COMPGS04/M024: Tools and Environments

Coursework 2 (Group Coursework)

Authors:
Jihyun Han
Elliott Omosheye
Sachin Pande
Luke Richardson
Yasaman Sepanj

March 26, 2014

1 Introduction

1.1 Background

In software engineering, cloned code is one of the fundamental issues of productivity. The programming community is generally recognising the duplicates as a bad practice [Refactoring: Improving the Design of Existing Code by Martin Fowler], although some argue otherwise. We tend to avoid it because we are taught that repeated code is inelegant and can cause difficulties whilst trying to maintain applications, or sometimes it may involve legal risks. However, due to time pressure, or limitation of programming languages, reusing code fragments by duplicating with or without variations is a common activity evident within software development.

Unlike plagiarism detection in literature, duplicated code does not solely depend on text matching, but consideration has to be made for the syntax, semantics and patterns of the codes. For instance, two source files that functions the same may have completely different identifier names for variables – this must also be detected as a clone. Different approaches to achieve the detection exists, and will be discussed later.

1.2 Project

In a 5 membered team, we are given the task of building a tool that analyses and reports similarity of two Java source files. This report introduces 5 different similarity detection mechanisms including Clone Detection and Plagiarism Detection, and we shall decide a suitable algorithm to implement for the tool. The tool will be tested with a given sample code and its variations to check our program's performance whilst analysing the similarities between the pair of codes.

2 Normalized Compression Distance

Vitányi, Paul MB and Balbach, Frank J and Cilibrasi, Rudi L and Li, Ming. Information theory and statistical learning: Normalized Information Distance. Springer. 2009.

2.1 Background

Normalized compression distance is a method of computing the similarity between two documents of any kind whether this be two text files or two music files. It measures the difficulty of being able to turn one document to the other. This method of determining similarities between two files is based upon the Normalized Information Distance (NID) between them.

2.2 Normalized Information Distance

The information distance between two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is defined to be the length of the shortest program, $\boldsymbol{p}$, for a universal computer to transform $\boldsymbol{x}$ into $\boldsymbol{y}$ and vice versa. The length of this program can be defined with the use of Kolmogorov complexity as follows:

$$|\boldsymbol{p}| = \mathbf{max}\{\boldsymbol{K}(\boldsymbol{x}|\boldsymbol{y}), \boldsymbol{K}(\boldsymbol{y}|\boldsymbol{x})\} \tag{1}$$

This distance is absolute and therefore needs to be normalized in order to provide the similarity in relative to both the files. Such an normalization provides the NID.

$$\boldsymbol{e}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\mathbf{max}\{\boldsymbol{K}(\boldsymbol{x}|\boldsymbol{y}), \boldsymbol{K}(\boldsymbol{y}|\boldsymbol{x})\}}{\mathbf{max}\{\boldsymbol{K}(\boldsymbol{x}), \boldsymbol{K}(\boldsymbol{y})\}} \tag{2}$$

2.3 Normalized Compression Distance

The NID is however impractical as it uncomputable. Therefore requiring a computable algorithm is requires, such a algorithm is Normalized Compression Distance (NCD). Transforming the equation through substitution of the uncomputable function $\boldsymbol{K}$ with a real-world compression function $\boldsymbol{Z}$ provides the NCD, defines by

$$\boldsymbol{e}_{\boldsymbol{Z}}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{Z}(\boldsymbol{xy}) - \mathbf{min}\{\boldsymbol{Z}(\boldsymbol{x}), \boldsymbol{Z}(\boldsymbol{y})\}}{\mathbf{max}\{\boldsymbol{Z}(\boldsymbol{x}), \boldsymbol{Z}(\boldsymbol{y})\}} \tag{3}$$

where $\boldsymbol{Z}(\boldsymbol{x})$ is the length of the compression of string $\boldsymbol{x}$ using the compression function $\boldsymbol{Z}$.

