Unscrambling Code Clones for One-to-One Matching of Duplicated Code

Pam Green
Peter C.R. Lane
Austen Rainer
Sven-Bodo Scholz

April 2010

Technical Report No. 502

School of Computer Science
University of Hertfordshire

Contents

1 Introduction **3**

2 Code clones **3**

3 The ‘over-reporting’ problem **4**

4 Method **6**

4.1 Unsubsume . . . 8

4.2 Uncopy . . . 9

4.2.1 Simple uncopying . . . 9

4.2.2 Overlap problem . . . 10

4.2.3 Amended uncopy . . . 11

4.3 Unoverlap . . . 12

4.4 Ordering the steps . . . 14

4.5 What difference does unscrambling make? . . . 14

4.6 Approximations . . . 15

5 Summary **15**

A Appendix **18**

A.1 Duplo . . . 18

A.2 Code Clone Finder (CCFinderX) . . . 19

A.3 Simian . . . 19

A.4 Example output . . . 19

List of Figures

1 Example files containing each type of clone . . . 6

2 Multiple clone example . . . 9

3 Overlapping multiple clones . . . 10

4 Overlapped clone example . . . 12

5 Unscrambling . . . 14

List of Tables

1 Duplicated clone example . . . 5

2 Example file clone data . . . 7

3 Unsubsuming . . . 8

4 Uncopying . . . 10

5 Uncopying multiple overlapping clones . . . 11

6 Unoverlapping . . . 13

7 Duplo matrix . . . 18

8 Duplo and Simian, and CCFinderX clones . . . 20

9 Sample clone detector output . . . 21

Abstract

Code clone detection tools find sections of code that are similar. Different tools use different representations of the code and different matching algorithms. This diversity makes clone detection tools attractive for other code matching tasks, particularly where code has been edited or rearranged. However, the tools report on *every* match found. In some applications we are interested in one-to-one matching, meaning that each section of copied code in one file is matched to just one section of code in the other file. In this report we explore the ways that clones reported by the detection tools can inflate the amount of matching code. We also explain, with the aid of a worked example, our method for unscrambling the output from clone detection tools to approximate one-to-one matching of the code in one file to that in another file.

1 Introduction

This report describes a method for using the output of clone detection tools to find sections of code in one file which have a one-to-one correspondence with another file. Clone detection tools normally report on every match between files which satisfies the given parameters. This can result in parts of one file being matched to more than one part of another file.

Different clone detection tools work with different representations of the code and use different matching algorithms. The flexibility and diversity offered by clone detection tools makes them attractive for other file comparison tasks. However, the tools may 'over-report' when answering questions such as "how much of the code in one file appears in another file?". In our work on origin analysis [7], we investigate the combination of similarity measures from a copy detection tool, Ferret [15, 17, 18, 22], with those from various clone detection tools: Duplo [1], Code Clone Finder X (CCFinderX) [10, 11] and Simian [8]. In this application we want to find a one-to-one correspondence between the code in files, a technique potentially useful in other areas.

In this report, code clones are introduced in Section 2 and the 'over-reporting' problem is explained in Section 3. Section 4 details the three-step unscrambling process, followed by a summary in Section 5. Examples of the output from three clone detection tools are provided in the Appendix for the interested reader.

2 Code clones

Code clones are similar sections of code. The similarity between the sections varies, depending both on the type of clone and on the similarity measurement. There are four types of code clone generally recognised (see, for example, [23]). The first three of the four types form a hierarchy of similarity: the simplest clones, known as type-1, are textually identical, with the possible exception of whitespace and layout; type-2 allows for differences in identifier names, in typing and in literal values; the additional changes introduced by editing such as additions, deletions and altered statements are acceptable for type-3 clones, which are sometimes known as "gapped clones" because of the breaks between fragments of similar code [3, 25]. Type-4 are semantic clones, which have the same function, but use different structure, syntax, or both [6].

Clone detection tools are usually used to find clones within or among files in a project, or between projects. Detecting within-file or within-project clones is valuable as an aid to consistency in maintenance [9, 20]. Knowledge of these clones means either that error correction or updating can be applied consistently, or that the cloned code can be abstracted, thus reducing maintenance effort. Code which matches across projects can suggest functionality for abstraction to a library file, or be useful in detecting plagiarism, for example in student assignments [21].

There are many methods used for clone detection, broadly these are based on text [1, 4], tokens [11, 16], abstract syntax trees [2, 24], program dependency graphs [12, 14] or metrics [13]. Within each method, various similarity functions are used. For example, text-based methods include hashed string matching of lines of code [5] or latent semantic indexing [19]. Matching may mean the exact, flexible or parameterised matching of the elements. Parameterisation covers a spectrum of options; for example, in string matching, from ignoring the case of otherwise matched strings, to accepting any identifier as a match for any other identifier.

3 The 'over-reporting' problem

Whichever method a clone detector uses, all instances of similarity between files which satisfy the constraints of the given parameters will be reported. In the context of origin analysis, the information required is the amount of code from one file which is present in another file. For these purposes, clone detection tools can 'over-report' by counting sections of copied code more than once. To use the tools for identifying one-to-one matches, the output from comparisons between files has to be interpreted so that each portion of code is only counted once. There are three ways in which the 'over-reporting' can occur:

1. when there are multiple copies of the same code in one or both of the files;
2. when there are subclones;
3. and when clones overlap each other.

Multiple copies, subclones and overlapping clones are illustrated in simplified form in Table 1 on page 5. The examples assume that there are two files being compared by the clone detector, that the matching is on exact copies of lines of code, and, for simplicity, that clones can be of any length. Column 1 names the problem, and the number of clones identified in the example. Columns 2-4 identify the file, the consecutively numbered location, and the reference, a combination of the two. For example, in the multiple copy section, the same 'code' is repeated three times in file 2, marked as locations 1, 2 and 3, and the copies are referenced as 2.1, 2.2 and 2.3. The code is shown in column 5 and the clones identified in column 6. Duplicated sections of the clones are shown in bold text. The clone pairs are listed by reference number in column 5, under the text.

Multiple copies of an identical block of code in the files will inflate the amount of copied code by [number of lines $\times ((m \times n) - (min(m, n)))$], where m, n are the number of copies in each file. For example, if one file has two copies of a block of code and the other file has the same block of code repeated three times, then a clone detector will report six clones, each of the three blocks in the first file paired with each of the two blocks in the second file. The information required for one-to-one matching is that there are two blocks in file 1 which are matched by two blocks in file 2. (For this example, the number of clones is 'over-stated' by $6 - 2 = 4$ pairs.)

