Java Archive Copy Detection using the Jarprint Generator

Andrew Runka, M.Sc.

***Abstract*—The paper introduces the Jarprint Generator: a copy detection application for Java Archives. The Jarprint Generator uses a Genetic Algorithm (GA) to determine a best-estimate of the highest similarity between two presented archives. This paper covers the implementation details of this application including a description of the GA used. Two phases of testing are performed to first ascertain that the application covers the base functionality required of a copy-detection system, and secondly to determine the performance quality of the system. The results suggest that the system is an effective, consistent, and reasonably efficient means of copy detection.**

I. Introduction

The Internet is an invaluable resource of knowledge and information, and although freedom of information is a desirable goal in this medium, protection of the rights of the authors who provide that information is paramount to maintaining the Internet as an environment suitable to the development of new knowledge. Copyright infringement, the unlawful copy or use of protected information, threatens the rights of these authors with the potential to turn their works into something they did not intend and do not condone. Copyright infringement has been a major issue in software development due to the simplicity of copying and distributing electronic information. This is further exacerbated in the domain of open source software development, where raw source code is freely available for all to see and copy. While often the licenses of this software dictate free use of this source code to all who would like it, copyright infringement may still occur when copied source code is used without adhering to the restrictions of the open source software license. The aim of this project is to develop a means of comparing two java archives (jars) to determine the level of similarity between them. This project uses a two stage approach to determine similarity. First, each class in each jar is given a fingerprint based on its composure. Secondly the class fingerprints are compared using a simple difference calculation to determine their similarity. In order to ensure that the class fingerprints are matched ideally between jars, a search of possible comparisons is performed to determine a best-match estimate of the organization between the two jar files. In this way the system is designed to detect jar files which have been slightly modified (altered, reorganized, extended) as well as jar files which have been included into larger packages. The number of permutations of class file orderings within a given jar file is exponential with regard to the number of class files. To search such a space deterministically could be quite costly as the size of the jar file increases. To this end, a Genetic Algorithm (GA) is employed a to stochastically search the solution space and provide a good estimate of the best ordering within a reasonable amount of time.

The remainder of this paper is structured as follows: Section II details the background information of both the Copy Detection problem and the Genetic Algorithm framework. Following this, Section III explains the system implementation and experimental details used in this study. Next, Section IV covers the presentation and discussion of the results that were found, and finally Section V presents the conclusions of this study.

II. Background

A. Genetic Algorithms

This section introduces the concept of the Genetic Algorithm. For an explanation of the implementation details for this project see the next section.

Genetic Algorithms (GAs) are a class of search heuristics, first popularized in 1975 by Holland et al. [1], and have since been applied to a vast assortment of problems [2], [3], [4], [5]. The fundamental idea of genetic algorithms is to iteratively evolve solutions using a conceptualized form of natural selection. Starting from a population of random solutions to the problem, GA effectively culls the weaker sub-optimal solutions, so that the more optimal solutions will continue to evolve towards the global-best. The advantage of a GA in these problems arises from its stochastic nature, which facilitates a balance between exploration and exploitation of the search space, allowing it to move quickly towards good areas of search while avoiding being trapped in local optima. GAs are commonly employed for combinatorial optimization problems where a deterministic search strategy would be too computationally expensive.

A pseudo-code framework for GA is presented in Figure 1. Initialization of a GA consists of creating a random population of solutions or 'chromosomes', which are each representative of some valid yet typically unsatisfactory solution to the problem at hand. An appropriate, problem dependent, representation must be selected for the chromosomes, such as bit strings or integer arrays. After initialization, the population of individuals are each evaluated according to some fitness function which ranks their solution to the problem at hand. Following this a new population of individuals are selected. A common means of performing the selection operation is with Tournament Selection. Here, k individuals are pulled at random from the population, and the best of those k is selected to be in the new population. The best solution found throughout the entire run is maintained using a technique called elitism. This dictates that the best solution is carried from generation to generation without any alteration. Once the new population is selected, it undergoes some means

of recombination. Recombination operators are broadly divided into two categories, crossovers and mutations. Crossover operators involve combining the solutions of two 'parent' solutions to create 'offspring'. Mutation operators involve a single individual and serve to further explore the area of the search space that is local to a given solution by applying small changes. The main evolutionary loop continues until some termination condition (typically a maximum duration of generations) is met.

```
GENERATE initial population
REPEAT
        EVALUATE population
        SELECT parents
        RECOMBINE parents to produce offspring
        JOIN offspring to form new population
UNTIL termination condition satisfied
```

Fig. 1. Pseudo-code of the Genetic Algorithm.

