

Fingerprinting Jar Files Using Winnowing

Cate Huston

Abstract—We applied the use of winnowing to fingerprint jar files, and attempted to detect jars that contained or had significant similarities to other jars. This approach was found to be effective; identical jar files and jar files included within other jar files were successfully flagged.

Index Terms—document fingerprinting, jar files, n-grams, winnowing.

I. INTRODUCTION

We attempt to use a small modification of a known algorithm with some additional criteria to attempt to detect potentially suspicious similarity between Jar files. Since Eclipse is released under the Eclipse Public License, it is important that contributed code of an incompatible license – such as the GNU Public License – is not included.

The goal is to identify potentially suspicious similarities between a set of 20 jar files. A series of methods were produced that compare aspects of two jar files, such as size, number of files, filenames, and finally fingerprints, similarly to the algorithm described in [1].

II. BACKGROUND

A. Related Open Source Software

The software described in the paper [1] is not Open Source. It is available for non-commercial use, free of charge, but it is patented and commercially available through Similix [2]. MOSS only analyzes code in certain languages, listed below [3], and does not accept jar files.

C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, MIPS assembly, HCL2.

JPlag [4], an alternative plagiarism detection service that operates in a similar way to MOSS [3] in that it runs on a remote computer. They do not give out accounts to “anonymous email addresses like Hotmail, Yahoo, Gmail, etc” and each application is reviewed [4]. JPlag is based on the “Greedy String Tiling” algorithm [5] detailed in [6]. Due to

Manuscript received November 4th 2008. Fingerprinting Jar Files Using Winnowing.

C. E. Huston is with the School of Information Technology and Engineering, University of Ottawa, Ottawa, ON K1N 6N5, Canada (e-mail: chust056@site.uottawa.ca).

the use of a “tokens”, JPlag only supports Java, C#, C, C++, Scheme and natural language text [4]. YAP [9] is based on the same algorithm but is designed to run locally. Although the code is released only non-commercial use is allowed as per the README in the download [9], and thus it is not Open Source either. It supports only single file submissions [13]. Plaggie [12] is similar to JPlag in functionality and UI, but is run locally. It is released under GPL, and supports only Java code [12].

SID [11] is a modification of a genome comparison algorithm. It works by comparing the amount of shared information between two programs [11]. It is not open source, and files must be submitted to the server in a “carefully formatted zip file” [11].

Sherlock [8] and SIM [10] do not advertise licenses, but make the source code and instructions for running the program on your home machine available. Sherlock formerly worked in two steps, first by taking signatures and then by comparing them, however it has since been modified not to. The original implementation is still available on the website [8]. SIM accepts only single file submissions and is quite specific to the local situation at Vrije Universiteit [13].

AC [7] is a GPL-licensed, stand-alone program that supports C, C++ or Java. It “incorporates multiple similarity detection algorithms found in the scientific literature” and displays results graphically [7].

The majority of these programs are unsuitable for this application as they compare the files. Our goal is to compare fingerprints of the files. There is a crucial difference here, as we may not wish to store all the jar files of all possible jars that we may wish to compare against. Storing fingerprints has a far smaller overhead.

III. APPROACH

A. Design

The Winnowing algorithm is very simple. Spaces are removed from text, and then “n-grams” are hashed. Not all hashes are stored, just the smallest hash in each window. If the first hash in each window were stored, adding an extra character at the start of a file would result in no matching fingerprints being found for otherwise identical documents. The approach of taking the lowest hash overcomes this issue [1].

Some other potentially interesting information about the jars is stored and compared. As well as comparing fingerprints (for

files with the same extension only), we also compare filenames, size of the jar file, the number of entities in the jar file, and the name of the jar file itself.

B. Decisions Made

Although the MOSS implementation of this algorithm removes information like variable names [1], this was not included in this implementation. Whilst in student assignments it may be trivial to alter variable names, this is less likely to be the case where an entire jar file has been included. Further, the .java files themselves make up a small part of the jar and may not even be included in it. Whilst it may be worth including a tokenizer and following this approach, it does not immediately seem worthwhile to implement it.

In order to improve performance, java library methods, for example the hashCode() method, were used wherever possible as these tend to be highly optimized.

