Interpretable Clustering of Students' Solutions in Introductory Programming

Tomáš Effenberger and Radek Pelánek

Masaryk University Brno
Czech Republic
tomas.effenberger@mail.muni.cz, pelanek@fi.muni.cz

Abstract. In introductory programming and other problem-solving activities, students can create many variants of a solution. For teachers, content developers, or applications in student modeling, it is useful to find structure in the set of all submitted solutions. We propose a generic, modular algorithm for the construction of interpretable clustering of students' solutions in problem-solving activities. We describe a specific realization of the algorithm for introductory Python programming and report results of the evaluation on a diverse set of problems.

1 Introduction

Learning environments often provide problem-solving activities, where students construct solutions that are automatically evaluated for correctness while still allowing for multiple approaches. We focus on introductory programming in Python, but similar types of problems are common in computer science education (e.g., regular expressions, SQL), mathematics (geometry constructions, logic proofs), or physics (gravity, electrical circuits).

Even for a simple problem, there may be many solutions; see Fig. 1 for a specific illustration for introductory programming. All these programs passed functionality tests, yet they differ significantly in their style and quality. Online learning environments collect a large number of solutions, and it is not feasible to analyze all of them manually. It is thus useful to use machine learning techniques to uncover structure in the solution set, particularly to cluster similar solutions.

Such clustering has several use cases. To teachers, it provides a summary of students' approaches, examples of poor style, or inspiration for class discussion [6]. The understanding of students' solutions is also valuable for content authors; the clustering can reveal that a problem is solved in an unexpected way, which is helpful for guiding revisions and the development of new content [13]. Another application is automating feedback to students [17,18]. If we are able to find sufficiently coherent clusters, we can use the same feedback message for the whole cluster. Clustering can also be used to improve student models since the cluster into which a solution belongs provides additional information about the student's state beyond the commonly used answer correctness and response time. For example, a solution to a programming problem can contain evidence of a misconception or insufficient understanding of some programming concepts.

a
```
def count_a(text):
    x = 0
    for i in range(len(text)):
        if text[i] == "A" or \
           text[i] == "a":
            x += 1
    return x
```

b
```
def count_a(text):
    n = 0
    for i in text:
        if i == 'a' or \
           i == 'A':
            n += 1
    return n
```

c
```
def count_a(text):
    x = text.count('a') \
        + text.count('A')
    return x
```

d
```
def count_a(text):
    count = 0
    for i in text.lower():
        if i == "a":
            count += 1
    return count
```

e
```
def count_a(text):
    text = text.upper()
    count = 0
    for letter in text:
        if letter == "A":
            count += 1
    return count
```

f
```
def count_a(text):
    count = 0
    for i in text:
        if i == "a" or i == "A":
            count = count + 1
    return count
```

Fig. 1. Examples of students' solutions to the following programming problem: *"Write a function that counts the occurrences of letters 'a' and 'A' in a text."*

Clustering of students' solutions has been tackled before, even specifically for the introductory programming [3,5,21]. Yet, no algorithm was developed that would lead to a small number of *interpretable* clusters, as needed for many of the outlined use cases. Consider, for instance, feedback writing. Having a guarantee that all solutions in the cluster contain `for` and `if`, and do not contain `ord` can save you from manually inspecting all the solutions.

Previously proposed algorithms that consider interpretability are based on some notion of exact matching (e.g., equivalence after canonicalization), which leads to hundreds of small clusters [6,9,14]. With so many clusters, the complete clustering is not well-interpretable, even if the individual clusters are. Another substantial limitation of these previous attempts is that they were evaluated on just 3 or 4 similar problems, and it is not at all clear how well they would generalize beyond them.

Outside of the educational domain, several interpretable clustering algorithms have been proposed. They describe clusters using either branches in a decision tree [2,4,15], frequent patterns [19], or the most relevant features in matrix decomposition [8]. These algorithms cannot be used off-the-shelf for clustering students' solutions since they are not designed to utilize varying importance of solution's features, e.g., the occurrence of recursion vs. addition.

