Clustering and Matching Student Solutions in Source Academy

Fredrik Olofsson
Abstract

Uppsala University uses a platform called Source Academy for the introductory programming courses, which handles homework, exercises and more. Homework is delimited and there are only so many ways that one can solve a given task, which leads to a lot of student submissions being very similar when solving a homework. The problem is that teachers going through these submissions will give similar feedback to many submissions, which is unnecessary. The solution to this problem is to identify these similar groups of submissions and provide that information to the teaching staff. This problem was solved by implementing a matching and clustering algorithm that based on specific criteria on submissions groups similar submissions together. The implemented algorithm groups the submissions into clusters that amount to less than 54% of the original number of submissions per homework. This algorithm can help teachers save time while grading and giving feedback on the Source Academy platform.
1 Introduction

Source Academy is used by Uppsala University for teaching programming in the introductory programming courses in the Computer Science and Information Technology programmes. This online platform is used both by teachers and students but in different ways. Students use this platform as a way to both experiment freely in their code editor but also as a way to complete exercises and most importantly submit homework. When homework is submitted teachers can start assessing all the submitted homework, correcting them and giving feedback to the students. Source Academy is very useful for these introductory programming courses, both for teachers and students.

Teachers spend a lot of time examining each student’s code, grading them individually and giving feedback. Currently, the only aid teachers have as integrated feedback is tests. This helps but it does not help identifying students with similar solutions, many students will have the same type of error or their entire solution is almost exactly the same as another student’s. What this leads to is that teachers will give these very similar programs essentially the same feedback, creating a lot of unnecessary work for teachers. If there was a way for teachers to find all these groups of solutions that are very similar, it would save them time as these solutions would get almost the same feedback. In this project we implement a matching and clustering algorithm for the functional Source languages used by Source Academy. This is done by extending the interpreter for these languages.

In this project we want to be able to match and cluster student solutions that are structurally alike and where all variables take the same values in the same order. Matching and clustering is the process of grouping programs that structurally are alike and where there exists relations between two programs variables. This essentially groups programs based on how similar they are to each other. This is done by first deciding whether programs are structurally alike, based on criteria such as the number of functions in the program, the number of recursive functions and more. Secondly, if two programs have a structure that is alike, we compare the values variables and return statements take during the execution of the programs. If the variables and return statements in the two programs in some way take the same values in the same order we can now say they should be a group or cluster. This project extends the Clara algorithm introduced in the paper "Automated Clustering and Program Repair for Introductory Programming Assignments" that was made for imperative languages by adapting it to functional languages [1].

The results of implementing this matching and clustering algorithm were positive as it successfully more than halved all of the given homework that were used for evaluation, except for one homework that is just above 50%. This in turn means that teachers will save roughly 50% of time when giving feedback and grading. Therefore, the problem has been mitigated, but the algorithm
could most likely be improved in the future.

2 Background

In this section some necessary background information is provided to facilitate the understanding of the implemented algorithm.

Abstract Syntax Trees

The algorithm implemented in this project works on abstract syntax trees (ASTs). ASTs are structures that represent the source code of a program in a tree-like structure [2]. This is relevant to this project as all the work is done by analyzing the ASTs of different programs and comparing them to each other. As can be seen in Figure 1, there's always a root node that is an ancestor/parent to all other nodes in the AST. The use of these structures are explained in the design and the implementation of the clustering and matching algorithm that will replicate the behavior of the original Clara algorithm [1].

Source Academy

In Source Academy there are four different languages used to teach students to code. These languages are divided up into two functional languages, Source 1 & 2, and two imperative languages, Source 3 & 4 [3]. Each of these languages does something different, for example, Source 1 does not have access to lists, but Source 2 does. The reason for these changes is because each of these languages cover one chapter of the book that Source Academy follows to teach students to program.

Source Academy also supports their own modules, which provides students with the ability to play sounds, draw curves and more.

