Duplicate Code Detection Algorithm

Todor Cholakov, Dimitar Birov

Abstract: *In this paper we propose an algorithm for detecting duplicate fragments of source code based on call graphs. The complexity of the proposed algorithm is estimated and the practical performance is tested using several executions of the algorithm on three different versions of open source Ant. An analysis is made of the influence of the algorithm parameters on its results. Additionally a visualization approach for displaying the results is represented, demonstrated by a tool implementing the algorithm.*

Keywords: *duplicated code detection, clone detection, source code analysis and manipulation, reengineering, refactoring*

INTRODUCTION

In this paper we present an algorithm, using call graphs for finding duplicate code in the source code of software products.

During the development process of a software product its code changes and evolves during the time of its lifecycle. These changes normally are performed by different developers, which have different knowledge about the structure and the design ideas of the software product. As a result it often happens that some functionalities are duplicated in different parts of the code. According to some authors [1; 2] between 7 and 23 percent of the code in large projects is duplicate. Although sometimes duplicate code is useful [3] , in most cases it causes problems on later development stages [4] – the code becomes difficult to read and changes to one of the duplicate fragment (for example bug fixing) are not automatically applied to the other "duplicate" fragments.

Such problems are usually detected by accident by the developers or during thorough review of the code in order to refactor it. However in most cases such problems stay unnoticed or even when they are discovered, nothing is done in order to resolve them, because a lot of time and efforts are needed to understand the code by the developers and any changes may cause unpredictable changes in the behaviour of the code. The automated code analysis makes the detection of duplicate code much more efficient.

There are lots of algorithms for detection of duplicate code [5; 6; 7; 8; 9], each of them having its advantages and shortcomings. In this paper we propose a new algorithm for effective detection of duplicate code within the source code of a software project. The main goals and results of our work are the following:

1. To create an algorithm for automatic and systematic discovery of duplicate code within the whole code base of the analysed software.
2. The algorithm and its implementation must be fast enough to process large code bases.
3. High accuracy of duplicate code detection. This means to minimize the number of code fragments, wrongly recognized as duplicates and to minimize the number of duplicate code fragments, which are not recognized as such.
4. High flexibility of recognition. This means that the maximum number of duplicate code fragments are detected, ignoring some insignificant differences.

CompSysTech'15, June 25-26, 2015, Dublin, Ireland

 ISBN 978-1-4503-3357-3/15/06...$15.00
http://dx.doi.org/10.1145/2812428.2812449

The proposed algorithm takes as input two call graphs, representing the analysed software on two different levels of detail.

Definition: A call graph for a given software project or part of it, is a graph whose nodes are methods, and whose arcs are method calls.

Depending on which constructs within the method body are considered calls, the call graphs may have different levels of detail:

1. The simplest case of call graph is to consider only methods as nodes and direct method calls as arcs in the graph;
2. The more complex case of call graph is to consider as nodes both methods and constructors. In this case the arcs are either method or constructor calls;
3. The most complex case of call graph is if we consider as arcs method calls, constructor invocations, field references, field assignments, loop operators and conditional operators as calls. This case specifies the most detailed and informative graphs.

In the proposed algorithm, we use call graphs, created according to **case 1** and **case 3** in different parts of the algorithm.

Depending on the labeling of the arcs, there are two kinds of call graphs:

1. In the canonical call graphs, each arc is labeled with the location of the call in the source code. There may be multiple arcs from node *A* to node *B*.
2. In the statistical call graph each arc is labeled by the number of calls denoted by it. There may be only one arc from node *A* to node *B*.

Before we describe the algorithm itself we are going to give some definitions of the term "duplicate code". Depending on the chosen definition we would get different results. The definitions below closely correspond to the classification proposed by Roy [2], except that the last case of his classification is not considered because it is impossible to be solved algorithmically.

DEFINITIONS OF "DUPLICATE CODE"

Definition 1. Two code fragments are duplicate when after removing all comments and whitespace and their replacement with a single space characters, the result code fragments are the same.

