A Simple but Effective Algorithm to Measure the Resemblance of Two Jar Files

Yifeng Li

Abstract

Code is easy to copy. Nowadays quite a few people like to borrow others' work without referring to it, which is strictly disallowed especially in academic settings. As a result, people are working on ways to detect whether two pieces of work are similar to each other. In this research project, we design and implement algorithms to determine the closeness between two Jar Files.

Introduction

Given two Java Archive (JAR) files, we want to determine whether they are identical or to what extent they are close to each other. For example, we would like to know if the two JAR files contain the same libraries and we are also interested in whether or not the classes defined in the JARs have the same fields and methods. The challenge is that there could variable kinds of files in the JAR file such as text file, image file, java class file and sometimes java source file. What's more, the Java source files are under rare circumstances in the JAR file, which is mostly the reason why we need to do researches on this problem.

Copying is everywhere. In a university or an institute, students in programming courses in particular tend to do so. Sometimes they hand in source code copied from their fellow classmates, which can be fairly easily detected by a simple computer program. Since we can see the source

code, it is no big deal for a student in computer science to come up with such programs. In many cases, however, the deliverables are just compiled class files or binary files, which makes it more difficult to detect the copying projects. Thus we are very interested in developing a simple but effective algorithm to measure the resemblance of two JAR files that mainly contain Java class files.

We aim for a simple but effective algorithm which can be used to measure the resemblance of two JAR files that mainly contain Java class. We assume that our JAR files under test only contain Java class files since the other factors are minor and easy to handle. Also, our assumption simplifies our algorithm a lot and places the emphasis on the most difficult part.

Given JAR files containing only Java class files, we first write a program to capture information about the JAR file itself and characteristics about the class files in it. We store the information in a data structure called Fingerprint. Then we write another program to compare each piece of data in a fingerprint to its counterpart in another fingerprint. Depending on the comparison results, we compute the percentage of common code the two JAR files are likely to share. After designing and implementing the comparison program, we do a reasonable number of tests which covers most of the cases that would happen in real world. From each testing result, we decide whether our capturing of the JAR file is good enough and whether our comparison algorithm works out well. And we keep adjusting our programs accordingly in order to improve the correctness and efficiency of our algorithm. When our programs works out well with all our well-designed test cases, the algorithm we would be using is likely to be a good one we are trying to find in our

research.

In the following sections, we introduce what can be in a JAR file, explain what information we capture in our fingerprint and why, describe how our comparison algorithm works and finally conclude with our test results.

Background

A JAR file is used to aggregates many files into one. In particular, .jar files are often used to distribute Java applications or libraries, in which case classes and associated metadata and resources such as text and images are bundled together in one JAR file. JAR files, which build on the ZIP file format, can be created or extract using the jar command that comes with a JDK.
A JAR file has a manifest file which is optional. The entries in the manifest file determine how one can use a JAR file.

In our implementation, we use the class java.util.jar.JarFile to process JAR files.

A Java class file is virtual machine-readable with a .class extension. Since Java is a platform-independent language, source code is compiled into an output file known as bytecode, which is stored in a .class file. If a source file has more than one class, each class is compiled into a separate .class file. These .class files can be loaded by any Java Virtual Machine. Therefore, a .class file compiled in one platform will execute in a Java Virtual Machine of another platform, which makes Java platform-independent.

There are 10 basic sections to the Java Class File structure. See the table below.

Magic Number	0xCAFEBABE
Version of Class File Format	The minor and major versions of the class file
Constant Pool	Pool of constants for the class
Access Flags	For example whether the class is abstract, static, etc
This Class	The name of the current class
Super Class	The name of the super 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 source file, etc)

In our implementation, we take advantage of the BCEL library to process class files.

Approach

A fingerprint of a JAR file captures the main characteristics and related data of that JAR file. It is nice to just store the fingerprint of a JAR file in text or binary format so that when we want to compare two JAR files, we only need to read and compare the most useful information stored as the fingerprints.

We store a fingerprint as a string, where each piece of data is a substring separated by a comma. And each substring contains data that captures one aspect of the JAR file. In our design, we capture the following data in the fingerprint.

- The name of the JAR file
- The size of the JAR file
- The number of class files in the JAR file
- The names of class files in the JAR file

- The size of each class file in the JAR file
- Major details of each class file in the JAR file

As listed above, we first write the name of the JAR file as a string into our fingerprint, followed by a comma. Immediately after the comma, we write the total size of the JAR file also as a string followed by another comma. Then follows the number of class files in the JAR file. Finally we capture the details for each class file as a separate string, respectively. Looking into the string that represents a certain class file, we see substrings separated by single space containing the following information.

- The name of the class file
- The size of the class file
- The number of constants in the class
- The number of fields in the class
- The names of fields in the class
- The number of methods in the class
- The names of methods in the class

Now we have finished describing the content of our fingerprint. It remains to come up with an algorithm to compare the two fingerprints and compute the corresponding indicator which is a percentage we will use to represent and determine the resemblance of two JAR files.