Where part of a large clone is found elsewhere in the file, both the clone and subclone will be reported, inflating the number of copied lines by the size of any subclones. In the example, the line "chunk of code" in block 2, file 2, is a subclone and need not be counted as matched code a second time.

When clones overlap, the overlapping portion of code is reported twice. In the example, the line "code in file" is 'over-reported' and need only be counted once.

Problem	File	Loc.	Ref.	Text	Cloned text
Multiple copies	1 1	1 2	1.1 1.2	Some repeated code over several lines Some repeated code over several lines	} } } **Some repeated code** } **over several lines**
	2 2 2	1 2 3	2.1 2.2 2.3	Some repeated code over several lines Some repeated code over several lines Some repeated code over several lines	} } }
6 clones				[1.1, 2.1] [1.1, 2.2] [1.1, 2.3] [1.2, 2.1] [1.2, 2.2] [1.2, 2.3]	
Subclones	1	1	1.1	A large chunk of code in file one	
	2	1	2.1	A large chunk of code in file two	A large **chunk of code** in file
	2	2	2.2	Another chunk of code elsewhere in file two	**chunk of code**
2 clones				[1.1 (lines 1-3), 2.1 (lines 1-3)] [1.1 (line 2), 2.2 (line 2)]	
Overlapping clones	1	1	1.1	A chunk of code in file which has two partial matches	
	2	1	2.1	A chunk of code in file	A chunk of **code in file**
	2	2	2.2	code in file which has two	**code in file** which has two
2 clones				[1.1 (lines 1 and 2), 2.1] [1.1 (lines 2 and 3), 2.2 (lines 1 and 2)]	

Table 1: Examples of three types of duplication in the clones reported by clone detection tools: multiple copies, subclones and overlapping clones. Column 4 shows a reference number, composed of the file number and the location. The reference number is used to identify the clones in column 5. The clones are shown in column 6, with the duplication in the clones shown in bold text.

4 Method

Unscrambling the clone detection tool output to assess the amount of code common to two files requires three main processes. Each of these deals with one of the ‘problems’ outlined in Section 3. They are, with the term used to describe them shown in bold:

Unsubsume to remove subsumed copies, or subclones,

Uncopy to remove duplicate copies, or multiple clones,

Unoverlap and to remove the overlapping section from one of a pair of overlapping clones.

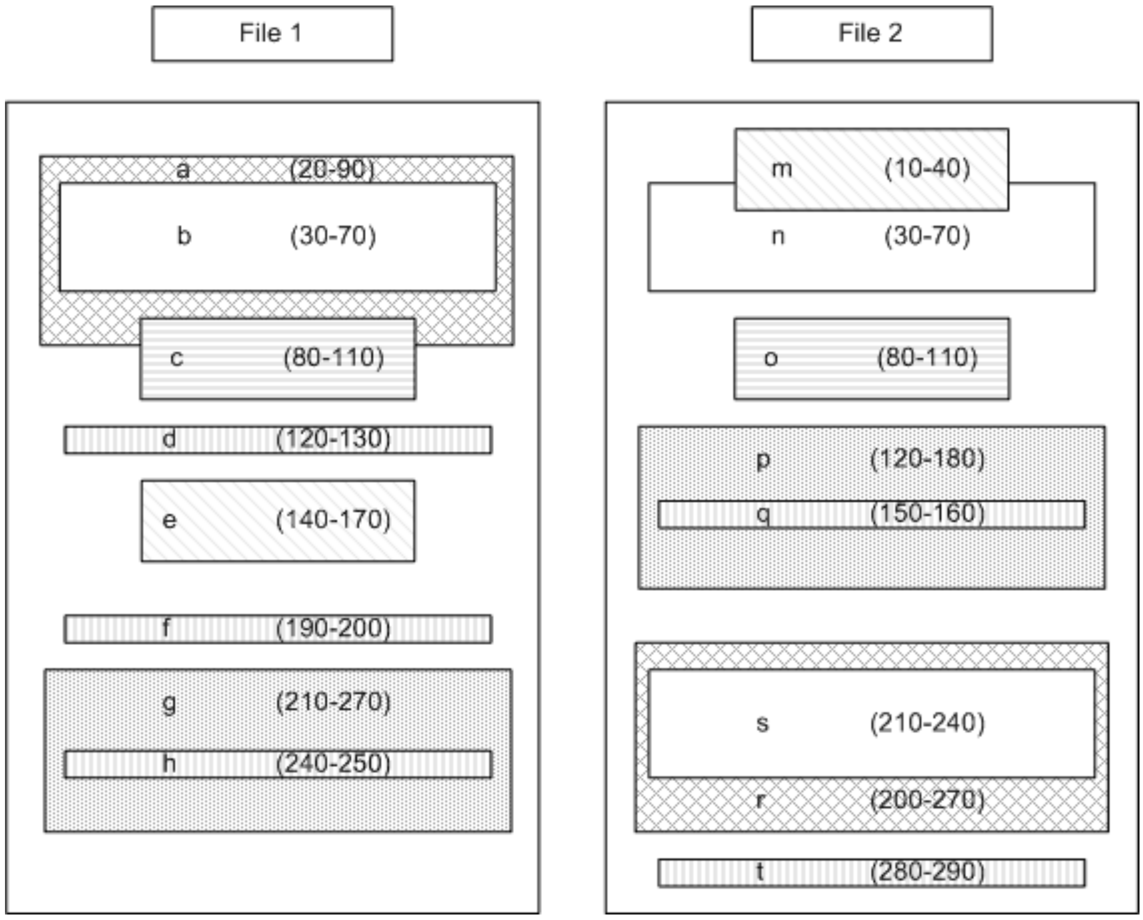


Figure 1: Example showing duplicated blocks in two files. Each block is labelled with a letter, and with its start and end line (or token) numbers. Identical blocks are shaded in the same way, for example, blocks a and r. The blocks d, f, and h are multiple copies in file 1, the same code being found twice in file 2, in blocks q and t. The blocks b, h, q and s are subsumed. The blocks a and c, and m and n overlap.