This is both the most simple and the most constraining definition, because in many cases there are insignificant changes between the two pieces of code and these cases will not be recognized as duplicate code.

The above definition is not suitable in most cases where the two code fragments have been written independently and later evolved to being similar. For such fragments some common refactorings may prove useful – for example extracting a method or a class. However if we want to detect more cases we need a new definition of the "duplicate code" term.

Definition 2. Two code fragments are duplicate code if after applying the code transformations described in **Definition 1**, the unification of the two fragments is possible through direct substitution of identifiers.

This definition covers all cases where the fragments are equivalent except variable names, but also covers the cases where different methods are called. The number of duplicate code fragments detected by this definition is much larger than the previous one. However here arises the question if it is possible to unify the duplicate fragments found and if there is an effective algorithm that would transform both fragments to a single one, and to preserve the functionality of each of them. The answer to this question is not simple and depends on the language on which the code fragments are written and on the code itself.

Some languages such as Java [10] and C++ [11] provide mechanisms for passing methods or fields as parameters, allowing the developer to call the passed method or

access or change the field in the method body. In some cases even the difference in the types can be resolved. However the usage of these mechanisms makes the resulting code more difficult to read and maintain and should be used with care. However, some cases of duplicate code, allow usage of heuristic algorithms, for the unification which do not increase the complexity of the code. As a conclusion we may state that unifying duplicate methods according to **Definition 2** is useful in the following cases:

- The methods differ only in the names of local variables, but there are no differences in the communication with the outside world.
- The length of the methods is big, and there are only a few differences.
- Heuristic methods for transformation are applicable, which do not increase the length and the complexity of code.

Although the second definition covers a large number of cases for duplicate code, it also has room for extension.

Definition 3. Two code fragments are duplicate code, when large parts of their code are duplicate code according to **Definition 2**.

This definition allows detection as duplicate code these code fragments where they have a lot of common code, but have some differencing parts. Such situations happen when a method is copied and then code has been added, changed or removed from the copy. The questions if and when is it possible and useful to unify the duplicate fragments rise again regarding such duplicates.

At first we are going to give an answer for the question whether it is possible to eliminate such duplicate code fragments. Let us consider two methods m_1 and m_2, which are considered duplicate code according to **Definition 3**. This means that they can be represented in the following form:

$$m_1 = A_1 x_1 A_2 x_2 \dots A_n x_n$$
$$m_2 = B_1 y_1 B_2 y_2 \dots B_n y_n$$

Where A_i is duplicate to B_i according definition for each $i \in [1, n]$, and x_j is different from y_j for each $j \in [1, n-1]$ and x_n is different than y_n or both are the empty word. It is possible to eliminate the duplicates in case each x_i and each y_j may be extracted as separate methods. After performing such transformation the two methods will become duplicate code according to **Definition 2**.

The question if and when such solution is useful is difficult to be answered. When the similar parts of the two methods are large enough, their refactoring may appear useful, because it would make the maintenance easier. At the same time each difference causes the appearance of two new methods according to the proposed refactoring technique. As a result when there are many differences the refactoring makes the code more complex, rather than simpler.

AN ALGORITHM FOR DUPLICATE CODE DETECTION

After we have specified the definition for duplicate code on which we base our work, it is time to describe the algorithm for detecting duplicate code itself. The algorithm is constraind to looking for duplicate methods only:

An algorithm for detection of duplicate code based on call graphs

1. A statistical call graph G_s for the whole product is created, and its arcs are denoted only by method calls. Each node n_i of the graph has a unique numeric id. There is no specific order of the ids.
2. For each node n_i of G an identifier string is defined, consisting of the frequencies of the calls to its neighbours ordered in decreasing order, followed by a colon and the corresponding nodes ids. For example if the method corresponding to the node n_5 calls the methods corresponding to n_4,n_7 and n_{12} 4,2 and 5 times respectively, the resulting identifier would be „5,4,2:12,4,7“
3. The identifiers from the previous step are sorted in increasing order.