2.4 Application

For the case of this project we are interested in similarities between Java source files, for which a variant of the NID can be used to measure the similarity. This variant named sum distance is defined by

$$\boldsymbol{e}_{\mathbf{sum}}(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{K}(\boldsymbol{x}|\boldsymbol{y}) + \boldsymbol{K}(\boldsymbol{y}|\boldsymbol{x})}{\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{y})} \tag{4}$$

The two files are tokenized then compressed with a customized compressor to approximate the sum distance.

2.5 Conclusion

NCD is able to provide a good score for the similarity between two files. Whilst the NCD algorithm and the variant sum distance are relatively easy to implement, the latter requires the need for the implementation of a custom compressor which is out of scope for this project. Furthermore these algorithms are unable to provide details about the nature of the similarities between the two Java source files.

3 Plagiarism Detection

3.1 Introduction

Plagiarism detection in code is the method and process used for locating instances of cloned code within a set of documents. The tool we have chosen to demonstrate and explore plagiarism detection is JPlag.

3.2 JPlag

JPlag is a system that finds similarities, software plagiarisms, among multiple source code files. It is robust against many attempts to disguise the copied code because it is aware of programming language syntax and program structure. Thus, it is very hard to deceive: 90% of plagiarisms are detected and the rest raise suspicion. [4] Moreover, JPlag is able to process a hundred programs with several hundred lines of code in seconds, making it a very robust and scalable tool. It has been successfully used for detecting plagiarism among students' Java programs but support for other languages such as C and C++ are also available.

JPlag's comparison algorithm is based on the "Greedy String Tiling" [5], which takes in a set of program source code as input and outputs a similarity score between pairs of programs and their corresponding similarity regions. To do so, it must initially convert the program source code in to token strings, this is the only language dependent process involve din JPlag's plagiarism detection. Tokens should be chosen in a manner, which captures the essence of the program's structure rather than the surface aspects, as a program's structure is harder for plagiarists to modify. JPlag does this phase whilst ignoring whitespaces, comments and names of identifiers. Figure 1 shows an example Java code and its corresponding tokens generated by JPlag. A list of possible JPlag tokens can be found in this technical report [6].

Once the token strings have been generated, JPlag beings the comparison phase by compar-

	Java source code	**Generated tokens**
1	`public class Count {`	BEGIN_CLASS
2	`  public static void main(String[] args)`	VAR_DEF, BEGIN_METHOD
3	`            throws java.io.IOException {`	
4	`    int count = 0;`	VAR_DEF, ASSIGN
5		
6	`    while (System.in.read() != -1)`	APPLY, BEGIN_WHILE
7	`      count++;`	ASSIGN, END_WHILE
8	`    System.out.println(count+" chars.");`	APPLY
9	`  }`	END_METHOD
10	`}`	END_CLASS

Figure 1: A sample Java Source code with the generated token strings

ing the two generated token strings. When comparing two strings A and B, JPlag's objective is to discover a maximal set of contiguous substrings. Each substring must occur in both A and B and must be as long as possible. These substrings (matches) must be unique in that each substring should not cover tokens already covered by other substrings. To avoid false matches, a minimum match length M is enforced.

3.3 Greedy Tiling Algorithm

The Greedy String Tiling algorithm itself has two steps shown in Figure 2. Firstly, all substrings common to both token strings are found with lengths equal to or greater than the minimum M. Secondly, all the tokens found in the substrings are marked so that they are no longer picked up by subsequent iterations of the first step. By marking its tokens, a match becomes a "tile". Thus, the similarity score between is calculated based on the fraction of tokens that were found as matches:

$$sim(A, B) = \frac{2 * coverage(tiles)}{|A| + |B|} \tag{5}$$

```
0   Greedy-String-Tiling(String A, String B) {
1       tiles = {};
2       do {
3           maxmatch = M;
4           matches = {};
5           Forall unmarked tokens A_a in A {
6               Forall unmarked tokens B_b in B {
7                   j = 0;
8                   while (A_{a+j} == B_{b+j} &&
9                           unmarked(A_{a+j}) && unmarked(B_{b+j}))
10                      j++;
11                  if (j == maxmatch)
12                      matches = matches ⊕ match(a, b, j);
13                  else if (j > maxmatch) {
14                      matches = {match(a, b, j)};
15                      maxmatch = j;
16                  }
17              }
18          }
19          Forall match(a, b, maxmatch) ∈ matches {
20              For j = 0 ... (maxmatch − 1) {
21                  mark(A_{a+j});
22                  mark(B_{b+j});
23              }
24              tiles = tiles ∪ match(a, b, maxmatch);
25          }
26      } while (maxmatch > M);
27      return tiles;
28  }
```

