Similarity Graph with Subgraph Isomorphism For Fingerprinting

James McAvoy

Abstract—Police use fingerprint to identify suspects at a crime scene. In JAR files, we want to create fingerprints to detect the present of *classes* from a given source. We assumed the difference between *classes* in a JAR file is unique and these difference can be capture using a data structure called a *similarity graph*. When comparing a JAR file with a fingerprint, a program creates two similarity graphs and determine if the smaller graph is a subgraph of the larger graph using subgraph isomorphism. This paper will show how this method as a viable approach for fingerprinting.

Index Terms—COMP5900, Fingerprinting, similarity graph, subgraph isomorphism, Java, JAR.

1 Introduction

Police use fingerprints to identify suspects at a crime scene. The premise is that fingerprints are uniquely associated with an individual. We want to apply this same principle with identifying a collection Java *classes* in JAR files with a fingerprint.

1.1 Problem

A non-trivial Java program can import several JAR files for its structure and functionality. These JAR files may have associated copyright and licensing agreements that are incompatible to the organization's goals or business model. A capability to recognizes the inclusion of code from different contributors will help detect copyright infringements or determine the application's providence.

1.2 Motivation

Graphs are powerful and versatile data structure to represents objects and their relationships in various fields of science. In the application of pattern recognition, object similarity is an important issue. If graphs are used for object representation this problem turns into determining the similarity of graphs, which is generally referred to as *graph matching*.

A Java program can be expressed as a collection of classes that cooperate together to achieve a result. Rarely, JAR files contain one **.class** file. Each **.class** file is different to each other. Can we exploit this observation to our advantage? A *Similarity Graph* is a data structure that can be use to express this dissimilarity or difference. We hypothesis that a JAR file containing a collection of **.class** files has an unique similarity graph, which is like a fingerprint. Using graph matching, we can detect this fingerprint in other JAR files that may contain these groupings of **.class** files.

1.3 Aim

The goal of this project is to determine the following:

1) A JAR file has a unique similarity graph which can be used as a fingerprint.
2) Subgraph isomorphisms can be used to determine if a JAR file's similarity graph matches a fingerprint's similarity graph.

1.3.1 Limitations and Constraints

This project narrowly focus on comparing Java **.class** files contained in a JAR file and omitted the other artifacts and file structure that one finds in a JAR file from comparison. This method should be used in conjunction with other techniques to create a more comprehensive solution.

1.4 Objectives

To demonstrate if this project meets its goal, a program will be created. The project will be consider a success if the program can accomplish the following objectives.

1) A fingerprint shall detect the JAR file it is derive from and other JAR file 100% of the time.
2) A JAR file with missing **.class** files shall be detected by a fingerprint.
3) A JAR file containing two different **.class** files from two different JAR files shall be detected by two fingerprints.
4) The fingerprint shall be less than 10% of the JAR file size.
5) Matching a fingerprint to a JAR file shall take less than 1 minute.

1.5 Outline

In this paper, section 2 will introduce the basic concept of similarity graphs and why this representation is suitable for fingerprinting. Section 3 will cover subgraph isomorphism as a means for detection. Then section 4 presents a program implementation using similarity graphs with subgraph isomorphism for fingerprinting. The results and validation of the project is presented in section 5. In the conclusion section 6, will summarize the results and discuss future follow on studies.

1.6 Notation

The word "class" is used several times in this paper and has several different meanings depending on the context. For clarity the word class will be presented below with the following meaning:

TABLE 1
Fictional JAR File containing five **.class** files.

Class	Constant Pool Length	Field Length	Method Length
1	66	20	1
2	41	10	2
3	68	5	8
4	90	34	5
5	75	12	14

- **.class**: The Java .class file.
- *class*: The Java programming language class.
- class: The everyday generic or programming concept of objects sharing similar properties and behaviour.

2 Similarity Graph

The problem of grouping "like" objects into classes based on properties of the object can be represented using *Similarity Graph*. The premise is that the grouping of "like" **.class** files in a JAR file is unique and using similarity graph is a viable approach to express this uniqueness.

To create a similarity graph we first record certain properties of a **.class** file. For purpose of instruction we recorded the following properties from a set of **.class** files in a fictional JAR file:

1) The constant pool length in a **.class** file.
2) The field length in a **.class** file.
3) The method length in a **.class** file.

See table 1 for an example of these values extracted from a fictional JAR file.