4. Each two identifiers from the resulting list, which are next to each other or have the same frequencies of the calls, are potentially duplicate code. If two identifiers have the same sets of frequencies, but different node ids, they may be duplicate code according to **Definition 2**, and if the node ids are also equal they may be duplicate according to **Definition 1**. All such pairs, which have more than ***minimumLimit*** arcs, are saved in a separate list for further analysis.
5. A canonical call graph G_c is created for the analysed code. The graph must include as nodes the constructors and the fields, too. Additionally the conditional and the loop operators may be included as special functions.
6. The pairs from step 4 are analysed, by using the canonical call graph in the following way:
   Let n_1 and n_2 are the currently analysed nodes and out_{n_1} and out_{n_2} are their corresponding numbers of outbound arcs.
   a. If $out_{n_1} \neq out_{n_2}$, then the currently analysed pair is not duplicate code.
   b. If out_{n_1} <**minimumDetailLimit** (see below) or out_{n_2} < **minimumDetailLimit**, the pair is skipped and is not analysed any more.
   c. The outgoing arcs of n_1 and n_2 are sorted by their place in the source code.
   d. The two lists of arcs are processed and the following operations are performed on each step:
      i. If the node at the end of the arc from the first list has the same temporary id as the node at the end of the arc from the second list, the algorithm continues to the next iterations.
      ii. If both nodes have no temporary ids assigned, they are assigned a new temporary id (the same id for both nodes).
      iii. If both nodes have different temporary ids or one of them has a temporary id and the other has not, the processing is stopped and the two pieces of code are considered as not being duplicate code.
   e. If after the processing of the two lists of arcs it is not stated that the two pieces are not duplicate code, the pair is remembered as being duplicate code. In this case the two nodes from the statistical call graph are added to the same cluster. If any of the nodes already belongs to a cluster, the other node is added to the same cluster. Otherwise n_1 and n_2 are added to a new cluster named after the first of them.
7. All nodes in the original statistical call graph, which do not belong to a cluster, are removed. As a result we get a graph containing only the duplicate methods, grouped in clusters by similarity. In some cases there may be more than two duplicate methods in a cluster.

The algorithm was executed on several versions of Ant [12] – 1.3, 1.5 and 1.7, having different sizes of the code base. The execution of the algorithm on Ant 1.5, which is of a medium size, detected 242 duplicate methods.

PARAMETERS OF THE ALGORITHM

The represented algorithm accepts the following parameters:

1. ***minimumLimit*** – This parameter constrains methods which have less than the specified number of outgoing arcs in the statistical call graph not to be analysed for being duplicates. Thus the simplest methods, that do not call other methods, are ignored.
2. ***minimumDetailLimit*** – This parameter constrains methods having less than the specified number of outgoing arcs in the canonical call graph not to be analysed

for being duplicate code. This parameter allows us to eliminate methods having too low complexity.

These parameters overlap to some extent, while the differences come from what structures are present in each of the graphs. In the first case only the direct method calls are considered. Usually they are a few – it is a rare case in practice to have a method calling more than 10-15 other methods. This means that the suitable values of these parameters are between 0 and 5. Bigger values of these parameters are useful if we are searching for the most complex duplicate methods.

The second parameter takes into account all possible constructs – a reference to a field, conditional operators and loop operators, method and constructor calls. A low value of this parameter would allow setters and getters to be considered duplicate, which is obsolete. A high value of this parameter may cause a significant decrease of the number of the detected duplicate methods. For the same version of Ant we changed this parameter from 3 to 15 and the number of detected duplicate methods decreased from 242 to 15 (using a value of 0 for ***minimumLimit***). A value of -1 for both parameters caused the appearance of 407 pairs of duplicate methods.

