Grouping Source Code by Solution Approaches — Improving Feedback in Programming Courses

Frank Höppner
Ostfalia University of Applied Sciences
f.hoeppner@ostfalia.de

ABSTRACT

Various similarity measures for source code have been proposed, many rely on edit- or tree-distance. To support a lecturer in quickly assessing live or online exercises with respect to *approaches taken by the students*, we compare source code on a more abstract, semantic level. Even if novice student's solutions follow the same idea, their code length may vary considerably – which greatly misleads edit and tree distance approaches. We propose an alternative similarity measure based on *variable usage paths* (VUP), that is, we use the way how variables are used in the code to elaborate code similarity. The final stage of the measure involves a matching of variables in functions based on how the variable is used by the instructions. A preliminary evaluation on real data is presented.

Keywords

source code distance, semantic analysis, variable usage paths

1. INTRODUCTION

Learning a programming language requires a lot of practice. Students gain knowledge and experience from both, homework and in-classroom exercises. It is, however, very important to give students feedback, e.g. discuss different solution approaches with them. Both, those who have and have not yet succeeded, will benefit from such discussions, either it widens their view (because they may not have thought about alternative approaches yet) or encourages them to try the exercise at a later point in time once more (once they got a glimpse on how to solve it). The sharing of different solution approaches and discussing the pros and cons of different approaches improves their algorithmic thinking skills. However, it is quite common in many programming courses, that the only (automatic) feedback consists of the number of passed or failed unit tests. Achieving a positive feedback then requires an already well developed solution and no credit is given for, say, getting the code structure right – which may frustrate novices noticeably.

We thus address the following research question: Can we support the lecturer in providing meaningful feedback about the solution approaches taken by the students? This requires automation as a lecturer does not have the time to review all solutions manually, especially not in online teaching situation. Providing meaningful feedback requires some kind of insight in the solution approach a particular student was following (semantics), how common the approach is, how many different approaches have been followed in the course, etc. We intend to achieve this by providing a similarity measure for source code that does not focus on results (as unit tests do) but to reach a higher semantic level by assessing the more abstract code structure: different solution approaches manifest themselves in different code structures.

Such a measure would be useful in many ways. For instance, during an in-classroom teaching situation it would enable the lecturer to pick two (or more) submissions following different approaches to start a discussion about their pros and cons. It could also support a lecturer in browsing the spectrum of approaches that were followed by the course members (how many different approaches were chosen how often) without having to check every solution manually. It may enable the lecturer to pick a solution that follows the same approach as a solution that has been discussed already, but did not pass all unit tests, thereby representing a live challenge to the course members ("spot the error"). Note that we are not aiming at grading the source with respect to its correctness, but leave this to the unit tests. As tests do not provide any useful feedback to students that do not yet have an appropriate code structure, the desired measure may close this gap, as it would allow us to differentiate code submissions that are far from working from those that follow a reasonable solution approach and got the code structure right, but only the tiny details prevents them from passing the tests. Such a more detailed inspection would enable a much more sensitive (automatic) feedback.

2. RELATED WORK

Similarity or distance measures for source code have been investigated for a long time. Many approaches have in common that they start from an abstract syntax tree (AST) that represents the code. Code is then compared by calculating some kind of *tree distance* on the corresponding ASTs: the minimal number of steps (node deletion, insertion, and relabelling) to transform one AST to the other. A survey on tree edit distance can be found in [2]. A tree, however, has no conscience about *variable identity*: the very same vari-

able occurs over and over again as a new node in an AST. To reflect which variable change affects other code, program dependency graphs (PDG) may be used. But comparing graphs is even more complicated than comparing trees, a survey on graph edit distance is given in [6]. To simplify the comparison, the AST may also be linearized such that much simpler string comparison measure (edit distance) may be applied (cf. [5, 7, 9]).

An extremly fast approach is to define a hash function on (possibly pre-processed trees) such that a similarity is detected when hash codes are identical [1]. The flexibility of such an approach is, however, very limited, as the similiarity measure is a binary one. It may nevertheless be useful for clone detection (plagiarism) or may be enhanced by calculating multiple hashes [4]. Tree edit distance has been used in, e.g., [10] to detect similar source codes. In [3] a distance measure for source code has been proposed, that transforms the AST into a sequence of tokens on which Levenshtein or edit distance is applied.