A similarity graph G is constructed as follows. The vertices corresponds to the **.class** files in a JAR file. A vertex is denoted (p_1, p_2, p_3), where p is the value of property i. We define a dissimilarity function s as follows. For each pair of vertices $v = (p_1, p_2, p_3)$ and $w = (q_1, q_2, q_3)$ we set

$$s(v, w) = |p_1 - q_1| + |p_2 - q_2| + |p_3 - q_3|.$$

If we let v_i be the vertex corresponding to a **.class** file i, we obtain

$$s(v_1, v_2) = 36, s(v_1, v_3) = 24, s(v_1, v_4) = 42,$$
$$s(v_1, v_5) = 30, s(v_2, v_3) = 38, s(v_2, v_4) = 76,$$
$$s(v_2, v_5) = 48, s(v_3, v_4) = 54, s(v_3, v_5) = 20,$$
$$s(v_4, v_5) = 46.$$

If v and w are vertices corresponding to two **.class** files, then $s(v, w)$ is a measure of how dissimilar the **.class** files are. A large value of (v, w) indicates dissimilarity, while a small value indicates similarity.

For a fixed number S, we insert an edge between vertices v and w if $s(v, w) < S$. In general, different values of S will create different similarity graphs. We say that v and w are in the same class if $v = w$ or there is a path from v to w. In the figure we show a graph corresponding to the **.class** files of the above table with $S = 25$. In this graph, only three **.class** files are grouped which are $\{v_1, v_3, v_5\}$. See figure 1.

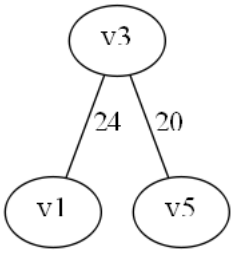

Fig. 1. Similarity graph of fictional JAR file where $S < 25$.

The project tried different values of S to determine the optimal value for detection. We want an S that provides reliable detection with good performance.

3 SUBGRAPH ISOMORPHISM

Isomorphism is the problem of testing whether two graphs are really the same. Certain pattern recognition problems mapped to graph or subgraph isomorphism detection. The structure of chemical components are natural described by label edges and vertices representing atoms. Identifying a molecule in a compound is an instance of subgraph isomorphism.

Assume two label graphs $G = (V_g, E_g)$ and $H = (V_h, E_g)$. We say that G and H are identical when $(x, y) \in E_g$ iff $(x, y) \in E_h$. The isomorphism problem consists of finding a mapping from the vertices of G to H such that they are identical. This mapping is called an *isomorphism*. Sometimes, the problem of finding this mapping is called *graph matching*.

In practice, exact matches are rare. The variety we are concern with is the problem of determining if graph G is contained in graph H. Instead of testing equality, we are interested in knowing whether a small of pattern graph G is a subgraph of H.

There are two distinct graph-theoretic notions of "contained in". *Subgraph isomorphism* asks whether there is a subset of edges and vertices of H this is isomorphic to a small graph G. *Induced subgraph isomorphism* is harder. It asks whether removing edges and vertices from H can we get G. For induced subgraph isomorphism, (1) all the edges in G must in H, and (2) no non-edges of G be present in H. In this project, the former, subgraph isomorphism, is sufficient for our purpose.

Labelling a similarity graph can improve detection and performance. The edges are labeled with the results of the dissimilarity function between two **.class** files and the vertices, which represent **.class** files, are labelled with their properties.

Computing graph isomorphism can be very expensive which is a drawback using this approach. There exists no polynomial-time algorithm for solving this problem but neither is it known to be NP-complete. One of the great unanswered mysteries in mathematics. The general accepted understanding is that isomorphism problems lies between P and NP-complete. In practice, if the size of the graphs are small (i.e. less than 100 nodes) and the graph

is labelled for both edges and vertices, performance is very reasonable.

4 Approach

This section outlines the implementation of the Fingerprint program used in this project. The program provides two operations which are as follows:

1) *Logging*. The operation inputs a JAR file and outputs a fingerprint of the JAR file. The program stores the resultant fingerprint in a repository, which the program reads later in the comparison operation.
2) *Comparing*. The operation inputs a JAR file and compares with fingerprints that are stored in a repository. The program outputs the results of each comparison.

The section will first introduce what is a Java **.class** file format, its structure and content. Then subsequent subsections will discuss how the logging and comparing operations were implemented.