III. Experimental Setup

A. Implementation

The implementation of the Jarprint Generator begins with first dissecting the class files from the jars presented. This is done using the java zip library. Classes are then parsed using the java reflect library. This had the unfortunate requirement of forcing all jar files to this system to be complete, that is, the system will throw exception to undefined references in a jar file. For each class a fingerprint is generated based on the following information:

- The number of each field modifier (public, private, etc),
- the number of each field type (short, int, etc),
- the number of each type of constructor argument,
- the number of each method modifier,
- the number of each method return type, and
- the number of each type of method argument.

The collection of all class fingerprints for an entire jar is called a 'jarprint'. This makes for a rather large array of integers, so for human readability each class fingerprint is shortened down into a hexadecimal sum of its parts, making the jarprint display as a list of hexidecimal pairs. Using this information in the jarprint immediately enables the system to detect similarity beyond obfustication in the form of renaming classes, methods or fields, as well as from the reordering of information within a class that may mislead text-based similarity measures.

In order to determine a similarity comparison, two jar files undergo the parsing process. Both jarprints are then fed into a GA to determine the best matching between classes. The larger of the two jarprints (or a random one if both are the same size) is kept aside for evaluation purposes, while reorganizations of the smaller jarprint constitute the search space of the problem. In this way the implementation determines the greatest possible similarity between the two jar files giving it the ability to overlook obfustication in the form of reordering the jar file or the removal or addition of classes.

Each individual in the GA population is initialized as a permutation of the order of the class fingerprints from the smaller jarprint. GA individuals are padded with empty fingerprints to make the smaller jarprint equal the size of the larger jarprint if necessary. This is done so that evaluation of the similarity between jarprints can be done on a class by class basis. Each ordering (or GA individual) is compared with the larger jarprint and given a difference score. This is simply the sum of all differences for each element in each class fingerprint, as seen in formula 1.

$$fitness = \sum_{c=0}^{C} \sum_{e=0}^{E} |J_{c_e} - j_{c_e}| \tag{1}$$

where *J* is the larger jar file, *j* is the smaller jar file, *C* is the number of classes in J and *E* is the number of elements in class c. Padded classes are ignored in this calculation.

Once each individual is evaluated, selection proceeds using the tournament selection method. The best match found throughout the duration of the search is maintained using elitism. Each individual is then modified using the swap mutation operator, where two class fingerprints in a solution exchange locations. No crossover is utilized in this implementation. Execution proceeds to loop through these operations until a set number of generations has passed or until an exact matching (zero difference score) has been found.

One final yet important note on the GA implementation: in an effort to improve the speed of convergence of the GA, class fingerprints that have been lined up as exact matches are 'locked' in place. This means that those classes (or alleles) are exempt from further mutation. This reduces the amount of redundant mutations by narrowing down the search space over time consequentially leading to increased speed and consistency.

Once the matching is complete, the final similarity percentage is calculated from the best difference score found via formula 2. This is done by normalizing the difference score over the sum of all values in the smaller jarprint. While this does have the potential to be greater than 1, it is assumed that differences that large denote distinct jar files and are given the value of 100% difference. The similarity percentage then the inverse of the difference percentage.

$$Similarity\% = 1 - min(1, \frac{best\_fitness}{\sum_{c=0}^{C} \sum_{e=0}^{E} j_{c_e}}) \tag{2}$$

where *best_fitness* is best result returned from the GA calculated as in formula 1, *j* is the smaller jar file, *C* is the number of classes in j and *E* is the number of elements in class c.

B. Experiments

Experimentation was broken down into two phases. The first phase performs the initial testing in order to ensure that basic requirements of the system are met. The second phase is used to determine the accuracy, consistency, and efficiency of the system.

