Method-Level Code Clone Detection on Transformed Abstract Syntax Trees Using Sequence Matching Algorithms

Kevin Greenan
kmgreen@soe.ucsc.edu
Department of Computer Science
University of California - Santa Cruz

March 16, 2005

Abstract

Current research shows that a large fraction of source code in many large-scale applications contains code clones [4]. The existence of code clones can introduce many instabilities within a software application, such as unnecessary duplicates. These instabilities can over-complicate routine maintenance tasks, since a change in one method may lead to changes across many methods. In addition, unnecessary duplicates can potentially induce the spread of bugs and prevent changes from being propagated [6]. Therefore, in order to prevent and treat these instabilities, we must figure out where potential clones occur. A properly annotated abstract syntax tree supplies a great deal of information about the structure of an application's source code. These trees can be easily transformed into a sequence of substrings. Borrowing a few ideas from biological sequence alignment, similarities between transformed subtrees can be identified. Once identified, an application architect or maintainer can accept (or reject) the similar blocks of code as a (non-)clone. Given the use of subtree transformation, I will present three code clone algorithms in this paper. The research presented serves as a starting point for code clone detection using transformed subtrees and sequence similarity. Thus, my results will justify intuition and open a few doors to future work in the area of code clone detection.

1 Introduction

What exactly is a code clone? Using a variation of the definition presented in [8], semi-formally, two methods are said to be clones if they are identical or near-identical. The word *identical* seems pretty vague. In fact, most of the literature on code clone analysis does not give a concrete definition to the word *identical* with respect to code clone analysis. Basically, if the cardinality of the intersection of two program entities exceeds a prescribed threshold, the two entities are clone candidates. These candidates are usually rejected or accepted through some sort of contextual analysis.

Various studies suggest that many programmers in the software development industry resort to copy and paste techniques, which is generally used as a form of reuse [6]. This form of reuse usually results in code

clones. Unfortunately, practicing copy and paste techniques can create very complicated maintenance tasks [9]. In addition, among many other factors, code clones can arise as a result of design decisions, poor cohesion between modules and poor communication between developers.

Code clones can be detected on an exact match basis. Unfortunately, using exact match criteria for code clone detection may not be sufficient in all cases. For instance, a programmer may copy a particular piece of code, paste it to another location in the system and proceed to change the pasted code such that it remains syntactically similar to the original code, but not exactly similar. Thus, some form of near-exact clone detection must be employed.

This paper presents one exact match and two near-exact match algorithms for code clone detection. These algorithms rely on an abstract syntax tree (AST), which stores attributes such as nonterminal production information, type information, parent file, parent class and line number. All other terminal symbols are intentionally not stored in the AST. Thus, the algorithms are designed to match the structure of the code, with respect to types, rather than match on attributes such as variable names, method names or literals. The effectiveness of the algorithms will be determined by a high-level analysis of the algorithms when run against selected modules in Eclipse and Hibernate. A more detailed analysis will be conducted on code generated manually for the purpose of analyzing the algorithms.

In the next two sections I will cover source code and AST transformation. Section 4 will present the code clone detection algorithms and define the matching criteria used for the algorithms. Then, in sections 5 and 6 I determine the effectiveness of the algorithms and present the results produced when the algorithms are run against modules from Eclipse, Hibernate and manually generated code. Finally, sections 7-8 cover threats to validity, future work and conclusions.

2 Code Transformation

In order to effectively parse the code within the context of my analysis, transformation of the actual source code text is required. These transformations include removal of comments, substitution of literals containing the terminal symbol "//", addition of filename identifiers and finally concatenation of all source files into one large source file. These transformations will be explained in the following paragraphs.

The substitution of string literals can be completed in one swoop, using a stream editor, such as *sed*. I wrote a small C program to remove the comments, without actually deleting the lines themselves. We do not want to delete the lines, since line information should remain consistent throughout transformation and analysis, since we would like to report LOC per method and relative line numbers when reporting method matches. Thus, as an example, Figure 1 illustrates how a source file will be transformed.

As a final step in the source code transformation phase, filenames are added to the first line of each source file. Then, all of the files are concatenated into one large file for the parsing step. By concatenating all of the files into one large file, all of the method subtrees can be extracted in one search.

```
-- Begin test.java}
\begin{verbatim}
// Programmer : John Doe
/* My test class */
/* This class implements the function func1 */

class TestClass {
        // Function : func1
        // Precondition : x is printable (has toString() method)
        // Postcondition : x + "//" is printed out to stdout
        void func1(Object x) {
                System.out.println(x + "//");
        }
};
```

```
-- Begin test.java



class TestClass {


        void func1(Object x) {
                System.out.println(x + LITERAL_SLASHES);
        }
};
```

Figure 1: Transformation of source code.

3 Parse Tree Generation and Transformation

After the source code is transformed, an abstract syntax tree is generated. The parser generator, *javacc*, was used to parse the source code. The package *jjtree*, which reads the parser code generated by *javacc*, is then used to create a parse tree, which only contains nonterminal information. Functionality must then be added to include attributes specific to method-level clone detection. Each node in the AST is attributed with line number, parent class, parent file and type information (where applicable). In addition, the structure of the Java 1.4 grammar downloaded from Sun Microsystem's repository forced most expression productions to degrade to linked lists. Instead of changing the grammar itself, I decided to write an algorithm to flatten the expression subtrees.

