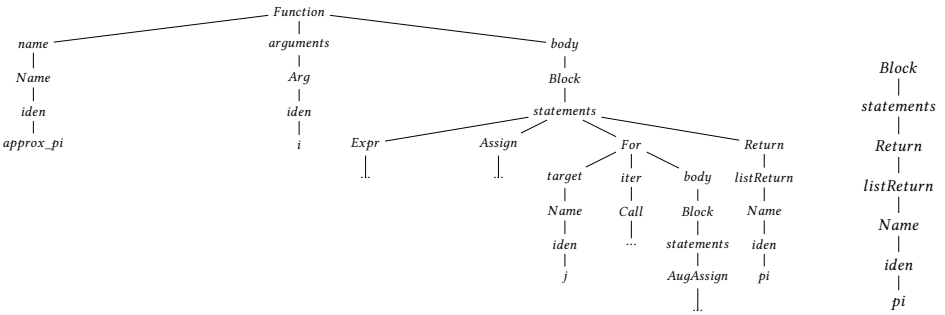
Given a minimum χ^2 threshold, *minChiSquare*, we consider a pattern p as sufficiently discriminative if the following constraint **C0** is satisfied:

$$\chi^2(p, D) \geq \textit{minChiSquare}$$

However, the number of ordered tree patterns that are discriminative can still be very large, with many such patterns being small, not interesting and hard to interpret.

Constraints. To find more useful patterns, we therefore impose additional constraints that each pattern should satisfy. These constraints were proposed and discussed in detail by Pham et al. [11]:
C1. We limit the labels allowed in the root of patterns. This enables the mining process to focus on finding patterns corresponding to specific kinds of language expressions, such as method declarations in Java or function bodies in Python.

[2] In our experiment this is done based on a score assigned by an automated grader to the programs corresponding to the ASTs. Programs in the good set have a score of 50% or more, programs in the bad set a score of less than 50%.




```
def approx_pi(i):
    """..."""
    pi = 0
    for j in range(i+1):
        pi+=4*((-1)**j/(2*j+1))
    return pi
```

Figure 1: A function in Python, part of its AST, and an induced subtree present in this AST

C2. We impose limits on the minimum and maximum number of leaves in the patterns.
C3. We impose a minimum total support that every pattern should have.
C4. We impose a minimum total number of nodes for each pattern.
C5. All leaf nodes in a pattern must have a label that is included in Σ_{leaf}, where Σ_{leaf} is the set of labels that occur in the leafs of the trees in the dataset.
C6. Given a node in an AST, it is often the case that some of its children are *mandatory* because they reflect a specific programming language construct. To avoid unnecessarily small patterns, all such obligatory children must be included in the patterns. We consider a label *structural* iff:

- in each of its occurrences, no two children have the same label;
- for all pairs of occurrences of the label, the order of the common child labels is the same.

For every label $a \in \Sigma$, we define its *obligatory child labels* $g(a)$ as the set of child labels common to all its occurrences. The constraint imposes that all such child labels are present in the pattern: let L be the structural labels in $\mathcal{D}$. For all nodes with a label $a \in L$, we require that its set of children in a pattern includes nodes with all obligatory labels $g(a)$.

We will denote the set of all patterns satisfying these constraints by $\mathcal{P}$. This set of patterns $\mathcal{P}$ still has two weaknesses: 1) in spite of the imposed constraints it is still too large; 2) whereas constraint **C2** is used and needed to reduce the time complexity and memory footprint of the algorithm, it often causes the patterns to be too small to be easily interpretable.

Post-processing: growing patterns to maximality. We address these weaknesses by post-processing the set $\mathcal{P}$. First, from $\mathcal{P}$ we consider only the k patterns with the highest χ^2 score (global constraint **G7**[3]). Subsequently each such $p \in \mathcal{P}$ is grown to obtain the patterns p' that satisfy the following conditions:

- p is an induced subtree of p' ($p' \geq p$);
- p' has exactly the same occurrences for its root node as p;
- p' satisfies all constraints **C1–C6**, except **C2**;
- no other pattern $p'' > p'$ exists that still meets these conditions.