In this paper, we formulate the problem of interpretable clustering of students' solutions in terms of desirable properties of such clustering. We then propose a generic algorithm to solve this problem, describe its specific realization for introductory programming in Python and report the results it gives for a diverse set of problems.

2 Interpretable Clustering Problem

The general aim of interpretable clustering of students' solutions is to compute clusters of solutions that are useful for the intended applications where the interpretability is indispensable. To facilitate interpretability, the output should consist of not just the clusters of solutions but also their succinct description. An

example of such description is "`for, if, or, no [i]`" for solutions that use `for`, `if`, and `or` and do not use indexing. Although the utility of a clustering depends on the specific application, we can formulate three general key properties of any interpretable clustering: homogeneity, interpretability, and coverage.

Homogeneity. Each cluster should be compact, i.e., the solutions within the cluster should be similar to each other. In addition, the clusters should be well separated from each other, i.e., the solutions from different clusters should be dissimilar. These two requirements apply to non-interpretable clustering as well, and many metrics to quantify them have already been proposed, e.g., variance ratio, Xie-Beni index, and Silhouette coefficient [16]. Many of these metrics define homogeneity as the ratio between within-cluster compactness and between-clusters separability. Compactness can be measured, for instance, as the average distance between two points in the cluster and the separability as the distance from the cluster centroid to the closest centroid of another cluster.

Interpretability. Each cluster should be accompanied by a succinct description. These descriptions should provide insight into students' approaches and facilitate the writing of useful feedback applicable to all solutions in the cluster. Some applications require *perfect recall* of the descriptions, meaning that the description applies to all solutions in the cluster. Without perfect recall, the description could easily mislead the user to write feedback that does not make sense for some of the solutions. This condition is also referred to as *strong interpretability* or *1-interpretability* [19]. Ideally, the description should apply *only* to the solutions in the cluster being described *(perfect precision)*. We may, however, trade off precision for improvement in other criteria.

Coverage. Each cluster should cover a reasonable portion of the solutions. Consequently, a small number of clusters should be sufficient to cover a vast majority of the solutions. For most applications, we do not need to have complete coverage — it is sufficient to cover all the typical solutions and report the rest as atypical. The appropriate number of clusters depends on the application; in our experience, 4 to 8 clusters are appropriate for writing feedback and providing insight to authors.

3 Interpretable Clustering Algorithm

In this section, we describe an algorithm that solves the interpretable clustering problem. The proposed algorithm is flexible — it can be applied to any problem type just by specifying appropriate features, and it can be adapted to different use cases by adjusting parameters that determine focus on individual criteria (homogeneity, interpretability, and coverage). Thus it constitutes a good starting point against which to compare more complex or specialized approaches.

In the description of the algorithm, we use the following terminology: *feature* is a property of a solution (e.g., usage of a concept like `if` or `nested loops`), *clause* is a single feature with an optional quantifier (e.g., `many if`, `no elif`), *pattern* is a conjunction of multiple clauses, and *label* is a short, possibly imprecise description of the pattern.

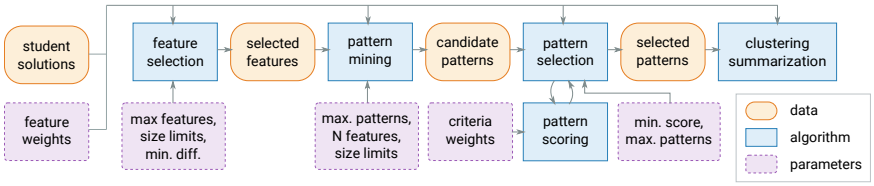

Fig. 2. Overview of the proposed interpretable clustering algorithm.

The input to the algorithm is a set of students' solutions to a given problem, represented in the form of a feature matrix. The features should be interpretable properties of the solutions, such as `recursion`. The algorithm describes the clusters by patterns over these interpretable features. In the final stage, the patterns are converted to short labels by omitting less important clauses.

