PLAGIARISM DETECTOR

By

Team 9

Pallav Gupta
Siddhant Benadikar
Siddhant Pasari
Vivek Nair

Managing Software Development
Fall 2017
Northeastern University,
College of Computer and Information Science

Under the guidance of
Dr Frank Tip

SYSTEM OVERVIEW

Our system consist of two major components

- Front end that has been developed in HTML5, CSS, AngularJs
- Back end, which is developed in Spring boot and the main logic is written in JAVA.

The UI created is simple, intuitive and easy to use. It consist of two pages, the upload page where you can add and upload one or multiple files and the results page which displays the results as a percentage of confidence on how similar are the two files. The results page displays the percentages in a tabular format after running the **ASTSIM-LCS** similarity and the **Jaccard** Similarity measure algorithms.

We have even provided a **help** button the user incase he runs into any error and immediately the system admin will be notified through a mail sent by the user and also our system handles all possible edge cases on the client side itself including those of empty files and non .java files too.

PROGRAM DESIGN

Our Java program is divided into multiple packages.

1. The controller package contains the **RestController.java** which is the spring boot controller, used to communicate with the angular application on the front end. The RestController sends a response to the front-end which contains the output of our algorithm.
2. The detector package contains the main detection algorithm that is running on the uploaded files. We are using **Factory** design pattern that we learnt in class to control the instance creation of each of the classes contained in the detector package, this made our code more flexible as adding extra functionality or using different detection algorithms has become much easier. Now we would have to only make changes to Detect Factory class making these objects or we can even inherit this class and override its methods to provide additional functionality.

Initially we were applying DFS on the generated ASTs to make a list of the nodes and their hash values. But to implement ASTSIM-LCS algorithm we needed to standardize method names of the files. Thus, we found out that using Visitor pattern instead of DFS makes it much easier to do this. The Javaparser already contains an implementation of the visitor pattern, but it doesn't do much. So we overrode each of the visits to extended functionality to generate a list of nodes and their hash values. Using the visitor pattern made our code more flexible as now we can add unrelated functionality as the AST hierarchy always remains static. The hash values generated for each node by the Javaparser is very elegant as it calculates it as the summation of hash values of the child nodes plus the hash value of current node. This made the hash values remain consistent over similar programs, which was used in calculating the Jaccard Similarity percentage.

ALGORITHMS USED

We used two algorithms to test the Java files we input:-

1) **Jaccard Similarity algorithm**
2) **ASTSIM-LCS algorithm**

We started off with implementing the **Jaccard Similarity algorithm**. It requires two sets of hash values generated by applying the visitor on the ASTs. The algorithm is essentially the intersection divided by the union of these two sets of node hash values. The algorithm being this trivial, is susceptible to differences in variable names and methods as two different strings create different hash values and thus would be deemed different even though they have same underlying utilization. Even with its shortcoming, Jaccard has a very good precision for very similar files by students who don't bother to make multiple changes to make their code submissions to make them look different. Hence, for this reason we decided to leave the Jaccard's similarity algorithm in our system.

To overcome the limitations of the comparisons that are affected by change in string name, we implemented the **ASTSIM-LCS** which compares the node type instead of its label. In this algorithm we compare each method of first AST with every method of the second AST. If their labels correspond (which we made sure by standardizing the method names) , we use LCS(Longest Common Subtree) rooted at the two nodes. We do this by comparing each of the method bodies with one another by applying visitor on two ASTs rooted at r1 and r2, where r1 and r2 are root nodes of the subtrees that represent the method bodies in T1 and T2, respectively. Then we find the maximum size of an optimal alignment between the nodes in subtree 1 and subtree 2. The nodes only align if they share the same label or are of the same type. Moreover, the parent need to share the same label or the nodes v1 and v2 need to be labeled BLOCK. After we have found this maximum size, we can find the actual alignment between the two sequences subtree1 and subtree2. When we use the LCS algorithm, we can align two nodes $v \in V1$ and $w \in V2$ without aligning the parents of v and w, when both v and w are labeled BLOCK. For instance, the parent of v can be labeled FOR STATEMENT, while the parent of w can be labeled WHILE STATEMENT but since it is of type BLOCK the children are still compared. For each node $v1 \in$ subtree 1 we check if a node $w1 \in$ subtree2 can be mapped to v1. If that is the case, then M[v1] = w1. After having found the mapping M and the size |M|, we can calculate the similarity score between T1 and T2. The shortcoming of **ASTSIM-LCS** algorithm is that since we are only comparing methods with each other, the files uploaded that don't contain any methods and just constructors and attributes will have 0% similarity even if they are the exact same file. But in any real world scenario usually finding java files without a single method is very rare.

FEW RESULTS OF GIVEN SAMPLE DATA

SET NAME	File 1	File 2	ASTSIM-LCS SCORE	JACCARD SIM SCORE
Set 6	MyIntToBin.java	IntToBinCopy.java	55.0%	81.0%
Set 8	SolutionA.java	SolutionB.java	40.5%	60.0%
Set 11	Checker.java	Checker.java	97.0%	100.0%

- We can observe that the major difference between samples in set6, is **indentation**. Our LCS algorithm does not handle spaces and tabs, so it gives a relatively low percentage match, whereas Jaccard's algorithm takes care of these changes and hence gives a high matching percent of about 81.0%.

- In Set8, there are changes in the structure of the programs in both the samples. In Sample 1, the logic is written in a single main function whereas the Sample 2 solution is more modular but essentially with same function body. There are multiple differences in naming and field values, but we can see that our detector gives a 40 - 60 % match, which can give the professor a hint that the code might be plagiarized.

- The checker program in both the samples in set 11, is essentially an exact copy, hence we get a good match almost touch 100%.

WHAT WE LEARNT FROM THE COURSE

This course taught us many industry standards related to design and testing and also how a modern Agile team works. We tried to incorporate our learnings into this project. To follow the **Agile** workflow we met every two weeks throughout the sem to give an update on our project progress, ofcourse it was much more frequent before the deadlines. We used the **Factory** and **Visitor** design pattern as previously mentioned to make our code more flexible, less cluttered and also used OO design principles like **Abstraction** and **Encapsulation**, to name a few, to keep our code more modularized. Also the various testing techniques like the **Whitebox** and **Blackbox** testing were conducted on our code to make it as robust as possible. We wrote **unit** tests for each of the classes in the detector package to make sure it handles the edge cases and null values. We also ran our algorithm implementations in a Black Box setting to see if it gave us desired results for multiple test data created by us. As we merely extended the Visitor of **Javaparser** it did not seem necessary to test their Visitors. Also, there were multiple cases where we did not get desired results for the algorithms or the code we were using, thus **refactoring** was extensively used in our project. We made sure we performed **regression** testing on our code every time we refactored or added new functionality to our system.

REFERENCES

1. Detection of plagiarism in computer programming using abstract syntax by Olav Skjelkvåle Ligaarden, pg 66: **(ASTSIM-LCS)** https://www.duo.uio.no/bitstream/handle/10852/9811/Ligaarden.pdf?sequence=1&isAllowed=y
2. An AST based Code Plagiarism detection algorithm by Jingling Zhao, Kunfeg Xia, Yilun Fu, Baojiang Cui