IPSJ SIG Technical Report

Vol.2016-SE-192 No.3
Vol.2016-EMB-41 No.3
2016/6/2

Estimating Code Size After a Complete Code-Clone Merge

Buford Edwards III[1,a)]   Yuhao Wu[1,b)]   Makoto Matsushita[1,c)]   Katsuro Inoue[1,d)]

Abstract: Code clone detection tools can identify portions of code that are the same. Using the data from these code clone detection tools we can estimate the number of lines of code that can be removed from a given source code if we could possibly remove all code clones and replace them with function calls to a shared function. The aims of this paper are to describe the methods and algorithms by which we can theoretically estimate the resulting code size. We also briefly discuss the application of the algorithm and take a look at a prototype implementing this algorithm.

1. Introduction

As software system evolves for a long time, it increasingly contains a lot of duplicate code (code clones) in its source files for feature extensions and/or maintenance purposes [3]. System owners can recognize the existence of code clones relatively easily with code clone detection tools [7], but it is not easy to determine whether or not those code clones are harmful for maintenance activities and if they need to be merged (refactored) [6].

One measure of whether or not to perform refactoring would be quantitative evaluation of code clones. We would want to know the total code clone size of the current system, and what the maximum reduction we will get in total is if we perform refactoring on all detected clones. The total code clone size can be easily calculated from code clone detection tools, but the latter is not known. These measures might be an important guide towards the decision of refactoring.

This paper aims to present an algorithm for determining the total amount in terms of lines of code that can be refactored from a source file by removing all instances of code clones from that source file without changing the functionality of the source code. We have named the algorithm Complete Code-Clone Merge (hereafter CCM), and we have implemented a prototype tool, which utilizes the output from a code clone detection tool, CCFinderX [11] as its required input, and which reports the sizes of the refactored system where all clone instances are merged and removed. This tool reports an estimated size of the ideally refactored system, but it does not show the code of the refactored system itself.

It has long been established that code clones can drive up maintenance costs for the developer [9]. Combined with the fact that companies spend a large portion of their resources devoted entirely to code maintenance already, code clones are an important issue for developers in charge of maintenance to take into consideration. Luckily there are many tools that have been developed that aid maintenance teams in locating code clones for the purposes of refactoring. We expect that this algorithm and the prototype tool built from it will be another valuable resource developers can use to aid in their maintenance of systems as it provides a way of estimating how much one can theoretically reduce through refactoring code clones from the source code.

Starting in Section 2 we will review code clones as well as what is meant by refactoring code clones, and discuss some related works in Section 3. The basic approach to this algorithm is described in Section 4 through illustration and the inclusion of formulas detailing how the size of the refactored source code is to be calculated in terms of total lines of code. This section begins with the most basic application of CCM, followed by more complex examples, and finally with a subsection devoted to a detailed look at the completed algorithm. In Section 5 and Section 6 we will take a look at the prototype tool which utilizes this algorithm and the results of its application. Finally we conclude the paper in Section 7.

2. Code Clones and Refactoring

Code clones are sections of code that are the same or very similar to each other. How similar the sections must be for them to be considered as clones depends on what kind of clone it is and how one measures their similarity. There are four types of code clones that are usually recognized [8]. Type-1 code clones are are identical, with possible variances due only to whitespace or layout. Type-2 clones may have differences in identifier names and in values. Type-3 clones, in addition to the differences in Type-2 clones, may have more changes such as additions, deletions, and altered statements caused by editing. Finally, Type-4 clones are semantic, meaning the sections of code have the same function, but may have a different structure and/or syntax. Code clones are often a target for code refactoring.

Code refactoring is the process of restructuring preexistent code without changing the external behavior or final execution result [2]. The purposes of refactoring are typically to reduce the

[1] Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka, 565–0871, Japan
[a)] BufordEdwards3@gmail.com
[b)] wuyuhao@ist.osaka-u.ac.jp
[c)] matusita@ist.osaka-u.ac.jp
[d)] inoue@ist.osaka-u.ac.jp