Note how constraint **C2** is ignored during this growing process, to ensure that all patterns p' thus obtained are sufficiently large to be interpretable. In practice we observe that in the set of patterns produced by the above process, some patterns are more specific than others ($p'_1 \geq p'_2$). [4] To reduce the number of patterns further, we only keep the largest pattern in this case.

Because ignoring constraint **C2** increases the time and memory usage of the algorithm again, we set an upper bound t on the time it is allowed to spend on this post-processing step (global constraint **G8**). This time gets divided over the k selected patterns so that the algorithm gets a time slot of maximum t/k per pattern to grow. The imposed time bound may imply that some patterns will not have been grown fully to maximality. If some time remains in the totally allocated time slot t, this can be reallocated again to those patterns that haven't grown to maximality yet, until the entire time budget is exhausted.

2.2 Implementation

As mentioned earlier, we use a modified version of our FREQTALS algorithm, presented in earlier work [11], which in itself is a modified version of the FREQT algorithm [1]. The main difference with our original FREQTALS algorithm is that in this work, we are interested in finding patterns p that satisfy the additional constraint **C0** that $\chi^2(p, D) \geq minChiSquare$. Algorithm 1 presents an outline of the FREQTALS algorithm.

Step 1: Mining subtrees. The first step of this algorithm is the initial search for patterns under the constraints **C0–C6**, which is performed by the ConstrainedFREQT algorithm 2.

[3] We label this constraint **G7** rather than **C7** as it is not a constraint on individual patterns like the previous constraints, but a more **G**lobal constraint used by the mining algorithm.

[4] This may seem contradictory since one of the conditions above stated that no other pattern $p'' > p'$ exists that still meets the conditions. The difference is subtle: the patterns that are identified when growing an initial pattern, are the largest for a given set of root occurrences. But many different sets of root occurrences are considered. It is possible that one set of root occurrences is smaller than the other, leading to two patterns where one is more specific than the other, while both are still maximally specific for their respective root occurrences. The final pruning will then still eliminate the more general pattern (which likely has a larger set of root occurrences).

input : $\mathcal{D}$, parameters of constraints C0–C4 and G7–G8
output: MP.
/* Step 1: mine subtrees under constraints C0-C6, using ConstrainedFREQT */
$\mathcal{FP} \longleftarrow$ ConstrainedFREQT($\mathcal{D}$)
/* Step 2: grow patterns to maximality */
Reduce $\mathcal{FP}$ to the k highest scoring patterns (constraint G7)
$\mathcal{ROM} \longleftarrow$ groupRootOccurrence($\mathcal{FP}$)
$\mathcal{MP} \longleftarrow \emptyset$
for *each* $r \in \mathcal{ROM}$ **do**
  $c \longleftarrow$ root label of r
  mineMaximalSubtrees($c, r, \mathcal{MP}$) with time bound t (constraint G8)
end
output($\mathcal{MP}$)

Algorithm 1: FREQTALS

$\mathcal{FP} \longleftarrow \emptyset$
$C \longleftarrow$ findLabels()
prune(C)
for *each* $c \in C$ **do**
  add ($\mathcal{FP}, c$)
  expand(c)
end
output($\mathcal{FP}$)

Algorithm 2: ConstrainedFREQT

function expand(f):
  $C \longleftarrow$ findCandidates(f)
  prune (C)
  for *each* $c \in C$ **do**
    add ($\mathcal{FP}, c$)
    expand(c)
  end

Algorithm 3: expand procedure

Algorithm 2 is a recursive depth-first search algorithm of which the expand procedure detailed in Algorithm 3 is the main component. At a high level, this expand procedure grows a pattern f larger in a number of steps. First, it determines in the data which new nodes can be connected to the *rightmost path* of the pattern, while also determining how often these new nodes appear in the data. The rightmost path of a pattern is the path from the root to the last node in the ordered tree; it can be shown that it suffices to only consider extensions of this path to enumerate all patterns.