Initial testing of the system was performed to test that the system meets the basic requirements for copy detection software. That is, it grade similar files with a high similarity, while grading dissimilar files with a low similarity score. Testing at this phase involved 3 main similarity comparisons:

1) Using two identical jar files
2) Using two distinct jar files
3) Using two similar or contained jar files

Ideally the system should conclude 100% similarity in case 1, a low similarity (0%-50%) in case 2, and a high amount of similarity (greater than 80%) in case 3. The jar files used in these tests are actual jar files containing libraries or applications worth of class files. This was done in an effort to demonstrate the real life applicability of this implementation.

The second phase of testing involved the creation of a series of archives. These files were filled with class stubs, that is classes with methods, constructors and fields, but no operable code. This was done in order to have a highly customizable array of archives tailored to the needs of testing. The testing in this phase is aimed at gaining insight to the performance of the implemented system. Firstly to test the accuracy of the system, 50 randomly generated classes were created and partitioned into 25 archives of 25 classes each. This was done as follows, the first archive contained the first 25 classes. The second archive contained the classes 2-26, the third contained 3-27, and so on until the final archive which contained classes 26-50. In this way each successive archive was then a single swapped class different from the previous archive. This array of archives was then used to determine the accuracy by comparing the first archive to every other archive.

Since a stochastic mechanism is used to determine similarity it is important to determine how consistent the results are in order to gauge the reliability of a single comparison. To do so several archives of ranging in size from 50 to 500 classes were generated and compared to determine the amount of variance from run to run. At the same time, the duration for comparison of these files is used to examine the relative speed with which these archives can be compared.

IV. Results and Discussion

All experiments were written in Java and run on a Core i7-820QM Quad-core mobile processor with 4GB RAM in a 64-bit Windows7 environment. All results are averaged over 15 runs.

A. Phase 1

Results begin by testing the basic requirements of the system. In the simplest case, the system should detect that two identical files are 100% similar. To test this, three jar files were selected. The first is the jar file for the ThoughtStack application [6] containing 30 classes, denoted in shorthand as TS(30). The second jar file is the implementation of this project itself (Jarprint Generator), containing 58 classes, shortened to JPG(58). Third is the jar file for the ECJ library [7] containing 403 classes, denoted ECJ(403). The results of this experiment are displayed in Table I.

Jar1	Jar2	Similarity	Duration
TS(30)	TS(30)	100%	1 sec
JPG(58)	JPG(58)	100%	4 sec
ECJ(403)	ECJ(403)	100%	263 sec

TABLE I
Results of comparing identical jar files

All three files were matched to 100% similarity when compared to themselves, demonstrating the effectiveness of this strategy against the base-case comparison. The duration to do so, however, increases exponentially with the number of classes in the jar file. This is largely due to the fact that the swap operator employed here moves a single class at each operation. So while it is a constant time operation, it must be performed many more times in order to accommodate the increased size of the search space. The issue of search speed (efficiency) is addressed in the results of the second phase of testing later in this section.

The second part of phase 1 testing requires the developed system to distinguish between two distinct jar files. That is, given two independently developed (non-copied) jar files the similarity comparison should result in a low score. The same jar files as above are used here. The results of this experiment are displayed in Table II.

Jar1	Jar2	Similarity	Duration
JPG(58)	TS(30)	20.63%	13 sec
ECJ(403)	TS(30)	23.11%	55 sec
ECJ(403)	JPG(58)	22.68%	79 sec

TABLE II
Results of comparing distinct jar files

All three comparisons between distinct jar files resulted in low similarity scores. A score of zero is unlikely due to the fact that all classes share the same basic structure, thus it is expected that some similarity is seen between large groups of classes. The maximum execution time in this set was below 2 minutes, which is a bearable duration for large comparisons that would be required only rarely in practice.

The third experiment in phase 1 concerns detecting similar material. To this end three versions of the Jarprint Generator (labelled JPG1, JPG2, and JPG3[1] in order of their release) were set aside throughout its development and are compared here. Alterations between versions of the Jarprint Generator consist of modifications to the bodies of methods and method calls (both of which are overlooked in this implementation), removal and addition of class files, and tweaks to method and field declarations. A fourth comparison attempts to detect the GA component from within the Jarprint Generator implementation. The results of this experiment are displayed in Table III.