After the AST is created and properly attributed, the tree must be transformed into a sequence of strings for input into the code clone algorithms. Since we are only interested in method-level details, the tree must be searched for method definitions. This search can be easily performed on the tree using a depth-first search algorithm. Fortunately, each node in the AST is of the same type, opposed to using n node types for n production rules, which is fairly common. Thus, the traversal of the tree is simply a straight forward depth-first search of an m-ary tree. The depth-first search algorithm identifies method declarations and calls a transformation function. This transformation function is responsible for transforming the method declaration subtree into a sequence of strings. Before transforming the method block into a sequence of strings, header information is added to the front of the sequence. This header contains parent file, parent class, the method name, method start line and method end line.

The remainder of the string sequence for a particular method contains the transformed method declaration subtree. The subtrees are transformed using a pre-order traversal of the method declaration subtree. Starting at the root of the subtree, the toString function is called for each node. The toString function essentially transforms each node into a string. This function has varied behavior, based on the information contained in each node. For instance, the toString function for certain nodes, such as ResultType, will have a unique name and type information, while other will only return a unique name. All nodes have a unique name, which is directly related to the right-hand side of the production rule from which it was produced. As an example, the function getSomething is given in Figure 2 with its corresponding string sequence (filename:classname:methodname:startline:endline;TransformedMethodSubtree).

```
int getSomething(int x) { return x; }

filename.java:className:getSomething:25:25;MethodDeclaration:ResultType:Type( int ):PrimitiveType:MethodDeclarator:MethodName:FormalParameters:
FormalParameter:Type( int ):PrimitiveType:VariableDeclaratorId:Block:BlockStatement:Statement:ReturnStatement:Expression:PrimaryPrefix:Name
```

Figure 2: Source code to string sequence transformation.

All of the algorithms are designed to handle these sequences, which take the form Header;TransformedMethodSubtree. Given the definition of how the sequences are constructed from the method declaration subtrees of the AST, the sequence matching/alignment algorithms for method-level code clone detection can now be presented.

4 The Algorithms

4.1 Definitions

Before presenting the algorithms for code clone detection, I would like to present a few definitions. The following definitions should aid in the understanding of my approach.

- Exact Match Definition
  - Given two sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$
  - A is an exact match of B if and only if $(\forall a_i \in A, \forall b_i \in B \mid a_i = b_i)$, where $0 < i < m = n$
- Near-Exact Match using LCS (definition of LCS taken from [10])
  - Given two sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$, and the sequence $C = \langle c_1, c_2, \ldots, c_k | \forall 1 \leq i \leq k, c_i \in B \rangle$.
  - C is longest common subsequence of A and B if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$, which maximizes the length of C, such that for all $j = 1, 2, \ldots, k$, we get $a_{i_j} = c_j$.
  - If $\frac{\frac{k}{n} + \frac{k}{m}}{2}$ exceeds a prescribed threshold, then a near-exact match exists between the sequences A and B [3].
- Near-Exact Match using Smith-Waterman (Optimal Local Sequence Alignment)[13][11]
  - Given two sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$, we can transform A into B, and vice versa, using the sequence of operations $OP = \langle Insert, Delete, Match \rangle$.
  - Each operation defined in OP has an associated score. $Insert$ and $Delete$ have a negative score, which $Match$ has a positive score.
  - The maximum score, for a subsequence, assigned when transforming the sequence A into B, is the score for the optimal local sequence alignment for A. The same applies when transforming B to A. Thus, scores for both A and B are produced.
  - In the context of code clone detection, the score associated with $Insert$ and $Delete$ will be equal, thus the scores for A and B will be the same. Therefore, the scores associated with A and B will be simply denoted with one score, denoted SWScore.
  - Let MaxSWScore be the maximum possible SWScore for the sequences A and B. Then, a near-exact match between two sequences A and B exists if $\frac{SWScore}{MaxSWScore}$ exceeds a prescribed threshold.

- Hash Table
  - A chained hashing table is used for all hash operations in the algorithms. Thus, in most cases comparisons are limited to the lists for a particular bucket in the hash table.

4.2 Exact Matching Algorithm

The exact matching algorithm accepts any number of transformed subtrees as input. Each sequence is immediately split into three pieces, header, method signature and method block. Every transformed subtree is hashed with the corresponding method signature and method block as the key, which is similar to techniques used in compilers [2] [1]. The header, method signature and method block are then stored as data in the hash table. Figure 3 gives the pseudocode for the exact match algorithm.

```
while !ListOfSubtreeSequences.empty()
        Let S = ListOfSubtreeSequences.getNext();
        (header, signature, block) = split(S);
        table[hash(signature + block)].add((header, signature, block));
end while

foreach key in table.keys()
        List list = table[key];
        while node = list.remove()
        compare(node, list, match_list);
        end while
end for

report(match_list);
```

Figure 3: Exact Match Algorithm

Each node in the match list contains two triples (header, signature, block), one for each method participating in a match. Thus, when reporting matches, file, class, method name and line number information is given for each method in the match.

4.3 Longest Common Subsequence Algorithm

The longest common subsequence algorithm accepts any number of transformed subtrees as input. As in the exact match algorithm, each sequence is immediately split into three pieces, header, method signature and method block. Every transformed subtree is hashed with the corresponding method signature as the key. The header, method signature and method block are stored as data in the hash table. The algorithm for LCS is similar to that of the exact match algorithm, except for the fact that the method signature is the only element in the hash key and the compare function is more sophisticated. The compare function calls

the LCS algorithm for each comparison between elements in a particular chain of the hash table. For brevity, the pseudocode for the LCS algorithm used to compare two sequences is given in Figure 4. The length of the longest common subsequence, which is used to measure similarity, can be extracted from `c[m-1]c[n-1]`.

```
Let x and y be sequences of length m-1 and n-1, respectively.
Let c denote an m x n cost martix.
For all 0 <= i <= m, 0 <= j <= n, c[i][0] = 0 and c[j][0] = 0.

for i = 1 to n
          for j = 1 to m
                if x[i-1] == y[i-1] then
                        c[i][j] = c[i-1][j-1] + 1;
                else if c[i-1][j] >= c[i][j-1] then
                        c[i][j] = c[i-1][j];
                else
                        c[i][j] = c[i][j-1];
                end if
          end for
end for
```

Figure 4: LCS matching algorithm. For brevity, note that the hashing and chain comparison follows from the exact match algorithm, therefore the details are not included in this figure.

