Substring Matching for Clone Detection and Change Tracking

J Howard Johnson

Software Engineering Laboratory,
National Research Council of Canada,
Ottawa, Canada K1A 0R6

Abstract

Legacy systems pose problems to maintainers that can be solved partially with effective tools. A prototype tool for determining collections of files sharing a large amount of text has been developed and applied to a 40 megabyte source tree containing two releases of the gcc compiler. Similarities in source code and documentation corresponding to software cloning, movement and inertia between releases, as well as the effects of preprocessing easily stand out in a way that immediately conveys non-obvious structural information to a maintainer taking responsibility for such a system.

Keywords: *legacy code, string matching, clone detection, program understanding, textual redundancy*

1 Introduction

Reverse engineering and design recovery [2,4,5] provide useful tools for program understanding, an important part of software maintenance of legacy systems. Such tools often mimic the front-end of the compilation process and yield much useful information that a compiler acquires to generate good code but which is normally not available to programmers. This information can be analysed, transformed, and summarized in a way that leads to significant insights into what is being done by the code as it executes.

A different approach is to consider the source as text and analyse it the way documents are analysed. This is closer to the way software developers and maintainers see code and provides, in a language-independent manner, a powerful set of tools that can access information inaccessible to the compiler-based approaches. Although not a replacement for these approaches, it provides information that is very difficult to obtain using them.

Identifying software clones and understanding how software changes between releases are two important issues for maintainers where a text-based approach is likely to be useful. The experience obtained using a prototype textual analysis system bears this out. Useful similarities in source do provide useful information.

Section 2 outlines the phenomena of software cloning and reorganization in large systems that will be studied by this technology. Section 3 describes the technology used to study textual matches. Following this, in Section 4, is an example of this approach that demonstrates its utility. Finally, conclusions and current research directions are outlined in Section 5.

2 Sources of textual similarity

Text in different files can be similar for a number of reasons. Two of these are of particular interest from a software maintenance point of view: software cloning and change (or lack of change) between versions of a system. A particular example of software cloning occurs in some approaches to the management of multiple configurations. Before one discusses any tool-based solutions, it is useful to describe these problems and their effect on software maintenance.

2.1 Software cloning

Maintenance of large software systems under pressure often leads to a phenomenon referred to as software cloning.

The scenario is typically something along the following lines:

- In the process of enhancing the functionality of the system or in removing a defect, a requirement for a sub-component similar to an existing sub-component is identified.

NRC 38301

0-8186-6330-8/94 $04.00

- The functionality of the required sub-component is sufficiently different from the existing component that substantial change is needed that will affect the existing uses of the existing component.
- The analysis of the usage pattern of the existing component is expected to be lengthy and difficult. Furthermore, significant regression testing will be required to ensure that old features still work correctly if it is changed in any way.
- In the interest of meeting production targets, a copy of the existing component is made and a systematic renaming is done to avoid naming conflicts and to tailor the component to its new use.
- Since the internals of the module are not completely understood (because of time pressure), unnecessary artifacts of the previous usage are preserved in the current code as "red herrings" for future maintainers. Some of these artifacts can never be accessed and constitute "dead code".

This cut-and-paste activity can happen for code fragments of a few lines or for modules of thousands of lines. It can happen in procedural code or in declarations. It can happen in documentation.

Software cloning has a number of negative effects on software maintenance:

- As mentioned above, red herrings and dead code are created.
- If the component that was copied is subsequently discovered to have a defect, the defect probably should be repaired in the other clones. These clones must be found and the impact of the correction assessed in each context.
- Errors in the systematic renaming can lead to unintended aliasing, resulting in latent bugs that show up much later.
- The bulk of the code grows much faster than it would by extending the functionality of the existing module to meet the new requirement.

The effect of all of these is a form of "software aging" or "hardening of the arteries" that results when even small design changes become very difficult to make.