The words clone and block mean the same in this report. Figure 1 represents two files with a number of blocks of code in common, and is used as an aid to explaining the unscrambling processes. Each block in the diagram is labelled with a letter and with start and end line or token numbers. Different blocks are shaded differently, identical blocks have the same shape and shading, for example, blocks e and m. The diagram illustrates subsumed blocks, for example, block b in block a and block q in block p; multiple copies, blocks d, f ,h in file 1, and blocks q and t in file 2; and overlapping blocks, for example, blocks a and c, and blocks m and n. Clone detection tools differ in their reporting of clones. Tools such as CCFinderX, which can be set so that they do not report intra-file matches, will not report clone s because it is subsumed by block r and matches only clone b in file 1, which is similarly subsumed by clone a. Other detectors, such as Simian, which report on both intra- and inter-file matches, will report clone s, because it matches clone n in the same file.

File 1		**File 2**	
Block	**Lines**	**Block**	**Lines**
a	(20 - 90)	r	(200 - 270)
b	(30 - 70)	n s	(30 - 70) (210 - 250)
c	(80 - 110)	o	(80 - 110)

File 1		**File 2**	
Block	**Lines**	**Block**	**Lines**
d f h	(120 - 130) (190 - 200) (240 - 250)	q t	(150 - 160) (280 - 290)
e	(140 - 170)	m	(10 - 40)
g	(210 - 270)	p	(120 - 180)

Table 2: The matched blocks in the example files shown in Figure 1, the block(s) in file 1 are shown on the left, and matching blocks from file 2 on the right of each section of the table.

Most clone detection tools identify clones by referencing the start position of each clone. Of these, many also explicitly provide the end location of the clones, the remainder give this information implicitly, either by giving the size or the content of the clones. In the remainder of this document, clones are represented as a pair of (start end) line or token numbers, For example, clone a is shown as (20 90). Matched clone pairs are shown as ((file-1-start file-1-end)(file-2-start file-2-end)) numbers. For example, the matched clones a and r are shown as ((20 90)(200 270)). The three processes, unsubsuming, uncopying and unoverlapping, are described in Sections 4.1 - 4.3, respectively.

4.1 Unsubsume

The unsubsuming step of the unscrambling process removes subclones, and matched pairs in which one or both of the blocks are subclones. In the example given in Figure 1, there are four blocks of copied code which are subsumed by other blocks, these are blocks b (in a) and h (in g) in file 1, and blocks q (in p) and s (in r) in file 2. This example is used to illustrate the unsubsuming process, with the steps shown in Table 3.

First, the start and end line (or token) numbers for the blocks in each file are collected separately. These are sorted with the start numbers in ascending order and, within this, the end numbers in descending order. This simplifies the checking for subsumed blocks because any subsuming block is just before the block(s) it subsumes. Once the end of the second block is after the end of the first block, the subsumption of blocks in the first block is ended. For example, in the sequence (20 90), (30 70), (80 110), the second block, (30 70), is subsumed by the first (20 90) because any block starts either at or after the preceding block's start by virtue of the sort, and the end point 70 is before 90, whereas the third block, (80 110), is not subsumed by the first because its end point, 110, is after that of the first block, 90[1]. Inter-file matched block pairs which include subsumed blocks are then removed.

In the example, the subsumed blocks b, h, q and s are removed. Block n is also removed as it no longer has a match in block b. The unsubsumed blocks are a, c, d, e, f and g in file 1 and m, n, o, p, r and t in file 2. There are 6 matched blocks: a and r, c and o, d and t, e and m, f and t, and g and p. Block t is matched twice, both to block d and to block f.

Step	**Clones or Clone pairs**
Matched blocks	((20 90) (200 270)), ((30 70) (30 70)), ((30 70) (210 250)), ((80 110) (80 110)), ((120 130) (150 160)), ((120 130) (280 290)), ((140 170) (10 40)), ((190 200)(150 160)), ((190 200) (280 290)), ((210 270) (120 180)), ((240 250) (150 160)), ((240 250) (280 290))
File 1 block numbers	(20 90), (30 70), (80 110), (120 130), (140 170), (190 200), (210 270), (240 250)
File 2 block numbers	(10 40), (30 70), (80 110), (120 180), (150 160), (200 270), (210 250), (280 290)
Unsubsumed file 1 blocks	(20 90), (80 110), (120 130), (140 170), (190 200), (210 270)
Unsubsumed file 2 blocks	(10 40), (30 70), (80 110), (120 180), (200 270), (280 290)
Unsubsumed matched blocks	((20 90) (200 270)), ((80 110) (80 110)), ((120 130) (280 290)), ((190 200) (280 290)), ((140 170) (10 40)), ((210 270), (120 180))
The matched blocks, blocks, by name	(a r), (c o), (d t), (f t), (e m), (g p)

Table 3: Unsubsuming: the subsumed blocks in file 1 and in file 2 are identified. In this example the subsumed blocks are (30 70) and (240 250) in file 1, and (150 160) and (210 250) in file 2. The 5 matched pairs which include these 4 blocks are removed.

[1]Any block subsumed by more than one other block is treated as subsumed by the block which has the closest starting point to it. For example, in the sequence ((20 90)(30 110)(30 60)), (30 60) is subsumed by both blocks, and is "caught" by (30 110). For this application, subsumption matters but the subsumer is unimportant.

4.2 Uncopy

The next step in unscrambling is the removal of duplicated matches. In the running example, after unsubsuming, there is only one instance of this type of duplication, the matching of block t to blocks d and f, (see the bottom row of Table 3, or Figure 5b(p.14)).

As the numeric information on duplicated copies has no context without further processing, an arbitrary choice must be made about which pairs of copies in the files to match. We simply match the first occurrence of the copy in file 1 to the first occurrence in file 2, and the second with the second and so on. In the running example, this means that block d will be matched with block t and that block f will be removed.

In this section, a simple method of matching multiple copies is first described. The problem caused by overlapping multiple clones is then explained. Lastly, an adaptation to the simple method which resolves this problem is detailed.

4.2.1 Simple uncopying

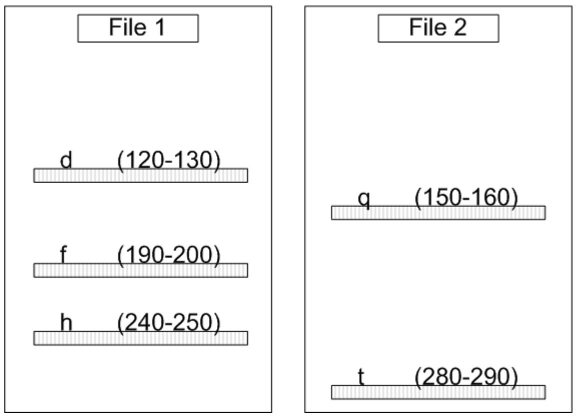


Figure 2: Five identical blocks of code in two files, three in file 1, two in file 2.