In many related papers the goal is to compare (student's) code to a (lecturer's) reference code; the higher the similarity, the closer is the student's submission to the correct solution. In this work, however, we want to identify source codes that are semantically similar, that is, they follow the same solution approach. Especially when novices start to code, their solutions are often more lengthy and redundant than those of an experienced programmer. This affects measures such as edit distance or tree distance dramatically. The approach in [3] did not work out as well as expected, and the authors hypothesized about one of the reasons that the length of the code dramatically influences the edit distance. While longer code may be less readable and more redundant, it is neither more likely wrong nor does it necessarily follow a different solution approach. As an example, consider codes (a) and (b) from Fig. 1. The task was to calculate the mean of all positive array elements. Both codes follow the same approach, but (b) uses a single loop instead of two and is thus more compact. But semantically, we consider both solutions as being identical. We are not aware of any source code similarity measure that tries to reflect that.

It is also common in the literature to replace variable names by a constant string (possibly depending on the variable's type), which eases matching sources that use different variable names. In plagiarism detection (see [5, 9] and references therein) this is considered as a countermeasure against disguising plagiarism by variable renaming. However, the way a single variable is used throughout the code is crucial for a solution approach. Fig. 1(d) utilizes the same statements (e.g., array-access in conditional statement of a for-loop), but does not calculate anything meaningful and thus does not represent the same solution approach as codes (a-c).

3. COMPARING SOURCE CODE BY VARIABLE USAGE PATHS

We argue that two solutions follow the same approach if they use variables in the same way. In the subsequent sections we show how we grasp variable usage in the code and then define similarity measures on this representation.

```
public double avg(double a[]) {
  if (a==null) return NaN;
  int n = 0;
  for (int j=0;j<a.length;++j) {
    if (a[j]>0) ++n;
  }
  double sum = 0;
  for (int j=0;j<a.length;++j) {
    if (a[j]>0) sum+=a[j];
  }
  return sum/n;
} // (a) [TwoLoopQuickExit]
```

```
public double avg(double a[]) {
  if (a==null) return NaN;
  int n = 0;
  double sum = 0;
  for (int i=0;i<a.length;++i) {
    if (a[i]>0) {
      ++n;
      sum+=a[i];
    }
  }
  return sum/n;
} // (b) [OneLoopQuickExit]
```

```
public double avg(double a[]) {
 if (a!=null) {
   int n = 0;
   double sum = 0;
   for (int i=0;i<a.length;++i) {
     if (a[i]>0) {
       ++n;
       sum+=a[i];
   } }
   return sum/n;
 } else {
   return NaN;
} } // (c) [OneLoopGuardedIf]
```

```
public double avg(double a[]) {
  if (a==null)
    return NaN;
  int n = 0;
  double sum = 0;
  for (int i=0;i<a.length;++n) {
    if (a[i]>0) {
      ++i;
      a[i]+=sum;
    }
  }
  return sum;
 } // (d) [Disordered]
```

Figure 1: Responses to a simple programming exercise: Write a function avg that yields the mean of all positive values in the array. Left: Code (a) uses two loops, while code (b) only one. While the main code is organized *after* an if-statement in (a) and (b), it is embedded in the conditional statement in (c). Code (d) uses similar instructions, but the variable usage is screwed up and does not solve the problem.

3.1 Variable Usage Paths

Solving a programming exercise requires to combine programming instructions such that a handful of variables jointly build up the final result (and return it to the caller). The key to a solution are thus the variables and how they are embedded in the (possibly nested) instructions. Code analysis often starts with an abstract syntax tree (AST); there, every variable usage is represented by a new node in the tree. This occludes important information: Where in the source code is the same variable used? We therefore rearrange the AST to a graph, where a node is unique for each variable and subsequent usages of the variable link to the same node. Fig. 2 shows an example for the code of Fig. 1(b). All variables are marked in red color. From the paths between the variable `sum` and function `avg`, we can see that the variable `sum` is declared, occurs in a conditional statement that is embedded in a loop, and finally occurs in the return value. We consider these paths (shown in blue in Fig. 2) as a kind of *fingerprint* for the role of this variable: code following the same approach requires variables with the same roles.