The results of this experiment demonstrate that highly similar files are detected as such. Changes between versions of the Jarprint Generator code were not significant enough to deceive the similarity comparison. Interestingly, the comparison between JPG1 and JPG3 results in a lower similarity than

[1]JPG3 here is equivalent to JPG(58) from the previous experiments

Jar1	Jar2	Similarity	Duration
JPG1	JPG2	99.58%	12 sec
JPG2	JPG3	99.52%	13 sec
JPG1	JPG3	98.96%	11 sec
JPG3	GA	100%	2 sec

TABLE III
RESULTS OF SIMILAR DISTINCT JAR FILES

between JPG1-JPG2 or JPG2-JPG3. This suggests the system has a degree of accuracy to its measurement that is studied in more detail in the section below. Finally, a comparison of the GA component with the JPG3 code which contains the GA component resulted quickly in a 100% match illustrating that this system is capable of detecting classes that have been added to larger projects.

The results of phase 1 of experimentation all followed directly with the expected behaviour of the system. This implies that system behaves as expected. That is copies and similar files are graded with a high similarity while dissimilar files receive a low similarity percentage. The next task is to gain some insight towards the quality of the system.

B. Phase 2

The second phase of experimentation aims to analyze the accuracy, consistency, and efficiency of the system. As described above in Section III, accuracy is measured by slowly increasing the disparity between compared archives. The previous phase of testing demonstrated the system capable of accurately determining exact matches where both archives contained the same classes. To gain a sense of the accuracy of the system, the calculated (actual) similarity versus the (expected) similarity determined by the ratio of class files in common between the two archives are measured and contrasted. The results of this test are displayed in Figure 2.

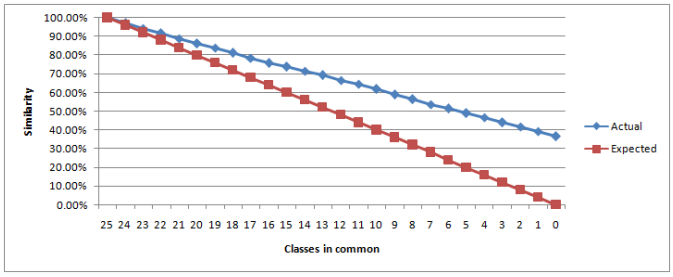

Fig. 2. Comparison of the similarity calculated using the Jarprint Generator vs. the similarity measured as a ratio of shared files

As seen in the above figure, the accuracy of the system (the distance between the actual and expected trend lines) decreases as the number of shared classes decreases. Specifically the implemented system overestimates the amount of similarity. This is due to the system's 'optimistic' approach. That is, it attempts to find the greatest amount of similarity between the two compared archives. Given that the archives in this comparison are randomly generated there is no guarantee on amount of dissimilarity between any two classes, and so the system is likely to pair classes together that exhibit some similarity without any actual relation between them. With this in mind the accuracy of the system, while not perfect, is reasonable enough to represent the likelihood of copyright infringement to a human observer. The potential for false positive identification may be an issue using this method of copy detection. Given 48% common material between the two compared archives the implemented system detected an average of 66.67% copied material. Commonalities between the structure of all class files may account for this discrepancy.

To examine the consistency of the system's similarity measures, this paper examines the standard deviation amongst multiple comparisons of the same archives. This test is performed for comparisons of multiple sized archives to determine if any correlation exists between the consistency and the comparison size. The results are provided in Figure 3.

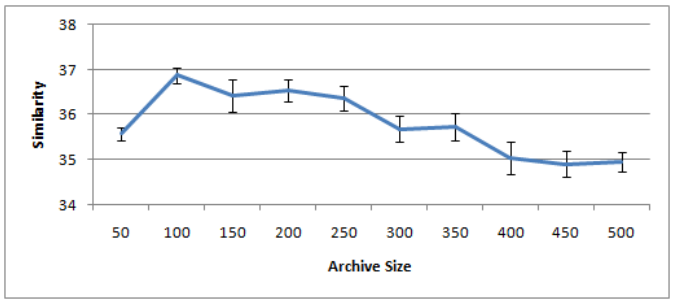

Fig. 3. Similarity calculations for varying sizes of archives. Vertical lines indicate standard deviation.