4.1 Java Class File Format

Developers writing programs using the Java programming language must compile the code into a portable binary format called *byte code*. The Java compiler creates a **.class** file for each Java *class* or *interface* with its data and byte code instructions. If there exists multiple *classes* in the source file, the compiler will create a **.class** file for each *class* definition. An interpreter (Java Virtual Machine aka JVM) loads these files and executes them. The JVM are available cross multiple platforms. Therefore, the **.class** file can be executed on multiple platforms, making the Java programming language platform independent.

The Java Specification Request (JSR) 202 defines the structure and binary layout of a **.class** file. A class file has 10 basic sections which are as follows:

- Magic Number: 0xCAAFEBABE.
- Version of the Class File: The minor and major versions of the **.class** file.
- Constant Pool: Pool of constants for the *class*.
- Access Flags: Specifies whether the *class* is abstract, static, etc.
- This Class: The name of the *class*.
- Super Class: The name of its parent *class*.
- Interfaces: Any interfaces in the *class*.
- Fields: Any fields in the *class*.
- Methods: Any methods in the *class*.
- Attributes: Any attributes of the **.class** (for example the name of the sourcefile, etc).

Later in this section, the paper will explain how these attributes of this structure were used to create a fingerprint.

4.2 Logging

The logging feature reads a JAR file as an input and outputs a fingerprint. The program stores the resultant fingerprint in a repository. Later in the comparison operation, the program will read these fingerprints and compare them against an query JAR file.

The operations first step is to read the JAR file and discovery where all the **.class** files are located. Then program iterates over all the **.class** files and extracts the following properties from the class file:

- Class Name;
- File Size;
- Constant Pool Length;
- Fields Length; and
- Methods Length.

A third party JAR file from Apache called Byte Code Engineering Library (BCEL) was used to access the a **.class** file's properties. The BCEL library's *classes* failed to read the **.class** files in the JAR, so we wrote out the **.class** file into the temporary file. Once this was completed, BECL's *class* instances were able to read the files.

A Comma Separated Value (CSV) was used for fingerprint format. It was decided early in the development stage to use this format for several reasons:

- Very simple to write and read a CSV file.
- Another programs can read this format, even database applications.
- Decouple the storage representation from the program's internal representation. This was the greatest reason. Early in the development cycle, internal program representation was vague. Similarity graphs was one candidate data structure of many. Therefore, having this decoupling provide more flexible during the development cycle of the project.

You will notice that the program only collected the physical attributes of a **.class** file and not its logical Java programming language constructs that defines a *class*. Using these properties and computing their dissimilarity was far easier than comparing object-oriented concepts, such as class IS-A and HAS-A relationships for example.

4.3 Comparing

Once the we created several fingerprints, we can start the comparison operation. Let F be set of fingerprints:

$$F = \{f_i : SG(f_i)\},$$

where each fingerprint i is represented by a Similarity Graph $SG(f_i)$.

Let q be the JAR file that we want to query against F. First, the program builds two similarity graphs; one for fingerprint and one for the query JAR file. Once build, the programs then compares the two similarity graphs and determines if $SG(q)$ *subgraph* $SG(f_i)$ or $SG(f_i)$ *subgraph* $SG(q)$.

To build a similarity graph SG for both f_i and q we need to define a dissimilarity function first. A vertex represents a **.class** file in a JAR file. A vertex is denoted (p_1, p_2, p_3, p_4), where p is the value of the following **.class** files properties:

1) The **.class** file size;
2) The **.class** file's constant pool length;
3) The **.class** file's field length; and
4) The **.class** files's method length.

For each pair of vertices in a fingerprint $v = (p_1, p_2, p_3, p_4)$ and $w = (q_1, q_2, q_3, q_4)$ we set

$$s(v, w) = |p_1 - q_1| + |p_2 - q_2| + |p_3 - q_3| + |p_4 - q_4|.$$

This process builds a graph $SG = (V_{SG}, E_{SG})$, where V_{SG} represents all the **.class** files and E_{SG} represents all the dissimilarities values between the vertices. It follows that SG is a *complete graph on n vertices* where n is the number of vertices and every vertex is joined to every other vertex by an edge. The number of edges in SG is $\binom{n}{2}$. Therefore, the total running time to build this data structure and store it is $O(n^2)$.