We thus do not operate directly on the graph, but paths from variable nodes to the enclosing function node. After applying some transformations to simplify the graph somewhat (e.g. removing *body* or replacing *for*, *while*, etc. by a subsuming label *loop*), we end up with string representations of the three blue paths like *sum/expression/return/avg*, *sum/dec-*

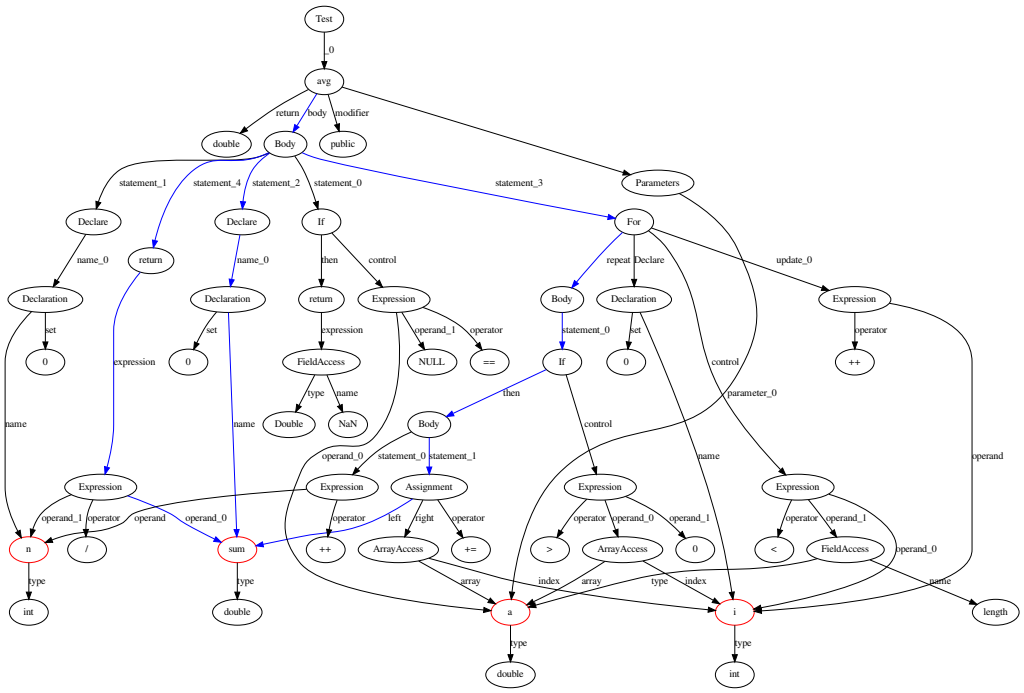

Figure 2: Graph representation of code in Fig. 1(b) (red: variable nodes; blue: VUPs of variable 'sum').

laration/declare/avg and *sum/assignment/if/loop/avg*.

Definition 1. Let I be a set of code instruction labels (such as *if*, *loop*, ...). For a given source code C (single class), let V_C be a set of variable-identifiers[1] and F_C the set of methods declared in C. A **path** $p = (v, i_1, \ldots, i_n, f) \in V_C \times I^* \times F_C := \mathcal{P}$ reflects that a variable v is used by instruction i_1, which is itself used by instruction i_2, etc., in function f. The **VUP-representation** (variable usage path) of code C is a set $P_C \subseteq \mathcal{P}$ of all paths occurring in C.

We thus intentionally drop many details from the original source code, e.g. numerical constants or full expressions. From the fact that two codes that are structurally identical (same VUP representation) we cannot conclude anything about the number of unit tests both codes may pass. But as we have mentioned earlier, we seek for a common code skeleton, which may indicate that the programmers were guided by the same underlying idea. The skeleton includes the information which variables need to be used in which instructions — but there are many different ways of coding expressions equivalently, so we simply stick with unit tests to check their correctness.