Obviously, comparison using the LCS algorithm is costly, since the LCS must be computed between all methods. Due to this cost, as shown above, I only compare methods with the same function signatures, which gives good results from a performance standpoint, but may leave a few clones undetected [7]. This behavior is explained in section 7.

4.4 Smith-Waterman Algorithm

The Smith-Waterman algorithm accepts any number of transformed subtrees as input. The hashing of methods is performed exactly the same as the LCS algorithm. As in the LCS algorithm, the Smith-Waterman algorithm is similar to the exact match algorithm, except for the hashing technique and the way sequences are compared. The compare function calls the Smith-Waterman algorithm for each comparison between elements in a particular chain of the hash table. The Smith-Waterman score, which is used to measure similarity, can be extracted from `maxScore`, given below. The pseudocode for the Smith-Waterman algorithm is given in Figure 5.

```
Let x and y be sequences of length m-1 and n-1, respectively.
Let c denote an m x n cost martix.
Let maxScore = 0.
For all 0 <= i <= m, 0 <= j <= n, c[i][0] = 0 and c[j][0] = 0.

for i = 1 to n
        for j = 1 to m
                if x[i-1] == y[i-1] then
                        matchVal = c[i-1][j-1] + matchScore;
                end if
                if c[i-1][j] >= c[i][j-1] then
                        c[i][j] = c[i-1][j] - insertDeletePenalty;
                else
                        c[i][j] = c[i][j-1] - insertDeletePenalty;
                end if

                if matchVal > c[i][j] then
                        c[i][j] = matchVal;
                end if
                if c[i][j] < 0 then
                        c[i][j] = 0;
                end if
                if maxScore < c[i][j] then
                        maxScore = c[i][j];
                end if
        end for
end for
```

Figure 5: Smith-Waterman matching algorithm. For brevity, note that the hashing and chain comparison follows from the LCS algorithm, therefore, the details are not included in this figure.

It is obvious at this point to note that the LCS and Smith-Waterman algorithms are very similar in structure. One important difference lies in the scope at which the algorithms match substrings. In general, LCS mainly computes global similarity between two sequences, while the Smith-Waterman algorithm computes local similarity. In addition, the Smith-Waterman algorithm can be tuned to facilitate matches among splits in code clones.

5 Determining Effectiveness

How will we ever know if the three algorithms are actually returning valid code clones? From a contextual standpoint, manual intervention is almost a requirement. The algorithms have no contextual bias, since they are simply matching based on sequence similarities in the method signature and code structure. Due

to the unbiased nature of the algorithms, method clones detected where the methods have only a few lines of code is expected. Therefore, I expect all of the so-called *setter/getter* functions to match. Technically, these functions are clones, but from a contextual viewpoint, they may not be clones.

In addition, I also expect the Smith-Waterman algorithm to match splits between clones. If the score is increased for a match, while the penalty for an insert/delete operation is held constant, the algorithm should allow larger gaps in local alignments. This case will be covered in a manually generated code example in section 6.

Although, results will be presented for a few large systems, the effectiveness will essentially be determined through manually generated code (i.e. code created for testing the code clone detection algorithms). Careful attention will be paid to data from the large systems. Data from these large systems does show evidence of valid matches, but does not show if possible clones are not considered due to deficiencies in the algorithms.

6 Results

Scripts were written to run the clone detection algorithms against Eclipse, Hibernate and the manually generated code. The scripts run each algorithm on a specified module with different levels of similarity (.5, .75, .9, .95). Functionality was put in place to detect clones between many modules (i.e. Eclipse vs. Hibernate). Unfortunately, that type of comparison was found to be beyond the scope of this study. All of the data is collected from the output generated by LCS and Smith-Waterman. The exact match algorithm was simply used as a verifier for the LCS and Smith-Waterman algorithms. All output was also transformed into summary files and data files to be used with R and GNUplot for graph plotting.

The graphs in Figures 6 and 7 show the number of method clone matches between methods with varying lines of code. These graphs agree with my initial intuition. The graphs show that most of the matched methods have very few lines of code. In fact, most matches had two lines of code. Upon further inspection, it became obvious that many of these matches were *setter/getter* methods. As previously mentioned, as per the definition given for a code clone, when we disregard context, all of these functions are code clones. This observation supports the use of human intervention in the code clone detection process. The graphs presented in Figures 8 and 9 also support this claim. The plots in Figures 8 and 9 show data taken from a run on the Hibernate base module. These plots have the same meaning as the plots in Figures 6 and 7, but give very little information. Very few relevant clones were extracted from the Hibernate base module. As previously mentioned, the only information given in the plots for the Hibernate base module is the overwhelming presence of method clones whose individual methods have very little lines of code.