The uneventfulness of this graph is a positive indication of the consistency of the implemented system. The standard deviation per experiment is never greater than 0.4%. Over all experiments neither the standard deviation nor the similarity calculation itself show any significant correlation with the size of the comparison.

Finally, the efficiency of the system was examined over the same experiments as above. A graph of the trend of average execution times can be seen in Figure 4.

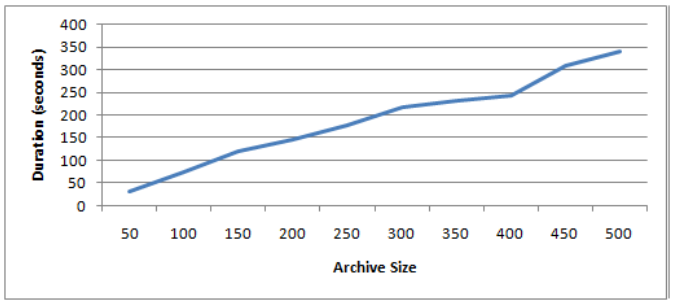

Fig. 4. Duration taken to compare varying sizes of archives.

The curve of the above graph indicates the level of efficiency of the system. It appears the duration of execution remains linearly correlated with the size of the archives being compared. While the size of the problem increases, so to do the number of swaps required to match each class. Although a constant time comparison would be ideal, such an algorithm is unlikely to exist.

V. Conclusion

The previous section suggests that the Jarprint Generator is an effective means of performing copy detection for Java archives. The base functionality testing indicates that the system is capable of distinguishing between copied, and not-copied content, while the performance testing suggests that the Jarprint Generator is both consistent and efficient. Accuracy was deemed to be of slight concern due to the systems tendency to detect the unintended similarities between distinct files, which creates the possibility for false positives. However, this risk appears to be within the realm of human comprehension and precaution.

One weakness of this approach is that it will have difficulty identifying copied material, when the number of copied classes is very small in comparison to both the source archive and the destination archive. In this case the dissimilarity of the majority of classes will have a greater impact on the similarity score than the few copied classes. Such a situation should however result in a greater similarity than an identical comparison with no copied content.

Future work should attempt to improve the accuracy of the system by revising the calculation of the final similarity measure to account for the common attributes of all class files. Also, a deconstructive approach to the similarity comparison, while potentially more costly, could yield a more informative report on specifically which classes are identified with high similarity. Finally, the use of more diversifying genetic operators could yield faster estimates, with an acceptable loss in accuracy.

References

[1] J. Holland, "Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence." 1975.

[2] D. Beasley, R. Martin, and D. Bull, "An overview of genetic algorithms: Part 1. Fundamentals," *University computing*, vol. 15, pp. 58–58, 1993.

[3] R. Haupt, S. Haupt, and J. Wiley, *Practical genetic algorithms*. Wiley Online Library, 1998.

[4] Y. Rahmat-Samii and E. Michielssen, *Electromagnetic optimization by genetic algorithms*. John Wiley & Sons, Inc. New York, NY, USA, 1999.

[5] B. Ombuki-Berman, A. Runka, and F. Hanshar, "WASTE COLLECTION VEHICLE ROUTING PROBLEM WITH TIME WINDOWS USING MULTI-OBJECTIVE GENETIC ALGORITHMS," in *Proceedings of the Third IASTED International Conference on Computational Intelligence*. Acta Press Inc,# 80, 4500-16 Avenue N. W, Calgary, AB, T 3 B 0 M 6, Canada,, 2007.

[6] A. Runka, "Thoughtstack," November 2010, `http://www.cosc.brocku.ca/ ar03gg/ThoughtStack/`.

[7] S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R. Hubley, A. Chircop, J. Compton, W. Haddon, S. Donnelly, B. Jamil, and J. O'Beirne, "Ecj: A java-based evolutionary computation research system," November 2010, `http://cs.gmu.edu/ eclab/projects/ecj/`.