International Institute of Information Technology
CS 306 Programming Languages
Supervisor: Prof. Sujith Kumar Chakrabarti

Measuring similarity between programs

June 7, 2020

Eric John IMT2017016
Srihari Vemuru IMT2017045
Seethamraju Purvaj IMT2017039

Integrated M.Tech

Contents

1 Introduction

It is common to find similar code in programs which are either syntactic or semantic due to certain coding practices. These could be due to copy and paste, lack of functional modularity etc. in Software Development or due to code manipulation including rearranging statements in case of a computer science student who is lazy to write his assignment.

Existence of similar code is generally undesirable as larger KLoc(lines of code) would generally mean higher cost of maintenance. But at the same time time coding constraints including time would make the programmer to directly reuse the code in the program over trying to link the appropriate module.

In the wake of gaining popularity of online programming and software engineering education, online coding courses have now become significant. These courses are taken usually by one or two instructors and will have thousands who are enrolled. Measuring the progress of those involved becomes difficult as the instructors have look through each program that they submit. The usual work around for this is giving a specified input and checking for the correct output which can be automated. But this approach is error prone as there are many workarounds as well as requirement that the code compiles. If a student's code does not compile the automated checker would likely give a low score which would also be fixed. Unlike coding competitions which would require you to pass all test cases learning courses will have to be supportive.

There are many approaches to finding similar code. But these approaches vary with the definition of what similarity we are searching for. Similarity is broadly divided as Representational and Behavioural. Representational similarity can be further divided into Textual, Lexical and Structural. Behavioural similarity is further divided as Functional and Execution similarity. While functional similarity looks at the input/output behaviour of a code fragment, execution similarity looks at sequence of execution statements like assembly code.

A naive approach to measure behavioral/semantic similarity between two programs is to run them on every input in the input domain, compare the outputs of the two programs, and then compute the portion of the agreed inputs. But, such an approach is impractical and infeasible for domains with infinite inputs.

Through this project we are look at one of the recent papers "Measuring Code Behavioral Similarity for Programming and Software Engineering Education"(1). The paper talks about three different metrics and suggests a combination of them for measuring similarity of two code fragments. It also talk about it's application on a real world online coding website and the results that it produced which we will be skipping.

The project also discusses some approaches that the paper has not explicitly mentioned which could be used to improve the evaluation algorithm that the paper proposed.

2 Functional Programming

Intuitively two terminating programs can be called similar if they implement a similar function. One approach for checking implementation of functions is in terms of similarity in the input-output relationship that is measuring the fraction of inputs for which two programs produce the same output. More complicated notions of input-output correspondence need to be contemplated when the input or output spaces are not finite.

3 Definitions

To measure behavioural similarity of two programs,the paper puts forward the following definitions:

Definition 1 (Program Execution). An execution of a program P is a function $exec : P \times I \rightarrow O$ that maps an input $i \in I$ to an output $o \in O$,where I is the input domain of P and O is the output domain of P.

Definition 2 (Behavioral Equivalence). Two programs P_1 and P_2 that share the same input domain I are behaviorally equivalent, denoted as $exec(P_1, I) = exec(P_2, I)$, iff $\forall i \in I, exec(P_1, i) = exec(P_2, i)$.

For example consider two programs that calculate the sum till a given n. Now irrespective of the nature of the function i.e. for loop, while loop or recursion the output and input will be same across all implementations. Therefore if A is iterative implementation and B is a recursive implementation, then A and B are said to be behavioural equivalent.

Definition 3 (Behavioral Difference). Two programs P_1 and P_2 that share the same input domain I are behaviorally different, denoted as $exec(P_1, I) \neq exec(P_2, I)$, iff $\exists i \in I, exec(P_1, i) \neq exec(P_2, i)$.

For example, Consider the above program A. Now if we have another program B that gives the product of numbers till n. Now irrespective of the implementation the output of the programs will not match except for n = 1. Hence A,B are said to be behaviourally different.