Figure 10 shows the clone similarity among all methods in the Eclipse runtime module, which gives a good summary of the total number of clones detected and proportion of similarity between the clones. The graphs in Figure 10 do not give much information about how well the algorithms are working. These graphs would give a great deal of information to a maintainer or manager. If the plot was populated with many long impulses (i.e. many high similarity clones), then someone should perform a detailed clone detection analysis of the system. Basically, the plot could be used as an indicator of high clone density in a system.

Little can be said about the validation of the algorithms from the graphs produced by Eclipse and Hibernate. Essentially, the results show that most of the clone matches involve methods with very few LOC. In addition, fewer clones are detected as the similarity ratio is increased, which is a fairly weak measure of algorithm validation. After a closer look at a sample of the clone matches, the algorithms are matching clones (which might be out of context or small accessor/mutator methods), but human intervention is still needed to reject or accept the clones.

Figure 11 depicts a manually created set of clones. The intention was to determine how well LCS and Smith-Waterman detected clones, which involve a split. This plot follows my intuition, since LCS matches globally with 75 percent similarity, while Smith-Waterman (with match score is 4 and insert/delete penalty is 1) loses momentum in the split. If a larger match exists after the split, Smith-Waterman will return a larger similarity. Based on this observation, it is possible that LCS and Smith-Waterman should be used to calculate a combined weighted average for similarity, since Smith-Waterman seems to give an inaccurate similarity in this case. This calculation will have to be incorporated in future work.

Figure 12 also shows a manually created set of clones. The figure illustrates a case where the exact match algorithm does not classify a set of methods as clones, when they are in fact clones. Both LCS and Smith-Waterman classify these methods as clones, with high similarity.

The summary presented in Figure 13 presents a comparison between LCS, Smith-Waterman and the exact match algorithm, by assigning frequencies of clones per class in the impl module of Hibernate. In addition, the contents of the table in Figure 13 support the validity of the clone detection algorithms. If the exact match algorithm is returning only exact matches and the LCS and Smith-Waterman algorithms return matches of varying similarity, then it follows that the exact match algorithm should return no more than the number of matches returned by LCS and Smith-Waterman, which is true by the data given in the table. Also, note that LCS seems to be much more liberal in terms of accepting two methods as clones, since the frequency of clones between classes is much larger for LCS.

In summary, through my analysis, I have determined three major outcomes. First, all of the algorithms presented match a great deal of methods, which have a small value for LOC. Instead of simply ignoring the clones with short methods, manual intervention is needed to reject or accept. Second, through manual analysis, it is apparent that all of the algorithms do in fact match clones, but more attention is needed when tuning the parameters to LCS and Smith-Waterman (mainly Smith-Waterman). Lastly, better results can possibly be achieved by using a combined weighted average of LCS and Smith-Waterman, since LCS seems to mainly give a global similarity measure, while Smith-Waterman tends to give a local alignment similarity.

7 Threats to Validity

Two very important cases threaten the validity of these algorithms. First, based on the analysis of the algorithms against Eclipse and Hibernate, most of the matches tend to be methods with 1-5 lines of code. While most of these methods were *setter/getter* functions, it is possible for a contextually valid set of clones

to contain few lines of code. Thus, simply rejecting clone candidates with few lines of code will not fix this problem. This threat is a result of the unbiased behavior of the clone detection algorithms. One way to alleviate this problem would involve a learning mechanism, which will be presented in future work.

Another threat involves the technique used to determine sets of methods to compare, as mentioned by [7]. As a result of the algorithms presented, in order for two methods to be compared, their signatures must match in return type, number of parameters and parameter type. The technique used to group methods for comparison was chosen for performance. For instance, if a module contains 300 methods, a more naive comparison mechanism would result in 300^2 calls to LCS and SW, which is very expensive. On the other hand, a great deal of work has gone into method signature analysis, which could still result in good performance. The inclusion of signature analysis will be discussed in section 8.

8 Future Work and Conclusions

Based on the results presented in this paper, we have learned how sequence matching algorithms can be used in the area of code clone detection. As I mentioned earlier, this study serves as a starting point, thus a lot of work is still necessary. Obviously, more work needs to go into tuning the parameters of the Smith-Waterman algorithm, within the context of code clone detection. Thus far, I have found that by increasing the distance between the scores for a match and insert/delete in Smith-Waterman, splits can be detected. Unfortunately, by increasing this distance, many false positives may arise. It is my belief that good parameters can be found for these algorithms, which support such code clone split detection between methods.

The technique used to hash the methods for comparison also has limitations. Hashing was used in this study to increase performance. It has become quite obvious that the hashing technique poses a threat. Thus, more work will be necessary in the field of signature analysis. Other areas of interest include a study of the learning behavior of the Smith-Waterman algorithm [11] and modifications to the LCS algorithm [5]. Modifications in the area of merge/split analysis and other forms of clone detection should also be incorporated into this work, such as work done by [14] [12].

Given the time constraints for this project I was unable to run a history analysis, which would show how clones arise as a system evolves. Most of the work completed for this study focused on implementing the algorithms and attempting to understand how they can be used in clone detection. Ideally, a history analysis should be put on hold until the algorithms can be tuned to a high level of accuracy and be used effectively without much human intervention. A visualization, such as a clone matrix was also planned for use in this project. Again, due to time limitations, such a visualization could not be created. The code created for this project was designed to include such a visualization, thus one can eventually be created. A prototype, which is shown in Figure 14, was created using gray-scale mapping.

Even though the algorithms seem to generate a decent amount of false positives, these false positives arise as a result of contextual analysis. The limitations imposed by contextual analysis can possibly be automated through some sort of learning. Through data analysis and example, I have shown that both the LCS and

Smith-Waterman algorithms can be used for code clone detection. Hopefully, more work will go into tuning the parameters, which could yield a very powerful tool in clone analysis.