Clara Algorithm

There is a clustering and matching algorithm named Clara, introduced in the paper "Automated Clustering and Program Repair for Introductory Programming Assignments" which this project heavily takes inspiration from [1]. The Clara algorithm was implemented for imperative languages like C and Python. Their algorithm works by first structurally matching programs based on looping structure which is connected to the control-flow of programs. This is because in the Clara algorithm, all loop-free code is treated as a single block and code containing loops increases the amount of blocks. The loops that the Clara algorithm looks at are for-loops and while-loops, which does not include recursive functions. The structural match is what makes the following variable matching possible. The variable matching looks at variables and the values they take including in which order they take them. Then, after having acquired this information, checks if there exists a total bijective relation to another given program's variables. A total bijective relation attempts to map each variable in
Figure 1: The structure of a simplified abstract syntax tree. This AST consists of one function that consists of three statements and one function call to this one function.

one program to a unique variable in another program, and only if every variable in both programs map to one in the other, is there a total bijective relation between them. Therefore, two programs match if they both structurally match and there exists such a total bijective relation between the programs variables.

3 Design

In this section, we go through the design of the clustering and matching algorithm for Source. First of all, the original Clara algorithm that was implemented for imperative languages would look at all functional programs as having a single block in their control-flow due to there not existing any loops. This makes the structural matching of the Clara algorithm unfeasible, as all functional programs would always structurally match. Therefore, let us look at an intuition of how we could structurally match programs in functional languages and in that way adapt the Clara algorithm to functional languages.

In Figure 2 there are three example programs that came from students. These three programs were selected as two of the programs belong to the same cluster and one does not. If we look at the first two programs, there are some patterns that are found.

- Programs one and two have the same number of functions.
- Programs one and two have the same number of recursive functions.
Figure 2: Example student programs.
• Programs one and two both have a return statement that consists of a conditional expression.

These properties are not shared with program three. Therefore, the algorithm doesn’t cluster them together.

The goal here is to make sure that there’s a possibility that two programs may match in the variable matching that comes after. Therefore, the structural matching can’t be too harsh and have too many criteria. If two programs do structurally match, but the two programs do completely different things, the variable matching that follows the structural matching will eliminate the possibility of these programs getting grouped together.

The variable matching is the process of mapping each variable in one program to a unique variable in another program, which is the total bijective relation that the Clara algorithm uses.

3.1 Structural Matching

Since the languages Source 1 and Source 2 are functional languages, some changes had to be made to structurally match programs due to the original Clara structural matching having been developed for imperative languages. The problem to solve in this case is if two programs structurally match or not in functional languages, by doing this, we only perform the variable matching if there exists a possibility of a match.

Logically speaking, two programs $P$ and $Q$ can only match in the variable matching if every return statement in $P$ is matched to another in the other program $Q$. Therefore, the first condition of a structural match is that the two programs must have the same number of function definitions. However, if functions were to have multiple return statements, this would have to be solved by the variable matching that follows the structural matching.

The second condition is that two programs have the same number of recursive functions. Since if one program uses recursive functions and another does not, but they have the same number of function definitions, the programs will likely not have equivalent executions, although they may have the same result. Thus, this makes up the second condition.

The third condition is that the programs $P$ and $Q$ have the same number of conditional expression returns. Conditional expression returns here are returns that look like the following:

\[
\text{return expression1 ? expression2 : expression3;}
\]

Since both $\text{expression2}$ and $\text{expression3}$ can also consist of a conditional expression, we gather information about the nesting depth of these conditional
expression returns. In a sense, this provides information of how many different returns the function really has, even though it is all in one return.

Therefore, the following conditions are what makes up a structural match between two programs $P$ and $Q$.

- The programs $P$ and $Q$ have the same number of function definitions.
- The programs $P$ and $Q$ have the same number of recursive functions.
- The programs $P$ and $Q$ have the same number of conditional expression returns. Further, every conditional expression return in $P$ matches a conditional expression return in $Q$ of the same nesting depth. Look at Figure 3 and imagine the left set of nodes as conditional returns in $P$ and the right set of nodes as conditional returns in $Q$.