SPEED AND COMPLEXITY

The complexity of the represented algorithm is calculated based on the following dimensions:

- n – the number of methods in the analysed project, roughly corresponding to the number of nodes in the call graphs.
- k – the number of the arcs in the canonical call graph.

The complexity of steps 1 and 5 is determined as O(n) and O(k) respectively, because it is necessary to process all methods in the project and all their calls in order to create the call graphs. The same argument is valid for step 2 where all nodes and their outbound arcs are processed in order to create the identifiers of the nodes.

The complexity of step 3 is O(n log n) – this is the complexity of the quicksort sorting algorithm which is one of the fastest possible sorting algorithms.

The maximum complexity of step 4 is $O(n^2)$. It is reached when all identifiers propose duplicate methods. In practice this is not the case.

The maximum complexity of step 6 is $O(n^2)$ for the same reasons. The difference here is that for each node all his arcs in the call graph are analysed. This means that for each two nodes their sets of outgoing arcs are processed. In the worst case each arc will be analysed n-1 times – one for each comparison in which the node takes part. As a result the complexity of step 6 is O((n-1)k).

Step 7 has complexity of O(n), because all nodes are processed.

As a result the maximum complexity of the algorithm is $O(n^2)$ regarding n, and O((n-1)k) regarding k. The minimum complexity is O(n.log(n)) regarding n and O(k) regarding k. This would be achieved in the cases where the identifiers corresponding to potentially duplicate code are grouped in pairs. In this case each arc will be processed not more than once in steps 4 through 6.

Even though the calculated complexity of the algorithm seems high, in practice it is closer to the lower bound of the range than to the higher estimation, because usually the number of duplicate methods is relatively small compared to the total number of methods in the analysed project.

In order to check what is the real complexity of the algorithm in practice, we executed it on three different versions of Ant [12] – 1.3, 1.5 and 1.7. Each version has more code than the previous and we can approximate the complexity of the algorithm by comparing the increase of time to the increase of the number of nodes and arcs in the graph. For the purpose of the experiment we provided the same environment for all the three runs of the algorithm – operating system, processor, Java version and so on. The setup of the

experiment limits the influence of "noises" – the measurements are done automatically at the beginning and at the end of each execution of the algorithm, and no other applications are run during the tests.

Version	n	k	t1(t2)	dt1(dt2)	d(nlog(n)) (dk)	$d(n^2)$ (d(nk))
1.3	16578	43901	6(0,092)			
1.5	32991	94254	20(0,12)	3,33(1,3)	2,13(2,15)	3,96(4,27)
1.7	52394	144194	37(0,26)	1,85(2,16)	1,65(1,52)	2,52(2,42)

Table 1. Comparison of the results of the algorithm for several versions of Ant

In the table above we have used the following specifications:

n – the number of nodes in the graph (the number of methods in the code)

k – the number of the arcs (the number of calls)

t1 – the execution time for the whole algorithm in seconds

t2 – the execution time of the algorithm excluding the time for creating the call graphs in seconds.

dt1 – the ratio between the execution times for the whole algorithm between two consecutive versions of the product

dt2 – the ratio between the execution times for the algorithm, excluding the times for creating the call graphs, between two consecutive versions of the product.

d(nlog(n)) – the ratio between the minimum estimates according to n for two consecutive versions.

d(k) – the ratio between the minimum estimates regarding k for two consecutive versions of the product.

$d(n^2)$ – the ratio between the maximum estimates regarding n for two consecutive versions of Ant.

d(nk) – the ratio between the maximum estimates regarding k for two consecutive versions of the product.

According to the table above, the ratio between the execution times of the algorithm never reaches the ratio of the maximum estimates, but is closer to the geometric mean between the best and the worst ratios.

It is important to state that the execution time of the algorithm is very good and is suitable for practical usage – Ant 1.7 has 1113 classes and about 200000 lines of code.

