Enhancing Computer Science Education by Automated Analysis of Students' Code Submissions

Lea Eileen Brauner and Frank Höppner[0000−0003−4170−5077]

Ostfalia University of Applied Sciences
Dept. of Computer Science, D-38302 Wolfenbüttel, Germany
le.brauner@ostfalia.de

Abstract. Lecturers of introductory programming courses are often faced with the challenge of supervising a large number of students. Reviewing a large number of programming exercises is time-consuming, and an automated overview of the available solution approaches would be helpful. In this paper, we focus on source code similarity at the level of students' selected solution approaches. We propose a method to compare Java classes using variable usage paths (VUPs) extracted from modified abstract syntax trees (ASTs). The proposed approach involves matching semantically equivalent functions and attributes between classes by comparing their VUPs. We define a F_1-based similarity measure on how well one student submission matches another. We evaluate our approach using students' submissions from an introductory programming exercise and the results indicate the effectiveness of our method in identifying different solution approaches. The proposed approach outperforms simplified comparisons and the widely used plagiarism detection tool JPlag in accurately grouping submissions by solution approach similarity.

1 Introduction

Lecturers of introductory programming courses often face the challenge of supervising a large number of students. Grading a large number of lab assignments is time-consuming, which is why automated unit tests are frequently used. However, such tests provide only feedback with respect to functionality, not to the efficiency or suitability of the approach. Feedback of this kind can be provided more easily, if code submissions are grouped or clustered according to their (syntactical) similarity, because the same feedback may be used for multiple solutions (see [3] and references therein). However, in this work we are not interested in grading (whereas the authors of [3] are), but feedback that may help students to revise their solution and learn from fellow students. Once an exercise has been solved, a student may benefit from other solutions that follow the same line of thought (but are much more elegant or compact) or alternative solutions that address the problem differently (not just syntactically, but in the underlying idea how the problem has been solved). We thus seek a similarity measure for code (as it is typically written in introductory programming courses) that focusses on

structurally different solutions. Note that solutions, which follow different ideas, typically differ syntactically, but syntactical differences do not automatically indicate a difference in the underlying idea.

2 Related Work

The proposed method for program similarity in [1] is based on the use of Graph Neural Networks (GNNs) to analyse Control Flow Graphs (CFGs) of Java functions. The authors state that they wanted to capture both, syntactical similarity as well as semantic similarity. However, this is achieved by estimating the semantics via a syntactic approximation. If the same program constructs are nested in a similar way, this has a positive impact on the semantic similarity. But to our experience this does not necessarily hold for code from novice programmers (cf. next section). Furthermore, single Java functions were used instead of full Java classes to reduce the runtime of the Graph Edit Distance problem, as this significantly reduces the number of nodes in each graph. However, it remains unclear how the approach can generalise to full Java classes with multiple functions.

Some approaches for detecting similar Java classes, like for code clone detection, also use abstract syntax trees (e.g. [6]). In conventional ASTs, a new node is created for each new variable access (even if an already referenced variable is accessed again). Thus, any information about the use of variables and thus about the data flow through the programme is discarded. This is also the problem with plagiarism detection tools like JPlag [5]. Here, all variable identifiers are replaced with generic identifiers, as the naming may not have any influence on plagiarism detection. However, for the detection of solution approaches in student solutions, it is important to obtain information about the data flow.

Many approaches utilize neural networks, which are trained using a large number of open source projects. In [7] individual statements (AST subtrees) are represented as embeddings (e.g. local variable declaration) and a sequence of embeddings is encoded using recurrent networks. However, similar to the clone detection, the sequential representation includes variable declaration statements, but actual dependencies between variables are not captured (that is, which variable is used where). This may not be a problem for well-written code from official repositories, because a tree of statements may suggest a meaningful variable usage itself. It is, however, a problem when source code from novice programmers is used, who struggle not only with the concepts of the language, but also with computational thinking in general. In that case, the wrong variables are used in the wrong places, turning a potentially useful code skeleton into a mess. The fact that conditional statements and loops are composed in a similar manner alone does not mean that the code does something similar.