Once all candidates have been determined in C, the constraints are used to make the set of candidates smaller. For details on how to prune for constraints **C1–C6** we refer to our earlier publication [11]. The core idea is that some patterns that do not satisfy a certain constraint, do not need to be grown further. For instance, if we make a pattern with too many leaves larger, it will still have too many leaves. Other constraints, on the other hand, cannot be used to prune: a pattern that is too small, can still be grown to become sufficiently large. Those extensions that lead to a pattern that satisfies all constraints, are added to the result set $\mathcal{FP}$, after which we recursively execute the expansion process on these extensions.

The χ^2 constraint **C0** is an example of a constraint that cannot be used straightforwardly to prune the set of candidates; following the approach of Nijssen et al. [10], we check a weaker version of this constraint while pruning, and the full constraint before adding the pattern to the result.

Step 2: Growing to maximality. Once the initial set of patterns $\mathcal{FP}$ has been determined by Algorithm 1, the algorithm proceeds to step two, the post-processing phase where patterns are grown to maximality. Since this is a more time- and memory-consuming process, we first limit the set of patterns to the k patterns with the highest χ^2 score (global constraint **G7**). Subsequently, we group the patterns by the occurrences of their root, as two different patterns with identical occurrences could be grown into a same pattern. We then start a search process for maximal patterns using mineMaximalSubtrees, a modified version of Constrained-FREQT in which we find only patterns with the given set of root occurrences. During this search, we compare these patterns with those already found, to make sure that only the largest patterns are reported. In practice we find that the run time of this search process can be very long; for this reason, we use a bound on the time spent to grow patterns larger (global constraint **G8**); this offers a trade-off between the quality of the patterns found and the time spent on finding them.

2.3 Clustering

In practice, for a human observer, even though they are different some of the mined patterns still look very similar to each other. To help the human analyst, we therefore group similar patterns together in a visualisation tool (cf. Figure 2). The grouping process relies on the k-means clustering algorithm, which takes two parameters: the number of clusters to be found and a feature vector representation of the patterns. Given that the aim of the clustering is to try and group patterns that humans would consider to be related, the feature vector representation chosen is based on the labels present in the patterns: for each possible label, the feature vector contains the absolute number of times that label is present in the patterns.

3 EVALUATION

To evaluate whether the pattern mining algorithm from Section 2 can discover source code fragments that are more representative for the source code of either high or low scoring groups of students, we set up an experiment where we mined for patterns in the source code produced by actual students taking an introductory programming course exam. Subsection 3.1 describes how we configured the mining algorithm and what method we used to analyse the discovered patterns. Subsection 3.2 then presents some quantitative and qualitative results of this analysis. Finally, subsection 3.3 draws preliminary conclusions from this initial experiment.

3.1 Setup

To mine for representative patterns of good and bad programming practices in Python source code, we set up the following experiment.

Input. As input we took the source code of over 500 students participating in an online exam for an introductory programming course in Python. The questions were corrected via an automated grading system, and based on the scores assigned by that system, for each question we divided the students' responses in two different groups. The first group consisted of the responses of students who obtained a score of 50% or more for that question (high score group); the second group consisted of those who failed that question (low score group). From the low score group we excluded all submissions with a score of 0, since most of those were empty submissions from which not a lot of useful information could be mined and which

could thus bias our pattern mining results. Table 1 summarizes the input that was given to the pattern miner: for each of the 6 exam *questions* it indicates the number of non-empty *responses* received as well as the number of responses in the *high* and *low* score group.