References

[1] Jeffrey D. Ullman Alfred V. Aho, Ravi Sethi. *Compilers Principles, Techniques and Tools*. Addison-Wesley.

[2] Steven W. K. Tjiang Alfred V. Aho, Mahadevan Ganapathi. Code Generation Using Tree Matching and Dynamic Programming. In *ACM Transactions on Programming Languages and Systems*, 1989.

[3] Junwei Li Yun Yang Arun Lakhotia Andrew Walenstein, Nitin Jyoti. Problems Creating Task-relevant Clone Detection Reference Data. In *IEEE Working Conference on Reverse Engineering*, 2003.

[4] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo SantAnna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance*, 1998.

[5] Walter F. Tichy James J. Hunt. Extensible Language-Aware Merging. In *ICSM*, 2002.

[6] Miryung Kim. An Ethnographic Study of Copy and Paste Programming Pratices in oopl. In *Proceedings of the International Symposium of Empirical Software Engineering*, 2004.

[7] Sung Kim. Signature analysis techniques. private communication, 2005.

[8] Bruno Lague, Daniel Proulx, Ettore Merlo, Jean Mayrand, and John Hudepohl. Experience report: Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance*, 1997.

[9] M. M. Lehman. Rules and Tools for Software Evolution Planning and Management. In *Annals of Software Engineering*, 2001.

[10] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*. MIT Press, 2003.

[11] Nam Tran. Local Alignment: Smith-Waterman Algorithm. tutorial, 2004.

[12] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *Proceedings of the International Workshop on Program Comprehension*, 2002.

[13] Philip Zigoris. The smith-waterman algorithm. private communication, 2005.

[14] Lijie Zou and Michael W. Godfrey. Detecting merging and splitting using origin analysis. In *Proceedings of the Working Conference on Reverse Engineering*, 2003.

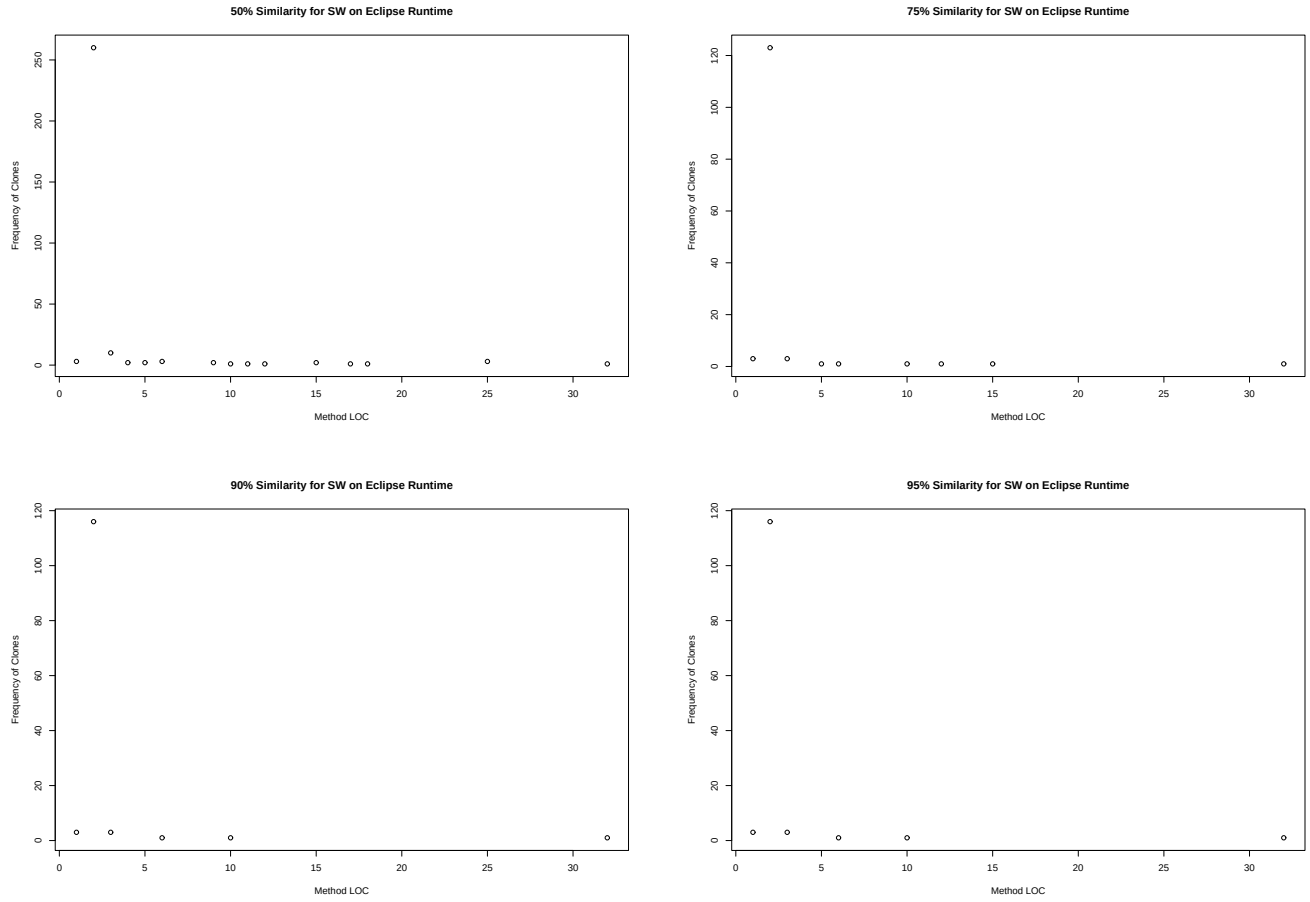

Figure 6: LOC vs. Number of Matches for Eclipse using the SW algorithm with similarity threshold .5, .75, .9 and .95.