This is a lot of vertices and edges to compare with other SG. We reduce the size of SG by grouping "like **.class**" files. Let S_l be the lower bound and S_u upper bound, using these values we create another similarity graph SG' where we select vertices pairs v and w from SG such that:

$$s(v, w) \geq S_l$$

and

$$s(v, w) \leq S_u.$$

This reduces the similiarity graph significantly and creates an unique graph for each JAR file, just like a fingerprint. During the experiment stage, several values of S were used to determine the best result. Users can change the S value in file, *.fprc.ini*. Figure 2 shows an example similarity group of a JAR file.

Next we compare the $SG(f_i)'$ with $SG(q)'$ using subgraph isomorphism. A successful operation returns the number vertices that matched and an array of the matching vertices, otherwise zero if no match was found.

The program used a graph matching C++ library, *VFLib*. A search for a comparable graph matching library written in Java was not found. A JNI .dll was create to allow access to this library from the Java program. The

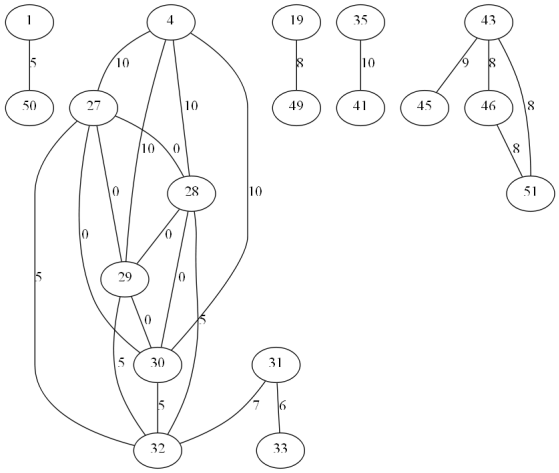

Fig. 2. Similarity graph of *org.apache.commons.el* JAR file where $S_l \geq 0$ and $S_u \leq 10$.

program used the subgraph isomorphism features that the library provided. A noticable increase of performance and accurancy was observed if the edges and vertices were labelled. The program labelled the edges with the dissimilarity values and each vertices were labeled with the **.class** file's constant pool, field and method length.

The program presented the results to the console standard output. It displayed the number of matches and the percentage of certainty of the match. Let m be the number of matches then percentage of certainty p is calculated as follows:

$$p = \left(\frac{m}{|V_{SG(f_i)}|} + \frac{m}{|V_{SG(q)}|} \right) * 50,$$

where $|V_{SG(f_i)}|$ is the number of vertices in the fingerprint's similarity graph and $|V_{SG(q)}|$ is the number of vertices in the JAR file's similarity graph.

5 Experiment

For the experiment, a subset of all the JAR files found in the Eclipse plugin directory was selected. The following ten JAR files are outlined in table 2. We assigned an ID for each JAR file for reference in later tables.

To meet the projects objectives additional JAR files were created. These JAR files are outlined in table 3.

TABLE 2
JAR Files under test.

JAR File ID	JAR File
1	com.jcraft.jsch
2	javax.servlet
3	javax.servlet.jsp
4	org.apache.commons.codec
5	org.apache.commons.el
6	org.apache.commons.httpclient
7	org.eclipse.core.commands
8	org.hamcrest.core
9	org.mortbay.jetty.util
10	org.sat4j.core

TABLE 3
Created JAR files.

JAR File	Comments
copy.jar	A copy of com.jcraft.jsch JAR file.
combine.jar	A combination of two JAR files.
missing.jar	A copy of com.jcraft.jsch JAR file with files missing.

5.1 Computer

The experiment was conducted on a laptop. Table 4 outlines the computer's specification.

5.2 Exact Copy Matching

First project objective was to determine a fingerprint can match with its derived JAR file with 100% accuracy and no other. We hope this method would reduce the occurrence of false positive results. Therefore, we ran the

TABLE 4
Computer platform used for the experiment.

Manufacturer	Dell
Model	Percision M4500
Processor	Intel(R) Core(TM)i7 CPU, Q 720 @ 1.60GHz
Installed memory (RAM)	8.00 GB
Operating System	Windows 7, 64 bit

TABLE 5
Result of an exact copy experiment.

FP for JAR File	Match %	Matches with JAR File
1	100	1, copy.jar
2	100	2
3	100	3
4	100	4
5	100	5
6	100	6
7	100	7
8	100	8
9	100	9
10	100	10

comparison operation against all fingerprints for each sample JAR file. The test used the following S values, $S_l \geq 0$ and $S_u \leq 5$.

The only fingerprint that should have two JAR file matches is *com.jcraft.jsch* because *copy.jar* is a copy of that file. Therefore, this method of fingerprint seems to reduce the occurrence of false positives. See table 5 for results.