The algorithm consists of four stages (Fig. 2), which are to a large degree independent and can be individually improved — or even approached in a distinctively different way than in our proposal. The four stages are:

1. **feature selection**: For the given problem, we select a small set of important, relevant, and distinct features.
2. **pattern mining**: Combining the selected features, we generate a set of candidate patterns that capture a large portion of the solutions, with the preference for short patterns with important features.
3. **pattern selection**: We score each candidate pattern with respect to its homogeneity, interpretability, and coverage. We then select the pattern with the highest score, remove matching solutions and repeat. We stop once we have enough patterns, or earlier if there is no pattern with a high score.
4. **clustering summarization**: We summarize each cluster by a short label derived from the pattern, together with a few examples of specific solutions from the given cluster.

A useful tool for understanding, implementing, and improving the algorithm is the feature matrix visualization with a column for each solution and a row for each feature (Fig. 3). If we cluster the solutions according to the selected feature patterns, we can see homogeneity and coverage of individual clusters at a glance.

3.1 Feature Selection

Different problem types need different features. For regular expressions problems, individual letters might be sufficient, while for programming problems, letters would be useless. Instead, we can extract keywords and compute statistics such as the number of variables from the abstract syntax tree of the program [7,17,20]. A completely different set of features can be obtained by similarity analysis, e.g., edit distances to other solutions [12,21], or dynamic analysis, e.g., variable values sequences [6,9].

Many of these features might be useful for non-interpretable clustering, but only the ones that are interpretable by themself are suitable for the interpretable

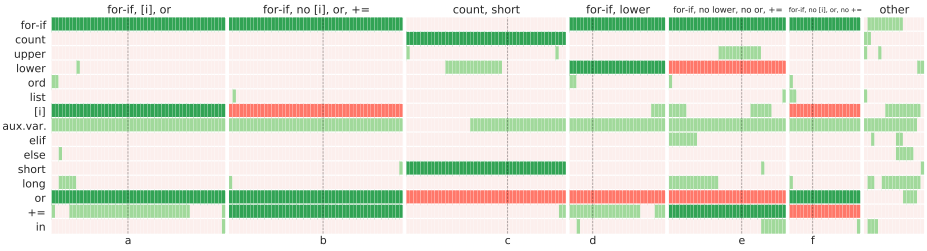

Fig. 3. Clustered feature matrix for the problem *Count A* with highlighted solutions `a`–`f` from Fig. 1. Each column corresponds to one solution, each row to one feature. Color hue denotes presence of features in solutions. Darker colors denote features in the corresponding pattern.

clustering. The more comprehensive and interpretable features, the better the output of the clustering algorithm, which is why some authors hand-crafted very specific features such as shape of the memoization array for dynamic programming problems [14], or whether a given sorting function is in-place [20].

Our algorithm can utilize domain knowledge about importance (interpretability) of the features in the form of *feature weights*. Instead of setting them manually, the weights can also be estimated from the data, e.g., based on the prevalence of the feature in the solutions.

If we define the features and their weights for a *problem type*, then only a fraction of these features might be relevant for any *specific problem*. Therefore, in the first stage, we select a set of useful features for the given problem. We use a greedy approach: considering one feature at a time, starting with the feature with the highest weight, we select the feature unless it is either extremely rare, used in nearly all solutions, or too similar to one of the already selected features. To measure the similarity between two features, we use the Jaccard coefficient (ratio of intersection and union) of the sets of solutions containing these features.

To illustrate the feature selection, let us consider the *Count A* problem (Fig. 1). The algorithm — assuming thresholds discussed later in section 4.1 — selects 15 features that are listed in Fig. 3. It skips 10 rare features, e.g., `recursion`, which was used in only 4 out of 240 solutions. It also skips 6 features too similar to other already selected, e.g., `if`, which closely coincides with more specific `for-if`.

3.2 Pattern Mining

The next step is to generate frequent patterns using the selected features. In contrast to the well-known apriori algorithm and other general pattern mining techniques [10], we take into account the interpretability of the patterns by preferring fewer clauses and important features (i.e., features with high weights). Our approach is similar to the depth-first *tree projection algorithm* for mining frequent itemsets [1], using feature weights for the ordering of the features.