Figure 2: Greedy String Tiling algorithm

4 Winnowing: Local Algorithms for Document Fingerprinting

Schleimer, Saul and Wilkerson, Daniel S and Aiken, Alex. Winnowing: local algorithms for document fingerprinting. 2003.

4.1 Introduction

If you wanted to compare two whole files to see if the are a clone then the obvious method would be to hash the files and compare the hash values. This algorithm applies this to finding partial clones. In short it works by removing irrelevant features from the text, splitting the text into parts of length k called k-grams and then hashing each k-gram. A small subset of these hashes is the derived and this is called the called the fingerprints of the document. The idea is that if two documents share one or more fingerprints then they likely share the same text.

The problem comes in finding the best way to decide which hash to use as one of the fingerprints of the file. This paper purports to give an efficient algorithm to select the fingerprints and also guarantees that at least part of any sufficiently long match is detected.

The paper compares their solution, called winnowing, to other algorithms. It uses the density of the algorithm and it defines the density of a fingerprinting algorithm to be the expected fraction of hashes that are selected from all the hash values computer when given a random input. It defines a local algorithm, which captures certain properties of a fingerprinting algorithm. It must define a window of size w to be w consecutive hashes of k-grams in a document and selects at least one fingerprint from each window. The algorithm is considered local iff the choice of fingerprint of each window only depends on the hashes in that window.

4.2 Algorithm

2 thresholds are chosen by the user, noise threshold k is the lower bound (no matching strings shorter than k), and the guarantee threshold t (match all substrings at least as long).
Noise threshold k is used to divide the text into k-grams. Bigger k reduces noise, but also reduces sensitivity to the reordering of contents.

Window size $w = t–k + 1$

For each window select the minimum hash value to be the fingerprint of the document. If there is a tie use the rightmost minimal value.
The winnowing algorithm is first compared to a $0 \bmod p$ algorithm, which is where the hash is selected if it is a divisor of p. It is then compared to other local algorithms, to see if there is another algorithm that performs better than winnowing. They prove that the winnowing algorithm has a lower bound on the density as good as the optimum local fingerprinting algorithm and therefore there does not exist a local fingerprinting algorithm with lower density.

4.3 Conclusion

Finally the paper shows the real world results with web data, and with plagiarism detection in the MOSS system. They find a problem due to the fact that there are large passages of repetitive low-entropy strings. Here winnowing encounters many identical hash values and therefore many ties for the minimum hash in the window. This causes poor behaviour; to solve this they define Robust Winnowing, which is not a local algorithm. Robust winnowing breaks ties by selecting the same hash as the previous window if possible. Otherwise it reverts back to normal winnowing behaviour. This is used to reduce the density on low-entropy strings. Moss uses this algorithm for its plagiarism detection. They discuss the implementation of Moss and how it stores the fingerprint data, how the comparisons are done against all the entries in the database, as well as how it presents its results.

5 Latent Semantic Indexing

Manning, Raghavan and Schütze. Introduction to Information Retrieval: 18 Matrix decompositions & latent semantic indexing. Cambridge University Press. 2008.

5.1 Overview of technique

Latent Semantic Indexing (LSI) makes use of a term-document matrix, this is simply an MxN matrix, where columns represent the number of documents in the collection and rows represent the terms. Calculation can then be carried out over this matrix to compute the LSI.

In the case of our tool, there would be two columns, for the pair of documents. The terms would be tokens, produced by some simple whitespace parser or potentially more complex grammar.