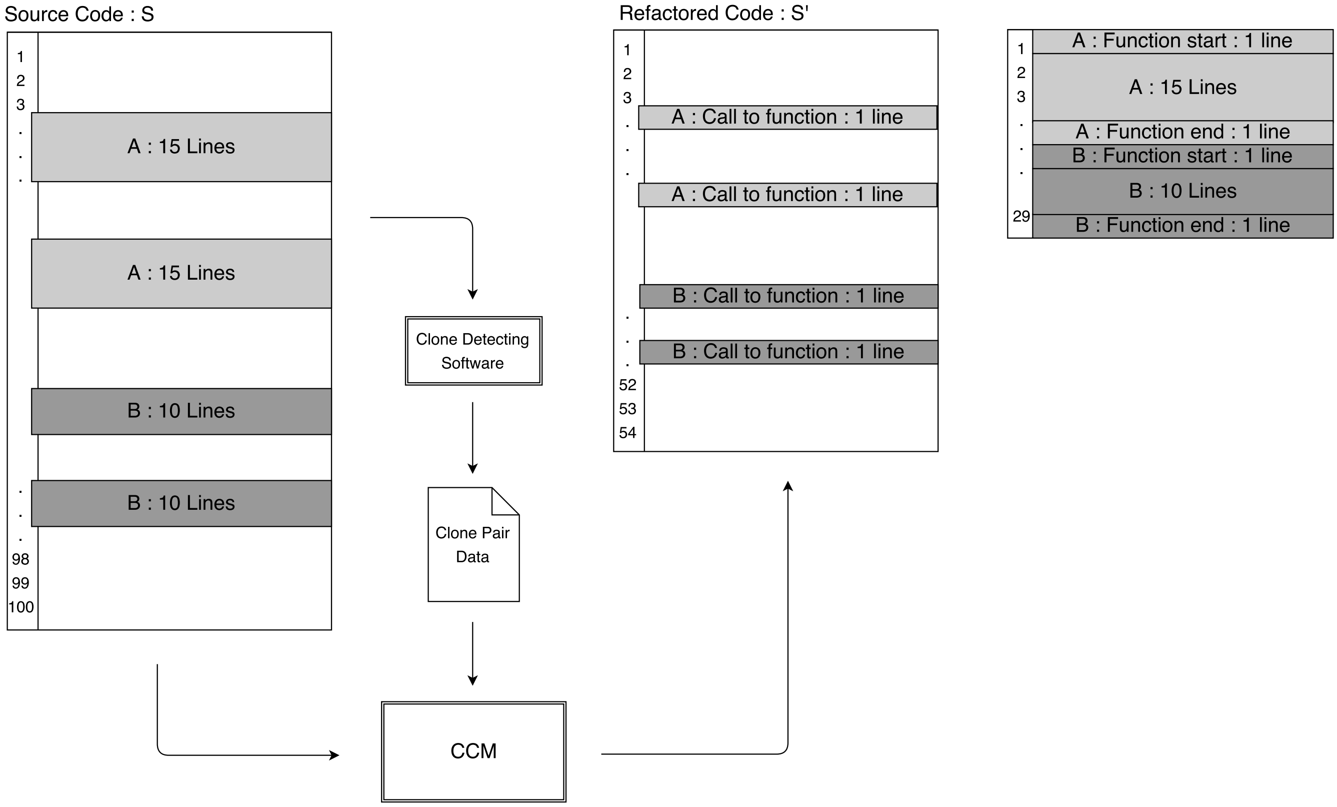
Source Code : S
1
2
3
.
.
.
A : 15 Lines
A : 15 Lines
B : 10 Lines
B : 10 Lines
98
99
100
Clone Detecting Software
Clone Pair Data
CCM
Refactored Code : S'
1
2
3
A : Call to function : 1 line
A : Call to function : 1 line
B : Call to function : 1 line
B : Call to function : 1 line
52
53
54
A : Function start : 1 line
A : 15 Lines
A : Function end : 1 line
B : Function start : 1 line
B : 10 Lines
B : Function end : 1 line
29

Fig. 1 Basic idea behind CCM.

overall complexity of the code which may reduce maintenance costs and improve overall readability. One technique to refactor code clones is to extract them from the code and create a shared function that contains the cloned portion, then create calls to that shared function [4]. This is the method by which we are structuring our algorithm.

3. Related Works

As previously mentioned there is a lot of research already existing regarding code clones, both how to detect them, and their impact on maintenance. We know code clones are have been shown to increase maintenance costs, and inconsistent changes to cloned sections of code can create faults in the program and lead to incorrect and unintentional program behavior [8]. Indeed, Juergens et al., in their report “Do Code clones Matter?” concludes that “nearly every second unintentionally inconsistent change to a clone leads to a fault” [5].

In addition, some other researches suggest that as a project increases in size it becomes much more likely for unintentional code clones to appear [1]. There are a number of reasons why this may be the case. Reasons may include, but are not limited to, things such as poor communication between programmers on projects that require multiple programmers, as well as development cycles with limited time constraints where copy-and-paste programming may yield quick short-term results but ultimately lead to more expensive maintenance costs. We recognize that if large size projects are especially susceptible to an increased number of code clones, then there is an inherent benefit to creating an algorithm to make this calculation for us, as doing so by hand no longer becomes a reasonable, time-effective procedure.

Due to the nature of code clones, the tools used to detect them, and refactoring, this paper later touches on the problem of overlapping code clones, which has been discussed in detail by other researchers [10]. Many code clone detection tools find portions of code that overlap one another, or detect clones that exist within another clone (embedded clones). This paper does not attempt to solve the issue of overlapping clones in clone detection software, but it does try to work around it.

4. Complete Code-Clone Merge

The idea of CCM is relatively simple in theory. We have a source file S of a certain line length $|S|$ and we wish to remove all code clones to create a refactored source file S' of a certain line length $|S'|$ by creating a shared function f for each unique code clone. Each unique code clone will be identified by an *ID*. The original code clones are then replaced with a call to their respective shared functions *fID* within the original source code so as to not alter the overall function of the source code. See Figure 1 for an illustration.