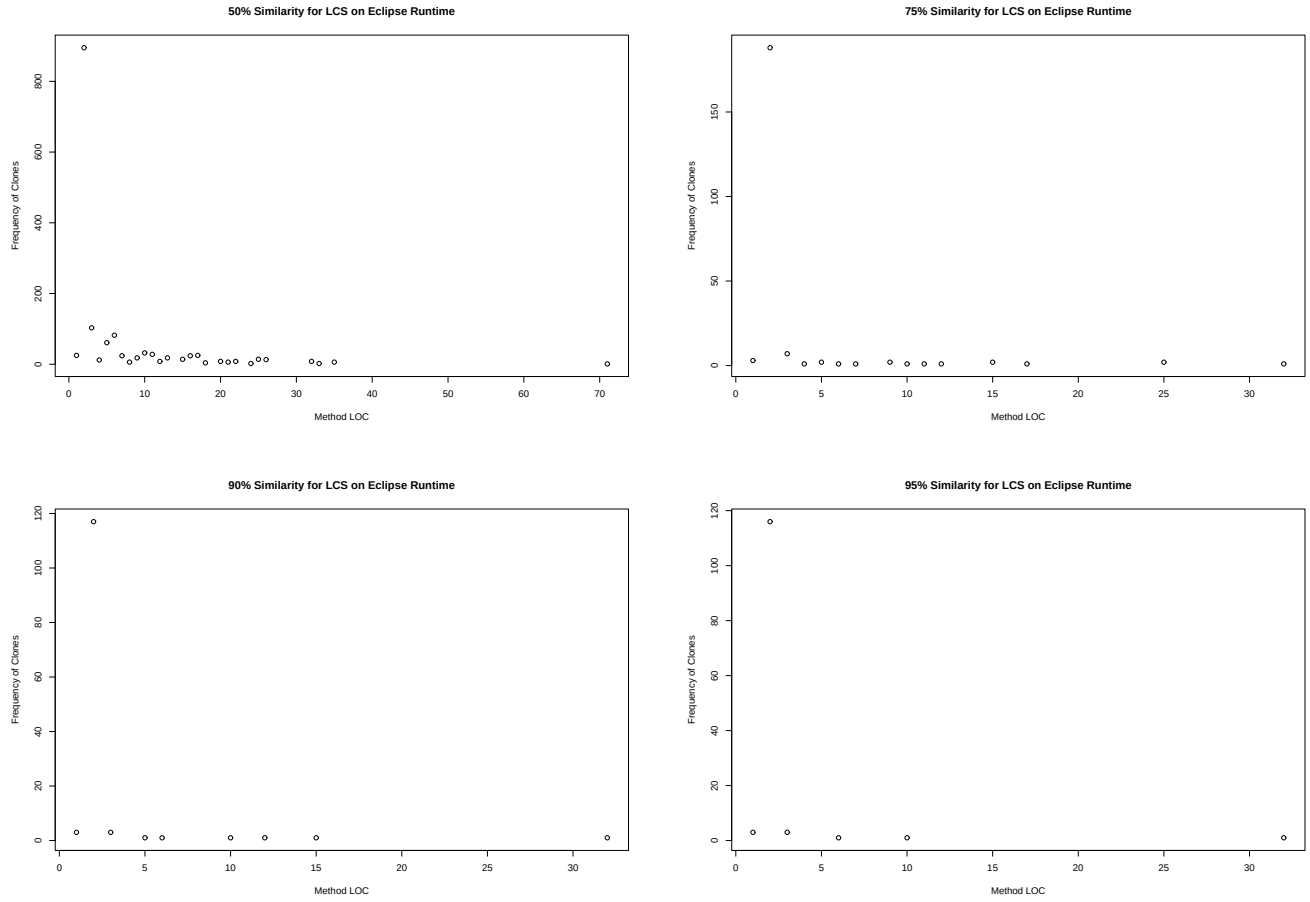

Figure 7: LOC vs. Number of Matches for Eclipse using the LCS algorithm with similarity threshold .5, .75, .9 and .95.