So for educational purposes (in introductory courses) we consider it as a major problem with other approaches, that they disregard the way in which variables are used. We argue that two submissions follow the same solution approach if they use variables in the same way. This work extends earlier work [2] by allowing a full comparison of Java classes (multiple functions and distinc-

```
class Range {

  int getRange(int[] arr) {
    int min = 1000;
    int max = -1000;
    for (int i=0;i<arr.length;++i)
      if (arr[i]<min) min=arr[i]; else
        if (arr[i]>max) max=arr[i];
    return max-min;
  }
}
```

```
class Range {
  int mn = 1000;
  int mx = -1000;

  int getRanges(int[] arr) {
    int min = arr[0], max = arr[0];
    for (int i=1;i<arr.length;++i) {
      if (arr[i]<min) min=arr[i];
      if (arr[i]>max) max=arr[i];
    }
    return max-min;
} }
```

```
class Range {
  int small = 1000;
  int large = -1000;

  void include(int a) {
    if (a<small) small=a; else
      if (a>large) large=a;
  }
  int range(int[] arr) {
    for (int i=0;i<arr.length;++i)
      include(arr[i]);
    return large-small;
} }
```

Fig. 1: Three solutions to the same exercise.

tion between local variables and attributes) and also addresses the problem of different functional decompositions.

3 Problem Definition

The goal is to detect different approaches to the same problem or exercise. While for an experienced programmer it may look like a task calls for a canonical, straightforward solution, students may find quite different approaches even to standard problems, as they have not yet developed a notion for *standard cases* or struggle with getting the standard approach straight. Figure 1 shows three (potential) solutions to an exercise which requests a function that returns the value range of the passed array (which carries elements of up to three digits only). With respect to the chosen solution approach, all three submissions follow the same idea and are considered similar (in contrast to the examples of Fig. 5, which are syntactically similar to the examples in Fig. 1 but solve a different problem). Keeping the code structure but mixing the variables, however, may easily destroy the functionality completely. A useful notion of similarity has to take into account which variable is used where. But still the similarity is not recognized easily, we are faced with the following challenges: The students have chosen different function names. Some use attributes, others only local variables. There are artefacts that indicate a change in the strategy (the solution in the middle declares some attributes but then decides not to use them). The solution on the right decomposes the problem and uses a second function to solve the task in exactly the same fashion as the solution on the left, but with a more compact code. Syntactically both solutions (left and right) are different, semantically they are identical. Ideally we would like to group code by ideas, but content ourselves with ways to group code by similar variable usage.

4 Proposed Approach

The problem of comparing two classes, say `C` and `D`, is decomposed into two steps, matching functions and then attributes. The use of variables, that is, how

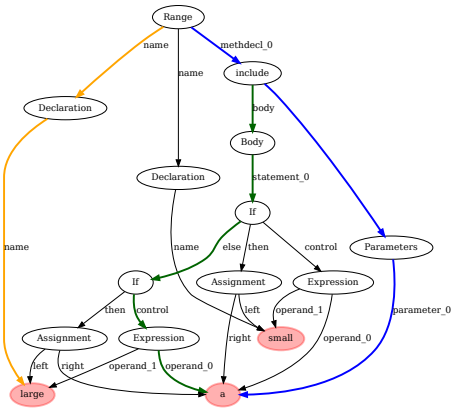


Fig. 2: Modified AST of class `Range` (in Fig. 1(right), only method `include`) where all nodes, that correspond to the same variable (filled red), are unified.