When creating a shared function let the function have the necessary lines for initialization and termination. We will represent the number of lines necessary for initialization and termination as some constant IT. Additionally, in place of the removed code clone let us add the necessary lines to call the shared function. We will assign constant FC to represent the number of lines necessary for the function call. For there to be a reduction in size, we assume each code clone length is at least of line length $IT + FC + 2$, thus $CloneLength \geq IT + FC + 2$. We know the amount

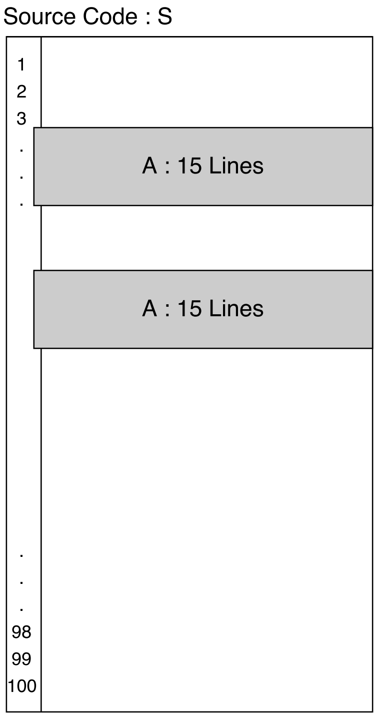

Fig. 2 Source code with 100 lines, only code clone A.

of times the code clone appears in the source code, henceforth referred to as *population* or *POP*, is at least 2 (there can be no *POP* less than this by definition). *FC* must also be at least one line in length, otherwise there is no replacement call and we would be altering the overall function of the source file. If the assumption concerning code clone length holds we can expect that if S contains any code clones, then after the complete code clone merge $|S'| < |S|$.

4.1 Basic Cases and Examples

Perhaps ones of the most basic cases we can consider would be only one code clone pair with no overlapping code clones, as there is only one clone, and a *POP* of 2. Take Figure 2 as the example. For this case, the equation to find $|S'|$ is as follows:

$$|S'| = |S| + (-CloneLength \times POP + FC \times POP + CloneLength + IT)$$

Using the formula we get the following result, assuming $IT = 2$ and $FC = 1$:

$$\begin{aligned}|S'| &= 100 + (-15 \times 2 + 1 \times 2 + 15 + 2)\\ &= 100 + (-11)\\ &= 89\end{aligned}$$

From this basic example we can see a total of 11 lines are removed from the overall length of the source code. We can discover the $|S'|$ value for Figure 1 as well. However, let us consider a simplified equation that will give us the same results if our assumptions hold true about the length of the code clones.

Whether by counting manually (as is possible to do here with the small size examples in Figure 1 and Figure 2) or through the use of clone detection software, we can count the total lines of code clones, henceforth referred to as $TLOC$. There are n number of code clones for any given source. For Figure 2, we need to only consider one code clone because n was equal to 1. However, in Figure 1, for example, $n = 2$. When there exists more than one kind of code clone there may also be a different number of FC required for each clone as this is determined based on the individual clone's *POP*. So for each individual clone we need to determine the summation of $CloneLength + IT + FC(POP)$. We will refer to this summation as the add back value, AB.

$$AB = \sum_{ID=1}^{n} CloneLength_{ID} + IT + FC \times POP_{ID}$$

Please note that this method for determining add back is not the completed solution to handle all cases, as we have not yet considered overlapping code clones. We will consider that more advanced case in Section 4.2. However, once we have determined AB, the basic formula for determining the refactored source code size is much more straightforward and will remain the same. It is as follows:

$$|S'| = |S| - TLOC + AB$$

4.2 Overlapping and Embedded Clones

Overlapping code clones are sections of code that are identified as code clones which share a portion of their code with another unique code clone. See Figure 3 for a visualization and example of an overlapping code clone. When overlapping code clones exist, it is no longer possible to simply insert a function call for each code clone. If this were done, the portion of the code that is overlapped would be executed more times than intended.

Using Figure 3 as an example, if code clones A and B are refactored out, without accounting for the overlap portion C, and if the the respective function calls for A and B are placed at the lines where code clones A and B begin, the overlap portion C would have its code executed one more time than in the original source. Because our goal is for S' to have the same execution result as S this would be unacceptable.

4.2.1 Chunking Method

The solution is then to recognize the overlapping portion of code as a code clone and separate it from any surrounding code clones it may be embedded in. In the case of the example in Figure 3, code clone C is embedded in both clones A and B. When C is explicitly separated from clones A and B, the original clones A and B are modified into new code clone "chunks." In the case of Figure 3, the chunks A', B', and C are recognized. We have nicknamed this solution the "Chunking Method." Now instead of making a function for which to call A and B, a function is created that only calls the lines containing the "chunk portion." In this case, for A and B, chunks A' and B' are the sections of code that do not include code clone C. Additionally, a function is created to call chunk C (which is itself a separate code clone). Chunks A', B', and C each have a chunk size of 1.