Let us take a look at an example of the conditional expression mapping. Looking at the return statements of programs one and two in Figure 2 there only exists one conditional expression inside both return statements and therefore their nesting depth is the same and match each other.

Now that the requirements for a structural match have been defined, an explanation of how this information is gathered is explained in section 4.2.

### 3.2 Variable Matching

For the variable matching, there is a difference between the Clara algorithm and the one used in this thesis project. The difference is that because the function bodies in the student programs are often short, the return statements are usually complex and consist of function calls. The arguments of these function calls are often moved out into variables to make the return statement easier to read.
and understand. Because of this, we opted to only compare return statements in student programs rather than both return statements and variables, which is what the Clara algorithm did.

The requirements for a match is therefore that every return statement in a program $P$ should find a unique return statement in a program $Q$ that takes the same values in the same order as the return statement in $P$. In the following list of steps $H_P$ is the variable history for the program $P$ and $H_Q$ is the variable history for the program $Q$.

1. During program execution for the programs $P$ and $Q$, the two programs store all return values in their variable history, $H_P$ for $P$ and $H_Q$ for $Q$. 
2. For every return statement $r_P$ in $H_P$ find a corresponding $r_Q$ in $H_Q$ that took all the same values in the same order as $r_P$.
3. If all return statements match, the two programs match.

4 Implementation of the Matching and Clustering Algorithm

We now move on from the design of the algorithm and how it theoretically works to how the implementation itself is done in Source Academy. The existing interpreter in Source Academy will not be modified, but a new modified version of it is be made. First, there is some preparation that is needed for the student programs to be able to execute.

4.1 Program Preparation for the Algorithm

For the student programs to be executed, certain pieces of code have to be added to programs. These pieces of code are shown in Figure 4, which also shows the entire process of preparing the student programs for the parser and the interpreter. The things that have to be added to the student code are module imports and predeclared functions. Then, these programs are turned into abstract syntax trees.

First of all, predeclared functions are introduced in each different Source language. These provide functionality such as lists, loops etc. In the frontend of Source Academy, these predeclared functions are handled automatically without students having to declare them themselves, so we need to add them to the code. The Source Academy language backend already has functionality for obtaining the predeclared functions for a specific Source language. Therefore, the solution was to prepend the obtained predeclared functions to the student code.

Similarly to the predeclared functions, modules are handled automatically in
the frontend of Source Academy. Thus, these modules have to be appended to the student programs as well in an import statement. Since there is no way to discern which module is used in a student program due to the modules sharing function names, the user is asked to provide this information to the algorithm via a command-line argument. The algorithm then prepends an import statement consisting of all function names in the given module, solving this issue and making the student programs executable.

At this point, all predeclared functions and modules have been added to the student programs and turning the student programs to abstract syntax trees remains. Source Academy already has functionality for turning code into abstract syntax trees, so this functionality was used to turn all student programs into abstract syntax trees. Now, the matching and clustering can start processing these abstract syntax trees in section 4.2.

4.2 Implementation of the Algorithm

A new copy of the interpreter is made to not affect the original one. This new interpreter is made to make changes that allow for the implementation of the matching and clustering algorithm. The structural analysis is coded in such a way so that it works essentially the same as the interpreter but without
executing anything. It visits all the nodes of a given abstract syntax tree and performs different work depending on which type of node it is. The structure that’s used for the matching and clustering algorithm is capable of storing the following information:

- Function declarations: For every function declaration that the algorithm encounters, it stores the name of the encountered function.

- Recursive functions: For every function that the algorithm determines is recursive, store the name of that function.

- Nesting depths: For every return statement that consists of a conditional expression, store the nesting depth of the conditional expression.

It is essentially storing everything needed for a program to be able to be compared to another, structurally.