To provide an example to illustrate uncopying, Figure 2 shows the five identical blocks of code present in the example files before unsubsuming, three in file 1 and two in file 2. In file 1 the sections start at lines (or tokens) 120, 190 and 240, while those in file 2 start at lines (or tokens) 150 and 280. There are six matched pairs.

A simple way to uncopy the clones is to sort the copy list on the start of the first clone in a pair, and within this, the end of the first clone, then the start of the second clone. This places any duplicated sections of code in file 1 in the order that they occur in the file, and the sections with which they are paired in file 2 ordered from the start to the end of the file. The first pair in the list are matched, and all other pairs in the list which include either the same file 1 clone or the same file 2 clone are removed from the list, because each of these sections of code now have a match in the other file.

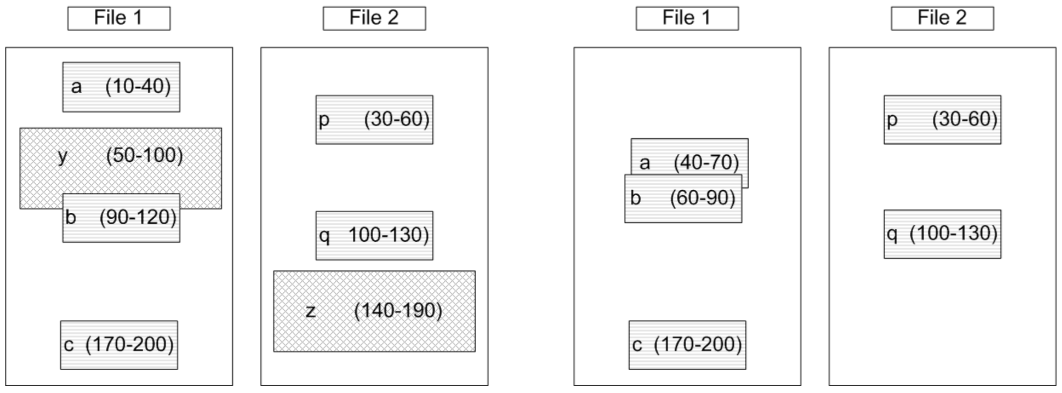
Table 4 details the steps in matching the blocks in the example in Figure 2, showing the 2 matches made between the first two blocks in each file and the second two blocks in each file, the third block in file 1 is unmatched.

Step	Clone pairs
$3 * 2 = 6$ Matched blocks	((120 130)(150 160)), ((190 200)(150 160)), ((240 250)(150 160)) ((120 130)(280 290)), ((190 200)(280 290)), ((240 250)(280 290))
Sorted pair list	((120 130)(150 160)), ((120 130)(280 290)), ((190 200)(150 160)), ((190 200)(280 290)), ((240 250)(120 130)), ((240 250)(280 290))
Matched pair 1	((120 130)(150 160))
Pairs with 1st clone (120 130) or 2nd clone (150 160) removed, leaving	((190 200)(280 290)), ((240 250)(280 290))
Matched pair 2	((190 200)(280 290))
Unmatched section in file 2	(240 250)

Table 4: Example of the uncopy process on the multiply matched blocks shown in Figure 2

4.2.2 Overlap problem

However, there is a problem with simply matching multiple clones in the order that they appear in the files when these clones overlap other clones. For example, assume there are multiple clones, say m clones in file a, and n in file b, where m>n and none of the n clones in file b overlap. If any of the first n of the m clones in file a overlap either with each other, or with another clone, then it may be better to match the clones in file b to later occurrences of the clones in file a, so that the amount of copying is not reduced by the overlapping code. This is illustrated in Figure 3. In each case it may be better to match the second clone (q) in file 2 to the third clone (c) in file 1, to maximise matching between the files.



(a) Multiple clone b overlaps clone y.

(b) Multiple clones a and b overlap.

Figure 3: Two clone arrangements which may cause problems for the simple uncopy method. Each diagram shows identical clones, a, b and c in file 1, and p and q in file 2; in both examples clone b overlaps another clone. It may be more appropriate to match clone q to clone c rather than to clone b.

4.2.3 Amended uncopy

To deal with this problem, the clones in file 1 and in file 2 are scanned to find those which overlap. Clone pairs in which either of the clones overlap another clone are moved to the end of the list, and are therefore matched after clones which do not overlap. The clones in Figure 3a are used to illustrate the process which is shown in Table 5. A further refinement, where many clones overlap, is to sort the overlapping clones according to the amount of overlap, with the most overlapped at the end of the list.

Step	**Clone pairs**
Sorted pair list	((10 40)(30 60)), ((10 40)(100 130)), ((50 100)(140 190)), ((90 120)(30 60)) ((90 120)(100 130)), ((170 200)(30 60)), ((170 200)(100 130))
File 1 clones	(10 40), (50 100), (90 120), (170 200)
File 2 clones	(30 60), (100 130), (140 190)
Overlapping clones	File 1 ((50 100), (90 120))
Pairs where 1st clone is (50 100) or (90 120) are put to end of list	((10 40)(30 60)), ((10 40)(100 130)), ((170 200)(30 60)), ((170 200)(100 130)) ((50 100)(140 190)), ((90 120)(30 60)), ((90 120)(100 130))
Matched pair 1	((10 40)(30 60))
Reduced list 1	((170 200)(100 130)), ((50 100)(140 190)), ((90 120)(100 130))
Matched pair 2	((170 200)(100 130))
Matched pair 3	((50 100)(140 190))
Unmatched section in file 1	(90 120)

Table 5: An example of the adapted method of uncopying which prioritises matches between non-overlapping multiple clones.

4.3 Unoverlap

The final step in unscrambling the clones is the removal of overlapping sections from one of each pair of overlapping clones. Here the overlapping section is removed from the smaller of the two clones, or from the second clone, if they are of equal size; another arbitrary choice. The same section of the matching block in the other file must also be removed. In the running example in Figure 5c, after unsubsuming and uncopying, there is only one overlapping pair of blocks, a and c in file 1, with lines 80-90 overlapping. As block c is smaller than block a, it is reduced to the lines 91-110. Block o in file 2 which matches block c is similarly reduced, see Figure 5d.