Take note that in some cases, it may be possible to have more than one clone embedded within another clone. In such cases, it

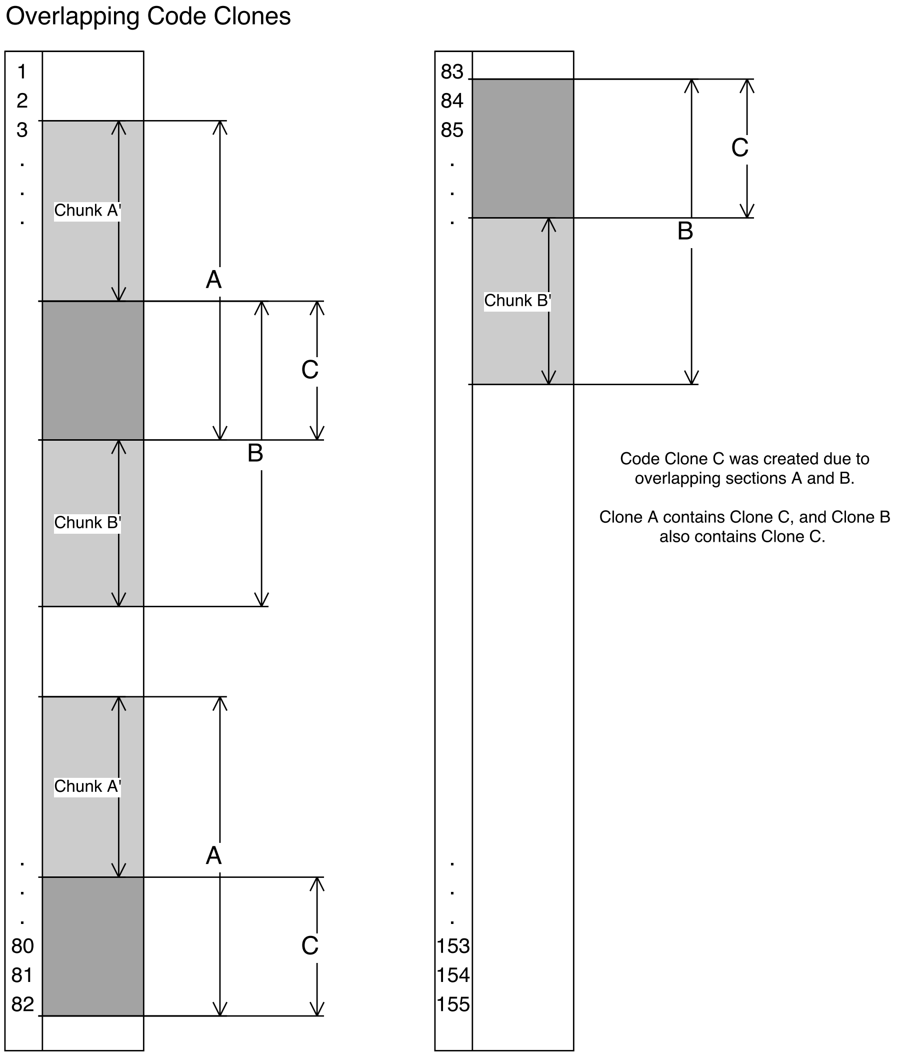
Overlapping Code Clones
1
2
3
Chunk A'
A
C
B
Chunk B'
Chunk A'
A
C
80
81
82
83
84
85
C
B
Chunk B'
153
154
155
Code Clone C was created due to overlapping sections A and B.
Clone A contains Clone C, and Clone B also contains Clone C.

Fig. 3 Overlapping Code Clone Example

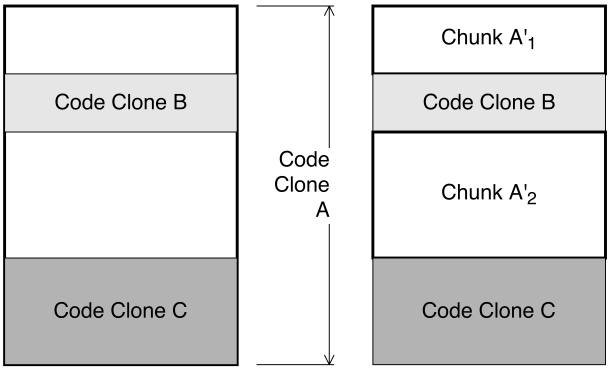

Fig. 4 Code Clone Requiring Multiple Chunks

may be necessary to split a code clone into even more chunks if an embedded code clone is located at a point that divides the code clone it is embedded within. See Figure 4 for a visualization and example. This type of case is not uncommon and will be seen again later on.

4.3 Attributes

In order to represent code clones overlapping for the purposes of this algorithm we will recognize the attributes of each line in the target source code S. Each line in S either contains a line that is detected as part of a code clone, or it does not. In some cases, as with embedded and overlapping code clones, a line may be part of more than one code clone. The *ID*, one or more, associated with a code clone will be part of that line's attributes. If the line has no attributes, we know it is not part of a detected code clone. We will use Figure 5 as an example how we construct the attribute list. This example corresponds to the overlap portion found in Figure 3. For the sake of simplicity, we will say code clone A and B are both 15 lines in length, while code clone C is 5 lines in length.

Looking at this attribute list it is easy to see where the overlap exists, and what lines exist to form code clone C. Likewise, it should be clear which parts form chunks A' and B'.