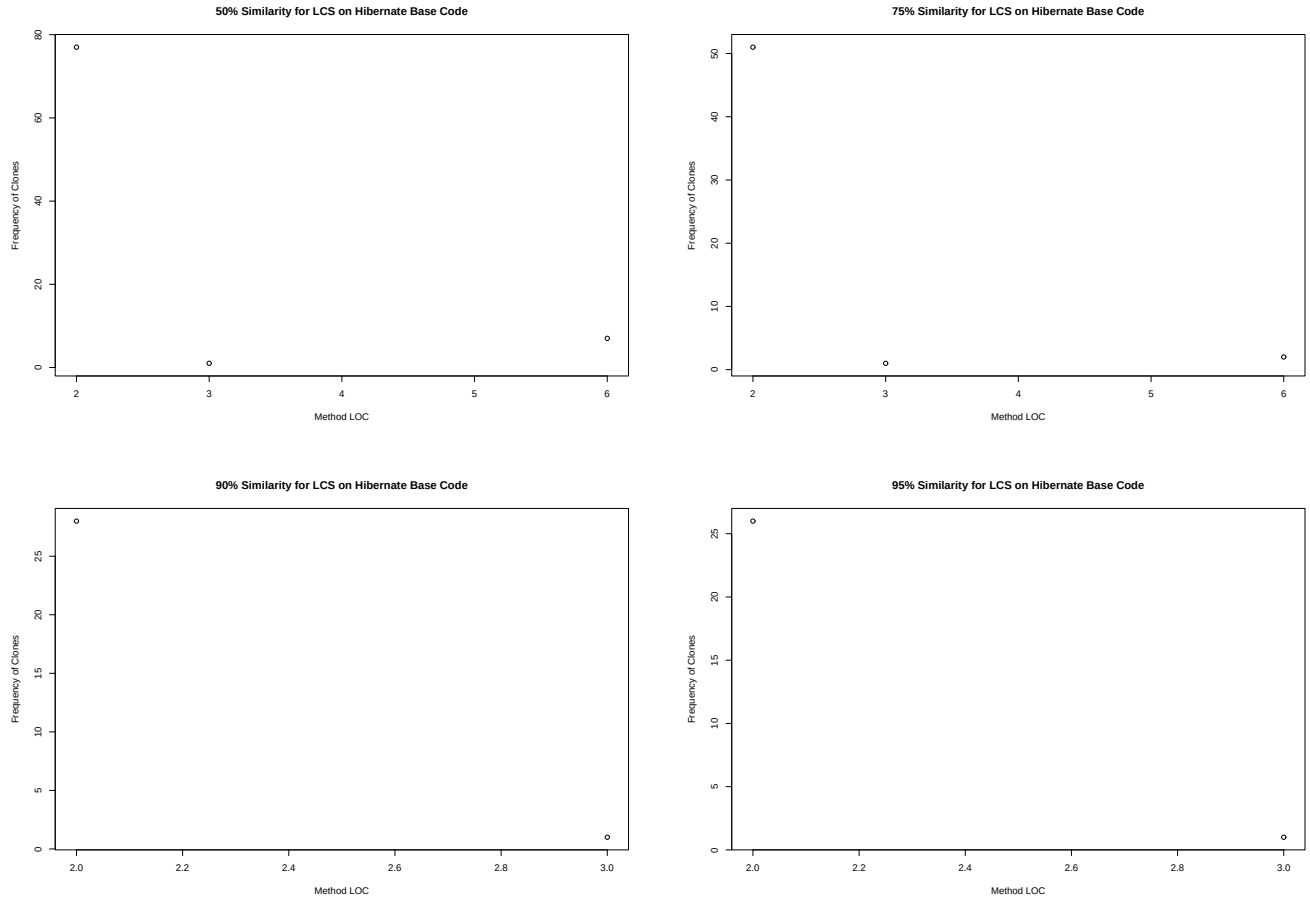

Figure 8: LOC vs. Number of Matches for Hibernate using the LCS algorithm with similarity threshold .5, .75, .9 and .95.