Definition 4 (Behavioral Similarity). The behavioral similarity between two programs P_1 and P_2 that share the same input domain I is $|I_s|/|I|$, where $I_s \subseteq I, exec(P_1, I_s) = exec(P_2, I_s)$, and $\forall j \in I - I_s, exec(P_1, j) \neq exec(P_2, j)$

For example, consider the program A in the 2^{nd} definition and another program which calculates the sum of numbers if $n \leq 100$ and product of numbers for $n > 100$. Now for the set {1..100} A and B give same output whereas for $n > 100$ A and B give a different output. Therefore both programs A and B are behaviourally different for the input domain of positive numbers.

Definition 5 (Behavioral Subsumption). Assuming two programs P_1, P_2 and a reference program P_r all share the same input domain I, P_2 behaviorally subsumes P_1 with respect to P_r, denoted as ($P_1 \leq_{P_r} P_2$), iff $\exists i \in I$, $exec(P_2, i) = exec(P_r, i)$ & $exec(P_1, i) \neq exec(P_r, i)$, and $j \in I, exec(P_1, j) = exec(P_r, j)$ & $exec(P_2, j) \neq exec(P_r, j)$.

Intuitively, a program P_2 behaviorally subsumes a program P_1 if and only if the set of inputs correctly handled by $_1$ is a proper subset of that correctly handled by P_2. An input is correctly handled by a program P_i with respect to the reference program P_r if and only if P_i and P_r produce the same output on this input.

Definition 6 (RS). P_r and P_c are two programs sharing the same input domain I. Let I_s be a set of inputs randomly sampled from I, and I_a be a subset of I_s such that $\forall i \in I_a, exec(P_r, i) = exec(P_c, i)$ and $\forall j \in I_s - I_a, exec(P_r, j) \neq exec(P_c, j)$. The RS metric is defined as $M_{RS}(P_r, P_c) = |I_a|/|I_s|$.

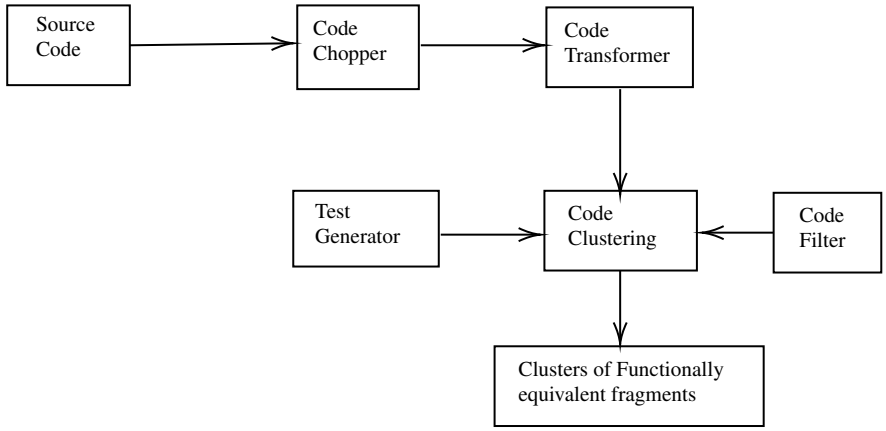
Definition 7 (SSE). P_r and P_c are two programs sharing the same input domain I, and P_r is the reference program. Let I_s be the set of inputs generated by DSE(Dynamic Symbol Execution)(4)(5) on P_r, and I_a be a subset of I_s such that $\forall i \in I_a$, $exec(P_r, i) = exec(P_c, i)$ and $\forall j \in I_s - I_a$, $exec(P_r, j) \neq exec(P_c, j)$. The SSE metric is defined as $M_{SSE}(P_r, P_c) = |I_a|/|I_s|$.

Definition 8 (PSE). Given two programs P_r and P_c sharing the same input domain I, we construct the paired program P_p in the form of $assert(exec(P_r, I) = exec(P_c, I))$, where $assert$ is a function that accepts

a condition as input and asserts that the condition is true. Let $exec(P_p, i) = T$ denote the execution of an input i on P_p that passes the assertion. Let I_s be the set of inputs generated by DSE on P_p, and I_a be a subset of I_s such that $\forall i \in I_a$, $exec(P_p, i) = T$ and $\forall j \in I_s - I_a$, $exec(P_p, j) \neq T$. The PSE metric is defined as $M_{PSE}(P_r, P_c) = |I_a|/|I_s|$.