VISUALISATION OF THE RESULTS

As a result of the execution of the algorithm a call graph is created, containing only the duplicate methods, grouped in clusters by similarity. As a result the most natural way to display the result is to show the resulting graph. The implementation of this approach allows that when a cluster is selected, all methods which are part of that cluster to be displayed as a popup menu. In an extended implementation the ability to open the corresponding code in an editor may be added, or even the ability to visually compare the duplicate method pairs.

On the picture is demonstrated a run of the algorithm on Ant 1.7.

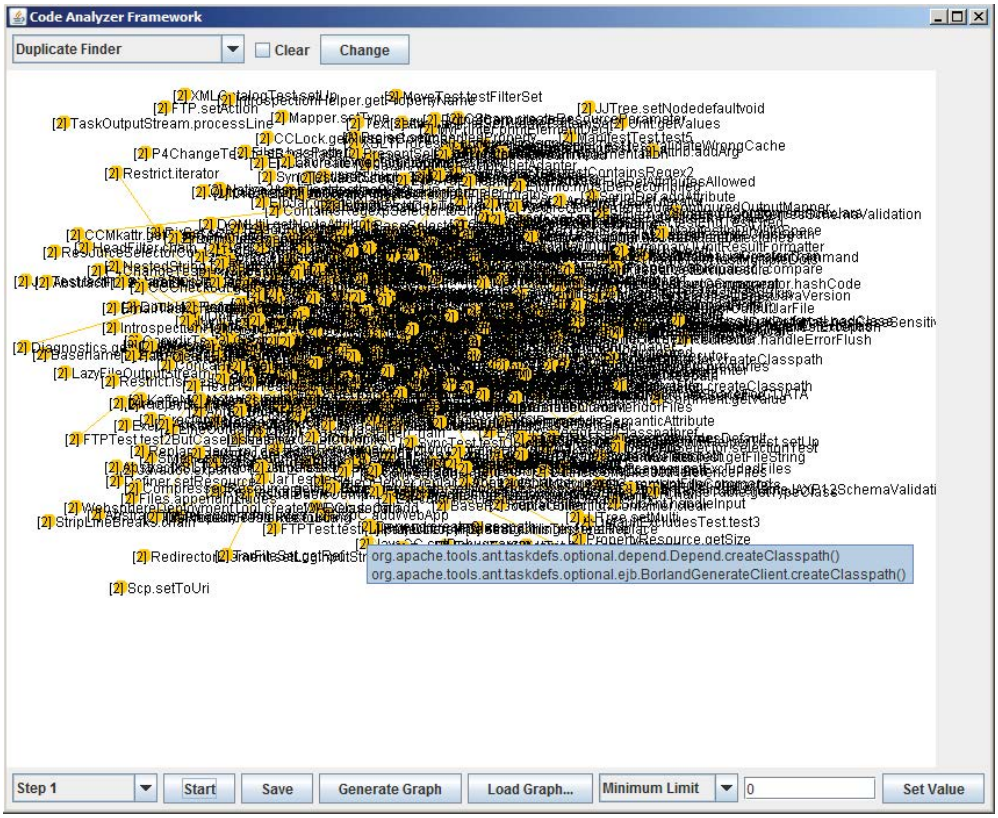

Fig.1.The result of the execution of the algorithm on the code base of Ant 1.7

As shown on the figure above, each cluster is represented by the last part of its name. This decreases the overlapping when having more complex graphs.

SIMILAR WORKS

J. Johnson [9] develops and algorithm for detecting duplicated code, based on text footprints. Although he uses heuristics for decreasing the number of footprints and their corresponding codes, the proposed algorithm has complexity of O(l.log(l)), where l is the number of lines in the code of the program. In practice, by using the proposed heuristics, the complexity decreases significantly (a single footprint may contain information for more than one line of code). We consider as drawback of the proposed approach that the program is analysed as simple text, ignoring the features of the programming language. Also a large part of the duplicate codes may not be detected because of variable or field renaming. This means that this algorithm detects only code that is duplicate according to **Definition 1**.