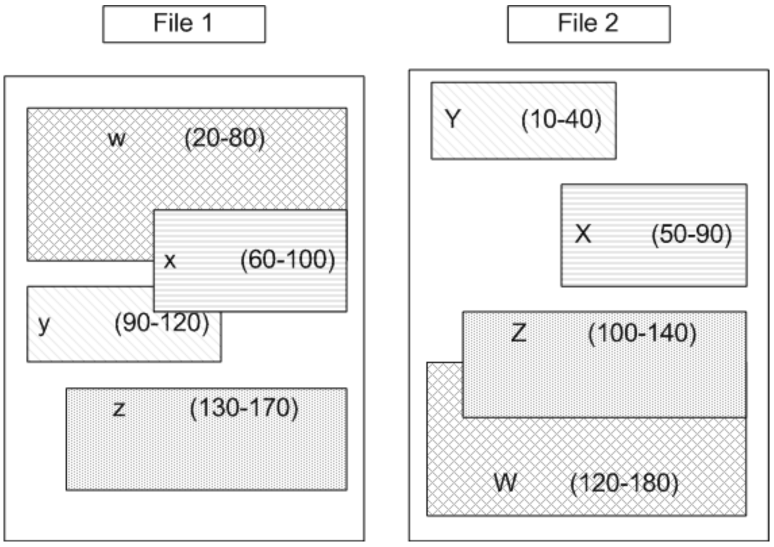
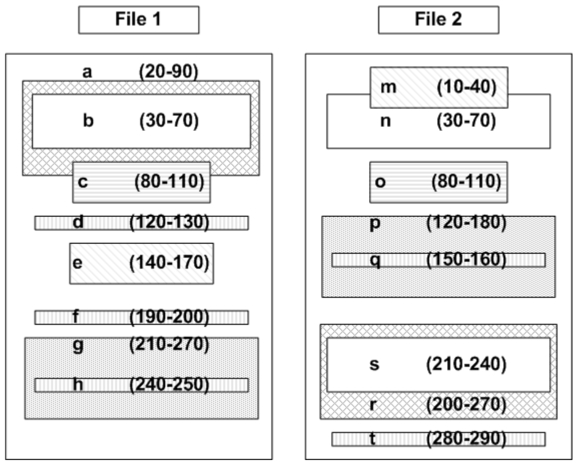


Figure 4: Two files with overlapping matched blocks

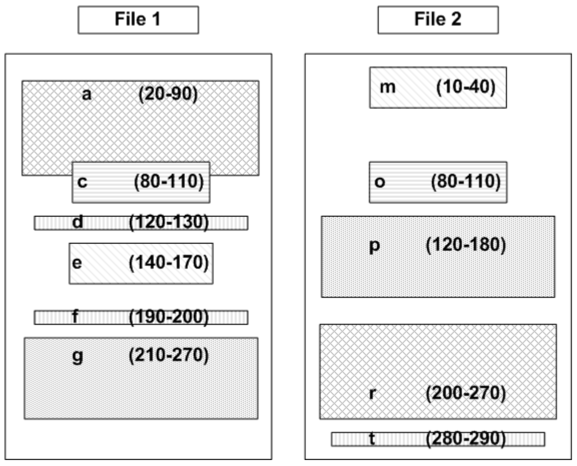
The example in Figure 4 is used to illustrate unoverlapping. The matched pairs are labelled with the same letter in lower or upper case, and have the same shape and shading. The pairs are w:((20 80)(120 180)), x:((60 100)(50 90)), y:((90 120)(10 40)) and z:((130 170)(100 140)). In removing overlapping sections of code, here file 1 is processed first, from the start of the file. The matched pairs are sorted on the clone start point in file 1. Consecutive clone pairs are checked for overlap. If two clones overlap, then the overlapping section is removed from the smaller of the two, and the equivalent part of its matching clone in file 2 is also removed. If a clone is reduced because it overlaps the previous one, then the reduced clone is used in subsequent pairwise checks. Once the clones in file 1 are checked and reduced as necessary, the reduced clones in file 2 are checked in a similar way. Table 6 shows the steps in unoverlapping for the example files in Figure 4.

Step	Clones or clone pairs
Checking file 1 blocks:	
Matched list, sorted on start of file 1 pair	((20 80)(120 180)), ((60 100)(50 90)), ((90 120)(10 40)), ((130 170)(100 140))
1st 2 blocks overlap, reduce smaller, and its pair	((20 80)(120 180)), ((81 100)(71 90)), ((90 120)(10 40)), ((130 170)(100 140))
2nd 2 blocks overlap, reduce smaller, and its pair	((20 80)(120 180)), ((81 89)(71 79)), ((90 120)(10 40)), ((130 170)(100 140))
3rd 2 blocks do not overlap, do nothing	((20 80)(120 180)), ((81 89)(71 79)), ((90 120)(10 40)), ((130 170)(100 140))
Checking file 2 blocks:	
New matched list, sorted on start of file 2 pairs	((90 120)(10 40)), ((81 89)(71 79)), ((130 170)(100 140)), ((20 80)(120 180))
1st 2 blocks do not overlap, do nothing	((90 120)(10 40)), ((81 89)(71 79)), ((130 170)(100 140)), ((20 80)(120 180))
2nd 2 blocks do not overlap, do nothing	((90 120)(10 40)), ((81 89)(71 79)), ((130 170)(100 140)), ((20 80)(120 180))
3rd 2 blocks overlap, reduce smaller, and its pair	((90 120)(10 40)), ((81 89)(71 79)), ((130 149)(100 119)), ((20 80)(120 180))

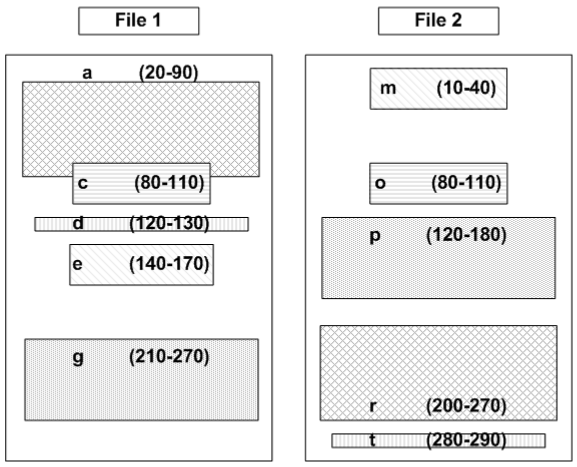
Table 6: The steps in unoverlapping the files shown in Figure 4