they are embedded into the control structures, captures the nature of the underlying functionality well and shall form the basis of a similiarity measure. We characterize a variable's usage by inspecting a modified abstract syntax tree (AST), where tree nodes that refer to the same variable are unified into a single node (which turns the AST into a graph). Figure 2 shows an example for the class `Range` (example on the right in Fig. 1, limited to the `include` function). The three variables (argument `a` and attributes `small` and `large`) correspond to nodes shaded red. Whenever a variable is used, there is an incoming edge to these nodes. A variable's usage can thus be described by the set of all paths from the root node (here: Range) to the variable node. (The root node is omitted from the path as it is the same for all paths.) For instance, the blue path `a/parameters/include` tells us that `a` is a parameter of function `include` of class `Range`. The green path `a/expression/if/if/body/include` informs us, that `a` is used in the control expression of a nested `if`-statement inside the `include` function. The orange path `large/Declaration/`$\perp$ misses a function node, because `large` is an attribute that is defined in the scope of the class rather than the scope of a function. We may thus describe a whole class by the set of all *variable usage paths* as follows:

Definition 1 (variable usage path). *Let I be a set of code instruction labels (such as* `if`*,* `while`*,* `expression`*, ...). For a given Java class C, let V_C be a set of variable identifiers*[1] *and F_C the set of functions declared in C. A* **path** $p = (v, i_1, \ldots, i_n, f) \in V_C \times I^* \times (F_C \cup \{\perp\}) := \mathcal{P}$ *reflects that a variable v is*

[1] Note that variable names themselves are not valid identifiers, as the same name may be used twice for variables in different functions. From the AST, we extract a unique node identifier for each variable. But in this paper, for simplification and readability, we use the variable names as identifiers.

used by instruction i_1, which is itself used by instruction i_2, etc., in function f (or, indicated by $\perp$, directly in the class definition in case of attributes). The **VUP-representation** *(variable usage path) of code C is a set $P_C \subseteq \mathcal{P}$ of all paths occurring in C.*

Classes `C` and `D` may then be compared by comparing the respective VUP representations P_C and P_D:

Definition 2 (VUP similarity). *Given two VUP representations P and Q, we measure their similarity by the F_1-measure:*

$$F_1(P,Q) = 2 \cdot \frac{p \cdot r}{p + r} \quad \textit{where} \quad p = \frac{|P \cap Q|}{|P|},\ r = \frac{|P \cap Q|}{|Q|}$$

If, however, function and variable names are disjoint in `C` and `D` (as it is the case in Fig. 1), $F_1(P,Q)$ is undefined, so in this case, we assume $F_1(P,Q) = 0$. We may replace all variable names and all function names by the same identifier, as it is frequently done by other approaches (e.g. plagiarism detection).

Definition 3 (simplified usage path). *Given a class C, let $\sigma : V_C \times I^* \times (F_C \cup \{\perp\}) \to \{\texttt{vn}\} \times I^* \times \{\texttt{fn}, \perp\}$ a transformation, that replaces all variables names by a constant identifier* `vn` *(generic variable name) and all function names f by*

$$f' = \begin{cases} \texttt{fn} \textit{ if } f \neq \perp \\ \perp \quad \textit{otherwise} \end{cases}$$

where `fn` *is a constant identifier. For a VUP P, we denote $\sigma(P)$ as a simplified VUP representation (SVUP) . By σ_f (σ_v, resp.) we denote the same transformation as σ except that it leaves the variable names (function names, resp.) unaltered.*

The discussed paths would thus read in the simplified representation `vn/Declaration/`$\perp$, `vn/parameters/fn`, and `vn/expression/if/if/body/fn`, resp. Comparing simplified representations $\sigma(P_C)$ usually yields a non-zero value for $F_1(\sigma(P_C), \sigma(P_D))$, but does not distinguish different function and variables at all, and thus gives only a rough similarity estimate.

By visual inspection of the examples in Fig. 1 we *know* that function `getRange` (left) and `range` (right) correspond to each other, but this information is not available to our similarity measure. The simple approach of averaging all possible pairwise similarities, such as comparing `getRange` with `include` and `getRange` with `range` is not a reasonable solution. To accurately assess similarity as realistic as possible, only the corresponding functions (and variables) should be considered (like `getRange` vs `range`). To solve this problem, we need some means to filter for usage paths of certain variables or functions:

Definition 4 (filter π on usage paths). *Let $\pi_{(x,y)} : \mathcal{P} \to \mathcal{P}, P \mapsto \{p \,|\, p = (v, i_1, \ldots, i_n, f) \in P \wedge (x = v \vee x = *) \wedge (y = f \vee y = *)\}$ be a filtering function that returns a VUP representation of only those paths that refer to variable identifier x (or all variables if $x = *$) and function identifier y (or all functions if $y = *$).*

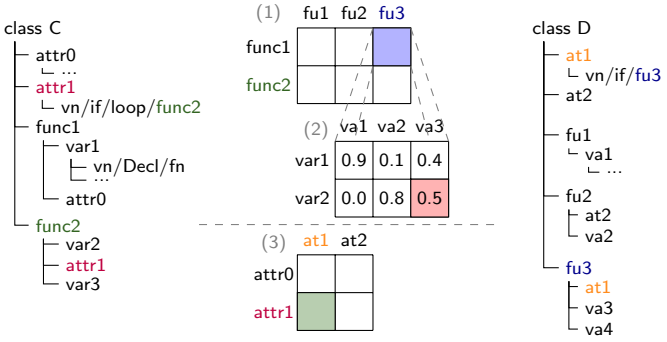


Fig. 3: Schema of class comparison.

The challenge is now to match functions and attributes from different classes dynamically. Figure 3 shows the proposed approach schematically. On the left, class C is decomposed into attributes (here: `attr0` and `attr1`) and functions (here: `func1` and `func2`). Function nodes f have a child node for every variable that is used in the function. The children of some variable v correspond to a simplified VUP representation of all related usage paths of function f, that is, $\sigma(\pi_{(v,f)}(P_C))$. The children of some attribute a correspond to $\sigma_v(\pi_{(a,*)}(P_C))$.

In order to find the best counterparts of functions in `C` among functions in `D`, a cost matrix is created (matrix (1) in Fig. 3). To fill the cost matrix we need to assess the similarity between, say, function `func1` of `C` and `fu3` of `D`, which shall be accomplished by comparing how similar both functions are in terms of their variable usage. To assess this particular value, another cost matrix is constructed (matrix (2) in Fig. 3), which compares all variables of `func1` against all variables of `fu3`. A single cell in this matrix captures the *cost* of assigning a variable from one class to a variable of the other. The cost value is obtained from calculating F_1 of the respective SVUPs ($1 - F_1$ enters the cell, turning similarity F_1 into cost/distance). For the marked cell (`var2` vs `va3`) we obtain its value from

$$1 - F_1 \left(\sigma(\pi_{(\texttt{var2},\texttt{func1})}(P_C)) \, , \, \sigma(\pi_{(\texttt{va3},\texttt{fu3})}(P_D)) \right)$$

The optimal match of variables is obtained, when the total assignment cost become minimal. This problem is known as an *assignment problem*, which is not trivial. The challenge is to efficiently determine the optimal assignment, as the number of possible combinations grows quickly with the size of the cost matrix. The Munkres algorithm [4] provides a solution by analysing the cost matrix and determining an optimal allocation with minimum total cost in cubic runtime complexity. In cost matrix (2) of Fig. 3 the optimal assignment would be $A_v = \{(\texttt{var1}, \texttt{va2}), (\texttt{var2}, \texttt{va1})\}$ with a minimal total cost of 0.1.

To reflect the optimal variable assignment for `func1` and `fu3` (as represented by some assignment A), the variable names can be renamed in all paths of P_C and P_D:

Definition 5 (renaming ϱ of usage paths). *Let $A \subset V_C \times V_D$ be the result of an assignment problem for classes C and D, that is, identifiers i_1 and i_2 have been assigned if and only if $(i_1, i_2) \in A$. Let $\varrho_A : \mathcal{P} \to \mathcal{P}$ be a renaming function, which replaces all occurrences of any identifier $(i_1, i_2) \in A$ in all variable usage paths by some new identifier $h(i_1, i_2)$ (e.g. obtained from a hash function h or any other unique, artificially generated name).*

Once the variables are renamed, we can fill in the cost value of the resp. cell in cost matrix (1), which is now obtained from

$$1 - F_1 \left(\sigma_f(\varrho_{A_v}(\pi_{(*,\texttt{func1})}(P_C))) \,,\, \sigma_f(\varrho_{A_v}(\pi_{(*,\texttt{fu3})}(P_D))) \right)$$

Having filled out all cells in this way, we obtain the cost-minimal match of functions from another application of the Munkres algorithm [4].