IV. RESULTS AND VALIDATION

A. Overall Results

The jar files tested fell into 3 groups. Firstly, there were 6 applets built with Processing [14], and 3 Processing libraries [14]. Secondly, there were 5 Substance Look and Feel [15] jars, one duplicated, and an application built using Substance. 5 miscellaneous jars (the third group) brought the total up to 20.

Table 1 shows the top 36 results, or all those comparisons that returned a similarity percentage of over 25.00%. At number 1, it can be seen that the program correctly determined which version of Substance Java Look and Feel [15] was duplicated.

Results 2-11 are the comparisons of the various combinations of the jars created using Processing [14]. 12-16 are 5 of the jars created using Processing [14] and the “core” library imported by all of them, the 6th appears at number 34 (this one is rather larger and more complicated).

At number 17, with 84.62%, the version of Substance [15] used in the application built using [15] is correctly identified. At 22 and 23, the previous and following version of Substance [15] both generate ratings of about 50%. 18-21, 24-25, and 28 are the comparisons of the various versions of Substance [15] against one another.

The remainder of the results, not shown in the table, ranked less than 25.00%. None of the expected comparisons returned a result of less than 25.00%; similarities at this level appear to be incidental and un-interesting, likely as a result of both jars using Swing libraries or similar reasons.

Table 1: Comparison of Results over 25.00%

	Jar 1	Jar 2	Result
1	substance.jar	substance5-0.jar	97.50
2	animatedweb.jar	hungry_pink_blob.jar	91.81
3	Birds.jar	PushPopCubes.jar	91.11
4	CubesWithinCube.jar	PushPopCubes.jar	91.09
5	Birds.jar	CubesWithinCube.jar	91.04
6	animatedweb.jar	PushPopCubes.jar	89.03
7	animatedweb.jar	Birds.jar	88.96
8	animatedweb.jar	CubesWithinCube.jar	88.96
9	hungry_pink_blob.jar	PushPopCubes.jar	88.91
10	Birds.jar	hungry_pink_blob.jar	88.88
11	CubesWithinCube.jar	hungry_pink_blob.jar	88.86
12	core.jar	hungry_pink_blob.jar	87.85
13	animatedweb.jar	core.jar	87.72
14	core.jar	PushPopCubes.jar	87.59
15	core.jar	CubesWithinCube.jar	87.52
16	Birds.jar	core.jar	87.30
17	minesweeper.jar	substance3-0.jar	84.62
18	substance4-0.jar	substance5-0.jar	66.40
19	substance.jar	substance4-0.jar	66.33
20	substance.jar	substance3-0.jar	61.94
21	substance3-0.jar	substance4-0.jar	54.19
22	minesweeper.jar	substance4-0.jar	51.99
23	minesweeper.jar	substance2-0.jar	50.33
24	substance3-0.jar	substance5-0.jar	47.29
25	substance.jar	substance2-0.jar	47.22
26	minesweeper.jar	substance.jar	45.62
27	minesweeper.jar	substance5-0.jar	45.62
28	substance2-0.jar	substance3-0.jar	41.39
29	animated_sierpinski_triangle.jar	animatedweb.jar	31.64
30	animated_sierpinski_triangle.jar	hungry_pink_blob.jar	31.48
31	animated_sierpinski_triangle.jar	PushPopCubes.jar	31.29
32	animated_sierpinski_triangle.jar	Birds.jar	31.27
33	animated_sierpinski_triangle.jar	CubesWithinCube.jar	31.27
34	animated_sierpinski_triangle.jar	core.jar	30.44
35	substance2-0.jar	substance4-0.jar	29.97
36	substance2-0.jar	substance5-0.jar	28.09