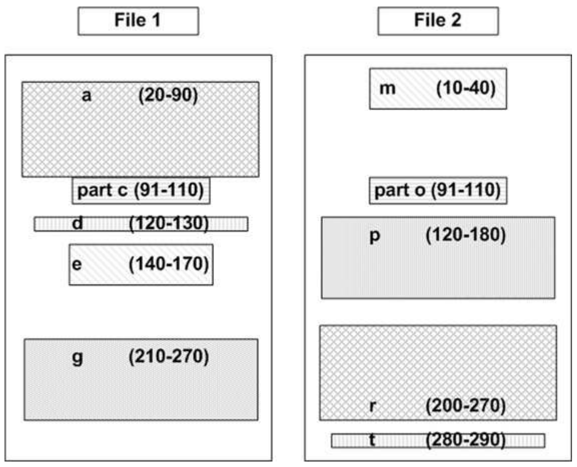
(a) Original clones



(b) Unsubsumed



(c) Uncopied



(d) Unoverlapped and finished

Figure 5: An illustration of the unscrambling process, based on the files introduced in Figure 1 (see p. 6) which is repeated as Figure 5a.

4.4 Ordering the steps

The order of the steps in unscrambling is important. Unoverlapping should be the last step, because until the clones are unsubsumed and uncopied, it is unclear which clones remain, and therefore which overlap. In the example in Figure 5a, if the blocks were unoverlapped first, then block m would be reduced because it overlaps block n, however block n has no match after unsubsuming and is therefore removed, so this reduction in block m is incorrect. Unsubsuming needs to be done before uncopying, so that multiple clones are not matched to clones which are then removed.

4.5 What difference does unscrambling make?

The amount of ‘duplication’ in the clones detected between a pair of files depends on the content of the files, on the size of clone specified for detection, and on other factors, such as the matching algorithm and the parameterisation of the code prior to matching. In some cases, the number of copied lines making up the raw clones exceed the number of lines in the file.

The example shown in Figure 5 is used to illustrate the difference between the number of copied lines (or tokens) in the raw clones detected and the number after unscrambling. Adding the number of lines in the raw clone data gives a total of 272 copied lines out of 300 in file 1 and 292 out of 300 in file 2, suggesting that most of the 300 lines are shared by the files.

Looking at Figure 5a (or Figure 1), it is clear that there can be no more lines shared between the files than the number of lines covered by the clones. These are lines 20-110, 120-130, 140-170, 190-200 and 210-270, or $91+11+31+11+61 = 205$ lines. When this is reduced by the 11 lines in the unmatched block f, the expected number of matched lines is 194, which should be the same in file 2 because of the requirement to match on a one-to-one basis. The lines contained by the unscrambled clones in Figure 5d confirm this, as there are $71 + 20 + 11 + 31 + 61 = 194$ in file 1, and $31+10+61+71+11 = 194$ in file 2, which better represents the amount of shared code.

4.6 Approximations

Two approximations are made in the unscrambling method described here. First, the arbitrary decision to match blocks sequentially when there are multiple matches does not necessarily give the 'best' matches. Second, the order of removing the overlapping sections of code, both the within-file ordering, from start to end, and the between-file ordering, file 1 before file 2, will affect the outcome. These decisions do not change the raw numbers of lines and tokens, but they may impact on block size and distribution. Further investigation may show that the context of the clones could be used to assess which are the most appropriate matches, and which are the overlaps to reduce.

5 Summary

Identifying the amount of code common to two files can be a difficult task. This is especially true when the code has been rearranged or when identifiers have been renamed. Clone detection tools can assist in this task because they are able to find matching sections of code wherever they are situated in the files. Another benefit of tools such as CCFinderX and Simian is that identifiers are, or can be, parameterised, so that renaming will not affect matching.

In this report, we have explained the three sources of duplication which can occur when using clone detection tools for this task. We have also described a method for disentangling clone detection tool output to approximate the one-to-one matching of code between files. We are aware that the method could be improved; however, it offers a reasonable solution to finding out from clone detection tool output how much of the code in one file appears in another.

References

[1] Christian M. Ammann. Duplo - code clone detection tool. Sourceforge project, 2005. http://sourceforge.net/projects/duplo/.

[2] I.D. Baxter, A. Yahin, L.M. De Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[3] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[4] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching: Research articles. *J. Softw. Maint. Evol.*, 18(1):37–58, 2006.

[5] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118. IEEE, 1999.

[6] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 321–330. ACM, 2008.

[7] Pam Green, Peter C. R. Lane, Austen Rainer, and Sven-Bodo Scholz. Building classifiers to identify split files. In Petra Perner, editor, *MLDM Posters*, pages 1–8. IBaI Publishing, 2009.

[8] Simon Harris. Simian. http://www.redhillconsulting.com.au/products/simian/. Copyright (c) 2003-08 RedHill Consulting Pty. Ltd.

[9] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *ICSE*, pages 485–495. IEEE, 2009.

[10] Toshihiro Kamiya. AIST CCFinder official site. http://www.ccfinder.net/index.html.

[11] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[12] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–56, 2001.

[13] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Reverse engineering*, pages 77–108, 1996.

[14] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eighth Working Conference on Reverse Engineering*, pages 301–309, 2001.

[15] P. C. R. Lane, C. M. Lyon, and J. A. Malcolm. Demonstration of the Ferret plagiarism detector. In *2nd International Plagiarism Conference*, 2006.

[16] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[17] C. M. Lyon, R. Barrett, and J. A. Malcolm. A theoretical basis to the automated detection of copying between texts, and its practical implementation in the Ferret plagiarism and collusion detector. In *JISC(UK) Conference on Plagiarism: Prevention, Practice and Policies Conference*, 2004.

[18] C. M. Lyon, J. A. Malcolm, and R. G. Dickerson. Detecting short passages of similar text in large document collections. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*. SIGDAT, Special Interest Group of the ACL, 2001.

[19] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.

[20] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244–, 1996.

[21] Ettore Merlo. Detection of plagiarism in university projects using metrics-based spectral similarity. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, volume 06301 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[22] A. W. Rainer, P. C. R. Lane, J. A. Malcolm, and S-B. Scholz. Using n-grams to rapidly characterise the evolution of software code. In *The Fourth International ERCIM Workshop on Software Evolution and Evolvability*, 2008.

[23] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[24] Robert Tairas and Jeff Gray. Phoenix-based clone detection using suffix trees. In *ACM Southeast Regional Conference*, pages 679–684, 2006.