Once all functions have been assigned to one another and all identifiers have been replaced in the usage paths accordingly, the attributes can be assigned in the same fashion by comparing their VUP representations (cost matrix (3) in Fig. 3). With $A_f = \{(\texttt{func1}, \texttt{fu3}), (\texttt{func2}, \texttt{fu2})\}$ being the union of all selected function assignments, the cost of the highlighted cell in cost matrix (3) is obtained from

$$1 - F_1 \left(\sigma_v(\varrho_{A_f}(\pi_{(\texttt{attr0},*)}(P_C))) \,,\, \sigma_v(\varrho_{A_f}(\pi_{(\texttt{at1},*)}(P_D))) \right)$$

At this point, all functions and variables (local variables as well as attributes) from class C have been assigned to the respective elements of class D, preserved in the assignments A_f and A_v.[2] The final similarity is then computed as a mixture of function and attribute similarity (we use $\alpha = 0.7$):

$$\alpha \underbrace{F_1(\varrho_{A_f}(P_C), \varrho_{A_f}(P_D))}_{\text{functions}} + (1-\alpha) \underbrace{F_1(\varrho_{A_v}(P_C), \varrho_{A_v}(P_D))}_{\text{attributes}}$$

Functional decomposition. So far, we have not tackled the difference between the code on the left and on the right of Fig. 1, which differs in the functional decomposition. The usage paths will of course differ: `range` (Fig. 1(right)) uses the argument `arr[i]` in the function call `include(arr[i])`, whereas `getRange` (Fig. 1(left)) uses `arr[i]` multiple times in the `if`-statements. To overcome these differences, we modify the way how paths are extracted from the AST in case of function calls. Figure 4 shows the graph for the `range` method, which calls the `include`-method (`call` node with edge to the `include` method). We would extract a path `arr/arrayaccess/call/for/body/range` for the array `arr`, which indicates that it occurs as a parameter in a function call. Whenever a path contains a `call` segment, we modify the path as follows: we replace the single `call` segment with all paths of the respective parameter from the called function, where (1) the argument is removed and (2) the path is cut off at the body of the function. In the case of the `include` function in Fig. 2 we have already discussed the path `a/expression/if/if/body/include`. We remove the parameter (`a`)

[2] or remain unassigned in case no counterpart exists

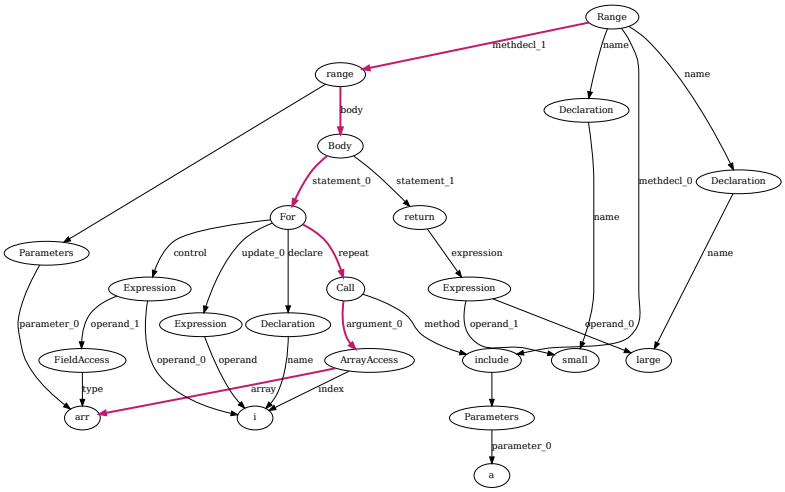


Fig. 4: Modified AST of class `Range` (in Fig. 1(right)), without implementation of `include`.