Table 1: Input to the Pattern Miner

Question	Responses	High	Low
1	509	470	39
2	489	360	129
3	528	300	228
4	350	273	77
5a	445	186	259
5b	150	107	43

Each exam question was conceived as a small coding assignment, such as the following one:

Exam question 1: Approximate the value of π. Several formulas exist to approximate the value of π. One of these formulas is the Gregory-Leibniz series :

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots)$$

Write a Python function which takes as argument an integer $i \geq 0$ and computes an approximation of π based on the first $i+1$ terms of this series:

```
def approx_pi(i):
    """
    @pre:   i is an integer >= 0
    @post:  returns an estimation of pi by
            summating the first i+1 terms
            of the Gregory-Leibniz series
    """
```

Preprocessing. To allow for variations in variable names, after generating ASTs from the students' source code and before starting the mining process, we anonymize all variable names by replacing them with a dummy identifier $*$. This is done when the algorithm reads the input ASTs and can be turned off or on by setting a flag.

Configuration. To configure the pattern mining algorithm we set the values for its various constraints as summarized in Table 2.

Table 2: Configuration for the Pattern Miner

Constraint	Parameter	Value
C0	Minimum χ^2 threshold	0.1
C1	Allowed root label	`Block`
C2	Minimum #leaves	2
	Maximum #leaves	3
C3	Minimum total support	5
C4	Minimum #nodes	20
C5	*No parameterization required*	—
C6	*No parameterization required*	—
G7	#Patterns with highest χ^2 score kept	1000
G8	Time bound for post-processing step	10 min

Note that **C2** takes two parameters and constraints **C5** and **C6** do not require any configuration. Regarding the root label constraint **C1**, for our experiment we selected `Block` as the desired root label of all patterns to be discovered. This is because, given the way the exam questions were phrased, we are interested in the code blocks provided by students for the Python functions they have to complete.

Clustering. As can be seen in Table 3, for each question the miner discovers hundreds of patterns. To make it easier to analyse these patterns, we group them in 30 clusters by using the k-means clustering algorithm explained in Subsection 2.3, with $k = 30$.[5]

Table 3: Number of patterns and clusters per question

Question	Responses	Patterns	Clusters
1	509	152	30
2	489	181	30
3	528	411	30
4	350	383	30
5a	445	298	30
5b	150	197	30

Analysis. To help us in analysing the discovered patterns we created a visualisation tool, depicted in Figure 2, which allows us to easily inspect code fragments, corresponding to particular occurrences of patterns belonging to a selected cluster.[6] The visualisation is conceived as a set of HTML files that can easily be browsed in any web browser without the need of installing a dedicated application.

The visualisation includes a code highlighting feature that highlights in yellow all the lines of code that are part of the pattern and in red, for each line, the expressions that are part of the pattern. For example, the code fragment shown in Figure 2 is an occurrence of a pattern that corresponds to a prototypical good solution, which explains why all of the lines are highlighted as part of the pattern. Only part of the formula on line 9 is coloured in red, since the first half of the formula is not part of the discovered pattern.

3.2 Results

We know take a closer look at some of the results obtained from a quantitative and qualitative point of view.

Quantitative analysis. As the minimum total support was set to 5, each discovered pattern contains at least 5 occurrences, some of which may belong to the high-score group and some to the low-score group. E.g., pattern 46 in Figure 2 had 15 occurrences, 13 of which in the high-score group and 2 in the low-score group. Patterns with more high-scoring occurrences are more representative of good solutions, and even the low-scoring occurrences for those patterns are often close to the good solution. Patterns with more low-scoring occurrences are more representative of typical programming errors or misconceptions, whereas the high-scoring occurrences for these patterns often still have some problems too.

[5] This value of k was determined experimentally because it yielded a set of clusters that was not too large, yet where most of the patterns in each cluster were sufficiently similar to be analysed together.

[6] Due to GDPR restrictions certain names in Figure 2 were anonymized.