3.2 Simplified Set Similarity

Given two source codes (a) and (b) from Fig. 1 and the corresponding VUP-representations P, Q. Code (a) determines the number of elements first before it sums the relevant values, while code (b) does both in a single loop. While the choice of one or two loops may affect the efficiency, they are semantically equivalent and thus *similar*. At first glance the variable usage paths (of, say, the loop counter) in all loops seem identical such that a set comparison (where duplicate elements are not counted) appears just right. However, the loop counter is named `j` in code (a) and `i` in code (b). Furthermore, (a) defines two variables with the same name `j` (one for each loop). Let us ignore these details by introducing a simplification (and revisit the problem later):

Definition 2. From a VUP representation P we obtain a **SVUP representation** (simplified VUP) representation P' by replacing all variable identifiers and all method identifiers by a constant identifier `vn` (generic variable name) and `fn` (generic function name), resp. ($I = \{vn\}$, $F = \{fn\}$).

Standard methods to measure set similarity may then be used to compare source code. For given SVUPs P and Q we use the F_1 measure[2]:

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

Fig. 3 shows a dendrogram for an average-linkage clustering using this measure (actually $1 - F_1$ to obtain a distance measure). Codes (a)-(d) correspond to TwoLoopQuickExit, OneLoopQuickExit, OneLoopGuardedIf and Disordered. Additional examples include: Nonsense (similar set of instruction

[1]Note the variable names themselves are not valid identifiers, as the same variable name may occur more than once in the same function, e.g. variable j in code (a) of Fig. 1.

[2]although based on asymmetric precision p and recall r, F_1 itself is symmetric and thus a similarity measure

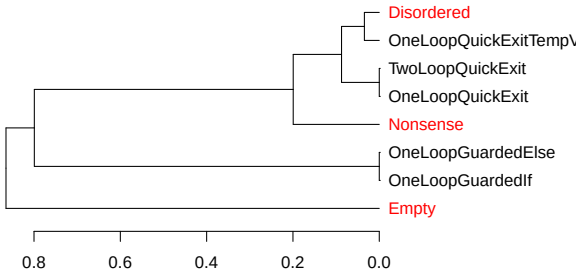

Figure 3: Dendrogram for source codes of Fig. 1 based on F_1-measure on VUP representation.

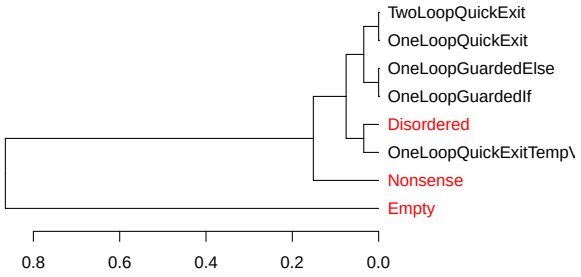

Figure 4: Dendrogram for source codes of Fig. 1 based on F_1-measure on SVUP representation.

Table 1: Example of comparing P and Q (see text).

			d	m(d)
P	1	vn/expr/if/loop/fn	if	(1,6), (2,7)
	2	vn/assign/if/loop/fn	assign	(3,8)
	3	vn/declare/fn	declare	(4,8)
	4	vn/assign/fn		
	5	vn/return/fn		
Q	6	vn/expr/loop/fn	step	un- / assigned
	7	vn/assign/loop/fn	0	1,2,3,4,6,7,8 / 5,9
	8	vn/assign/declare/fn	1	3,4,8 / 1,2,5,6,7,9
	9	vn/return/fn	2	4 / 1,2,3,5,6,7,8,9

set but variables are mixed randomly), OneLoopQuickExitTempVar (the mean value is assigned to a temporal variable before it is returned) and OneLoopGuardedElse (same as OneLoopGuardedIf (Fig. 1(c)) but inverted if-condition). From the dendrogram we learn that codes TwoLoopQuickExit and OneLoopQuickExit become identical as desired. But we can also see that code OneLoopGuardedIf, where a conditional statement encloses the main code, is recognized as very dissimilar. The additional *if*-statement occurs in *all* paths and the simple Jaccard similarity finds this code to be completely different. We address this problem next.