and cut off the path at the body of the function and obtain `expression/if/if`. That is, in the original path of the `include(arr[i])`-call we replace `call` by `expression/if/if` (same for all other paths that include this parameter). This leads us to `arr/arrayaccess/expression/if/if/body/range`, which is the same path we would have obtained if the function `include` was inlined in the `range` function. In this way we generate the same paths that are extracted from Fig. 1(left). Finally, minor transformations are applied to the extracted path sets to purify them, such as filtering out artefacts or nested *bodies* (such that `if(..){{stmt;}}` equals `if(..){stmt;}`).

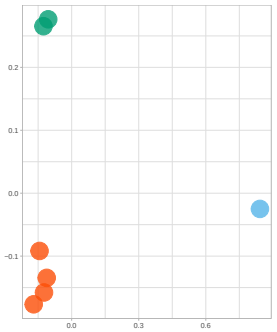
5 Experimental Evaluation

We first discuss the results for the example classes from Fig. 1. As a distractor we add two more source codes to them, shown in Fig. 5, which are similar in terms of statement nesting, but dissimilar to the others in terms of variable usage. The following sources were compared pairwise: *oddeven* and *oddevenfunc* from Fig. 5, the *left*, *mid*, and *right* code from Fig. 1, an *empty* class and a clone *mid2loop* of *mid* where each of the two conditional statements gets its own for-loop. The result is a 7×7 distance matrix. One cell in this distance matrix then corresponds to the total distance between two classes `C` and `D`. Multidimensional scaling (MDS) was applied to the distance matrix ($R^2 \approx 0.90$), which projects the individual data points in the two-dimensional space, while trying to preserve the pairwise distances. For the final evaluation of the quality of the realised class comparison, hierarchical cluster analysis with the single-linkage method

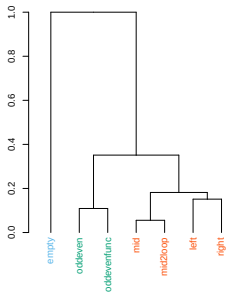
```
class Range { // oddeven

  int cnt1 = 0;
  int cnt2 = 0;

  int oddeven(int[] a) {
    for (int i=0;i<a.length;++i) {
      if (a[i]%2==0) ++cnt1;
      else ++cnt2;
    }
    return cnt2−cnt1;
  }
}
```

```
class Range { // oddevenfunc
  int cnt1 = 0;
  int cnt2 = 0;

  void f(int a) {
    if (a%2==0) ++cnt1;
    else ++cnt2;
  }

  int oddeven(int[] a) {
    for (int i=0;i<a.length;++i)
      f(a[i]);
    return cnt2−cnt1;
} }
```

Fig. 5: Two sources with similar code structure (conditional statement within loop over array) but different variable usage.