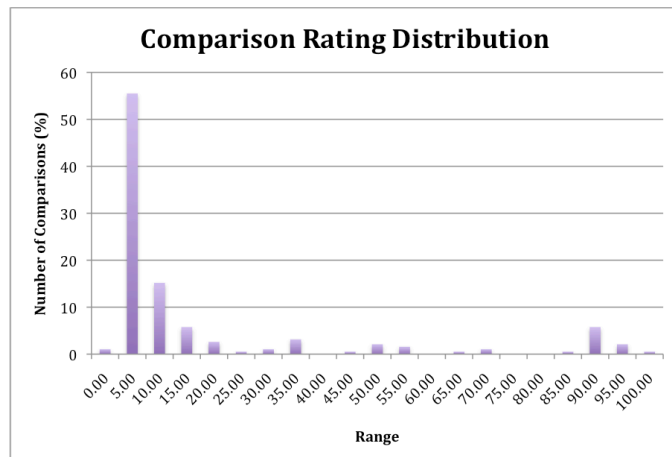


Figure 1: Comparison Rating Distribution

The overall distribution is shown in Figure 1. It is clear that the vast majority of comparisons are uninteresting, with over half of the comparisons generated falling in the range of 5-10%. What is also apparent is some clustering of the data.

B. The Effect of the Different Factors Compared

The predominant factor in the comparison result for two jar files is the number of overlapping fingerprints, which is given a rating of 65%. The weights of the different metrics are shown in Table 2, below.

Table 2: Metrics and Weightings

Metric	Weight
Name of Jar	2.5
Number of Files in the Jar	2.5
Size of the Jar	5
Filenames of Entities in the Jar	25
Fingerprints or Jars with corresponding file extensions	65

In Figure 2, we can see how the different metrics being considered are scored for the top ranking comparisons (numbers are the same as in Table 1). We can see that the more highly ranked jar files score highly on fingerprints, filenames, and number of files with lower ranking files being less consistent. Files 29-34, for example, score more highly on filenames than they do on fingerprints. Number of files is only a factor on higher ranked comparisons. No comparison ranks on name, and only the top ranking comparison (the duplicated file) ranks on size.

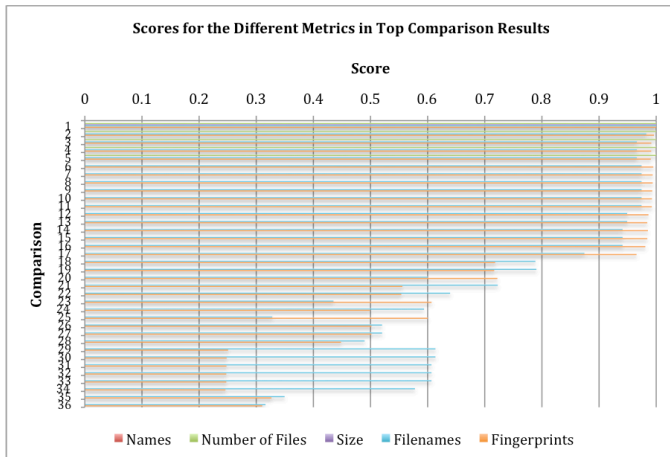


Figure 2: Scores for the Different Metrics in Top Comparison Results

The breakdown of the score is shown in Figure 3. The importance of each metric in the eventual score is shown in Figure 4. We can see that the importance of the fingerprints is fairly consistent at around 60%, with the exception of comparisons 29-34 where the filenames make up a larger proportion of the eventual score. Note that these comparisons are all combinations of applets built using Processing. They share the same core library (core.jar) but other than that the

code is fairly distinct. Note that this additional code makes up a fairly small proportion of the jar.