3.3 Reflecting Instruction Embedding

A single conditional statement may introduce an new element in many paths turning them all different (as in codes (a),(c) in Fig. 1). For compensation we could measure a partial similarity between paths (e.g., 80% of path p is contained in q), but a *single instruction* (such as the conditional statement in (c)) would then still weaken *all path similarities*. We therefore propose a different approach in the fashion of edit distance, where we pay a constant cost for a missing path element, which may then be used in many paths.

Given two SVUP representations P, Q, we compare all paths $p \in P, q \in Q$ to identify missing path elements $\delta(p,q) \in I^*$. (For example, δ(*vn/loop/if/fn*,*vn/loop/fn*)=*if*). Many different path combinations may lead to the same $\delta(p,q)$, so by $m(d)$ we denote the set of all pairs $(p,q) \in P \times Q$ with $\delta(p,q) = d$. The matching of paths in P to paths in Q is done iteratively: In the first iteration, we match all paths from P and Q that are identical ($\delta(p,q) = ()$). To match the SVUP representations at minimal cost, in subsequent iterations we identify the missing path element d that unifies the largest number of paths (that is, choose $d = \text{argmax}_x|d(x)|$). Before entering the next iteration, all pairs are removed from $m(\cdot)$ that have been assigned already. We reflect the cost of adding a missing path element by adding it as a *virtual path* to the SVUP P or Q (depending on where it was missing).

Table 1 shows a detailed example. The table on the left shows the paths belonging to SWUP of P and Q. All paths have been numbered for easier reference. The initial map $m(\cdot)$ is shown on the top right; for instance, the path element *if* unifies path #1 of P with #6 of Q, as well as #2 of P with #7 of Q. At the bottom right each line corresponds to an iteration of the matching process. In step 0, paths #5 and #9 are already identical. In the second iteration the path element *if* is chosen from map $m(\cdot)$, because it unifies 2 paths (all others only one). We have thus matched 6 paths in total (step 1 of bottom right table), and only #3, #4, #8 remain unassigned (gray). The map $d(\cdot)$ now offers two alternatives (*assign* and *declare*, both $|m(\cdot)| = 1$), we arbitrarily choose *assign* as the second missing path element for the third iteration. This leaves only path #4 unassigned. The choice of $m(declare)$ has covered paths #4 and #8, so we remove all pairs from $m(\cdot)$ containing any of these (already covered) paths. This ends the matching phase (all $|m(\cdot)| = 0$). We had to add the first missing path element *if* to paths in Q; likewise we had to add the second path element *assign* to P to match #3 against #8. As a penalty for the missing path elements we add them to the respective SVUP, that is, P becomes $\{1,2,3,4,5,assign\}$ and $Q = \{6,7,8,9,if\}$. Taking the established identity of paths into account, this gives us an F_1 value of

$$p = \frac{|P \cap Q|}{|P|} = \frac{4}{6},\ r = \frac{|P \cap Q|}{|Q|} = \frac{4}{5},\ \text{so } F_1 = 2 \cdot \frac{16/30}{44/30} = \frac{8}{11}.$$

Fig. 4 shows the resulting dendrogram. Now OneLoopGuardedIf (Fig. 1(c)) became much more similar to One/TwoLoopQuickExit (Fig. 1(a-b)). But we are still dissatisfied with the high similarities towards Disordered: it uses the same set of embedded instructions (causing the high similarity), but the variable usage is mixed up. The instructions may look identical, but the author did not get the role of variables right and that should degrade the similarity.

3.4 Matching Variables

So we finally revisit the simplification of definition 2. We have hypothesized that similar solution approaches use variables at specific places in the code skeleton. Up to now, we have mixed the usage paths of different variables in the SVUP representation. This will be sorted out next.