(a) MDS of example codes

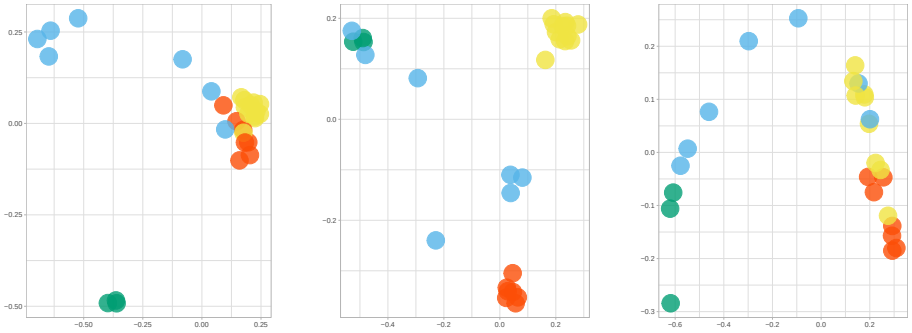


(b) Dendrogram of ex. codes

Fig. 6: Evaluation of example codes

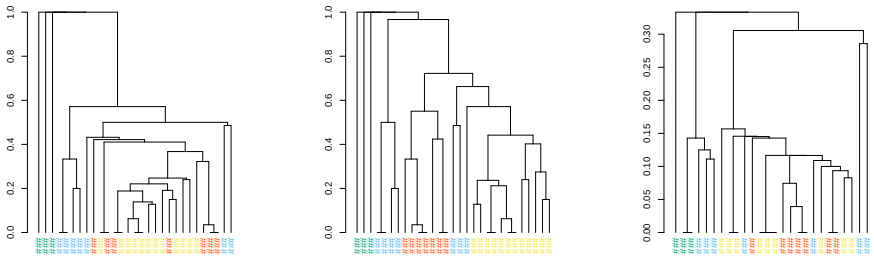
was applied to the distance matrices. Figure 6 shows the results. On the left we see the MDS embedding into 2D. The classes that use variables in different ways are optimally separated from each other. This is also reflected in the dendrogram. We can see that the submissions that use variables in a similar way and follow a similar approach are close to each other, as desired (e.g. mid and mid2loop, which barely differ in terms of variable usage).

For a more comprehensive evaluation of the proposed approach, we analysed real students' submissions resulting from different exercises. The evaluation results of one of these exercises are presented below. In the task TimeKeeper (TK), a working time administration had to be implemented. A complete realisation typically contains four methods, a constructor and about four attributes. Two predominant approaches can be found in the submissions, which perform the necessary calculations in a different way (re-calc all values with every change versus calculation of single values upon request). First, the submissions were grouped



(a) simplified similarity (b) proposed similarity (c) JPlag similiarity

Fig. 7: MDS 2-dimensional projection, C_{green} = empty classes, C_{blue} = partially implemented classes, C_{yellow} and C_{red} = two predominant solution approaches



(a) simplified similarity (b) proposed similarity (c) JPlag similarity

Fig. 8: Dendrograms of hierarchical clustering

manually, four clusters were identified. C_{green} contains completely empty classes without implementation, C_{blue} contains an implementation that has been started but for which no specific solution approach is recognisable yet (only some variable and method declarations but only little functionality), and C_{yellow} and C_{red} contain the two predominant solution approaches. We compare the results of the proposed approach with the simplified representation $(F_1(\sigma(P_C), \sigma(P_D)))$ and the JPlag [5] tool to derive an additional distance matrix. JPlag is a frequently used and well established tool for source code comparison, most commonly used to detect plagiarisms. Figure 7 shows the results of MDS and Fig. 8 presents the resulting dendrograms. Note that some jitter was applied to the 2D projections to make data points visible that lie very close to each other.

Figure 7a shows the 2D representation of the submissions, which were analysed with the simplified variant. The submissions belonging to C_{green}, representing empty classes, are perfectly separated from the remaining submissions. On

the other hand, the submissions in C_{blue}, which represent partially implemented classes, are partially separated from the other groups. Some of these submissions form a distinct cluster in the top-left region. Among these submissions, those that include more code and are gradually evolving towards one of the two solution approaches are more similar to the fully implemented classes. Consequently, they tend to be positioned in close proximity to the fully implemented classes. The fully implemented classes are concentrated in a single cluster. Within this cluster, however, the upper region consists mainly of submissions that follow the yellow approach, while the lower half comprises mostly submissions that follow the red approach. Thus, an approximate separation is already noticeable here using the simplified comparison.

Figure 7c shows the results of the pairwise analysis with the JPlag tool. The empty classes from C_{green} are separated from the other classes, but not as distinctly as in Fig. 7a. This can be explained by the fact that we classified a class as "empty" when no functionality was implemented at all. However, a class declaration or the declaration of the main function may still be present in the file. Thus, the submissions differ in the syntactic comparison with JPlag, but not in their variable usage paths. On the right side of the figure, the fully implemented classes as well as isolated half-finished classes occur in one group. There is a slight tendency to separate the approaches in C_{red} and C_{yellow}, but the approaches are mixed and clusters overlap. Submissions of C_{blue} are widely distributed between the empty and fully implemented classes, two almost reaching C_{yellow}.