CCFinder [8] uses an algorithm based on dividing the code into lexemes. The tool allows ignoring the names of the identifiers, the comments in the code and the whitespace. The instrument processes the program code, allowing unification of slightly different constructs. This makes it much more flexible compared to the text based algorithms. In order to analyse large amounts of code the algorithm has to process all combinations of lexemes (in practice some of the combinations are skipped, but the order remains the same). According to our approach a footprint is created for each method, based on the method calls within its body and then only the potentially duplicate combinations are processed. This significantly increases the performance, because of the filtering on most of the possible pairs of duplicate code.

CloneDr [7] uses hash functions on abstract syntax trees which results in significant increase of the speed. At the same time the hash function chosen by the authors allows elimination of the leaf nodes of the tree (the identifiers) and the differences when having symmetric operations (i.e. *a+5* and *5+a* are considered equivalent expressions). Our

approach is very close to theirs, but instead of defining a hash function on the whole syntax tree, we use a small part of it – the method calls. This makes it possible to detect as duplicates much larger class of code pairs.

CloneDigger [6] uses a technique for comparison of syntax trees, allowing to ignore some changes in code, if the overall structure is preserved. The resulting trees are grouped by equivalence in a way similar to that of CloneDr.

The algorithms represented here are only a small part of the existing algorithms for duplicate code detection and each of them is representative for a specific class of approaches for duplicate code detection – starting from text based to algorithms based on syntax trees and neural networks.

CONCLUSION

The algorithm presented here is able to detect a large class of duplicate methods within the code base of the analysed software project. Its complexity allows low execution times even for large code bases, making it usable in everyday practice. It has advantages compared to all similar algorithms either in speed or the range of the detected duplicates.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the financial support by the Bulgarian National Science Fund with project DO 02-102/23.04.2009.

REFERENCES

[1] **Baker, B.** On Finding Duplication and Near-Duplication in Large Software System. Proceedings of the 2nd Working Conference on Reverse Engineering., 1995. pp. 86-95.

[2] **Roy, C., Cordy, J.** An Empirical Study of Function Clones in Open Source Software System. Proceedings of the 15th Working Conference on Reverse Engineering.,

[3] **Aversano, L., Cerulo, L. и Di Penta, M.** How Clones are Maintained: An Empirical Study. Proceedings of the 11th European Conference on Software Maintenance and Reengineering., 2007. pp. 81-90.

[4] **Juergens, E., et al.** Do Code Clones Matter? Proceedings of the 31st International Conference on Software Engineering., 2009.

[5] **Komondoor, R., Horwitz, S.** Using Slicing to Identify Duplication in Source Code. Proceedings of the 8th International Symposium on Static Analysis., 2001. pp. 40-56.

[6] **Bulychev, P., Minea, M.** Duplicate Code Detection Using Anti-Unification. Spring Young Researchers Colloquium on Software Engineering., 2008.

[7] **Baxter, I. et al.** Clone Detection Using Abstract Syntax Trees. Proceedings of the 14th International Conference on Software Maintenance., 1988. pp. 368-377.

[8] **Kamiya, T., Kusumoto, S. и Inoue, K.** A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering., 2002. pp. 654-670.

[9] **Johnson, J. H.** Identifying redundancy in source code using fingerprints. Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering., 1993.

10. **Gosling, J, и др.** The Java Language Specification. 2014.

11. **Kerningan, B. и Ritchie, D.** The C Programming Language. 1978.

12. Apache Ant User Manual. , http://ant.apache.org/manual/., 19.11.2014.

ABOUT THE AUTHORS

Todor Cholakov, PhD student, FMI, Sofia University "st. Kliment Ohridski", Phone +359 888 517 225, E-mail: todortk@abv.bg

Dimitar Birov, assoc. Professor, FMI, Sofia University "st. Kliment Ohridski", Phone +359 895805032, E-mail: birov@fmi.uni-sofia.bg