4 Basic structure of evaluation algorithm

This section briefly explains the components of a general evaluation algorithm by Lingxiao and Zhendong (6).



4.1 Code Chopper

Since we are considering functionally equivalent codes of various sizes, instead of the whole program or function, we use a code chopper to extract code fragments from a program as candidates for functionally equivalent ones. It takes a function definition and parses it into a sequence of statements, then it extracts all possible consecutive sub sequences from the statement sequence, and each of the sub sequences is considered a candidate for functional equivalence. The chopper fixes a value on number of statements that need to be in a sub sequence and can treat internal boundaries as a bigger statement(simple loops etc).

4.2 Code Transformer

As we define the functional similarity in terms of input-output behaviour each code fragment has to be executed to observe it's output value. Therefore all code fragments have to compile and run. A method would be to run through the whole program and include all the library statements in each fragments as well as store the data type declaration statements and later use it with or without casting to a more common type.

4.3 Code Filter

The code chopper may extract many code fragments that overlap with each other.This is because the code chopper divides a function into sub sequences and some will overlap as they are from same location . It is not feasible to use these fragments as it will increase computation cost. The code filter finds such sub sequences and take only one among those to represent that group of fragments.

4.4 Test/Input Generator

Since the executions of each code fragment require random inputs, the input generator component takes a code fragment and its input variables and generates random values for the input variables. Functionally equivalent code fragments should exhibit the same behavior on even invalid inputs.

4.5 Code clustering

The code clustering component takes a set of code fragments that have been compiled and then executes each code fragment with the same random inputs which will be generated by the input generator, and separates two code fragments into two different code clusters whenever the outputs of the two code fragments differ.

5 Metrics To Measure Similarity

5.1 Random Sampling(RS)

The idea behind RS is that by uniformly sampling a significant portion of inputs from the input domain, the behavioral similarity computed based on the sampled inputs can provide an accurate approximation to the actual behavioral similarity. To compute the value for RS, first random test generation is used to generate inputs uniformly distributed over the input domain. Then both programs are run on each test input separately and the proportion of the agreed inputs over the sampled inputs is the value for RS, [Definition 6].

RS treats the program as a black box, and does not analyze the actual intricacies of the program for test generation. Therefore, the cost of generating test inputs and measuring RS is rather low or even negligible. But due to the same reason, it may miss some inputs, like corner cases, that reveal behavioral differences between programs.

5.2 Single-program Symbolic Execution (SSE)

One way to consider the program's algorithm, not considering it as a black box, while generating inputs is to find the set of inputs for which the program runs differently for each of them. If we pick one input for each path in the program's computation graph, then these inputs explore the representative behaviors of the program. To compute SSE, [Definition 7], one program is chosen as the reference program, and apply DSE to generate test inputs that capture the behaviors of the reference program. Then the other program is run on these test inputs and the proportion of agreed inputs is computed over the generated inputs as the value of SSE.

Thus, these test inputs are more likely to cover those corner cases of the program. Revealing such corner cases is helpful in distinguishing programs with small behavioral differences. But still, SSE has some limitations. First, SSE never considers the program under analysis, but generates test inputs based on only the reference program. The generated test inputs do not necessarily capture all behaviors of the program under analysis. Second, programs may have infinitely many paths when there are loops in the program whose iteration count depends on unbounded inputs. For these programs, it is also infeasible to enumerate all program paths. To alleviate this issue, we can either bound the input domain or the loop iteration count to enumerate a subset of program paths as an approximation.

5.3 Paired-program Symbolic Execution (PSE)

To address the limitation that SSE may fail to reveal behaviors in the program under analysis, the Paired-program Symbolic Execution (PSE) metric, [Definition 8], is computed by constructing a paired program from the reference program and the program under analysis, and generating test inputs by exploring the paths in the paired program. The paired program shares the same input domain with the two programs, and it feeds the same input to both programs and asserts the outputs of the two programs to be the same. Hence, the proportion of test inputs that pass the assertion is the value of PSE.