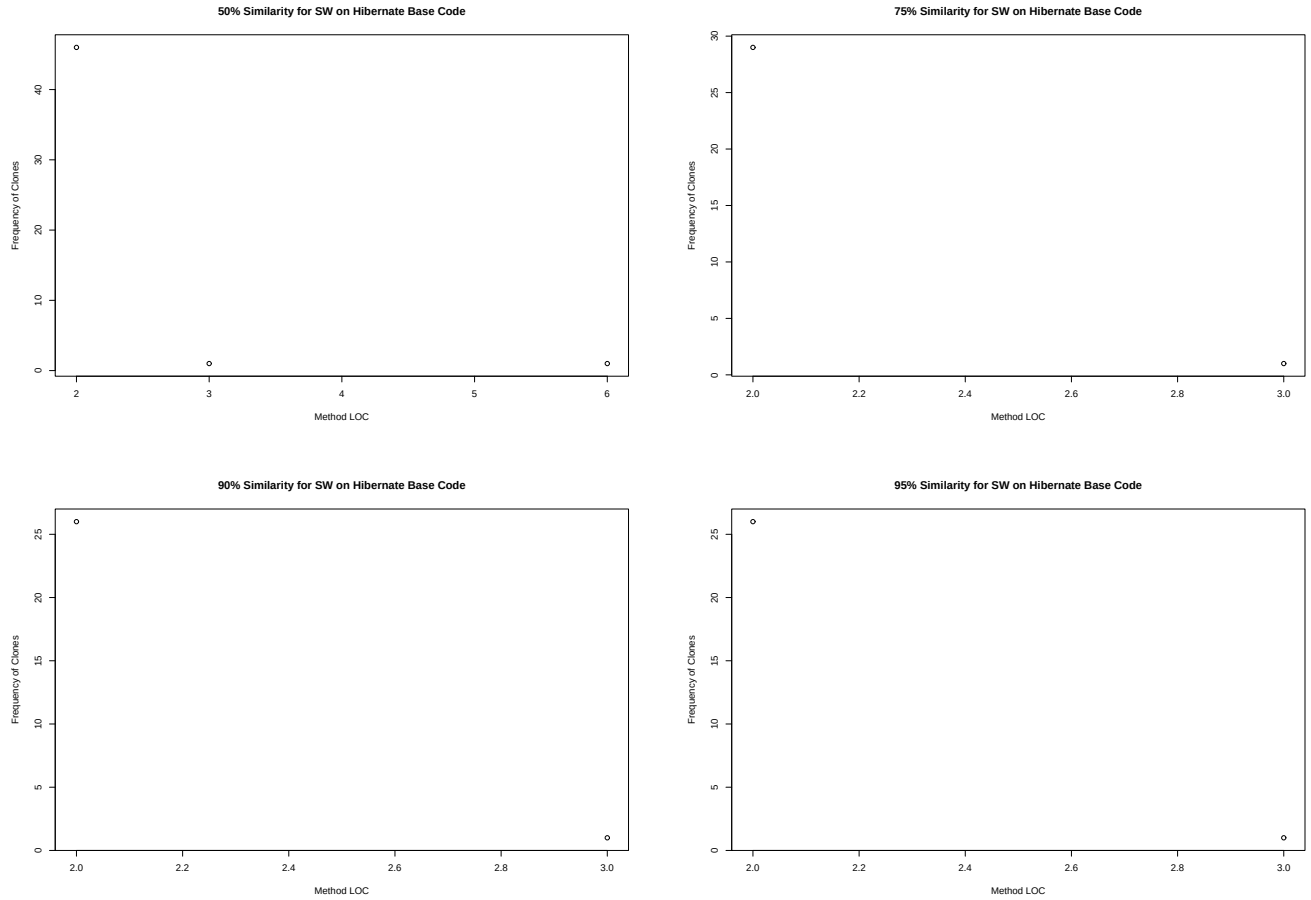

Figure 9: LOC vs. Number of Matches for Hibernate using the SW algorithm with similarity threshold .5, .75, .9 and .95.

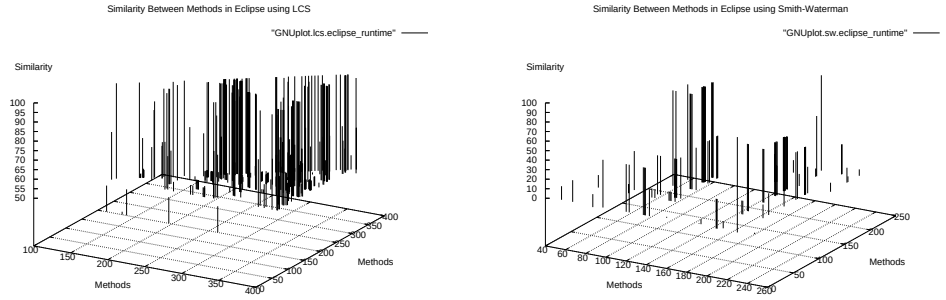

Figure 10: Method vs. Method vs. Similarity for Eclipse.

```
float testSplit1() {
       float x, y;
       if(checkAnotherThing("SOME_CONSTANT")) {
              x = foo();
       } else if(checkAnotherThing("ANOTHER_CONSTANT")) {
              y = bar();
       }

       for(;;) {
              System.out.println("Hello World!");
       }

       float barBaz = foobar("BAZ");
       float bazBar = barbaz("FOO");

       return barBaz + bazBar;
}
```

```
float testSplit2() {
       float x, y;
       if(checkAnotherThing("SOME_CONSTANT")) {
              x = foo();
       } else if(checkAnotherThing("ANOTHER_CONSTANT")) {
              y = bar();
       }
       return x + y;
}
```

Figure 11: Splitting Example for Smith-Waterman tuning. LCS matches these methods with similarity = 78.77 percent, while the Smith-Waterman algorithm matches with 50.94 percent similarity.

```
int typesMatterNamesDoNot1() {
       int x=1, y=5;
       if(x==1) {
              if((x & y == 1 && x | y == 5) && x + 4 == y) {
                     x = y = 0;
              }
       }
}
```

```
int typesMatterNamesDoNot2() {
       float z=1, t=5;
       if(x==1) {
            if((z & t == 0 && z | t == 4) && z + 5 == t) {
                  z = t = 1;
              }
       }
}
```

Figure 12: The exact match algorithm will not classify these methods as clones, since the declaration at the beginning of the methods differ in type. The LCS algorithm matches these methods with similarity = 98.64 percent, while the Smith-Waterman algorithm matches with similarity = 97.97 percent .

Class	LCS	Smith-Waterman	Exact
ScheduledCollectionRecreate	1		
ScrollableResultsImpl	118	41	7
SessionFactoryImpl	241	76	7
CollectionEntry	931	92	7
ScheduledDeletion	6	1	
ScheduledCollectionRemove	2	1	
ScheduledEntityAction	6		
ScheduledUpdate	9	3	1
RowSelection	167	74	11
NonBatchingBatcher	3		
EntityEntry	5		
EmptyInterceptor	13	3	3
CollectionPersister	158	89	22
ScheduledCollectionAction	4		
DatastoreImpl	19	5	2
ScheduledCollectionUpdate	3		
BatcherImpl	5	1	
ScheduledInsertion	12	3	
FilterImpl	8	1	
IteratorImpl	19	11	1
SessionFactoryObjectFactory	55	5	1
Status	20	10	1

Figure 13: Frequency of clones for classes in a module of Hibernate. This table can be used to compare the exact match, LCS and Smith-Waterman algorithms.

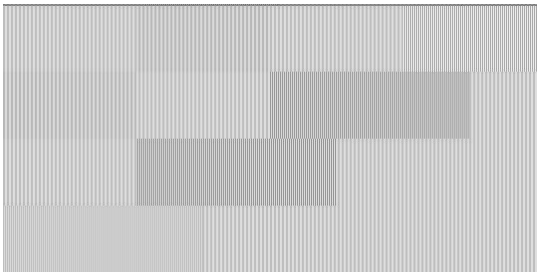

Figure 14: A rather crude idea for a visualization. This is a clone similarity matrix. Each method has a row, which intersects all other methods. The lighter a cell (method vs. method relationship) is colored, the higher the similarity. Ideally, a user can click on a cell and get code listings for each method.