The first step of the algorithm is to get the abstract syntax trees for all of the student solutions. This is done by putting all the solutions for a specific homework in a certain folder and reading all the files in that folder. By doing this, we get the file name and the code at the same time. From there, we get the abstract syntax trees for the student programs and import all required modules. The second step is to analyze the structure of all student solutions. This is done by mimicking the behavior of the interpreter but without evaluating anything and just analyzing. The following node types were important when doing the analysis:

- **Function declarations**
  - Store the function name of the function in the function declarations in the structure.

- **Return statements**
  - If it consists of a conditional expression, analyze the nesting structure of the conditional expression and store the nesting depth for that return statement in the structure.

- **Function calls**
  - If the called function is the same as the currently analyzed function, store it as a recursive function in the structure.

- **Conditional expressions** The algorithm had to check if the current node is the root of the nesting analysis. This is to be able to keep track of left and right from the root node and not change it in the case of nesting.
  - If the current node is root: analyze the left and right subtrees for possible nesting.
Figure 5: A conditional expression where the depth of the left subtree is two and the right subtree is one.

- If the current node isn’t root: check if we are analyzing the left or right subtree and increase the depth of the corresponding tuple value if the current node is a conditional expression.

Figure 5 shows a conditional expression with depth two and one, specifically the following expression:

`expression ? (expression2 ? 1 : 2) : 3`

In Figure 5 the top node is the root node. Then, we analyze the left subtree. The algorithm notices that this node is a conditional expression and not the root node. This node also analyzes its own left and right subtree but there are no more conditional expressions. Therefore, we get the depth of two in the left subtree. Now the algorithm goes back to the root node and looks at the right subtree. Following the same process, the depth of the right subtree is one as there are no additional conditional expressions in it.

An example of running the structural analysis on program two in Figure 2 will provide the following information:

```plaintext
definitions = {"cone"}
recursive_functions = {"cone"}
nesting_depths = {return → [1,1]}
```

This information can be verified by analyzing program two manually.

After the above process of analyzing the structure of all student programs is
completed, all student programs are run and their variable histories are stored in the aforementioned structure as another field. The structure now keeps the following information:

- Function declarations: For every function declaration that the algorithm encounters, it stores the name of the encountered function.
- Recursive functions: For every function that the algorithm determines is recursive, store the name of that function.
- Nesting depths: For every return statement that consists of a conditional expression, store the nesting depth of the conditional expression.
- Variable history: For every return statement, the algorithm stores all the values the return statement took during program execution.

As described in the design of this algorithm, the requirements for a variable match are that every return statement in a program $P$ must match a unique return statement in program $Q$. This match is checked by analysing the information that was received by executing the programs with a slightly modified version of the original interpreter.

Finally, the end of this algorithm is the actual matching and clustering. Everything up to this point has provided enough information about all programs to analyze both their structure and their variable histories which is exactly what's needed for the clustering. The clustering works according to Algorithm 1 which shows the pseudocode for the implemented algorithm.

The algorithm takes as input all information about all of the programs, which includes their analyzed structure and variable histories, which exists in the above structure. First of, the algorithm initializes the clusters by initializing the first cluster which initially only contains the program $p_0$. This is done so that all other programs always have a cluster to compare against. Further, for every remaining program, go through all of the clusters and attempt to structurally match the current program against the representative program of the current cluster. The representative program of a cluster is the first program that was added to the cluster, meaning the program that initialized that cluster. Therefore, if the algorithm first structurally matches these two programs and then also pass the variable matching, the current program joins the current cluster. However, if no cluster was found for the current program, we initialize a new one with the current program as the representative program of that cluster.