5.1.1 Low-rank approximation and computing similarity

LSI first requires an approximation of the term-document matrix into lower-dimensional space using singular value decomposition, an extension of symmetric diagonal decomposition. Symmetric diagonal decomposition is a technique in which a square matrix can be factored into the product of matrices derived from its eigenvectors.

This approximation is required as even medium sized term-document matrices can contain tens of thousands of terms, so this is required for LSI to be scalable.

The mathematics of low-rank approximation is too complex to cover in a short summary, but the end result is a reduction of the size of the term-document matrix, while still maintaining differences and similarities between documents close to that of the original matrix.

Once the low-rank approximation has been computed, the similarity between two documents can be calculated using cosine similarity.

5.2 Performance

Dumais 1993 and Dumais 1995 conducted experiments on TREC documents and tasks. They managed to achieve results with precision at or above the TREC median, achieving a top score on 20% of topics. Therefore we can conclude that the accuracy of LIS is, in general, good.

The performance of LSI is noted to be poor in terms of computational complexity. It is noted that "there have been no successful experiments with over one million documents". Although this is something of concern for some, for us this would be a tolerable limitation, considering we are only dealing with two input files.

LSI is noted to "work best in applications where there is little overlap between queries and documents". For calculating document similarity, ideally we would have more linear performance and be similarly good at working with considerable overlap.

LSI is a vector space model and as such treats documents as a 'bag of words'. This could either prove to improve similarity matching or degrade it, as source code can be sufficiently reordered to produce a program with distinct behaviour, but LSI will still produce results that show a close match. Conversely if someone was to rearrange code to attempt to disguise plagiarism, LSI matching would be unaffected.

Finally there are two other limitations of LSI related to its vector space representation, its inability to cope with synonymy and polysemy. Synonymy refers to two distinct words having the same meaning and polysemy one word being having different meanings dependent on use. This would be an issue for programming language code, where certain words, such as access modifiers and Java keywords are used frequently throughout text.

5.3 Conclusion

Overall LSI seems to be a good similarity matching algorithm. However, for our particular use case, matching Java source code, the weaknesses due to it being a vector space model are likely to put it at a disadvantage to other models which take into account word ordering.

6 Measuring similarity using Clone Detection approach

One of the approaches to detecting similarity in code is the Clone Detection. It is able to find the similarities between the code files and also provide the position of duplicates.

6.1 Code Detection

Typically, clone detection is done by comparing the two streams of data and spotting out any identical segments that exists in both of the streams. The comparisons can be based on text, tokens, syntax or structures. The clone detection steps are as follows [1]:

- Raw code is fed into the pre-processing stage where it removes uninteresting codes such as embedded code such as SQL queries in a Java file. It determines the comparison unit such as tokens, hash or characters, etc.
- The processed code is then transformed to the form suitable as input to the actual comparison algorithm. It can tokenize or parse depending on the comparison unit and it removes whitespaces and comments to normalise the code.
- The transformed code is then fed into a comparison algorithm. It aims to find the longest contiguous sequence of similar units. It outputs a list of matches that were found in the transformed code.
- It then maps the detected clones back to the source code by their line numbers.
- Then it can be extracted from the source to be visualised for manual inspection, such as the scatter dot plot.
- Sometimes, in order to reduce the amount of data, it can return only clone pairs as the result.

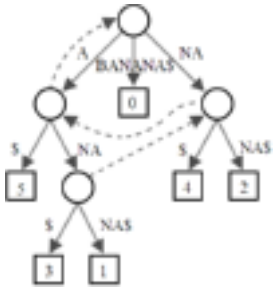

Figure 3: Clone detection

One of the applications that utilises clone detection is the CCFinder. It aims to be used in an industrial scale and to be applicable to million-line size system. The tool takes token sequences as comparison unit through a lexical analyser and applies rule-based transformation to the sequence, which enables the tool to be able to detect cloned code portions that have different syntax but may have similar meaning. CCFinder utilises suffix tree matching algorithm to detect clones. Suffix tree matching provides the position of the detected segments as a tree with sharing nodes that represents duplicates. CCFinder searches the leading nodes on the tree for clone detection.