Definition 3. Let $\pi_{(v,f)} : \mathcal{P} \to \mathcal{P}, P \mapsto \{p \,|\, p = (v, i_1, \ldots, i_n, f) \in P\}$ be a filtering function that returns only those paths that refer to variable identifier v and function identifier f. From a VUP representation P with a set V of variable identifiers and F of function identifiers, we obtain a **GVUP representation** (grouped VUP) $P' = \{\pi_{(v,f)}(P) \,|\, v \in V, f \in F\}$. That is, a GVUP is a set of VUPs, one VUP set for each variable in each function.

For instance, the code in Fig. 1(b) consists of 4 variables in one function, so the GVUP representation is a set of four VUP representations (one set for each variable). Now, as the GVUP representations P and Q of two source codes are sets of sets, how do we generalize the F_1 calculation? Consider the following example, where we use abbreviated instructions $I = \{a, b, c\}$:

$$\begin{aligned} P &= \{\{x/a/c/f, x/b/c/f\}, \{y/a/f, y/c/f\}, \{z/b/f\}\}, \\ Q &= \{\{x/a/f, x/c/f\}, \{y/b/f\}\} \end{aligned}$$

Formally, a direct calculation of $|P \cap Q|$ yields 0 as none of the sets in P is contained in Q. But this does not meet our intention. Note that variable y in P plays the role of x in Q (same for z in P and y in Q). Variables may be renamed without changing the semantics, in fact $|P \cap Q|$ should be 2. The desired semantics is to match variable and function names appropriately, rename them accordingly and calculate the F_1 measure on the obtained $\bigcup_{X \in P} X$ and $\bigcup_{Y \in Q} Y$.

How do we match the sets $X \in P$ against $Y \in Q$? All paths in X or Y refer to the same variable and function name, so we can safely transform the VUP to a SVUP representation and use the measure from section 3.3 to construct a pairwise cost matrix. We employ the Munkres algorithm [8] to find the optimal assignment based on this cost matrix. As the Munkres algorithm performs a 1:1 least-cost assignment, some variables may not get assigned. We match them afterwards in a second pass to the least costly counterpart. (This allows us to match multiple identifiers in one program to the same variable in another, as required when comparing codes (a) and (b) of Fig. 1.)

As an example, for the abovementioned P and Q we would create a cost matrix of all pairwise F_1-values (P-paths in rows, Q-paths in columns):

$$\begin{pmatrix} 0.67 & 0.50 \\ \mathbf{1.00} & 0.50 \\ 0.50 & \mathbf{1.00} \end{pmatrix}$$

The Munkres algorithm optimally assign two of the P-paths to two of the Q-paths (bold face). The third P-path is then associated with the first Q-path, which matches best according to the higher F_1-value. We have thus assigned the variables x and y of P to variable x in Q and may think of renaming all these variables in both codes to, say, u. In the same fashion we may rename the variables of the second assignment to v; reunifying all SVUPs leads us to:

$$\begin{aligned} P' = \{ & \quad u/a/c/f, u/b/c/f, u/a/f, u/c/f, \quad v/b/f\}, \\ Q' = \{ & \quad u/a/f, u/c/f, \quad v/b/f\} \end{aligned}$$

The final similarity is obtained by applying the calculations of Sect. 3.3 to both sets P' and Q' (this time with different variable names rather than just generic variable names vn). Although the example did not include different function names, it works in the same way. Fig. 5 shows the resulting dendrogram. As desired, the similarity of code Disordered (Fig. 1(d)) is now worst among all solutions that follow the same solution approach.

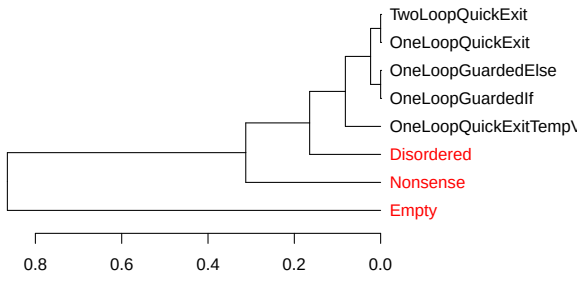

Figure 5: Dendrogram for source codes of Fig. 1 based on F_1-measure on GVUP representation.