5.3 Missing .class Files

In this experiment, we removed files from the *com.jcraft.jsch* JAR file and created a new one and named it *missing.jar*. We compare *missing.jar* with the fingerprint repository and the program correctly detected that *missing.jar* could be from *com.jcraft.jsch* JAR file. The test used a very small S value of $S_l \geq 0$ and $S_u \leq 5$. See table 6 for results.

5.4 Combined JAR Files

In this test, we combined two JAR files into one and named it *combine.jar*. We compare *combine.jar* with the fingerprint repository and the program correctly detected that *combine.jar* could be derived from *org.apache.commons.codec* and *org.apache.commons.el* JAR files. The test used a very small S value of $S_l \geq 0$ and $S_u \leq 5$. See table 7 for results.

TABLE 6
Result of missing file experiment.

JAR ID	% certainty
1	59.37
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0

TABLE 7
Result of two combine JAR files experiment.

JAR ID	% certainty
1	0
2	0
3	0
4	63.33
5	86.66
6	0
7	0
8	0
9	0
10	0

5.5 Fingerprint Size

We want fingerprint size to be less than 10% of the size of the JAR file. Table 8 shows that the program was able to create fingerprints less than 10% and in fact most were less than 5%. Having small fingerprints facilitates efficient storage.

Size of the fingerprint could be smaller if the file storage format was changed from text to binary or store the fringerprints in a database.

5.6 Performance

Performance was measured by comparing the *copy.jar* file against the fingerprint repository ten times. The

TABLE 8
Result of the fingerprint size experiment.

JAR ID	JAR (KB)	Size	FP Size (KB)	%
1	221		5	2.2%
2	117		3	2.5%
3	62		3	4.8%
4	54		2	3.7%
5	126		4	1.5%
6	314		12	3.2%
7	105		6	5.7%
8	28		1	3.6%
9	189		9	4.8%
10	190		9	4.7%

TABLE 9
Performance experiment results.

Run	time (ms)
1	23,288
2	22,846
3	23,082
4	22,863
5	22,848
6	22,801
7	23,112
8	22,958
9	22,847
10	22,942

program called *System.currentTimeMillis()* at the start of the comparsion operation and again at the end. The total time is the difference between end and start time. The program output the total time to standard console output after each run. Table 9 displays the results after ten runs. The test used a very small S value of $S_l \geq 0$ and $S_u \leq 5$.

To compare one JAR file against ten fingerprints, the program took approximately 22,953 ms or 23 seconds on average. Therefore, on average comparing a JAR file against one fingerprint takes about 2.3 seconds.

6 Conclusion

Using similarity graph as a data representation to uniquely identify a JAR files and using subgraph isomorphism for detection seems to provide a viable method for fingerprinting. We were able to achieve the objections specified in the introduction with a limited set of JAR files. It was demonstrated this method can detect JAR files with missing **.class** files and JAR files containing two JAR files. Performance was accepted when using a small S value. The fingerprint files size meet the project objective but could be smaller if fingerprints were stored from a text to binary format.

The project scope and goals were limited on purpose. More research needs to be conducted to answer the following questions:

- Determining the best value of S to create a similarity graph that provides optimal performance and detection accuracy.
- Determining if fingerprint stored in a binary format will improve storage size and detection performance.
- What are the best properties to use for calculating dissimilarities.
- Could this method detect JAR files containing obfuscated **.class** files.

References

[1] R. Johnsonbaugh, *Discrete Mathematics*, 2nd ed. New York, NY USA: Macmillan Publishing Company, 1990.

[2] J. Ullmann, *An Algorithm for Subgraph Isomorphism* J. ACM 23 1 (January 1976), 31-42.

[3] H. Bunke, *Graph Matching: Theoretical Foundations, Algorithms and Applications* .

[4] S. Skiena, *The Algorithm Design Manual*, 2nd ed. London, UK: Springer-Verlang.

[5] A. Buckley, *JSR 202: Java Class File Specifiation Update*, October 2006. Sun Microsystems, Inc.

[6] L. Cordella,P. Foggia,C. Sansone,M. Vento, *Performance Evaluation of the VF Graph Matching Algorithm* Proc. of the 10th ICIAP, IEEE Computer Society Press, vol. 2 (1999), 1038-1041.

PLACE
PHOTO
HERE

James McAvoy Presently, a part-time first year graduate student at Carleton University and working full-time as an independent software consultant.