6.2 Rabin-Karp algorithm used for Clone Detection

The native string search used for Clone Detection may be too slow as it iterates through every message space available. A variant of the algorithm is the Rabin-Karp algorithm. The algorithm is a string search algorithm created by Michael O. Rabin and Richard M. Karp, and is one of the clone detection algorithm. It calculates a hash value for a pattern and for each character substring of the given text. Instead of comparing every combinations of characters or tokens, it compares the numerical value (hash) to detect clones. The hash function returns a shortened codes compared to the raw codes, hence the speed of matching becomes faster than a brute-force string matching.[3]

7 Tool requirements

7.1 Overview

The following list of requirements are formed from the defined set of tool requirements in the coursework brief and based on our algorithm choice, JPLAG plagiarism detection.

7.2 Requirements

1. The tool will compute a similarity measurement for two input Java source code files using an implementation of the JPLAG plagiarism detection algorithm.
2. The tool must work via command line and should not have a graphical interface.
3. Input taken should be the names of two files as command lines arguments.
4. The output should be a similarity measure to STDOUT as a percentage.
5. Input Java source files will be parsed using a Java parser library, ANTLR.
6. The tool will be written in Java (version 6) to meet the requirements for the parser library.
7. The tool will not invoke the JPLAG binary or any other similarity testing tool.
8. The output of similarity measurement by plagiarism detection algorithm will complete in reasonable time for expected inputs. Expected inputs are generally Java source code files and specifically under testing TOH.java, variants of TOH.java and arbitrary dissimilar test inputs.
9. The tool will have an automated testing mechanism to carry out pairwise comparison of the said files.
10. The tool should work with all platforms the JVM works on that support ANTLR. The tool will be tested and run on Linux and Windows platforms.

8 UML Diagrams of architecture

8.1 Sub-section

9 Implementation

The development of the plagiarism detector involved implementing Michael Wise's "Greedy String Tiling" algorithm, as introduced before. In order for the algorithm work flawlessly, we have prepared our own data holder classes: a "Token" class responsible for objectifying each token every loop, and a "Match" class that tracks where the duplicates were detected. The implementation of the pseudo-code of Greedy String Tiling was done in Java. Our tool's workflow is as follows.

9.1 Tokenizing

The Greedy String Tiling compares tokens instead of each characters. It is required that the tool is able to understand the source code (written in Java) and recognise the grammar. We used ANTLR and a Java grammar (from Github, under BSD licence) for ANTLR to convert the source files into token strings. Initially, the given Java grammar dealt with all Java statements in one expression. However, in order to be able to classify different semantics behind the tokens, we needed to add separate method for if statements, for loops, etc. A Listener object was created for the parser to be able to tokenize according to the aforementioned rules. The tokenized string is then passed to the main method to be manipulated for plagiarism detection.

9.2 Algorithm

The Java implementation of the Greedy String Tiling algorithm is as follows: Given the list of Token objects of two source code files, set A and set B, we iterate through A and B with two nested loops. The algorithm compares the tokens to see if they are identical, and to see if the two tokens are not 'marked' (initially set to false). If the condition succeeds, another innermost nested loop repeats this process until the condition does not hold. While it does, it counts the how far the duplicate has extended, and creates a Match object with Index of the duplicated tokens from both sets, and the length of the duplicate - thus it eventually collects the set of all longest common substring. The algorithm has a mechanism where it keeps only the largest duplicates. The program checks whether, in each loop, the match is at least as long as the longest ones found before, whose length is stored in maxMatch variable. If the length is equal to maxMatch, another maximal match is found and added to the global set called matches. If the length is larger than maxMatch, the global set matches is cleared and newly adds the current duplicate. The maxMatch is now the length of the current duplicate. The global set 'matches' is then emptied to the result set called 'tiles'. The matches set is cleared and the above process repeats until maxMatch is lower than a fixed threshold, which we set to be 3. As more duplicates are found, the algorithm will eventually halt when it cannot find any more duplicates that are long than length 3.