[25] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, page 327, Washington, DC, USA, 2002. IEEE Computer Society.

A Appendix

The method for unscrambling code clones described in the report can be used with any clone detection tool, provided that the start and end locations of the clones can be determined.

This appendix provides examples of three clone detection tools and their output. In our application, we combine the three tools, Duplo, CCFinderX and Simian. The tools were selected because they are freely available, stand-alone, command-line tools which work with C code, and provide complementary matching techniques. Duplo is text-based and reports on exact matches between blocks of code. CCFinderX is token-based and matches code which is simplified, parameterised and tokenised. Simian falls between the two, as it provides parameterisation options and uses line-by-line matching.

Here the tools are briefly described, and, using a comparison between two programs, sample output generated by each tool is shown. The programs compared are one of five lines, and a copy with one repeated line, to give a 'toy' illustration of overlapping clones. Duplo, CCFinderX and Simian are described in Sections A.1, and A.3. Examples of the output each tool produces and the detected clones are shown in Section A.4.

A.1 Duplo

Duplo is an open source program similar to 'Duploc' [4, 5], which uses hashed string matching of lines of code for clone detection. Duplo allows comparison of two (or more) files to find blocks of matched code. Two parameters determine a block: the minimum number of characters in a line and the minimum consecutive matched lines in the block. The default parameter settings for the program define a matching block to be at least four lines, each of at least three characters. The code is preprocessed to remove whitespace, comments and, optionally, preprocessing directives.
A line such as "`while (( c = getchar()) != EOF ) \* some comment *\`"
becomes "`while((c=getchar())!=EOF)`".
The lines with fewer than the minimum number of characters specified are ignored. To optimise matching, each line is hashed using the MD5 algorithm. A matrix is coded to show whether each pair of lines in the files match. The matrix is scanned to find contiguous matches and those of at least the minimum number of lines are reported. Table 7 shows the matrix for the example files, in which a 1 indicates a matching pair of lines. There is one sequence of three copied lines, and one of two, which are highlighted in different colours, and correspond with the output shown in Tables 8 and 9.

	main()	int a = 1 + 2;	int b = a + 5;	int b = a + 5;	int c = a * b;
main()	1	0	0	0	0
int a = 1 + 2;	0	1	0	0	0
int b = a + 5;	0	0	1	1	0
int c = a * b;	0	0	0	0	1

Table 7: A matrix for a comparison between the example files shown in Table 8, here 1 is used to indicate a matching pair of lines, and 0 a non-matching pair. Sequences of 1s across the left to right downward diagonals, such as the two highlighted, show consecutive copied lines.

The report provided by Duplo details the files and line numbers where each matching block starts, the code which forms the block, and summary figures which include the total number of lines in the compared files, the number of duplicated blocks and duplicated lines. An example of Duplo output is shown in the fourth column of Table 9.

A.2 Code Clone Finder (CCFinderX)

CCFinderX is a token-based code clone detector [10]. The preprocessing stage filters out parts of the file which are considered to be uninteresting, such as comments; or parts which may produce false clones because of their repetitious structure, such as declarations, preprocessing directives and initialisations. CCFinderX can be used with any of the following filetypes: C, C++, C#, Cobol, Java, Visual Basic and text files. The 'interesting' code is tokenised. The tokens are transformed using language-specific rules to produce a sequence of key words and parameterised identifiers.

There is flexibility in the size of clone detected and in the type of matching performed. Clone size is determined by specifying the minimum number of tokens in a sequence and the minimum number of token types. Token types include identifiers, typed literals, one for each operator, such as "`*`" or "`=`", and for each delimiter, such as "`;`" or "}". Identifiers can be matched by one-to-one replacement or not.

CCFinderX outputs information about code clones in the form of two sets of numbers for each match. These numbers indicate the start and end token numbers of the matched section of each file. For example, "1.10-60 2.35-95" means that the section of file 1 including tokens 10-60 matches tokens 35-95 in file 2.

A.3 Simian

Simian ignores whitespace, curly braces, comments, imports, includes and package declarations. Simian tokenises to allow a number of optional parameters for relaxed matching; for example, matching identifiers regardless of names; matching any string to any other string; or by ignoring the case of letters in strings. Matching is based on the hashed values of significant lines of the transformed code text. Simian supports Java, C (and #, ++, objective), COBOL, Ruby, Lisp, SQL, Visual Basic and Groovy.

The user is able to specify the minimum number of lines in a clone. The tool outputs a list of all matches, both within file and between files. The output can be parsed to find the start and end line numbers for each inter-file clone.

A.4 Example output

Two small example files, with an unusual repetition of one line to illustrate clones which overlap, and the result of comparing them using both Duplo, CCFinderX and Simian are shown in Table 9. For these comparisons, the Duplo parameters, the minimum number of lines in a block and characters in a line, are both set to 2; the CCFinderX parameters, the minimum number of tokens in a clone and different types of tokens, are set to 5 and 2 respectively; and the Simian threshold is set to 2, all other parameters are the default settings.

The clones are shown in Table 8, the two found by both Duplo and Simian are on the left and the four by CCFinderX on the right. The files are repeated, with the clone highlighted in bold text and the line or token numbers shown at the top of each file. To show the relationship between token numbers and the cloned code, the tokens generated by CCFinderX for each file are listed in the lower section of columns 2 and 3 in Table 9. Dashed lines have been added to indicate the line breaks in the code.

The Duplo clones in the example, constructed from the start line number and a count of the number of copied lines reported, are ((0 2)(0 2)) and ((3 4)(5 6)). Note that file 2 precedes file 1 in the output. Also note that lines with fewer than the minimum specified characters are ignored, therefore the constructed end line number is lower than expected. For example, the first clone pair should be ((0 3)(0 3)), but as line 2 has only one character it is ignored in the reported code. This does not affect multiple clones or subclones, as the code will be treated in the same way if it is identical. However, inaccuracy may occur when processing overlapping clones. For example, the two clones in file 1 overlap at line 3, but this is not apparent from the constructed line numbers in the matched pairs. To suit our application, we have adapted the method used by Duplo to provide the actual line numbers for the end of the clones.

The Simian and CCFinderX clones are taken directly from the output by parsing the text and are Simian : lines - ((4 5)(6 7)), ((1 4)(1 4)); and CCFinderX : tokens - ((6 24)(6 24)), ((10 17)(24 31)), ((10 27)(17 34)) and ((17 24)(10 17)).