4. EXPERIMENTAL EVALUATION

We started to evaluate the proposed approach on real student code submissions and demonstrate its performance on some examples. The submissions were inspected manually and grouped by approach (defining ground truth). While the author of an exercise may have a specific solution in mind, a group of students usually finds multiple ways to solve the exercise. The real data contains nearly identical submissions (potential plagiarism) as well as submissions that appear somewhat chaotic, have superfluous declarations and calculations, or contain artefacts indicating a change in the solution approach over time. Most codes can nevertheless be assigned to solution approaches, but the strong variation in novice's code length misleads edit- and tree-distance such that resulting dendrograms do not match the approaches.

We show results for two exercises, the first exercise asks for the most frequently occurring element in an integer array. To inspire the students to elaborate on different solutions, an additional restriction was given that all values v in the array satisfy $0 \leq v < 10000$. The two main solution approaches were: (1) iterate over all elements in an outer loop, count the frequency of the current element in an inner loop, remember the element that occurs most often; (2) instantiate an array of size 10000 (associating a counter with each possible element in the original array), increment the respective counter while looping over the array and identify the largest entry in the counter array. Apart from these two dominating solutions, three solutions (3) sorted the array first, such that identical values are grouped together, which simplifies frequency counting in a single loop over the array.[3] Finally, (4) there are some exotic solutions which may be considered as a mixture of the discussed solutions. Possibly students became aware of the other approaches by discussing approaches among each others, but had difficulties in solving the task and switched back and forth between them. Usually, elements of all other solutions can be found in them. Fig. 6 shows average-linkage hierarchical clustering

[3]However, when this exercise was handed out, sorting algorithms were not yet discussed (so this solution required background knowledge or a student's own initiative).

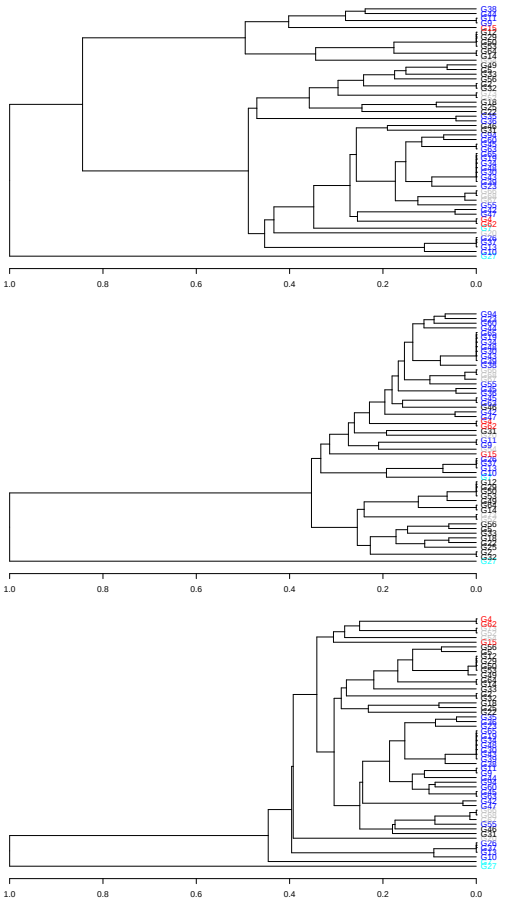

Figure 6: Dendrogram for exercise 'most frequent element' based on F_1-measure (measure from Sect. 3.2, 3.3, and 3.4 from top to bottom).