We generate the candidate patterns recursively, starting from an empty pattern. For each feature and each possible quantifier, we try to extend the parent pattern by one clause (quantified feature). We then check whether the extended pattern is sufficiently frequent (i.e., whether there are enough solutions that match the pattern) while simultaneously not too similar to the parent pattern (i.e., whether there are enough solutions that match the parent pattern but not the extended pattern). If both conditions are met, we include the extended pattern into the candidate set and search for more specific patterns recursively.

The search tree can differ a lot between problems, so it is impossible to have a single set of universal thresholds. We circumvent this issue by *iterative deepening*: we start with tight thresholds, run the search and iteratively loosen the thresholds until we find a sufficient number of patterns (e.g., 1000).

We introduce two additional modifications that increase the interpretability of the generated patterns. First, we increase the thresholds on the pattern inclusion in proportion to the length of the pattern, expressing the preference for shorter patterns. Second, we increase the number of considered features in each iteration (*iterative broadening*), expressing the preference for important features.

3.3 Pattern Selection

To select patterns, we use a greedy approach known as *sequential covering* [11]. In each iteration, we score all candidate patterns, select the best, and remove matching solutions. This process is repeated until we select a prespecified number of patterns or until there is no pattern with a score above a prespecified threshold. Instead of using a constant threshold for the minimum score, we can increase it in each iteration; this is useful when the problems are diverse: starting with a low threshold ensures that at least some patterns are selected, while increasing it after each iteration avoids selecting an excessive number of patterns.

Pattern scoring reflects the desirable properties of homogeneity, interpretability, and coverage. We operationalize these properties using scores with a value between 0 and 1; a higher value is better. In the following discussion, we highlight the high-level idea and rationale behind each part of the scoring function. We also briefly mention specific formulas used in our realization of the algorithm.

Homogeneity consists of two aspects — *hard* and *soft* — which are averaged. *Hard homogeneity* is the degree to which *all* solutions in the cluster share some features (or their absence). Hard homogeneity is closely related to interpretability and actionability of clusters since exact matches are easy to understand and act upon (e.g., in feedback). We quantify hard homogeneity as the sum of weights of the shared features, normalized by the sum of weights of all relevant features selected for the problem. *Soft homogeneity* is the degree to which solutions in the cluster are similar and differentiated from other clusters. Soft homogeneity also applies to non-interpretable clustering, but the standard measures like Silhouette coefficient [16] must be adapted to work with an incomplete clustering. We quantify soft homogeneity as $\max(0, 1 - D_{in}/D_{out})$, where D_{in} is the mean distance from a solution within the cluster to the centroid, and D_{out} is the mean distance from a solution outside the cluster to the centroid.

Interpretability considers four properties of the pattern; the interpretability score is the product of the individual criteria. The pattern should be *short* (long patterns are harder to interpret) and contain features that are *important* and *positive* (negative clauses are harder to interpret). The fourth aspect is *precision*, which expresses the preference to avoid "false positives," i.e. solutions matching the pattern that are already assigned to one of the previous patterns. Specifically, the length score is b^{L-1}, where L is the number of clauses and $b = 0.95$; the importance score is the average of the maximum and mean feature weights, normalized by the maximum weight over all selected features; the positivity score is $1 - c \cdot$ (proportion of positive clauses), using $c = 0.5$; and the precision is the ratio of the number of solutions in the cluster to the number of all solutions matching the pattern.

Coverage is based on the size of the cluster. A straightforward approach would be to make the coverage score equal to the relative size r of the cluster. However, it is more important to distinguish clusters of the sizes 3% and 6% than clusters of sizes 53% and 56%, so it is preferable to use a nonlinear scoring function. We use a simple, one-parametric piecewise linear function. Specifically, the coverage score is $\frac{1-k}{k}r$ for $r < k$ and $\frac{k}{1-k}r + 1 - \frac{k}{1-k}$ for $r \geq k$, using $k = 0.2$.

Overall score. To combine these three criteria into a single score, we take their harmonic mean. The harmonic mean is more sensitive to the lowest value than the arithmetic mean, which better suits the requirement that all three criteria should be reasonably satisfied — even perfect coverage cannot make up for poor interpretability. If one of the criteria is more important for the considered use case, *weighted* harmonic mean can be used.