Figure 7b analogously shows the results of the proposed similarity presented in the context of this paper. At first glance, four groups and two outliers can be identified. In the upper left there are the empty submissions from C_{green}. Within this group there are two submissions from C_{blue}. This is due to the fact that these two solutions contain only attribute declarations and implement no functionality. Since the declared attributes are never referenced in functions, they are identified as artefacts and removed before class comparison, which results in empty VUP-representations as well. In the middle there is a group of partially implemented submissions (C_{blue}), which have reached a similar stage of completion. Two other partial solutions represent intermediate versions, which are thus located somewhere in between. However, the different approaches have been separated perfectly and form two distinct clusters.

The 2D representation resulting from MDS provides an intuitive way to get an overview of the structure in the data, clusters can be identified with a single glance. However, structures may be present in the data that are not recognisable in 2D space. A coefficient of determination of $R^2 \approx 0.70$ is solid, but not optimal. The dendrograms from a hierarchical cluster analysis are shown in Fig. 8. Again, the proposed approach separates the different approaches (C_{yellow} and C_{red}) best. The dendrograms also show that the proposed comparison is the only one in which the two solutions C_{yellow} and C_{red} can be perfectly separated by a cutplane ($\delta \approx 0.6$). For the other two dendrograms no such separation is feasible.

In summary, the results suggest that the composed comparison is a better discriminating method than the simple one and the JPlag Tool. The sample

is small, but it becomes clear that on the test data, with the help of the chosen implementation, manually distinguishable solutions are distinguishable by a machine as well.

6 Conclusion

In conclusion, this paper presents an automated method for improving computer science education by analysing code submitted by students. The proposed approach compares Java classes in a semantic way using variable usage paths (VUPs) extracted from modified abstract syntax trees (ASTs). The method presented in this paper outperforms simplified comparisons and the JPlag tool in accurately grouping submissions by similarity of solution approaches. The evaluation results demonstrate the effectiveness of the approach in identifying different approaches. Through a preliminary evaluation using real student code submissions, it was shown that this method provides valuable insights for students and teachers. Lecturers can get an overview of the prevailing solution approaches in the submissions and specifically discuss individual ones in the lecture. We intend to offer students, who have finished their exercise, a solution from a fellow student that follows a different approach. We then ask the students for pros and cons of both solutions, which encourages students to overthink their own solution as well as comprehending given code (and will also be used to assess the validity of the method on a larger scale). Both applications can have an immediate positive impact on the students' learning success.

Acknowledgements This work was partly funded by the German Federal Ministry of Education and Research under the grant no. 16DHBKI056 (ki4all).

References

1. Hellendoorn, V.J., Devanbu, P.: Are deep neural networks the best choice for modeling source code? In: Proc. 11th Joint Meeting on Found. of Software Eng. pp. 763–773 (2017)
2. Höppner, F.: Grouping source code by solution approaches - improving feedback in programming courses. In: Proc. Int. Conf. on Educational Data Mining. Int. Educational Data Mining Society (2021)
3. Joyner, D., Arrison, R., Ruksana, M., Salguero, E., Wang, Z., Wellington, B., Yin, K.: From clusters to content: Using code clustering for course improvement. In: Proc. Tech. Symp. on Computer Science Education. pp. 780–786 (2019)
4. Munkres, M.: Algorithms for the Assignment and Transportation Problems. Journal of the Society of Industrial and Applied Mathematics **5**(1), 32–38 (1957)
5. Prechelt, L., Malpohl, G., Phlippsen, M.: Jplag: Finding plagiarisms among a set of programs. Tech. rep., University of Karlsruhe (2000)
6. Sager, T., Bernstein, A., Pinzger, M., Kiefer, C.: Detecting similar java classes using tree algorithms. In: Int. Workshop on Mining Software Repositories. p. 65–71 (2006)
7. Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X.: A novel neural source code representation based on abstract syntax tree. In: Int. Conf. on Software Eng. pp. 783–794. IEEE (2019)