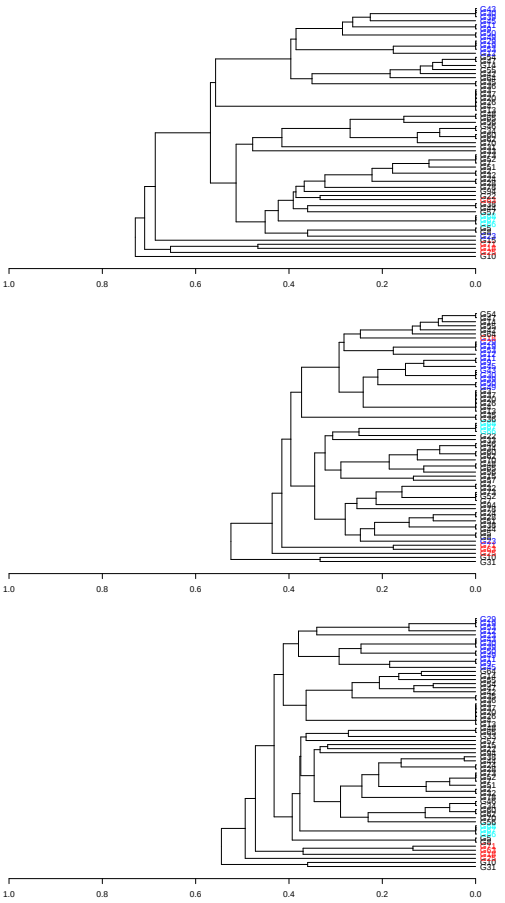

Figure 7: Dendrogram for exercise 'happy number' based on F_1-measure (measure from Sect. 3.2, 3.3, and 3.4 from top to bottom).

results for the similarities from Sect. 3.2, 3.3, and 3.4 from top to bottom. The ground truth is shown by means of the node colors: 1: black, 2: blue, 3: red, 4: gray, incomplete / unfinished: cyan. While the clusters in the top clustering mixes even the two large approaches (1) and (2), the clustering in the middle is much better but does not separate solution (3). This is only achieved by the GVUP-clustering (bottom), which corresponds best to the ground truth. The bubblesort used in (3) uses very similar loops as the main task, so it was crucial to distinguish the role of variables as done in the GVUP-approach.

A second exercise deals with the identification of *happy numbers*[4]. The calculations require the sum of squares of each digit and the submissions differ mainly in the way how this is solved. The intended solution (black) was a loop that successively splits the last digit (using integer division and modulo). Another popular solution is based on string-conversion (blue), a few (somewhat restricted) approaches (red) dealt with different numbers of digits individually (avoiding a loop). Again, the average-linkage clustering for all three proposals is shown in Fig. 7 and the GVUP-approach separates them best. The modulo-approaches (black) subdivide into two major branches, which differ in the way intermediate variables are used. Approaches that use the same kind of utility variables (e.g. to store digits, square of a digit, etc.) are closer matches than approaches that use different sets of utility variables.

[4]`https://en.wikipedia.org/wiki/Happy_number`

5. CONCLUSIONS

Assessing the variety in student's solutions to a programming exercises without having to inspect all codes manually can help a lecturer in many ways. We have proposed a measure that captures how variables and instructions are coupled by means of *variable usage paths*, and use this *fingerprint* to match code from different solutions while at the same time being tolerant to code repetitions. The approach needs to be evaluated further, but the first results appear promising. The nature of the comparison is set-based, which allows us not only to assess similarity (using F_1), but also to use recall and precision. This enables further applications, for instance, we may assess *partial solutions* by the degree how many elements of a *complete solution* they contain (using recall only) or assess the student's degree of programming maturity by investigating the amount of superfluous statements (using precision only).

6. REFERENCES

[1] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Int. Conf. on Software Maintenance*, pages 368–377. IEEE, 1998.

[2] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337:217–239, 2005.

[3] J. Broisin and C. Herouard. Design and evaluation of a semantic indicator for automatically supporting programming learning. In *Int. Conf. Educational Data Mining*, pages 270–275, 2019.

[4] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 243–247. IEEE, 2009.

[5] Z. Djuric and D. Gasevic. A source code similarity system for plagiarism detection. *The Computer Journal*, 56(1):70–86, 2013.

[6] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and Applications*, 13(1):113–129, 2010.

[7] O. Karnalim. Syntax trees and information retrieval to improve code similarity detection. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, pages 48–55, 2020.

[8] M. Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[9] C. Ragkhitwetsagul, J. Krinke, and D. Clark. A comparison of code similarity analysers. *Empirical Software Engineering*, 23(4):2464–2519, 2018.

[10] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar java classes using tree algorithms. In *Int. Workshop on Mining software repositories*, pages 65–71, 2006.