3.4 Clustering Summarization

Finally, it is useful to provide a short description of each cluster, as the full patterns are sometimes too long. A basic step is to remove implied features (e.g., `for-if` implies `if`). We can also simplify patterns by omitting some less important clauses. For example, negative clauses like `no import` are only informative if the feature appears in many solutions; otherwise, the user is likely to assume that the feature is not used unless specifically mentioned in the pattern.

4 Application to Python Programming

We have developed a proof-of-concept implementation of the algorithm and applied it to introductory Python programming data. The data come from an online learning environment `umimeprogramovat.cz`, which is used by both high school and university students. The environment offers quite a standard interface for solving programming problems: students see a problem statement and a sample testing data, write the code inside the browser, and after each submission, their solution is evaluated on hidden tests. If the submitted program is incorrect, the student can improve it and submit again. In this work, we consider only the correct solutions (i.e., the solutions that passed the tests).

The problems in the environment cover most topics typically included in the first university programming course (CS1). The simplest problems are one-line programs, such as writing a logic condition. The most difficult ones can still be solved with up to 15 lines of code but involve non-trivial concepts like lists and nested loops and take an average student around 15 minutes. The number of collected solutions ranges from 80 to 550 per problem.

4.1 Methodology and Setting

We have developed the algorithm iteratively using 11 problems (2 358 solutions). We have manually labeled a subset of solutions from these problems to clarify the desirable output of individual stages and perform experiments to refine the algorithm and find reasonable values for parameters. After this design phase, we reached the algorithm as described above. Then, we tested the algorithm on 11 new problems (2 598 solutions) without any change to the algorithm, parameter values, or feature weights. The number of problems may seem small, but our dataset is actually much more diverse than the datasets used in previous work [6,9,14], which contain just 3 or 4 similar problems.

The algorithm requires specification of features relevant for a given problem type, together with their weights. We automatically extract about 100 features from the abstract syntax tree. Most features correspond directly to a node in the abstract syntax tree (e.g., `for`, `if`), but a few are derived from relationships between multiple nodes (e.g., `recursion`, `for-if`). In addition, we use features `short` and `long` for programs that are below the first or above the last quartile in the number of lines for a given problem.

To set the feature weights, we used a semi-automatic approach. We started with weights estimated by a heuristic based on how soon and how frequently the feature appears in students' solutions (considering the ordering of problems in the learning environment), and then we manually adjusted some of the weights according to our experience with feedback writing.

To set values for other parameters, we used the training set of 11 problems. The advantage of the modular approach is that individual stages are largely independent, and thus each parameter can be set by analyzing inputs and outputs of a single stage. Using this approach, we reached the following setting:

- *feature selection*: max. 20 features, rel. size limit 0.02, min. difference 0.1,
- *pattern mining*: max. 1000 patterns; $12 + i$ features and relative size limit $0.09 - 0.01 \cdot i$ in the i-th iteration ($i \in 1, 2, \ldots, 8$),
- *pattern scoring*: unit weights in the harmonic mean; coverage score function with $k = 0.2$, length score base $b = 0.95$, positivity effect $c = 0.5$,
- *pattern selection*: max. 10 patterns, min. score $0.05 \cdot i$ in the i-th iteration.

4.2 Results

Fig. 4 contains a compact overview of the obtained clusters for the 22 problems. The problems are displayed in the same order in which they appear in the learn-