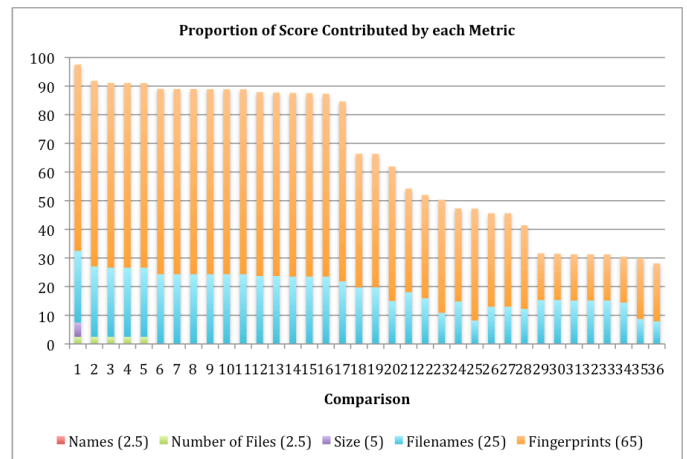


Figure 3: Comparison Score by Component

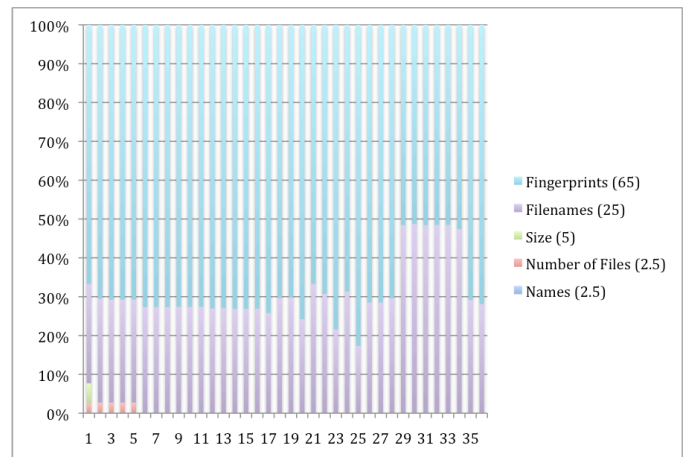


Figure 4: Proportion of Score Contributed by Each Metric

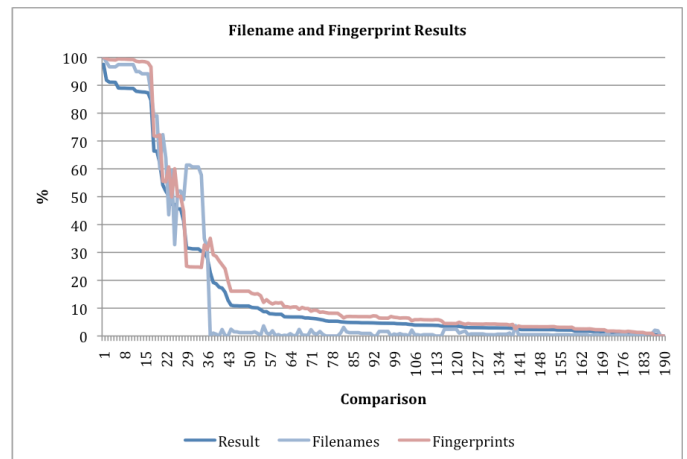


Figure 5: Filename and Fingerprint Results

Figure 5 shows the relationship between the total result, and the results for filenames, and fingerprints. The cutoff for interesting results was 25%, and in this graph we can see how that the result follows the fingerprint percentage with no strong correlation with the filenames. Above 25% the graph has more variation.