9.3 Analysis

The similarity percentage is calculated as follows:
Coverage = sum of all lengths of duplicated codes
Similarity percentage = 2 * Coverage/length of file A + length of file B

Higher percentages indicate a higher chance of plagiarism. It should be noted that given two identical code, if one was to add more content to one of them, the similarity percentage decreases, due to the nature of the calculation.

10 Testing

The aim of testing in this project is to gain an understanding of the chosen algorithm's effectiveness in identifying file similarity. This has been achieved through the writing of a simple testing script in Java and a number of test file inputs.

The test script is written in Java and simply invokes the main method of our similarity testing main class. All possible pairs are tested together, including the reversal of pairs (although these will likely yield identical results), ie x,y and y,x combinations are tested.

The following test inputs have been created and categorised using the criteria of the different clone types:

- Type 1: Identical code segments, except for differences in layout and comments.
- Type 2: Structurally identical segments, except for differences in identifiers, primitive types, layout, and comments.
- Type 3: Similar segments including additions, modifications, and removal of statements, including coding style such as non-OO/OO.
- Type 4: Semantically equivalent segments, ie a clone such that the original code is unidentifiable.

(Source: Tools and Environments slides) The following listed test inputs have been generated with the different types of clones in mind, in order to be able to analyse the results of our pairwise comparison effectively and are therefore listed by category of clone type.

Type 1 Clones
test3: Method order rearrange and additional comments added
test5: Change identation of file

Type 2 Clones
test1: 'Optimize import statements' and access modifiers changed public to protected.
test2: Variable and method names changed.
test4: All ints changed to longs, while loops changed to for loops and for loops changed to while.

Type 3 Clones
test7: Program refactored to 3 methods.
test8: Program altered to be Object-oriented.
test9: Extra methods created through extracting code segments.
test10: A combinbation of extra methods, optimized imports and object oriented style.
test11: Changes in test10 as well as access modifiers.

Type 4 clones
test14: IDE code inspector 'fixes'.
test12: Applied IDE code inspector 'fixes', coding style changed to OO and extra methods added.
test13: Test 12 and all int literals changed to long objects.

Different code for Tower of Hanoi
test 15: Alternative 1: Sanfoundry solution
test 6: Alternative 2: Dickinson college solution

Dissimilar code, solving a different problem
test 16: SimpleWordCounter
test 17: AirlineProblem

11 Results of pairwise comparison

11.1 Type 1 Clones

12 Evaluation of results and project review

12.1 Sub-section

13 Responsibilites

Sachin Pande - Group Leader

- Responsible for setting up the project
- Researched Normal Compressions Distance
- Implemented the parse listener
- Overridden the default ANTLR listener method in order to create a customized token strings
- Implementation of testing code

Yasaman Sepanj

- Researched Plagiarism Detection
- Implemented the Comparision process using the Greedy String Tiling algorithm in Java
- Implementation of testing code

Elliott Omosheye

- Researched Winnowing
- Modified the Java grammar
- Implemented methods that analyses the result and returns the similarity percentage

Jihyun Han

- Researched Clone Detection
- Written the report
- Testing methodology and writing of tests

Luke Richardson

- Researched Latent Semantic Indexing
- Implementation of testing code
- Testing methodology and writing of tests
- Written the report

13.1 References

[1] Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Chanchal K. Roy, James R. Cordy, Rainer Koschke.
[2] CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. Toshihiro Kamiya, Shinji Kusumoto, Member, IEEE, and Katsuro Inoue, Member, IEEEE.
[3] Computer algorithms: Rabin-Karp String Searching from Stoimen?s Web log.
[4] Prechelt, Lutz, Guido Malpohl, and Michael Philippsen. "Finding plagiarisms among a set of programs with JPlag." J. UCS 8.11 (2002): 1016.
[5] Michael J. Wise. String similarity via greedy string tiling and running Karp-Rabin matching, Dept. of CS, University of Sydney, December 1993.
[6] Prechelt, Lutz, Guido Malpohl, and Michael Philippsen. "JPlag: Finding plagiarisms among a set of programs." Technical Report 2000-1, March 2000