At this point, all programs have been analyzed and grouped together based on structure and variable matching, so the matching and clustering algorithm is complete.
Data: programs
\[ p_0 \cup \text{remaining} \leftarrow \text{programs}; \]
\[ \text{clusters} \leftarrow \{ \{ p_0 \} \}; \]
for \( p_i \) in remaining do
  for \( c_j \) in clusters do
    if structuralMatch\((p_i, \text{representative program of } c_j)\) then
      if variableMatch\((p_i, \text{representative program of } c_j)\) then
        \( c_j := c_j \cup \{ p_i \}; \)
      end
    end
  end
if no cluster was found then
  \( \text{clusters} := \text{clusters} \cup \{ \{ p_i \} \}; \)
end
end

Algorithm 1: The matching and clustering algorithm.

5 Evaluation

The implementation of the matching and clustering algorithm has been explained so we are now going to look at how well the algorithm works in practice.

5.1 Clustering

Since the goal of this matching and clustering algorithm is to help teachers save time correcting and giving feedback to student solutions by clustering solutions that are similar both structurally and with the values variables take. First of all, consider the graph in Figure 6 that shows the amount of student programs that ran with no errors per homework. All of the different homework have at the minimum above 200 valid student solutions with homework three clearly taking the lead with over 500 valid solutions.

The graph in Figure 7 shows the amount of valid student solutions per homework and the corresponding number of clusters for each homework. The data for these graphs were obtained by allowing the matching and clustering algorithm to finish executing on each homework and then extracting the needed information from each homework.

In Table 1 the difference between the amount of clusters and valid solutions per homework is shown as a percentage, where lower is better. This table makes it clear that all but one homework made the number of clusters into less than 50% of the number of valid student solutions.
Figure 6: A graph that shows the number of valid student solutions per homework.

Figure 7: A graph that shows the number of clusters and valid student solutions per homework. Fewer clusters is better.

<table>
<thead>
<tr>
<th>clusters</th>
<th>homework 1</th>
<th>homework 2</th>
<th>homework 3</th>
<th>homework 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>programs</td>
<td>43%</td>
<td>29%</td>
<td>53%</td>
<td>46%</td>
</tr>
</tbody>
</table>

Table 1: The number of clusters compared to the number of valid submissions per homework. A lower percentage is better as it means more programs were clustered into larger clusters.
Table 2: The performance of the structural matching and the variable matching for 100 programs per homework. These times were calculated from the total amount of time it took for the structural matching and variable matching, calculating the average for both and multiplying the average by 100.

<table>
<thead>
<tr>
<th></th>
<th>homework 1</th>
<th>homework 2</th>
<th>homework 3</th>
<th>homework 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural Matching Time</td>
<td>2.54 ms</td>
<td>2.61 ms</td>
<td>2.19 ms</td>
<td>0.57 ms</td>
</tr>
<tr>
<td>Variable Matching Time</td>
<td>1349.41 ms</td>
<td>9089.21 ms</td>
<td>188.78 ms</td>
<td>12.51 ms</td>
</tr>
</tbody>
</table>

5.2 Performance

The performance of the implemented algorithm can be seen in Table 2, which shows the average time it takes each homework to structurally match 100 programs and also perform variable matching on 100 programs. This data was obtained by summing up the time spent in the structural matching function and the variable matching function 10 times for each homework and then calculating the average. The data was extracted when the algorithm had finished running the current homework. The performance data was obtained on a computer with the following specifications:

Operating System: Windows 11 Pro running WSL 2
Processor: Intel(R) Core(TM) i9-9900 CPU @ 3.10GHz 8 cores
Installed RAM: 32.0 GB
System type: 64-bit operating system, x64-based processor
GPU: NVIDIA GeForce RTX 2080 SUPER

6 Related Work

Perry et al. implement a new type of clustering technique which they call SemCluster for imperative programming languages [4]. Algorithms like the one implemented in this thesis project and Clara require programs to essentially perfectly match certain program features, which they saw as a problem. As this results in an excessive number of clusters due to the very nature of the matching itself. SemCluster works by instead focusing on the high-level algorithmic solution strategy and ignoring small syntactical and implementational differences. According to them, this strategy to clustering results in much fewer clusters compared to algorithms like Clara and those in that category. This is understandable as our algorithm and the Clara algorithm can generate quite a lot of clusters based on small differences in programs, as we look at specific properties rather than the program as a whole.