PSE improves SSE by avoiding the situations where the generated test inputs capture behaviors of the reference program but not the program under analysis. However, PSE still faces the same challenge of

handling infinite paths in some paired programs. To alleviate this issue, we can again bound the input domain or loop iteration counts. In addition, the paths to be explored by PSE in the paired program are the combination of paths in both programs. Thus, PSE has a higher cost in path exploration than SSE does.

6 Evaluation

The paper tries to find behavioural similarity when used in an online coding website. The scenario that it took is the following. You are given a set of programs. One is the reference program, the other's the code that people who are working on a programming task has produced. The paper compares the programs with the reference program and tries to quantify the progress of a individual person with respect to the group as well as the reference program.

How effective are our metrics in ordering programs based on their progress towards the reference program? How accurate are our metrics in approximating the behavioral similarity?

A random set of programs were chosen from the data set with the following properties. The sequence has a later program behaving equivalently as an earlier program or the sequence has a later program behaviourally subsuming an earlier program with respect to the reference program[Definition 5]. To do this one has to correctly identify the sequence behaviourally equivalent code. An online coding website provides some level of help for the same as the programs are labelled with the task that is associated to along with their time stamp. They chose to do find the sequences manually.

Then they compute all three metric values for each program in each sequence, and verify whether the computed values of a metric over a sequence conform to the sequence order. Two equivalent programs should have the nearly the same value, and a program subsuming the other should necessarily have a higher value than the other.

A metric is marked as correct indicating the progress over the programs in the sequence if the computed values of the metric conform to the order of the sequence. A higher percentage for a metric indicates that the metric is more accurate in indicating the ordering of programs. Next using the manual ordering as the true solution they compare their metric with respect to it and compute absolute error as well as standard deviation.

7 Outcome

PSE is sensitive to changes in the submitted program. They found two kinds of changes in the submitted program that was effectively detected by PSE:

control-flow changes that result in different or additional paths in the paired program.
non- control-flow changes that make previously infeasible paths in the paired program feasible.

PSE can detect these changes because the path exploration on these two paired programs (before and after changes) identifies two different sets of paths. The metric values computed based on the path exploration are then different. If such changes in the submitted program correctly handle more inputs, PSE could typically generate more passing test inputs and thus increase the metric value. Thus PSE effectively indicates the progress between programs. The other two metrics, RS and SSE, do not work well on producing the correct orders over the sequences, because they are not sensitive to the changes of the submitted program (not generating test inputs based on the submitted program). In other words, given a reference program, they both run any submitted program on a fixed set of test inputs. Hence, they often fail to distinguish two submitted programs with slight behavioral differences.

RS approximates the actual behavioral similarity much better than the other two metrics. The reason for its accurate approximation is that RS ensures that the randomly generated inputs are uniformly distributed across the input domain.

SSE and PSE do not approximate the behavioral similarity well because they compute the metric values based on only path exploration. They implicitly give an equal weight to each path by doing simple path counting. However, the corresponding input partition of each path may vary in size. In order to provide a more accurate approximation, they proposed assigning a weight to each path based on the size of its corresponding input partition.

The cost of computing the three metrics is acceptable. As expected, RS has lower cost than the other two metrics. The major cost for computing RS is running the sampled inputs on two programs.

8 Viewing as a Evaluation Algorithm

One of question's that frequently came to our mind while reading the paper was "How do they do it?. Isn't that too specific? Can you actually use it?". Thus we tried to convert the paper into it's equivalent evaluation algorithm (A brief overview of the algorithm is shown in section 4) with resources that they gave as references.

8.1 Code Chopper

The paper had taken for granted the programs that were submitted had few lines. Thus they took each submission as an independent program for their calculations. Though this is justified to some degree, we found out that in many practical cases this is not a good approach.