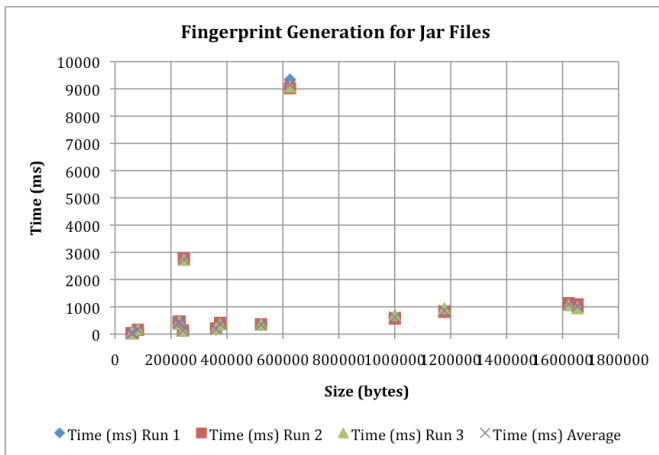


Figure 6: Time vs. Size

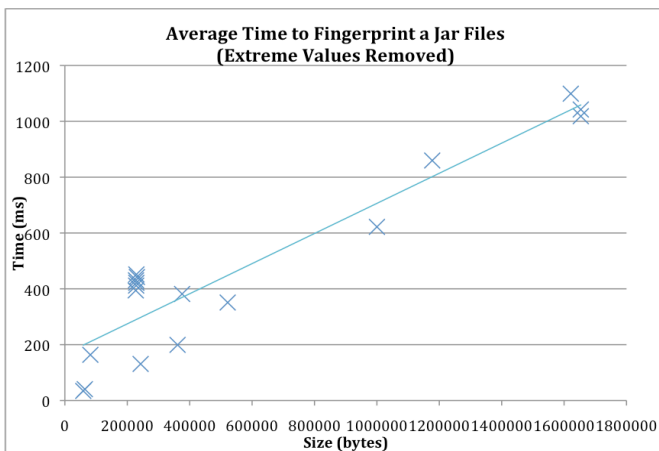


Figure 7: Average Time to Fingerprint a Jar File (Extreme Values Removed)

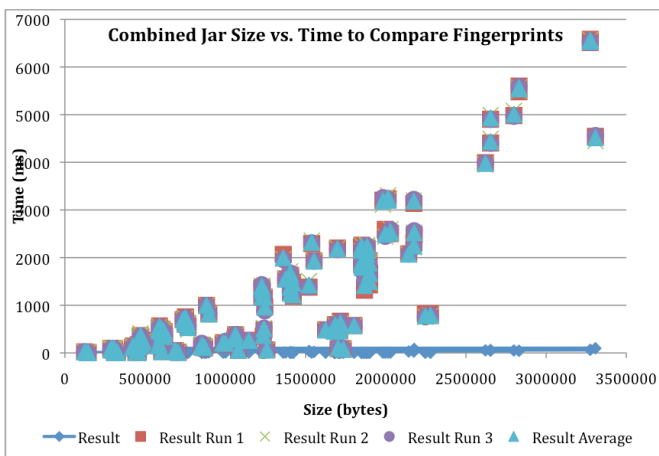


Figure 8: Combined Jar Size vs. Time to Compare Fingerprints

A. Running Time

The running time of the algorithm is approximately linear. As shown below in Figure 6, the length of time to fingerprint the jar typically increases with size, with few exceptions. Time values were obtained running on a 1.8Ghz MacBook Air and

are fairly consistent, with a Standard Deviation of 29.88. There are two extreme values in this data; it is likely these jars have a large number of entities in them. Average times for the other 18 jars are shown in Figure 7, where the linear relationship is shown more clearly.

The running time for fingerprint comparisons was linear for smaller jar files was not linear, as can be seen in Figure 8, but with an upper bound of $O(n^2)$ as shown in Figure 9. Again, results are fairly consistent.

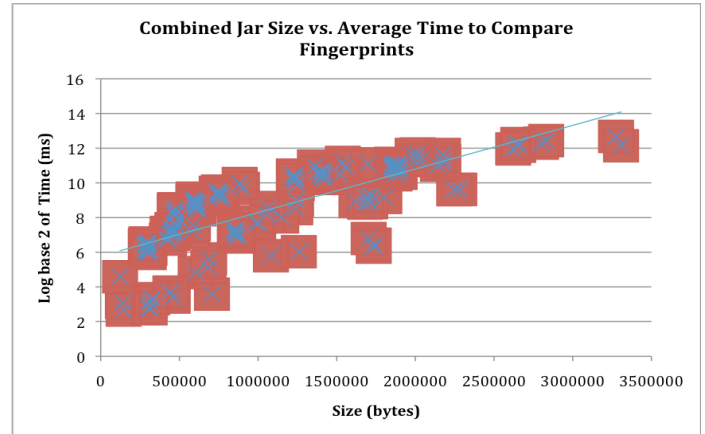


Figure 9: Combined Jar Size vs. Average Time to Compare Fingerprints

V. CONCLUSION AND FUTURE WORK

A. Conclusion

Fingerprinting using the Winnowing method [1] appears to be an effective way to detect potentially suspicious similarities between jar-files. Using the test file, there were no false-positives ranking above 25.00% and no false-negatives ranking below 25%. With the data used in this case, 25% was an effective boundary below which similarities were incidental and above which similarities were expected given the data, and more suspicious ones did, for the most part, rank higher. The duplicated file was clearly detected.