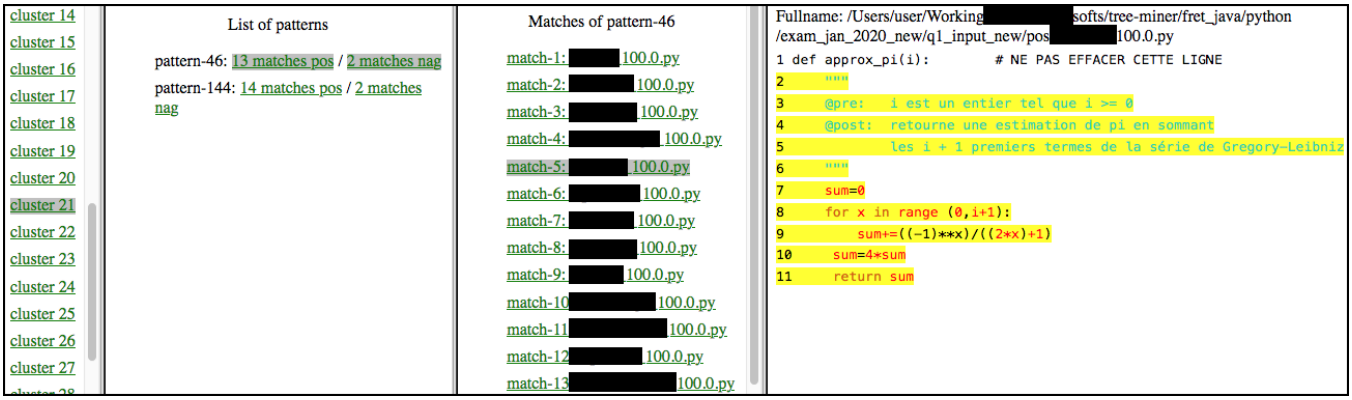


Figure 2: Using the pattern browser to inspect discovered patterns.

Table 4: Running Time of the Pattern Miner

Question	Responses	Patterns	Step 1	Step 2	Fin.
1	509	152	3	603	no
2	489	181	4	602	no
3	528	411	6	607	no
4	350	383	4	291	yes
5a	445	298	4	603	no
5b	150	197	2	602	no

Table 4 gives an indication of the running time of the mining algorithm, for each question. For completeness we also repeat the number of responses given as input and number of patterns discovered. The column *Step 1* shows the running time, in seconds, of the first step of the algorithm (mining subtrees under constraints **C0-C6** using ConstrainedFREQT). We observe that for each of the questions, step 1 takes only a few seconds. Step 2 (growing patterns to maximality by dropping constraint **C2**) takes significantly more time. Even more, we can observe that, while we imposed a time bound of 10 minutes (= 600 seconds), that time bound was reached for 5 of the 6 questions. [7] Only for question 4, step 2 managed to finish completely in the allocated time. Even though having reached the time bound implies that some of the patterns may not have been grown to their maximum size, it doesn't make the patterns found useless. We decided to keep the time bound of maximum 10 minutes for this experiment because we wanted to get a feeling of the quality of the patterns mined under such a time constraint.

Qualitative analysis. We will now take a closer look at a selection of illustrative patterns that were found by the miner in the students' responses to exam question 1. For each of these patterns we give a brief explanation of what kind of good or bad coding idiom they seem to capture. It is not our purpose to be exhaustive here but only to give the reader a feeling of the quality of the patterns that were discovered.

[7] In each of these cases we even went a few seconds over the total allocated time budget of 600 seconds. This can be explained by the fact that this time budget was divided over the growing of many different root occurrences to maximal patterns, but some time may also have been spent on switching between root occurrences to grow.

► *Good solution using a for loop.* Pattern 46, which had 13 occurrences in the high-scoring responses, captures the essence of a prototypical solution to the question that makes use of a *for* loop. Here is one such occurrence. The other occurrences are very similar up to the names of the variables being used and slight variations in how the formula is constructed.

```
sum = 0
for i in range(0,i+1):
    sum += (-1)**i/((2*i)+1)
sum = 4*sum
return sum
```

► *Good solution using a while loop.* Pattern 137, which had 5 occurrences in the high-scoring responses and 0 low-scoring occurrences, captures the essence of a prototypical solution to the question that uses a *while* loop. Here is one such occurrence:

```
pi=0
c=0
while c<=i:
    r=(4*(-1)**c)/(2*c+1)
    pi +=r
    c+=1
return pi
```

As a side-note it was surprising to discover that there were relatively few patterns that actually captured the use of a *while* loop for this particular question. It seems that most of the students preferred implementing a solution that made use of a *for* loop.