For example, in a college most students have to write a large program. There are pieces of codes that are relevant and those that are not relevant. For a competitive programming problem, input-output statements provide little to no importance to the program. While on the other hand one conditional statement could make all the difference. Taking the whole program as a measure thus does very little in distinguishing apart an input-output statement error from a conditional statement error.

Thus breaking the code into multiple sub sequences and giving them weights is a justified approach to improve the evaluation strategy based on what you are doing. Here for competitive programming giving higher weightage to all code fragments that contain the conditional statement should provide a more accurate grade for the students.

Implementation:
Theoretically (6) , the job of the code chopper is to generate possible sub sequences of "statements" from the original code. Foe that we will be needing a syntax tree. The pre order traversal of the tree is generated and then a sliding window of variable length is slid across to get the continuous sub sequences. However the current implementation takes the program's lexical representation and generates the sub sequences from that. However the fragments that have unequal number of bracket "{", "}" are eliminated using simple algorithms. The *code_chopper.cpp* if the file that take input any ASCII and gives the fragments in the vector of strings.

8.2 Code Transformer

As soon as you cut a code into it's fragments a new problem arises. For any of the metrics in the paper to be applied, there is a major requirement. The code should compile so that you can provide it input and see it's output behaviour, be it valid output, segmentation fault or any other runtime error.

The fragments therefore have to be converted to compilable programs. "How can you make a program compile?". Adding header files, declaration of data types etc where easy guesses. But stopping there can still lead to error's. One example could be missing semicolons in case of C++. Though this seems simple problem, the error it causes is significant. But if you think care fully, you will be give a program that compiles, you can never run to these issues if the code chopper breaks the program bases on valid statements. An original program, that has compiled as a whole, when broken down will have header file, data type declaration errors generally. If compilation error persists even after that just remove that fragment. The other consecutive fragments above or below will have most of the information it had.

Another thing to do is correctly identifying the input,output variables. We found an good approach in one of the reference papers (6). Take all the variables that exist in the fragment(using a lexer). Any variable that was non declared in the fragment before transforming is an input. Any one that is declared and used in a statement(updated/ used to update another variable) is an internal variable. Any one that is declared but not used is an output. Now you have found the effective inputs and output for a code fragment.

8.3 Code Filter

The code filter has to find similar codes from the fragments as representational similar code will be to a large extent functionally similar. Therefore it is easier to remove them before clustering. But taking structural or lexical similarity can be an overkill. If the code is lexically or structurally similar then to a large probability they were made by someone else. In online programming example that the paper took this is going to be very rare. No student would copy his code and then change it such that it is functionally same but textually different. We do not need to check explicitly for such cases as they will anyway be handled by the clustering algorithm.

In order to implement the above, we modified the string matching algorithm *codefilter.cpp* . We used suffix array to get the longest common prefix values and later use them to group up code that are having a certain amount of matching characters. The implementation takes *O(logn (nlogn))* for producing the result and can handle 50K lines of code. Any more that that is likely to be rare in a single program file for an online programming course. A representative from each group that the algorithm produces then goes into the code clustering

8.4 Test Generator

The paper talks about taking inputs and running them on two programs and comparing them based on the matrices that they provided. They do not talk about how they have figured out the mapping between programs variables. Internal variables can be skipped but what about input and output variables. Of course if you are comparing the same students submissions at different times probably all the variable names will remain same. But what if you are comparing it with another student. Therefore for the programs in comparison the input and output have to be mapped. A naive way is to permute over all inputs and check for set equivalence of outputs.

This can be computationally expensive. Thus we tried to find to simplify it. One of the foremost thing that you can do is to make input more general. You can give the same input to all variables and keep changing the value of that input to see the output behaviour. Programs that produce the same output set are likely to be functionally same. But Clearly this is a bad way to do it. You can write a program that gives one fixed input when all values are same but behave differently when at least one input value differs from the rest. We felt that it is better permute and check the possibilities if accuracy has a concern. Even this is not enough. Different programs can have different data types to represent the same variable. If one fixes a variable to be long there is no guarantee that the other program which as the same intended use fixes it to be long(lets say it was fixed to long long). But this is not mistake and you will have to accommodate this as well.