From the results, we can conclude that whilst comparison of entity names within the jars is useful, it is not clear that the comparison of the other metrics considered is. These could likely be disregarded for general purpose comparisons.

Performance is good, and almost linear for smaller jar files, although with larger jar files the exponential relationship is noticeable. This is mitigated, however, as the implementation only compares files within a jar against those files in the comparison jar with the same file extension. This should mean that whilst bounded by $O(n^2)$ the runtime should be significantly less (provided the jar contains a mix of file types). The longest amount of time to generate a fingerprint for a jar file was less than 10 seconds, and no comparison took more than 7 seconds. Given that fingerprint generation has a

significant overhead, the option to store the fingerprint as a string will potentially be helpful.

Although performance could likely be further improved this may necessitate using an alternative programming language. A functional programming language might work well in this context, but this may be incompatible with the goal of creating an eclipse plug-in. Java also has more options in terms of graphical display of the data.

B. Future Work

One of the limitations of this implementation is that it returns just one number for each comparison. The MOSS implementation [3] returns 2 - each files similarity against the other. This would be helpful in this situation as we would be able to see where a jar had been wholly included (as with the sustance3-0.jar in minesweeper.jar) which is not currently the case. A relatively small jar included in a relatively large one may therefore rank lower than it should.

For the prospective application of this program, it is necessary to determine a baseline similarity between Eclipse plug-ins. It would also be helpful to determine whether the shape of the graph in Figure 5 is typical; if so, then this relationship could be used to determine the cutoff for interesting similarities dynamically.

REFERENCES

- [1] 1. Schleimer, S, Wilkerson, D. D, Aiken, A. Winnowing: Local Algorithms for Document Fingerprinting. SIGMOD 2003. Available at <http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf>.
- [2] 2. Similix, Available at <http://www.similix.com/>. Last retrieved October 29th, 2008.
- [3] 3. MOSS: A System for Detecting Software Plagiarism. Available at <http://theory.stanford.edu/~aiken/moss/>. Last accessed October 29th, 2008.
- [4] 4. JPlag: Detecting Software Plagiarism. Available at <https://www.ipd.uni-karlsruhe.de/jplag/>. Last accessed November 1st, 2008.
- [5] 5. Prechelt, L, Malpohl, G, Phlippsen, M. JPlag: Finding plagiarisms among a set of programs. March 28th, 2000. Available at <http://page.mi.fu-berlin.de/prechelt/Biblio/jplagTR.pdf>. Last accessed November 1st, 2008.
- [6] 6. Wise, M. J. String similarity via greedy string tiling and running Karp-Rabin matching. December 1993. Available at ftp://ftp.cs.su.oz.au/michaelw/doc/RKR_GST.ps. Last accessed November 1st, 2008.
- [7] 7. AC: An anti-plagiarism system for programming assignments. Available at <http://tangow.ii.uam.es/ac/>. Last accessed November 1st, 2008.
- [8] 8. The Sherlock Plagiarism Detector. Available at <http://www.cs.su.oz.au/~scilect/sherlock/>. Last accessed November 1st, 2008.
- [9] 9. Plagiarism Detection - YAP. Available at <http://luggage.bcs.uwa.edu.au/%7Emichaelw/YAP.html>. Last accessed November 1st, 2008.
- [10] 10. The software and text similarity tester SIM. Available at <http://www.cs.vu.nl/%7Edick/sim.html>. Last accessed November 1st, 2008.
- [11] 11. SID - Plagiarism Detection. Available at <http://genome.math.uwaterloo.ca/SID/>. Last accessed November 1st, 2008.
- [12] 12. Plaggie. Available at <http://www.cs.hut.fi/Software/Plaggie/>. Last accessed November 1st, 2008.
- [13] 13. AC: Other Systems. Available at <http://tangow.ii.uam.es/ac/?q=other-systems>. Last accessed November 1st, 2008.
- [14] 14. Processing. Available at <http://processing.org/>. Last accessed November 4th, 2008.
- [15] 15. Substance Java Look and Feel. Available at <https://substance.dev.java.net/>. Last accessed November 4th, 2008.