Another way to perform clustering is by analyzing core statements in the given student solutions. This is what Marin et al. does [5]. The authors also try tackling the problem of small differences determining if a program should be in a
cluster or not. They do this by for each program creating a program dependence graph which gives information about control and data dependencies. Then by analysing all of these program dependence graphs with different methods such as approximate graph alignment they retrieve the core statements that appear in most programs. By doing this, they cluster based on these core statements and in this way get rid of the noise that may cause more clusters in other algorithms. Thus, differentiating itself from the way our algorithm and the Clara algorithm works by trying to eliminate noise in the solution and looking at what most programs have in common instead.

Similar to SemCluster, the algorithm PaCon by Fu et al. tries to look at the program at a high-level and identifying which tactic the student approaches the problem with, rather than the exact way it is implemented [6]. They do this by looking at path conditions from symbolic execution and clustering based on the path conditions. This differs from our algorithm as well, as it tries to find the high-level approach that the student took, rather than looking at specific criteria.

It does seem like clustering is moving away from matching based on specific properties and instead focusing on identifying strategies as SemCluster and PaCon does [4][6]. This approach attempts to discern how a student approached a problem with their solution rather than exactly how the student implemented their solution.

7 Discussion

In this section we will discuss some unexpected problems and the results from the evaluation.

7.1 Unexpected Problems

During the implementation and evaluation phase several unexpected problems occurred. Some of which were more troublesome to deal with than others. Here two of the unexpected problems are explained.

Module Imports

One of the necessary steps for the variable analysis to work for most homework solutions is the module functionality. These modules allow for things such as graphics and playing sounds. As mentioned in the implementation, the required modules are imported at the start of the program to allow usage of them in the homework solutions, which are necessary for the variable analysis to work. However, there happened to be an issue with this, which came very unexpectedly. Namely, that the module code which was located on their website had been updated to work with another kind of structure when evaluated. Because of this,
the external module code couldn’t be evaluated in the version of the code base which work had been done on. Therefore, the solution to this was to update the code base manually and everything worked with no issues.

**Manual Labour**

For the matching and clustering algorithm to work, it is essential that all programs share the same tests and that no inputs for these tests have been changed. Not even the order. This, however was an issue when analyzing why some programs appeared in different clusters and many times were in a cluster with only themselves. This then had to be fixed by analyzing hundreds of student solutions per homework and manually fixing the incorrect or modified tests.

When starting to work on the third homework which involved a module which played sounds in the browser by using a function called play it was noticed that all the programs that used this function gave errors. After some quick analysis, the error in itself was because there was no browser which the function could use to play the sounds for. To solve this, all student solutions that used the function play were modified to no longer use that function.

**Memory Constraints**

The amount of memory required for each homework varies a lot and depends on which module the homework uses. For example, homework two which handled curves, required a lot of memory to store information about all variables and their values. This is because the curve object as mentioned, consists of point objects. Thus, having to store for example even one curve with 10000 points for hundreds of programs is heavy on the memory. The other homework did not have any issue with memory, it was specifically homework two which successfully went over the heap memory limit in Node.js, which lead to an increase in the amount of memory that the clustering is allowed to use, to avoid memory issues. This solved all memory issues for every homework and allowed every homework to execute successfully.

### 7.2 Results of Evaluation

Here we discuss the results obtained from the evaluation of the implemented algorithm.

**Clustering**

Looking at Figure 6 which shows the amount of submissions per homework, one can see that there are a lot of submitted solutions per homework. The algorithm successfully clusters more than half of all these programs in three out of the four given homework which is shown in Figure 7 which has the number of valid programs and the number of generated clusters for each homework side by side. The number of programs that are clustered in any given homework
seems to vary a lot. The reason for this is believed to be in the way that these homework are structured. In all of these homework, students were using different modules to achieve different goals.