4.4 CCM Algorithm

CCM requires a target source code S and cloned snippet sequence L composed of three tuples, {unique clone ID (*ID*), starting line (*SL*), ending line (*EL*)} as input, and the structure of the refactored source code set *S'* where clones in L are merged into new shared functions f and |*S'*| (size of *S'*). CCM consists of following five steps:

Line #	Attribute
1.	
2.	
3.	A
4.	A
5.	A
6.	A
7.	A
8.	A
9.	A
10.	A
11.	A
12.	A
13.	A B
14.	A B
15.	A B
16.	A B
17.	A B
18.	B
19.	B
20.	B
21.	B
22.	B
23.	B
24.	B
25.	B
26.	B
27.	B
28.	
29.	
30.	

Fig. 5 Attribute list example

Step 0 (preparation): Remove all lines in S that are either empty (whitespace) or contain only comments. For each line in S, prepare an attribute set of clone ID, which is initially empty.

Step 1: For each tuple t in L, add an ID to the attribute set of all lines between SL and EL (including those lines).

Step 2: For each ID, create one complete cloned snippet as a new shared function fID. The attribute set of each line is the union of the corresponding line's attributes for each clone instance with the same ID.

Step 3: Scan S from the beginning.

- If the change of attribute happens during scanning, we insert a calling statement for a function, except for the lines going back to empty.
- Delete lines with non-empty attribute.
- Count the number of lines deleted, and also added for the function calls.

The resulting source code is a part of S', and the count is also a part of $|S'|$.

Step 4: In the same manner, scan each fID,

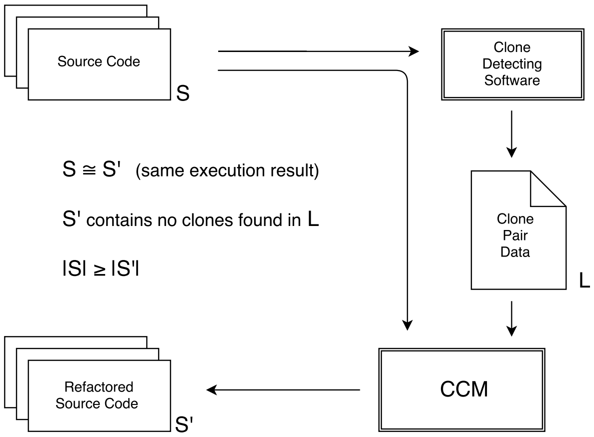

Fig. 6 From S to S'

- Delete the lines having a lower ID than fID.
- If a change of attribute happens, insert a termination statement just before the change, and also insert an initialization statement just after the change (if the line is not deleted).
- Delete the lines with attributes with lower ID than fID.

The resulting collection of fID will be another part of S', and the count of the lines remaining and inserted are another part of $|S'|$.

Step 5: Merge and add the results of Step 3 and 4, and formulate S' and $|S'|$ respectively.

The time complexity for above Step 0 through 5 are O(1), O($|L|$ × MaxOverlapLevel(L)), O($|S|$ × MaxOverlapLevel(L) × MaxOverlapLevel(L)), O($|S|$ × MaxOverlapLevel(L)), O($|L|$ × MaxOverlapLevel(L)) and O(1) where MaxOverlapLevel(L) is the max number of levels of overlapping snippets L. Note that Step 2 can be performed by scanning S and creating each fID and its attributes at the same time. This process can be merged into Step 1. If we assume MaxOverlapLevel(L) is no more than C (a constant value), the total time complexity becomes O($|S|$) + O($|L|$). The space complexity of CCM is O($|S|$ × MaxOverlapLevel(L)) + O($|L|$ × MaxOverlapLevel(L)); if same assumption shown above holds here, O($|S|$) + O($|L|$).

5. CCM Prototype

A prototype tool based on the CCM algorithm has been developed. Using the target source code S and clone data gathered through the use of clone detection software CCFinderX as input the tool will generate data concerning total refactor size as well as an S' file with comments substituting for function calls (Figure 6).

6. Prototype Application

Three examples of source codes and results the CCM Tool has been run are detailed below. They are written in Java or C. However, CCM is not limited to Java source codes. The prototype tool itself is limited to only those languages which CCFinderX can detect clones for.