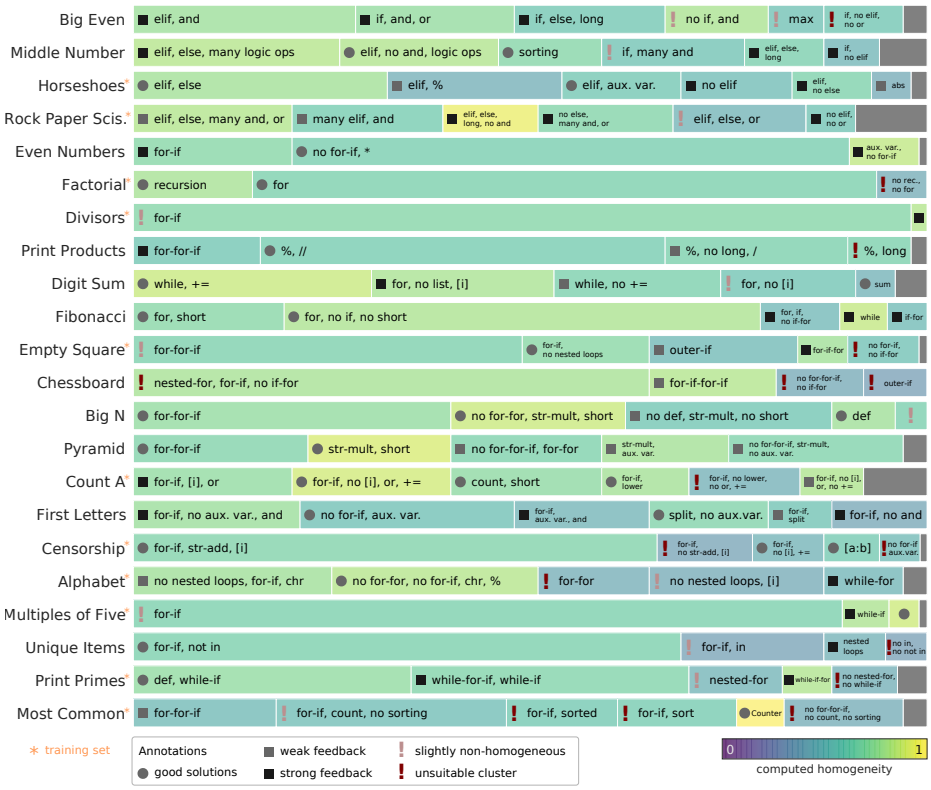

Fig. 4. Overview of clusters found by the algorithm, together with a manual rating of their quality. The gray rectangles correspond to the remaining unclustered solutions.

ing environment, i.e., approximately from easier to more difficult. The problems used for training are marked by an asterisk.

Each rectangle represents one cluster: width corresponds to coverage, color to computed homogeneity, and the label can be used for basic assessment of cluster interpretability. For each cluster, we created a detailed report — complete patterns and a sample of solutions belonging to the cluster. Based on these reports, we manually classified each cluster into one of five categories: *good solutions* (a homogeneous set of solutions that do not require feedback), *strong feedback* (a set of solutions for which we can provide clear and useful advice that is applicable to all of them), *weak feedback* (similar to strong feedback, but the feedback is rather a hint or a suggestion), *slightly non-homogeneous* (the cluster makes sense, but is not completely homogeneous), and *unsuitable cluster* (highly non-homogeneous or hard to interpret).

Overall, the results show that the algorithm can generate useful and interpretable clusters. For many clusters, we can provide strong feedback, e.g., in the *Count A* problem, there is a large "`for-if,or,[i]`" cluster (Fig. 1a) for

which we can provide the following feedback: *"This problem can be solved more elegantly without indexing."* Negative clauses are often indicative of useful feedback, especially if the feature in question is important and used by most of the other solutions. For example, in the *Rock Paper Scissors* problem, the 3rd cluster contains long solutions that use many nested `if`s. Useful feedback is to show how the solution to such problem can be greatly simplified using logical operators. Similarly, students in the 6th cluster write complicated code without `elif`; they might even not know this useful construct.

As another example, consider the simplest problem in the dataset: *Big Even* (*"Write a function that returns True if the larger of the two numbers is even."*) Students were expected to solve this problem with one line of code. These compact solutions are in the cluster "`no if, and`," which is slightly non-homogeneous due to presence of a few longer solutions. The output of the algorithm reveals that students solve the problem in other ways than anticipated and provides useful impulse for system designers (e.g., for the development of new, scaffolded problems). The three largest clusters also afford clear and useful feedback.