► *Missing loop.* Pattern 72 (with 4 low-scoring and 1 high-scoring occurrence) and pattern 8 (with 5 low-scoring and no high-scoring ones) capture mainly a few bad solutions where there is either no loop at all to incrementally calculate the sum, as in this occurrence:

```
s = ((-1)**i)/(2*i + 1)
return 4 * s
```

or where there is a loop but the formula does not appear in it so the loop makes little sense, as in this occurrence:

```
sum = (-1)**i/(2*i+1)
for i in range(i+1):
    i+=1
```

```
return 4*sum
```

► *Wrong iteration variable.* It was interesting to observe that pattern 46, mentioned earlier as capturing a prototypical good solution to the question by making use of a *for* loop, in addition to having 13 high-scoring occurrences, also had 2 low-scoring responses.

One was due to the use of a wrong formula; the other was due to a wrong use of iteration variable in the loop. Indeed, the pattern did allow for slight variations in the formula, as well as for variance in variable names. As a consequence of the latter, the pattern does not impose that the iteration variable used inside the formula (for example *i*) is the same as the one used by the for loop (for example *j*), as can be seen in this low-scoring occurrence of the pattern:

```
s = 0
for j in range(0, i+1):
    s += (-1)**i/(2*i+1)
pi = 4*s
return pi
```

► *Sum update and return at the same indentation level.* Pattern 91 occurs in 1 high and 4 low-scoring responses. The pattern contains, at the same indentation level, an assignment instruction like `s=((-1)**i)/(2*i+1)` together with a return statement like `return 4*s`. The four low-scoring responses all fail the question because having the return statement of the final calculated sum at the same level as the assignment statement in most cases leads to a wrong behaviour:

Two of the low-scoring responses were wrong because no sum assignment is nested inside the loop. Here is a bad occurrence with a *for* loop, the other bad occurrence was similar using a *while* loop.

```
sum =(-1)**i/(2*i+1)
for i in range(i+1):
    i+=1
return 4*sum
```

Another low-scoring response was clearly wrong because it contained no *for* or *while* loop at all.

```
s = ((-1)**i)/(2*i + 1)
return 4 * s
```

The fourth wrong response matching this pattern had the sum update in the right place inside the *for* loop, but the *return* statement as well, thus causing the function to exit after the first iteration.

```
sum=0
for n in range(i+2):
    r=((-1)**n)/(2*n+1)
    sum+=r
    return 4*sum
```

Surprisingly, one response matching this pattern did score high:

```
a=i
pi=((-1)**(i))/(2*i+1)
while a>0:
    a-=1
    pi+=((-1)**(a))/(2*a+1)
return 4*pi
```

As shown by the code below, this is because the sum assignment is done once outside the loop, at the same indentation level as the return statement (thus matching the pattern) but then the sum is also updated inside the loop. Whereas this code is not the most compact solution possible because of the duplicated formula, it did pass the tests and thus the student succeeded this question.

3.3 Preliminary Observations

It is not trivial to find the right configuration and set of constraints to use for the pattern mining algorithm to produce the best results. In prior code mining experiments [11] we used configurations close to the one used in this particular experiment, to produce patterns revealing interesting insights in an acceptable amount of time, for different programming languages. Without claiming that the configuration used in this experiment is the best one, it did allow us to find interesting patterns representing relevant good and bad coding idioms with relatively little effort.

Clustering the results proved helpful as well, even though here too it is not trivial to find out what clustering algorithm and parameters produce the best result. With the current k-means clustering algorithm, for a value of $k = 30$ the 30 clusters produced were not too large containing mostly a handful of similar patterns, which made them easy to compare. Nonetheless there were quite some clusters with 1 pattern only, as well as clusters with 10 patterns, which were less relevant.