6.1 Multilap.java

The first is a Java source code of relatively few lines written

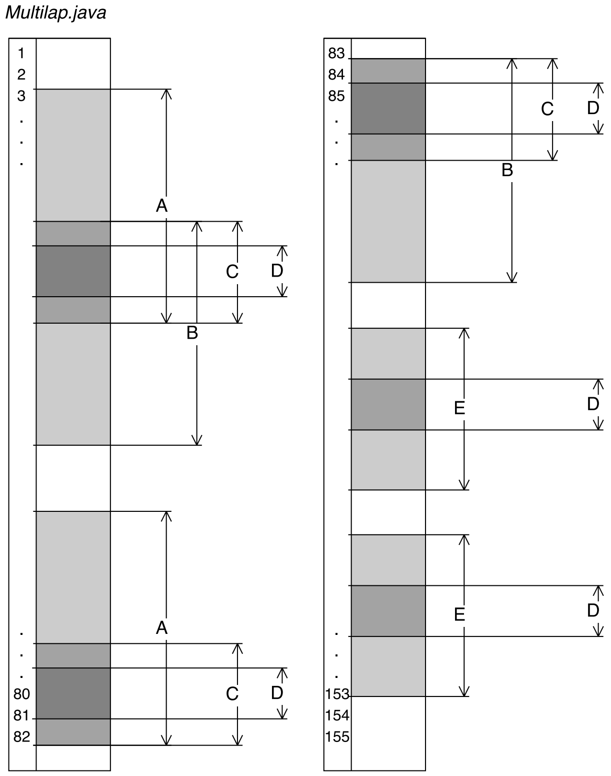

Fig. 7 Multilap.java Visualization

Table 1 Multilap.java Clone Data

CID	POP	LENGTH	CSIZE	CHUNKS
A	2	30	8	1
B	2	31	9	1
C	3	22	15	2
D	5	7	7	1
E	2	19	12	2

Table 2 Multilap Results

Initial Size:	154
Total Clone Length:	138
Reduced Size:	100
LOCs Reduced:	54
Percent Reduction:	35.064934

purposely to contain code clones and to demonstrate how CCM handles both overlapping code clones and multi-layered overlaps. Multi-layered overlaps are when code clones are embedded within two or more other detected clones.

In the example presented by Figure 7, we have a total of five different code clones and each code clone is assigned a unique clone ID. Although CCFinderX assigns different numbers for each clone ID, we will replace these with letters A through E just for the sake of explanation, the tool itself keeps the ID CCFinderX assigns.

In this example, clone A and B overlap, and the overlap portion we will refer to as clone C. Within code clone C, there is clone D, which splits code clone C into two chunks. Clone D can also be found splitting code clone E into two chunks as well.

It is from the data in this graph that is used for the final calculations. We first determine the total number of lines that must be included from the code clone sections themselves which comes from a summation of the CSIZE (chunk size) column of the data table (Table 1).

$$
\begin{aligned}
TotalCSIZE &= 8 + 9 + 15 + 7 + 12 \\
&= 51
\end{aligned}
$$

Next we count the number of chunks that are required per code clone. Clones A, B, and D each have a chunk value of 1. However, because they must be split due to another embedded code clone, clones C and E both have a chunk value of 2. We then multiply this number by 2, as this tool assumes the initialization and termination statements for each of the shared functions are both one line each.

$$
\begin{aligned}
TotalIT &= TotalChunks \times IT \\
&= (1 + 1 + 2 + 1 + 2) \times 2 \\
&= 14
\end{aligned}
$$

Finally we determine the number of function calls that are required by the source code. Beginning from the top of the source code, whenever a new code clone appears we insert a function call statement which we will assume to be 1 line in length. For example, the first clone that appears is A, so we insert a calling statement in place of that code clone. As we continue we next come across C, we insert; we find D, insert; come across C, insert, then B and so on until the end of the source code. Each time we increment the function call counter FC.

$$FC = 19$$

We then add all of these values together to determine the AB value and complete the final calculation. AB is now taking into account all cases, including overlapping and embedded clones.

$$
\begin{aligned}
AB &= TotalCSIZE + TotalIT + FC \\
&= 51 + 14 + 19 \\
&= 84 \\
|S'| &= |S| - TLOC + AB \\
&= 154 - 138 + 84 \\
&= 100
\end{aligned}
$$

Once we have done this calculation we now know the Initial Size ($|S|$) of the source file, the Total Clone Length, and Reduced Size ($|S'|$) of the source code. Using this information we can determine the Lines of Code (LOC) Reduced by calculating $|S| - |S'|$. We then use LOC Reduced to determine the Percent Reduction.

$$
\begin{aligned}
PercentReduction &= (LOCReduced/|S|) * 100 \\
&= (54/154) * 100 \\
&= (.35064934) * 100 \\
&= 35.064934
\end{aligned}
$$

Table 2 summarizes the results.

6.2 Java JDK

The prototype tool was also run on the entirety of the Java JDK 1.8.0_77-b03 and results were gathered [12][13]. See Table 3 for the results. Note that the total code length in terms of line numbers is not including lines of whitespace.