The algorithm sometimes produces clusters that are not sufficiently homogeneous or satisfactory. This is mostly the case of the last clusters. These cases could be partially resolved by further tinkering with the algorithm parameters, but partially it is a consequence of the basic greedy strategy used in the algorithm. The problematic cases are distributed relatively uniformly among the training and test set, i.e., it is not the case that we have overfitted the training set, but rather a sign that some problems would require a more tuned or improved algorithm. However, for some cases, it would be challenging to provide a high-quality interpretable clustering even for a human expert.

5 Discussion

The central aim of this work is to highlight the issue of *interpretability* in the context of clustering of students' solutions in problem-solving activities. For this purpose, we propose a generic, modular algorithm and demonstrate its application to data from introductory Python programming. The algorithm is able to produce useful, interpretable, and actionable clusters — they provide useful insight for content authors and allow efficient distribution of feedback to students.

The limitation of the presented work is that it is based solely on qualitative evaluation by the algorithm authors. The algorithm also contains quite a few choices and parameters. Although our experience suggests that the approach is reasonably robust and we have not observed any significant degradation of performance on the test set, the setting of the algorithm parameters needs further exploration. Since the presented algorithm is able to produce reasonable clusters, it provides a good starting point for a search for improved versions. These improvements can take the form of better parameter optimization, but also of non-greedy alternatives to individual stages or even significantly different approaches. Another important direction for future work is the exploration of the generalizability of the proposed algorithm to other problem-solving activities.

References

1. Ramesh C Agarwal, Charu C Aggarwal, and VVV Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of parallel and Distributed Computing*, 61(3):350–371, 2001.
2. Jayanta Basak and Raghu Krishnapuram. Interpretable hierarchical clustering by constructing an unsupervised decision tree. *IEEE transactions on knowledge and data engineering*, 17(1):121–132, 2005.
3. Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences*, 23(4):561–599, 2014.
4. Sanjoy Dasgupta, Nave Frost, Michal Moshkovitz, and Cyrus Rashtchian. Explainable k-means clustering: Theory and practice. In *XXAI Workshop, ICML*, 2020.
5. Lei Gao, Bo Wan, Cheng Fang, Yangyang Li, and Chen Chen. Automatic clustering of different solutions to programming assignments in computing education. In *Proceedings of the ACM Conference on Global Computing Education*, pages 164–170, 2019.
6. Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):1–35, 2015.
7. Elena L Glassman, Rishabh Singh, and Robert C Miller. Feature engineering for clustering student solutions. In *Proceedings of the first ACM conference on Learning@ scale conference*, pages 171–172, 2014.
8. Derek Greene and Pádraig Cunningham. Producing accurate interpretable clusters from high-dimensional data. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 486–494. Springer, 2005.
9. Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, 53(4):465–480, 2018.
10. Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *Data mining and knowledge discovery*, 15(1):55–86, 2007.
11. Jiawei Han, Micheline Kamber, and Jian Pei. Data mining concepts and techniques, third edition. *The Morgan Kaufmann Series in Data Management Systems*, 5(4):83–124, 2011.
12. Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*, volume 25, 2013.
13. David Joyner, Ryan Arrison, Mehnaz Ruksana, Evi Salguero, Zida Wang, Ben Wellington, and Kevin Yin. From clusters to content: Using code clustering for course improvement. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 780–786, 2019.
14. Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 739–750, 2016.

15. Bing Liu, Yiyuan Xia, and Philip S Yu. Clustering through decision tree construction. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 20–29, 2000.
16. Yanchi Liu, Zhongmou Li, Hui Xiong, Xuedong Gao, and Junjie Wu. Understanding of internal clustering validation measures. In *2010 IEEE international conference on data mining*, pages 911–916. IEEE, 2010.
17. Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*, pages 491–502, 2014.
18. Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. Learning program embeddings to propagate feedback on student code. volume 37 of *Proceedings of Machine Learning Research*, pages 1093–1102, Lille, France, 07–09 Jul 2015. PMLR.
19. Sandhya Saisubramanian, Sainyam Galhotra, and Shlomo Zilberstein. Balancing the tradeoff between clustering value and interpretability. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, pages 351–357, 2020.
20. Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pages 83–92, 2012.
21. Hezheng Yin, Joseph Moghadam, and Armando Fox. Clustering student programming assignments to multiply instructor leverage. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, pages 367–372, 2015.