We also observed that some of the mined patterns were quite similar. For example, patterns 46 and 144 with 13 respectively 14 occurrences in the high-score set and 2 in the low-score set, describe nearly the same pattern and correspond to the same students (except one). A possible explanation (that would require further investigation) could be that due to the time bound in step 2 they were not grown into a single larger pattern. Luckily however, because they are so similar the clustering algorithm did present them together in a single cluster (cluster 21), so that their similarity was easily discovered. The same seems to be the case for several patterns grouped in other clusters.

In some cases the patterns also tend to be quite specific leading to very detailed patterns with relatively few occurrences. More generic patterns would match more occurrences and lead to fewer patterns overall. But then again they shouldn't become too generic so that they are no longer revealing interesting idioms.

Finally, the visualisation and browsing tool, and the ability to immediately see the code occurrences matching the patterns was crucial in analysing the results. Especially the tool's ability to highlight those parts of the code that match the pattern made it easy to understand the pattern and whether it captured some interesting solution, idiom or programming misconception.

3.4 Another Experiment

Coming back to the discussion on what would be an ideal configuration for the algorithm, we decided to run another quick experiment on question 1, with a time bound of 60 minutes for the post-processing step (as opposed to 10 minutes), and keeping 100 (instead of 1000) patterns with highest χ^2 score. By setting a higher time bound and keeping less patterns with high χ^2 score, we hoped to give the algorithm more time to find larger and less patterns, but of higher quality.

A first thing that we observed was that the algorithm still reached a time out. We were also surprised to notice that the algorithm

didn't find less patterns than before but even a few more. There were also a few patterns (like pattern 46) that we found before that now didn't appear anymore and vice versa (see below). Whereas all this can be explained in terms of some of the subtle details of the algorithm, it confirms our statement that it is not easy at all to find an ideal configuration for the mining algorithm.

In this second experiment we also set the number of clusters to 10 instead of 30, to avoid finding many clusters with only 1 pattern. And even though there were indeed no more clusters with only 1 element, we now got clusters with more than 20 elements, which seemed too large.

To conclude this section we couldn't help but show you this little "pearl" that was discovered in this second experiment because of the high redundancy in this solution:

```
if i == 0 :
    return 4
if i == 1 :
    x = 4*(2/3)
    return (x)
if i == 2:
    x = 4* (2/3 + 1/5)
    return (x)
if i == 3:
    x = 4* (2/3 + 1/5 + 1/7)
    return x
""" THIS WENT ON UNTIL i == 12 """
```

4 CONCLUSION AND FUTURE WORK

In this paper we adapted an existing source code mining approach [11] to apply it to a new setting for discovering good and bad coding idioms from student code. In this section we will discuss some future work ideas to build upon the promising results of the initial experiment presented in this paper.

Although the experiment in this paper was using the Python programming language, the pattern mining algorithm itself is not specific to a particular language and has been applied in the past, though not yet to distinguish good from bad student code, on other programming languages as well (Java, Cobol and C#). We could envision conducting experiments similar to the one presented in this paper to student code written in Java, since we have all the necessary data available from an introductory programming course taught in Java.

To group the discovered patterns in clusters, the pattern visualisation and browsing tool made use of k-means clustering, which is one of the most well-known clustering algorithms. Other clustering algorithms could be tried out as well, such as affinity propagation clustering, which is a relatively fast clustering algorithm that does not require setting a specific input parameter k as is the case for k-means clustering. Initial experiments that we conducted seem to indicate that this algorithm would also yield relatively good clusters. Exploring what clustering algorithm and configuration is most optimal for our particular setting remains a point of future work.

We should also explore in more detail what values for the parameters of the different constraints of the pattern mining algorithm would yield the best results. Once the mining and clustering algorithms and their parameters have been fine-tuned to provide the best results possible, so that we can easily discover high-quality patterns representing good or bad programming idioms or misconceptions, a next important step would be to explore how to exploit this information in a *source-code based recommendation system* [9] that could help students learn from their errors, or teachers to learn about typical misconceptions of their students.