In homework one, the students were to draw certain shapes and this was quite a simple homework and the class that the module used to represent these drawings was not too complex and therefore the clustering worked quite well.

The second homework involved drawing curves and it was done in a controlled manner. The students were given a mathematical function and they had to make sure it was drawn correctly. This resulted in homework two having a small number of clusters.

The third homework was the most troublesome homework as it involved sounds and students were also asked to copy their solution from an earlier question into their solution, which introduced a lot of noise. This is because for each task, a student can solve it in many different ways, so when a homework consists of multiple tasks, it makes it so the student code can vary a lot more. Another issue with this homework is that it highly depends on in which order things are done, for example you can create the same sound by first applying a certain amount of operations, but then also by creating the same sound with a different order of the same operations. This may introduce floating-point precision errors which it did look like in homework three.

The fourth homework was a bit troublesome, the homework itself was quite defined but the way a student could solve it varied a lot. This is because the fourth homework was about navigating binary trees and what caused a lot of the clusters is the depth of the conditional expression returns. Students think differently about this navigation causing a lot of variation in the conditional expression returns.

In general, the amount of programs that can get clustered in each homework depends on the creativity of the students and also how structured and delimited the homework is.

**Performance**

For the performance of the algorithm itself it highly depended on which homework it was. As the structural matching is simple and just looks at the structure of the program it takes basically no time to finish. In Table 2 we can see that homework four seems to be much less complex structurally, which is correct. The structural matching is much simpler than the variable analysis and therefore it is understandable that it is fast and efficient.

Compared to the structural matching, the variable matching is much slower. The performance of the variable matching can also be seen in Table 2. This
table shows that homework two is comparatively much slower than all of the other homework in the variable matching and that homework four is very fast in the variable matching. The reason that homework two is much slower than all of the other homework is understandable because homework two handles curves. A curve in Source Academy consists of a number of points. In this specific case, each curve in homework two is made up of 10000 points which all have several attributes. Then the problem is that for each program it is expensive to compare curves with another program, as the algorithm will have to go through each of the points and compare them individually.

8 Conclusions and future work

In this thesis project, we successfully implemented a matching and clustering algorithm that extends the Clara algorithm to functional languages. This implementation does what it is supposed to do however there are of course weaknesses in it. First of all, since the Source Academy interpreter is written in TypeScript, which itself is built on top of JavaScript, there is no possibility of implementing parallelism. So what this meant for this project is that first of all, executing all of the hundreds of student solutions per homework takes quite some time, sometimes even multiple minutes. Referring back to Table 2, if we could run all programs in parallel it would make the entire process of matching and clustering much faster. Another weakness is that the algorithm only detects direct recursion and not mutual recursion, which means that programs with mutual recursion may get clustered incorrectly.

When the algorithm is finished, we get a nice amount of clusters compared to the original amount of student solutions as can be seen in Figure 7. As mentioned in the evaluation, the reason for some of the artificial clusters is that students could easily change the tests that were in the code, and also which data was used in their programs rather than keeping it the same for each single program.

In the future, the implemented algorithm’s output, which is the clusters and which programs are in each cluster could be used by a parser to quickly extract all the clustering information. This in turn can help establish a connection between the frontend of the Source Academy platform and the language backend. As currently, the algorithm can only be used in the backend and has no connection to the frontend. Thus, in the future one could work on bringing the algorithm together with the frontend for easier usage. To further aid the algorithm, it would benefit greatly from structuring homework in such a way that clustering can easily be applied. Meaning that homework is done in a structured and delimited manner to avoid artificial clusters. This also includes either filtering away all of the tests and the used test data for the students and instead generating them, or removing the ability for students to change them.
In general, the matching and clustering algorithm works well, with some weaknesses here and there and can be further improved in the future with direct or indirect actions on the algorithm or the student solutions themselves.
References