Our comparison algorithm is fairly simple, which goes as follows. First of all, we compare the

names of two JAR files. If they are identical, we add a percentage value of 5% to the total percentage which we keep track of through the whole comparison process and has an initial value of 0%. We continue with comparing the sizes of the two JAR files. If the difference between their sizes is no greater than 500, we consider them to be highly suspected targets, in which case we add another 10% to the total percentage. Otherwise we add nothing to the total percentage. Then we would like to take a look at the numbers of class files in each JAR file. If they get the same number of class files, chances are fairly high that they are copies of each other. In this case, we add another 10% to the total percentage; otherwise we do nothing. Next, we look into the details of the class files. Beginning with the names of the class files in each JAR file, we count the number of class files which bear the same names in both JAR files. We compute the ratios of that number to the total numbers of class files in each JAR file. Let's take the bigger ratio of the two and multiply it by 25%. Then we add the product to the total percentage. Now we consider the sizes of the class files in both JAR files. Let's make a design decision first. We regard two class files as of the same size as long as the difference between their sizes is less than 100. Then we count the number of class files with the same size and get the ratios as we did before. However, we only assign a weight of 15% to this part, which means the ratio times 15% would be added to the total percentage. Now we have taken 65% of the total percentage into account. Whatever percentage is left over will be determined by the details of the class files.

The remaining part is computed as follows. Now we only consider those class files either with the same name or having the "same" size as defined above. Let's count the number of such class file pairs. Each of such class file pairs is assigned a total weight of 35% divided by that number. For

each class file pair, we proceed with comparing their constant pools, fields, and methods respectively. If the numbers of their constants are the same, we add 1/3 of the class file weight to the total percentage. If over half of the fields have the same names, we add 1/3 of the class file weight to the total percentage. If over half of the methods have the same names, we add another 1/3 of the class file weight to the total percentage. Otherwise we do nothing.

We finish our comparison algorithm here. It finally gives us a percentage value which can be used to indicate the resemblance of the two JAR files.

Results

In order to demonstrate that our design is able to meet our goal, we conduct the following experiment. We devise and generate a suite of test cases which will cover all possible scenarios. The inputs of our test program are simply fingerprints which we carefully design and make up for the purpose of testing. The fingerprints have one-to-one correspondence to real world JAR files. As a result, we don't use real JAR files as inputs to our test program, which simplifies the testing process and places the focus on the actual data processing.

The table below illustrates the test cases we have made and the corresponding results.

Name of JAR	Size of JAR	Number of classes	Ratio of classes with same name	Ratio of classes with "same" size	Number of constants	Names of fields	Names of methods	Total percentage
Same	Same	Same	1	1	Same	Over half	Over half	100.0
Not same	Same	Same	1	1	Same	Over half	Over half	95.0
Not	Same	Not	0.2	0.8	Same	Over	Over	65.0

same		same				half	half	
Same	Same	Not same	0.5	0.5	Not same	Over half	Less than half	46.7
Same	Not same	Not same	0.2	0.0	Same	Over half	Less than half	11.3
Same	Not same	Not same	0.1	0.7	Not same	Less than half	Less than half	21.0
Not same	Same	Not same	0.2	0.9	Same	Less than half	Less than half	43.7
Same	Same	Same	0.0	0.9	Same	Over half	Less than half	66.3
Not Same	Not same	Not same	0.2	0.4	Not same	Over half	Less than half	23.7
Not same	Not same	Not same	0.0	0.2	Not same	Less than half	Less than half	4.0

The suite of test cases is not an exhaustive one and it is not so practical to come up with all the combinations. Thus we have carefully designed these test cases in the light that they are most likely to happen in the real world.

Conclusion

We have designed and implemented an algorithm to measure the resemblance of two JAR files as well as conducted an experiment with performing adequate testing on our program. Through the test results shown above, we claim that it have been a simple but effective algorithm for accomplishing such a comparison task. With our program, we are able to generate a fairly accurate result regarding the relationship between two JAR files. If a professor would use our program in an academic setting, there could be a high possibility that he get a relatively satisfactory output from our program.

However, our algorithm could generate quite inaccurate result depending on the given JAR files.

Though the possibility is quite low, we still need to minimize it further in the future. This bug results from the fact that we haven't captured all information in our JAR files completely. Sometimes the information we store in the fingerprint is only a small part of the whole JAR file we are trying to capture. For example, we just ignore those resource files in the JAR file, which are not captured in our fingerprint. In our following work, we are going to capture more information such as parameters of each method in our fingerprint.

Another source of inaccuracy could be the weights we assign to each part of the total percentage. We need to do more research on this to ensure that each part really deserves the weight assigned to them.

There are two more approaches that we may want to incorporate into our algorithm in the near future. First, we can try to decompile the class files and get a version of source files which we are able to look into. Various kinds of decompiling libraries have been out there. We will try some of them and pick one of the best as a tool to get java source files from which we can gain better insight into the class files. In the other approach, we can place our focus on a specific method that is defined by a class. Inside each method, there are classes it references as well as other methods it invokes. We can store such kind of relationships in a graph structure. Differences and similarities of these graphs are likely to shed us more light upon our class files.

References

http://www.angelfire.com/tx4/cus/jasper/

http://www.murrayc.com/learning/java/java_classfileformat.shtml

http://download.oracle.com/javase/tutorial/reflect/index.html

http://jakarta.apache.org/bcel/

http://en.wikipedia.org/wiki/JAR_(file_format)

http://en.wikipedia.org/wiki/Class_(file_format)