A suggestion of how this could work is provided by Pattern 46, which occurred in many good answers, but also a small number of incorrect ones. In this case, it could be informative for the students who had incorrect solutions to see a correct solution that had the same pattern. Conversely, as was exemplified by the last occurrence of Pattern 91, a solution that got a high score but belongs to a pattern containing many low-scoring solutions may be suspect and worth exploring in more detail by the teacher as the high-scoring solution may not represent the best solution possible.

Integrating such a recommendation system in an automated grading tool would be enriching so that students not only know that their code is wrong and what tests it doesn't pass, but also why it is wrong and how it could be improved. Even students who solved a question correctly could receive recommendations on how to improve their coding skills.

ACKNOWLEDGEMENTS

Work partially conducted in the context of an industry-university research project between UCLouvain, Vrije Universiteit Brussel and Raincode Labs, funded by the Belgian Innoviris TeamUp project INTiMALS (2017-TEAM-UP-7) and with the support of the Erasmus+ programme of the European Union.

Co-funded by the Erasmus+ Programme of the European Union

REFERENCES

[1] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, and Setsuo Arikawa. 2004. Efficient substructure discovery from large semi-structured data. *IEICE TRANSACTIONS on Information and Systems* 87, 12 (2004), 2754–2763.

[2] Céline Deknop, Simon Baars, Kim Mens, Ana Oprescu, and Johan Fabry. 2019. Clone Detection vs. Pattern Mining: The Battle, In 18th Belgium-Netherlands Software Evolution Workshop (BENEVOL2019). *CEUR Workshop Proc. 2605.*

[3] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. 2015. 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.

[4] Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond).* 49–50. https://doi.org/10.1109/BLOCKS.2015.7369000

[5] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *44th ACM Technical Symposium on Computer Science Education (SIGCSE '13).* ACM, 579–584. https://doi.org/10.1145/2445196.2445368

[6] Aída Jiménez, Fernando Berzal, and Juan Carlos Cubero Talavera. 2010. Frequent tree pattern mining: A survey. *Intell. Data Anal.* 14, 6 (2010), 603–622. https://doi.org/DOI:10.1007/978-3-319-07821-2_2

[7] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ System and its Pedagogy. *Computer Science Education* 13, 4 (2003), 249–268. https://doi.org/10.1076/csed.13.4.249.17496

[8] Thomas Lancaster and Fintan Culwin. 2004. A Comparison of Source Code Plagiarism Detection Engines. *Computer Science Education* 14, 2 (2004), 101–112. https://doi.org/10.1080/08993400412331363843

[9] Kim Mens and Angela Lozano. 2014. Source Code-Based Recommendation Systems. In *Recommendation Systems in Software Engineering*, Martin P. Robillard,

Walid Maalej, Robert J. Walker, and Thomas Zimmermann (Eds.). Springer Berlin Heidelberg, 93–130. https://doi.org/10.1007/978-3-642-45135-5_5

[10] Siegfried Nijssen and Joost N. Kok. 2005. Multi-class Correlated Pattern Mining. In *Knowledge Discovery in Inductive Databases, 4th International Workshop, KDID 2005, Porto, Portugal, October 3, 2005, Revised Selected and Invited Papers (Lecture Notes in Computer Science, Vol. 3933)*, Francesco Bonchi and Jean-François Boulicaut (Eds.). Springer, 165–187. https://doi.org/10.1007/11733492_10

[11] Hoang Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, Coen De Roover, Johan Fabry, and Vadim Zaytsev. 2019. Mining Patterns in Source Code Using Tree Mining Algorithms. In *Discovery Science*. Springer. https://doi.org/10.1007/978-3-030-33778-0_35

[12] Lutz Prechelt, Guido Malpohl, and M. Philippsen. 2002. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science* 8 (2002), 1016–1038. https://doi.org/10.3217/jucs-008-11-1016

[13] Ricardo Alexandre Peixoto Queirós and José Paulo Leal. 2012. PETCHA: A Programming Exercises Teaching Assistant. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)*. ACM, 192–197. https://doi.org/10.1145/2325296.2325344