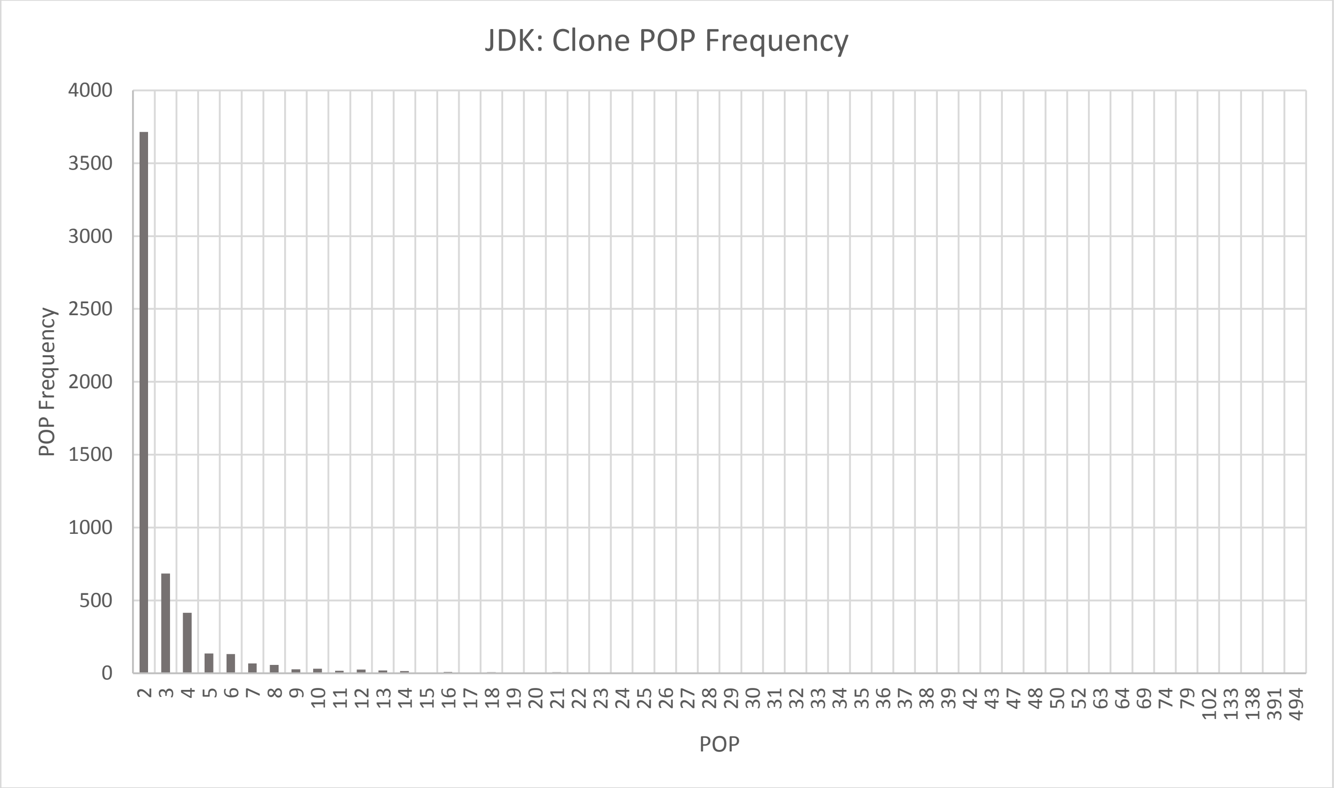

Fig. 8 Java JDK POP Frequency Graph

Table 3 Java JDK Results

Initial Size:	813546
Total Clone Length:	207072
Reduced Size:	708139
LOCs Reduced:	105407
Percent Reduction:	12.95649

Table 4 Quake engine Results

Initial Size:	216722
Total Clone Length:	49098
Reduced Size:	194324
LOCs Reduced:	22398
Percent Reduction:	10.3349

As can be seen in Table 3 the total percentage of code that can be theoretically refactored using the method we have described in this paper is approximately 13%. This result seems reasonable if we consider the graph displaying the *POP* frequencies (Figure 8). From the graph we can easily recognize that, at least in the case of the Java JDK, code clones with a *POP* of 2 are the most common without several thousand more code clones appearing to have a *POP* of 2 compared even to those with a *POP* of 3. Considering that code clone volume for this example is approximately 25% $(TotalCloneLength/InitialSize \times 100)$, if we assumed every code clone had a *POP* of 2 then we would expect to reduce the total volume by about half using the algorithm we have described in this paper. This is because we are essentially reducing the *POP* of each code clone to 1 by introducing a shared function and substituting function calls where these code clones may appear.

6.3 Quake engine

Another example of a large scale source code yielding similar results to that of the Java JDK is the Quake engine. The Quake engine is a game engine written in the C programming language by id Software to run their video game Quake. It was chosen for testing because the source code is available to the public and the code itself consists of several hundred-thousands of lines of code [14]. The results of the test can be observed in Table 4.

From these results we can observe that the total code clone volume is approximately 22.7%. Because the results tell us the percent that can be refactored by removing code clones is approximately 10.3%, we can expect for clones with a *POP* of 2 to again be the overwhelming majority of clones that are detected (Figure 9). And again that is precisely what we observe when we graph the frequency of *POP* values from this source.

7. Conclusions

Through the application of this algorithm we are able to determine what percentage could theoretically be refactored if we could merge all instances of code clones, effectively removing all clones from a given source code. The results discussed in the application section seem reasonable as we took a look at the frequency of different *POP* values. It may be of further interest to research *POP* frequency from many different large sources. If a trend appears where the most common *POP* of a code clone is consistently 2, then it may be possible to make the assumption that the total code clone volume that can be refactored by merging code clones would be approximately half that of the total code clone volume as was the case with the Java JDK and to a somewhat lesser extent the Quake engine.

