Drew VanAtta & Jake Bailey

CS4100 – Project 1 – Plagiarism Detector for C code

Regular Expressions and Tokenization

For our project we identified 10 tokens, and 15 regular expressions:

TKVARNAME 1

_[a-zA-Z0-9] //for variable names that start with underscore

[a-zA-Z0-9]+[\n]* //for all other variable names

This token was used to assign variable names into, because students could change those to anything to try and hide plagiarism. We output “aa” for the token (we tried to do 001, but eventually ran into compiler problems thinking that 009 was an invalid octal number, so just stuck with characters)

TKFUNCTION 3

[a-zA-Z]+[\n][a-zA-Z_]+[(] //function empty declarations

[a-zA-Z_]+[\n]*[(] //function call

This token was assigned for functions to hide function names that could be changed. The output from this token was “ad”.

TKTYPE 4

{T}[\t\n\f] //T is a macro with all C data types

This token was made to assign all C data types, because we thought it might be possible for a student to change from data types such as an int to double, trying to hide the plagiarism. This token output “ac”.

TKVAL 5

[0-9][;] //integer values

["|'](\[^"\n]|\\(.|\n))*["|'] //Finds string literals, "values" given to strings

[{D}].[{D}] //floating point values

This token was created for the values that appear in code, such as integer values, string literals, and floating point values. Again, it pairs with the token because students could change an int to a floating point value very easily. We originally has a token for string values, but it just got combined with this one. The output from this token was “ae”

TKCOMMENT 6

"/*"([^*]|*+[^*/])**+"/" //multi line comment

"//"(.*)   //single line comment

This token was created so that we could ignore comments that students could add to try and fluff the code and make it look different. Nonetheless, the comments are unimportant towards the plagiarism process so we just ignore them.

TKINCLUDE 7

"#"(.*)   //picks up on any # (#include, #define, #pragma, etc)

We created this token when we were testing on the files because we realized none of the other regular expressions would pick up on the includes at the start of the files

TKLOOP 8

[while|for|do]+[\n][(]   //picks up anything students could change out to hide plagiarism

During the experimentation process with the files we also noticed that we needed a token and regular expression for the loops in the files. We included all of them into one token because students would be able to slightly change a loop to appear not plagiarized. This token output "af"

TKIF 9

if[\n]*[(]

TKELSE 10

else[ ]if[ ]*[(]

These two tokens go hand in hand, again during the testing process on the actual files we noticed there were if statements and else that needed to be tokenized. The TKIF token output "ag" while tkelse output "ah"

TKIGNORE 255

.

Finally we have our ignore token, that just skipped over everything else that was unimportant in the files (such as ;, or any other value that wasn't picked up in other tokens).

We also had a lone regular expression:

[\t\n\f]   {   Addline(yytext[0]);   }

This regular expression was mainly for lex to be able to go through the files

3. The results of your analysis, including identifying any submissions found by the algorithm that

you believe may be plagiarism

Implementation

To implement the Winnowing algorithm, we first had to read in our string of continuous tokens that was created during the bashing script into tokens.txt. We created a vector of sets that contained ints, so that we could hash and fingerprint in the same function.

A for loop went for the number of files, calling our generateKGrams function on each tokenized file. The generateKGrams function would first generate each kgram, then immediately hash it and store it in a vector of ints. It did this with a double for loop, first one looping through the token string size minus the kgram length, so that we stay in bounds when creating the last k-gram. The second loop was to hash the k-gram, which it accomplished by looping through the length of the k-gram, and for each character adding its value to a placeholder, and then multiplying the placeholder by the loop increment + 1 (to avoid multiplying by zero). The final value was then taken mod 1007 for our final hash value.

After all of the kgrams were created, hashed, and stored, we created a set of ints to store our fingerprints. Our function then went through a double for loop. The first loop went to through the size of the kGram vector minus the window size, so that we could create the appropriate window size without going out of bounds (similar to how we created the k-grams). The second loop would go through the window size, and check for the new smallest and add it to the set.

Because of the nature of sets, we didn't have to check for adding the same smallest number repeatedly. Essentially, we searched through our window of k-grams and searched for the smallest hash in each window, and saved it to the set, resulting in our final fingerprint set to represent the file. To compare each file, we found the intersection of each set (the hashes that were in both sets representing its respective file) and compared every file against each other to see how many were similar. We only output the matches from files a to b that contained at least a 75% match between the fingerprints.

Results

During our testing, we played around with our k-gram size and window size a bit, and found our sweet spot was k=11 and w =21. With these values, we were able to find 346 matches between file comparisons that were at least 75% similar, with 47 of them being 100% matching fingerprints (though 28 are outliers and will be explained later). We also found that the function that had the biggest factor in the files was the dispense_bills (or whatever the student named it), as that was the function that seemed to allow for the most variation on how they could do it, and if it matches it was a dead giveaway.

The biggest group of guaranteed matches was files 3, 9, 53, and 54. After checking each of these files, we concluded that they are most definitely plagiarized as they are near exact copies just with minor changes that you wouldn't even need this program to see. Another group of 100% matches was 6 and 15, which after review, also seemed to be very similar and likely to have been plagiarized off each other as well, as the only change are the variable names and how they make the code look, but the math in the dispensebills function on both does the same exact thing. File 22 also matches 100% of hashes when compared to three other files: 8, 25, and 47. After review, the file definitely matches with file 47, but 25 and 8 are a bit different, but the math behind the function is the exact same. File 49 also matches very highly with these files, as it has the same math in its dispenseBills functions that the others also share. Files 6 and 15 were 100% copies off each other, and separately files 30 and 10 were definitely copied off each other as well.

This leads to the last group of 100% matches, file 44 to a bunch of other files. This is because file 44 isn't really a real c file, and just some script or something of the output of what the file would have looked like running (which we found hilarious). The file literally only had one hash value (leading to a bunch of 100% matches) because our lex program was looking for c code, not the output of some guy's c code.

We also noticed that our results aren't always 100% accurate. For example, we know that the group of files 22, 8, 25, and 47 are copies of each other, but 47 compared to 22 yields 94.44% similarity in the fingerprints, while it matches 100% with file 8. This shows that there are at least some inconsistencies between the hashes, but that's to be expected. Not every single fingerprint will match, but if a high percentage of them do, its very likely that they are copies.

NOTE: program output is in the PlagarismReport.txt file, adding it here would've made this 16 pages instead of four