**BACKGROUND**

Clone types can be categorized into four types based on both textual and semantic similarities (taken from Roy et al).

Type-1: Identical code fragments except for variations in whitespace, layout and comments.

Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type-4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

**ORIGINAL CMCD TECHNIQUE**

To make our contribution clear a good understanding of the original CMCD technique is needed, which we provide below. The modifications we made are described in the next section.

The CMCD technique determines the similarity between two code fragments by modelling each with a count matrix and comparing the count matrices. A count matrix is made up of a set of count vectors. In the original CMCD, there is one count vector for each variable that appears in the code fragment. The values in a count vector come from a set of counting conditions that are applied to the variable that vector represents. The counting conditions represent how a variable is “used”. The intuition is, if two code fragments are indeed clones, then a variable in one fragment will have a counterpart in the other fragment that is used in very similar ways, so the count vectors will be very similar. Also, most variables in one fragment will have counterparts in the other fragment, so the count matrices will be very similar. If, on the other hand, the fragments are very different, then there is a high probability that many variables in one fragment will have no obvious counterpart in the other, so the count matrices will look different.

THE ORIGINAL COUNTING CONDITIONS.

1. Used
2. Added or subtracted
3. Multiplied of divided
4. Invoked as parameter
5. In an if-statement
6. As an array subscript
7. Defined
8. Defined by add or subtract operation
9. Defined by multiply or divide operation
10. Defined by an expression which has constants in it
11. In a third-level loop (or deeper)
12. In a second-level loop
13. In a first-level loop

Two count vectors are compared by computing the normalized distance between them. The original technique uses Euclidean distance and normalizes (roughly) by dividing by the vector lengths (see paper for full details.

After computing the count vectors for each variable for each code fragment, the resulting count matrices need to be compared to determine similarity. An issue arises in that, while each variable in one fragment may have a very similar counterpart in the other fragment, this may not be obvious if the order of the count vectors is different in the count matrices, that is, it is not enough to just compare the first row of one matrix with the first row of the other, and so on. CMCD resolves this issue using maximum weighted bipartite matching as follows.

Each row in the two matrices being compared is treated as a vertex in a graph, and each vertex from one matrix has an edge to every vertex in the other matrix. Each edge is weighted by the distance between the two respective count vectors. This results in a weighted bipartite graph. The maximum weighted bipartite matching of this graph is then a pairing of count vector from one matrix with a count vector in the other matrix that maximizes the sum of the count vector distances. This sum is then the measure of similarity between the code fragments.

**REFINEMENTS:**

We have added the following features in additions to original list

1. Defined by String Literals
2. Defined by Character Literal Expression
3. Defined by Null Literal Expression
4. Defined by Boolean Literal Expression
5. Defined by Numeric Literal Expression
6. Defined by another Variable
7. Assigned by Expression with Literals
8. Assigned by Add or subtract operation
9. Assigned by multiply or divide operation
10. Assigned by another Variable
11. Third Level Member Accessed
12. Second Level Member Accessed
13. First Level Member Accessed
14. Defined by Type
15. In Case Statement
16. In Switch Statement
17. In Default Statement

In addition to additional features we have added the following heuristics

1. If the number of nodes difference is greater than 3 on each level of abstract syntax tree, the method will be evaluated as different.
2. If the total number of nodes in abstract syntax tree is greater than 10, the method will be evaluated as different.
3. If the total variable difference between two method is greater than 7 the method will be evaluated as different
4. If the number of variables is less or equal 5, the distance between will not be normalized, instead we used the absolute difference as score.
5. If the number of variables is less or equal 3, the distance between will not computed and the method will be skipped, since there is enough variable to get the full context of the methods and this will avoid false positive.

**METHODOLOGY**

1. Implement original CMCD to the best of our understanding
2. Perform evaluations on the 4-types of clones and on production source code
3. Apply our modifications on the original CMCD techniques and measure the improvement using the steps 2 above.

**Evaluations Results**

1. Scenario based evaluations:

Both original CMCD and refined CMCD can detect the four type of clones.

|  |  |  |
| --- | --- | --- |
| Clone Type | Original | Refined CMCD |
| Type-I | YES | YES |
| Type-II | YES | YES |
| Type-IV | YES | YES |
| Type-III | YES | YES |

1. Evaluation on real world source code (Movie segment frontend source code in search engine.)

|  |  |
| --- | --- |
| CMCD Types | Precision % |
| Origin CMCD | **72.6%** |
| Refined CMCD | **84.71** |

Total methods in source code is **928**, refined CMCD detected **133/928= 14.33/%** of the source code as duplicate and can be refactored. That means about **7.1**% of the methods source can be refactored since 14.33% is method pair.

We didn’t measure the recall for both version of CMCD because, it needs human labelling and its daunting task to go through each methods and mark pairs as duplicate or not. If we run the original CMCD without heuristics such as minimum number of variable need to compare; the precision is below 30% since it will detect most of them methods similar if the count matches, but minimum variable length is needed to gather full context of the methods.

**Future works:**

Both original and refined CMCD struggles when the methods are short and it is just wrapper method, we will leave this plus detecting clone at block level and performance improvement as futures works.

REFERENCES

1. Y. Yuan and Y. Guo, “CMCD: Count Matrix Based Code Clone Detection,” in 2011 18th Asia-Pacific Software Engineering Conference. IEEE, Dec. 2011
2. Chen, X., Wang, A. Y., & Tempero, E. D. (2014). A Replication and Reproduction of Code Clone Detection Studies.
3. C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Science of Computer Programming
4. I. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). IEEE Comput. Soc, 1998.