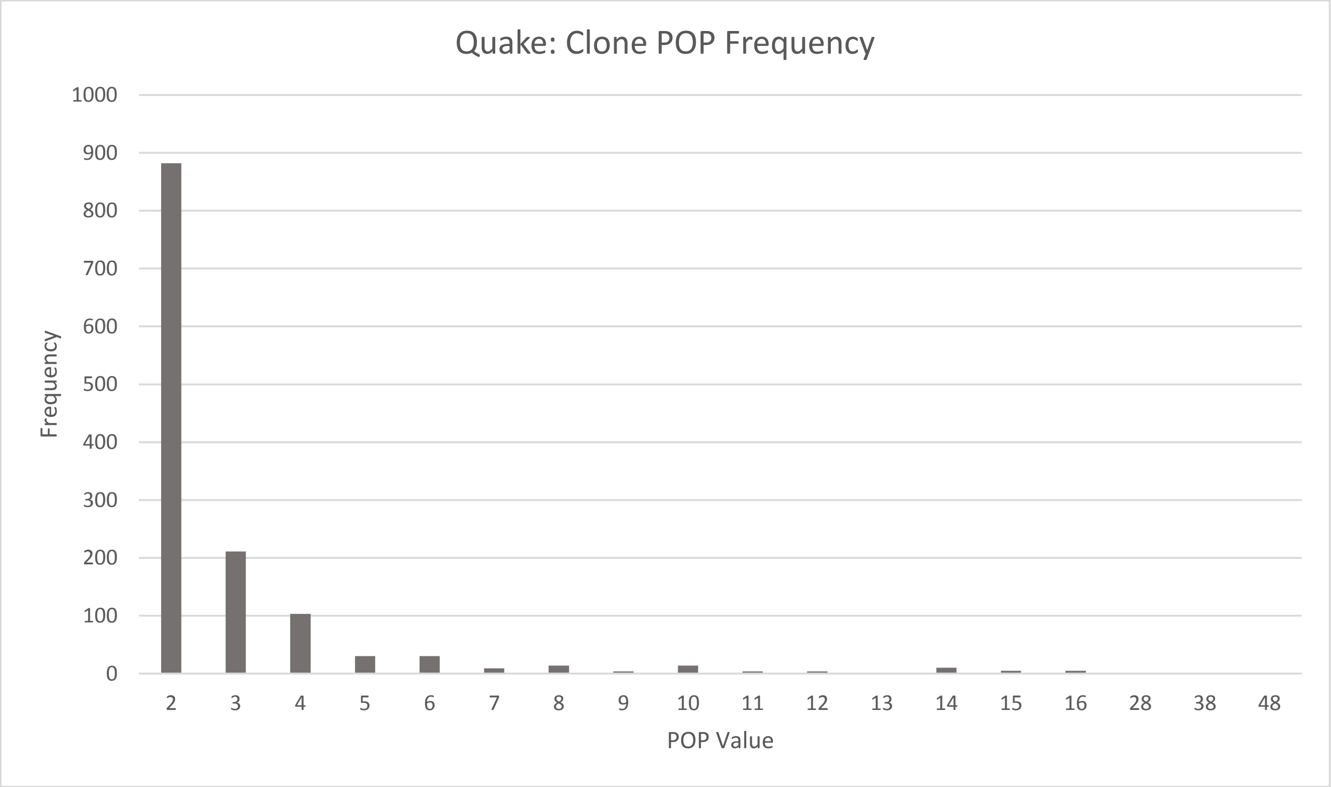

Fig. 9 Quake engine POP Frequency Graph

References

[1] Michel Dagenais, Ettore Merlo, Bruno Laguë, and Daniel Proulx. Clones occurrence in large object oriented software packages. In Proceedings of the 8th IBM Centre for Advanced Studies Conference (CASCON ’98), pp. 192-200 (1998).

[2] Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley (1999).

[3] Nils Göde, Clone Evolution, Dissertation of University of Bremen, May, 2011.

[4] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, Refactoring Support Based on Code Clone Analysis, In Proceedings of 5th International Conference on Product Focused Software Process Improvement, pp.220-233 (2004).

[5] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, Stefan Wagner, Do code clones matter?, In Proceedings of the 31st International Conference on Software Engineering (ICSE ’09), pp.485-495 (2009).

[6] Cory J. Kapser, Michael W. Godfrey, “Cloning considered harmful” considered harmful: patterns of cloning in software, Empirical Software Engineering, Volume 13 Issue 6, pp.645-692 (2008).

[7] Chanchal K. Roy, James R. Cordy, Rainer Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming, Volume 74, Issue 7, pp.470-495 (2009).

[8] Chanchal K. Roy, James R. Cordy, Rainer Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Sci. Comput. Program., Vol.74, No.7, pp.470-497 (2007).

[9] Stefan Wagner, Asim Abdulkhaleq Kamer Kaya, Alexander Paar, On the Relationship of Inconsistent Software Clones and Faults: An Empirical Study, In Proceedings of 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER2016), pp.79-89 (2016).

[10] Pam Green, Peter C.R. Lane, Austen Rainer, Scen-Bodo Scholz, Unscrambling Code Clones for One-to-One Matching of Duplicated Code, In Technical Report No. 502, School of Computer Science, University of Hertfordshire, April, 2010.

[11] CCFinderX Official site, http://www.ccfinder.net/ .

[12] Java SE | Oracle Technology Network | Oracle, http://www.oracle.com/technetwork/java/javase .

[13] Java™ SE Development Kit 8, Update 77 Release Notes, http://www.oracle.com/technetwork/java/javase/8u77-relnotes-2944725.html .

[14] GitHub - id-Software/Quake: Quake GPL Source Release, https://github.com/id-Software/Quake .