You can either type cast all the inputs to the largest data type or to the smallest data type. Each has it's own advantages and disadvantages. If type casted to larger group, then you will have more inputs representing the domain, but can slow down the clustering algorithm as well as cause runtime errors due to out of domain input which were handled differently. On the other hand if the inputs were type casted to smallest data type clustering algorithm is faster and less likely to have runtime errors as the input is valid for all variables. The issue with this is the fact that a program can behave the same for a specific range and can change the behaviour over something out of this range. The type casting to smallest data type will not be able to handle this. You can always group up the primitive data types and derived data types as separate and then permute which will give increase the efficiency.

8.5 Code Clustering

The paper assumed that it had been given sequences which are behaviourally similar and then they sorted it with respect to the metrics they came up with. How did they find behaviourally similar code in the first place? On closer inspection we can find that the problem they chose would already have the behaviourally

similar code in one place. All submissions to one program has to be behaviourally same to be qualified as a result. What if the codes were not grouped based on the question? How are we to find it? The code clustering becomes useful here.

One of the reference papers that the Microsoft paper provided does the following[reference]. It tries to execute all code fragments on one input first and partition them into smaller sets . Thus, the whole set can be gradually partitioned into functionally equivalent clusters with more and more inputs. Thus gradually forming clusters.

Our take on this was to find the pairwise similarity of all fragments initially using random sampling. Then using some clustering algorithm like k-means break them into cluster with the metrics results as the effective measure of clustering. Soon enough k-means was ruled out following the fact that finding the appropriate number of clusters become difficult. If reference programs are given the clusters can be approximated to the number of fragments produced from the reference programs.

9 Conclusion

In this report we introduced different forms of similarity between programs and did an in-depth analysis of a particular type of similarity: functional similarity. Measuring functional/behavioral similarity is particularly interesting not only because of it's complex nature, but also because of it's scope of usage in the real world. There are various metrics discussed in this paper to measure similarity between programs. Then, an algorithm to cluster fragments of behaviorally similar codes is discussed.

With the advance of information technology, online programming and software engineering education has gained a lot of popularity. Allowing large-scale classes while maintaining the quality of the education is challenging. Tasks, such as grading and providing customized feedback on programming assignments, require instructors to go through and understand students' code. This is just one of the real world problems that require "Measuring Similarity between Programs". The methods discussed in this paper can be applied to online programming websites, hint generation, reducing repeated code fragments in large code databases.

References

[1] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, Nikolai Tillmann *"Measuring Code Behavioral Similarity for Programming and Software Engineering Education"*

[2] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, Jurgen Wolff von Gudenberg, and Toshihiro Kamiya *"Similarity in Programs"*

[3] Chen T.Y., Leung H., Mak I.K. (2004) *"Adaptive Random Testing"*. In: Maher M.J. (eds) Advances in Computer Science - ASIAN 2004. https://doi.org/10.1007/978-3-540-30502-6_23

[4] N. Tillmann and J. de Halleux. *"Pex–White box test generation for .NET"*. In Proc. TAP, pages 134–153, 2008.

[5] N. Tillmann, J. de Halleux, and T. Xie. *"Transferring an automated test generation tool to practice"*: From Pex to Fakes and Code Digger. In Proc. ASE, pages 385–396, 2014.

[6] Lingxiao JIANG, Zhendong SU *"Automatic mining of functionally equivalent code fragments via random testing"* https://doi.org/10.1145/1572272.1572283

[7] K. K. Sharma, Kunal Banerjee, Chittaranjan Mandal *"A Scheme for Automated Evaluation of Programming Assignments using FSMD based Equivalence Checking"* https://dl.acm.org/doi/10.1145/2662117.2662127

[8] Hyun-Je Song, Seong-Bae Park, and Se Young Park *"Computation of Program Source Code Similarity by Composition of Parse Tree and Call Graph"* http://downloads.hindawi.com/journals/mpe/2015/429807.pdf

[9] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy *"Similarity in Programs"* https://drops.dagstuhl.de/opus/volltexte/2007/968/pdf/06301.SWM.Paper.968.pdf