As expected, each of the clone detection tools counts line 4 in file 1 twice, matching it to both line 4 and line 6 in file 2. For Duplo and Simian, this is clear from Table 8: both the first four lines and the last three lines of the files are matched. This means that line 4 in file 1 is matched both to line 4 and to line 5 in file 2.

It is less simple to see the duplication in the CCFinderX output. Totalling the tokens in the clones identified by CCFinderX gives 53 tokens. The clones cover 22 tokens (6-27) in file 1 and 29 (6-34) in file 2, the 7 extra tokens being those in line 5, file 2, which is an extra match to line 4 in file 1.

(Duplo) and [Simian]		**CCFinderX**			
File 1 (0 2) [1 4]	File 2 (0 2) [1 4]	File 1 (6 24)	File 2 (6 24)	File 1 (10 27)	File 2 (17 34)

main()
{
int a = 1 + 2;
int b = a + 5;
int c = a * b;
}
	main()
	{
	int a = 1 + 2;
	int b = a + 5;
	
	int b = a + 5;
	int c = a * b;
	}
		main()
		{
		int a = 1 + 2;
		int b = a + 5;
		int c = a * b;
		}
			main()
			{
			int a = 1 + 2;
			int b = a + 5;
			
			int b = a + 5;
			int c = a * b;
			}
				main()
				{
				int a = **1 + 2;**
				int b = a + 5;
				int c = a * b;
				}
					main()
					{
					int a = 1 + 2;
					int b = **a + 5;**
					
					int b = a + 5;
					int c = a * b;
					}

File 1 (3 4) [4 5]	File 2 (5 6) [6 7]	File 1 (10 17)	File 2 (24 31)	File 1 (17 24)	File 2 (10 17)

main()
{
int a = 1 + 2;
int b = a + 5;
int c = a * b;
}
	main()
	{
	int a = 1 + 2;
	int b = a + 5;
	
	int b = a + 5;
	int c = a * b;
	}
		main()
		{
		int a =**1 + 2;**
		int b = a + 5;
		int c = a * b;
		}
			main()
			{
			int a = 1 + 2;
			int b = a + 5;
			
			int b =**a + 5;**
			int c = a * b;
			}
				main()
				{
				int a = 1 + 2;
				int b =**a + 5;**
				int c = a * b;
				}
					main()
					{
					int a =**1 + 2;**
					int b = a + 5;
					
					int b = a + 5;
					int c = a * b;
					}

Table 8: The reported clones: on the left, the two found by both Duplo and Simian, Duplo line numbers are shown in round brackets and Simian in square brackets, Duplo indexes from 0 and Simian from 1. On the right, the four clones found by CCFinderX. In each case, the duplicated code is shown in bold text.

Line	File 1	File 2
0	main()	main()
1	{	{
2	int a = 1 + 2;	int a = 1 + 2;
3	int b = a + 5;	int b = a + 5;
4	int c = a * b;	
5	}	int b = a + 5;
6		int c = a * b;
7		}

CCFX Tokens	File 1	File 2
1	(def_block	(def_block
2	id—main	id—main
3	c_func	c_func
4	(paren	(paren
5	)paren	)paren
6	(brace	(brace
7	r_int	r_int
8	id—a	id—a
9	op_assign	op_assign
10	l_int	l_int
11	op_plus	op_plus
12	l_int	l_int
13	suffix:semicolon	suffix:semicolon
14	r_int	r_int
15	id—b	id—b
16	op_assign	op_assign
17	id—a	id—a
18	op_plus	op_plus
19	l_int	l_int
20	suffix:semicolon	suffix:semicolon
21	r_int	r_int
22	id—c	id—b
23	op_assign	op_assign
24	id—a	id—a
25	op_star	op_plus
26	id—b	l_int
27	suffix:semicolon	suffix:semicolon
28	)brace	r_int
29	)def_block	id—c
30	eof	op_assign
31		id—a
32		op_star
33		id—b
34		suffix:semicolon
35		)brace
36		)def_block
37		eof

Duplo output

```
/clone-examples/overlap2.c(0)
/clone-examples/overlap1.c(0)
main()
int a = 1 + 2;
int b = a + 5;

/clone-examples/overlap2.c(5)
/clone-examples/overlap1.c(3)
int b = a + 5;
int c = a * b;

Configuration:
Number of files: 2
Minimal block size: 2
Minimal characters in line: 2
Ignore preprocessor directives: 0
Ignore same filenames: 0

Results:
Lines of code: 9
Duplicate lines of code: 5
Total 2 duplicate block(s) found.

Time: 0 seconds
```

CCFX output

```
version: ccfx 10.2.6
format: pair_diploid
option: -b 5
option: -s 2
option: -u -
option: -t 2
option: -w f-g+w-
option: -j +
option: -k 60m
option: -ig 1
option: -ig 2
option: -preprocessed_file_postfix
.cpp.1_1_1_3.default.ccfxprep
preprocess_script: cpp
source_files {
1 ../clone-examples/overlap1.c 30
2 ../clone-examples/overlap2.c 37
}
source_file_remarks {
}
clone_pairs {
1 1.6-24 2.6-24
4 1.10-17 2.24-31
6 1.10-27 2.17-34
4 1.17-24 2.10-17
1 2.6-24 1.6-24
4 2.10-17 1.17-24
6 2.17-34 1.10-27
4 2.24-31 1.10-17
}
clone_set_remarks {
}
```

Simian output

```
Copyright (c) 2003-08 RedHill Consulting Pty. Ltd.
All rights reserved. Simian is not free unless
used solely for non-commercial or evaluation purposes.

 ..... [options list] ..... threshold=2
Found 2 duplicate lines in the following files:
Between lines 6 and 7 in ../clone-examples/overlap2.c
Between lines 4 and 5 in ../clone-examples/overlap1.c
Found 3 duplicate lines in the following files:
Between lines 1 and 4 in ../clone-examples/overlap1.c
Between lines 1 and 4 in ../clone-examples/overlap2.c
Found 10 duplicate lines in 4 blocks in 2 files
Processed a total of 9 significant (14 raw) lines in 2 files
Processing time: 0.032sec
```

Table 9: On the left of this table at the top are two small example files with an artificial overlap. Below this, the CCFinderX tokens for the files are shown, with dashed lines placed to show line breaks. Clone detection tool output from comparison between the two files is shown on the right: that from Duplo and CCFinderX at the top of columns 4 and 5 respectively, and from Simian below.