This is not to say that software cloning is all bad. First of all, it meets the short-term goal of quicker and more reliable change. It is also sometimes necessary because the programming language employed lacks an abstraction mechanism. For example, an abstract or generic algorithm may result in very different code in different contexts and the language does not support instantiation of generic procedures. The programmer thinks of the instances of the algorithm as being somehow the same but there is no way of saying this just once.

Whether cloning is good or bad, the understanding of a large source depends on finding and understanding cases of how code has been copied and modified. This is the motivation for tools for clone detection.

2.1.1: Other approaches: Since a common approach to understanding source is to perform lexical analysis on it followed by some semantic analysis, it is natural to look for clones as subtrees of the syntax or semantic trees that match in all or in part. This has the advantages that it can be combined easily with other semantic analysis and that it also identifies common recurring code fragments or clichés. This is the approach of a group from McGill [3]. A number of software metrics are calculated for each subtree and a clone is signaled if the metrics all agree.

Another approach similar to that discussed here is that of Baker [1]. She calculates a position tree for the whole of the source and uses the information contained in it to identify large matches. The approach described here has a slightly different view about matches and is designed to scale up to much larger sources than can likely be handled using position trees.

2.2 Change in large systems

In the natural evolution of large systems, the content of individual files changes, large files split into smaller ones, files are renamed, and directories are reorganized. These introduce confusion for the maintainer, since things are not where they were before. The maintainer needs to be able to visualize how the present system relates to his or her understanding of a previous version.

Change also shows where activity is going on to remove bugs or enhance functionality. To identify the important parts of the system for maintenance purposes it is useful to consider what has been changed previously.

Identifying change for both of these reasons can be supported usefully by tools, especially in large systems.

2.3 Support for multiple configurations

Large systems are usually expected to run in more than one environment. There are several strategies for maintaining a number of similar but different versions of modules. One such strategy involves maintaining separate files for each variant, only one of which is used in any specific system build.

Understanding the structure of these multiple configurations can be important for maintenance since modifications may affect the environments in different ways.

3 Substring matching in large bodies of source

The approach taken can be summarized as

(1) For each file being considered, apply a text-to-text transformation to discard characters not to be considered for matching. For exact matching, this is an identity transformation (output equals input). Various types of approximate matching can be accommodated by discarding different parts of the input.
(2) Generate a set of substrings that together cover the source (i.e., every character of text appears in at least one substring).
(3) Identify matching substrings (i.e., those with the same sequence of characters).
(4) Transform this database of raw matches into a form that more concisely expresses the same information.
(5) Perform task-specific data reduction.
(6) Summarize high-level matches.

Steps (2) and (3) are information collecting phases, (4) is an information-preserving transformation, (5) is an aggregation and simplification phase, and (6) presents results in a useful form. Phase (1) provides greater sensitivity for particular types of input.

In the following subsections, these phases will be described for a prototype implementation of a tool that identifies clusters of files with significant text in common. Section 4 will show, by means of a realistic example application, how this tool can be used to identify where code is the same as a result of software cloning and where code has changed across releases.

3.1 Text-to-text source transformation

To ensure that the approach scales up, it is based on the idea of exact matching. This restriction means that sorting can be used to bring together simultaneously all identical substrings. This approach will not work for the usual models for approximate matching and they require more complex (and more expensive) approaches.

Exact matching can be generalized in a weak fashion to handle one form of approximate matching that can be referred to as "exact matching on partial information." A text-to-text transformation is applied to the source before substrings are generated. The following candidate transformations seem plausible:

(1) Remove all white space characters (blank, tab, carriage return, line feed). The resulting matches are not sensitive to different layout on the page effected using only white space.
(2) Remove all white space except for line separators. Results are similar to (1), but line boundaries are preserved.
(3) Replace each (maximal) sequence of white space characters by a single blank. There are variants of this approach based on (1) and (2). Matches will then be found if white space is used in the same places.
(4) Remove comments.
(5) Retain only comments.
(6) Replace each identifier by an identifier marker.
(7) Various combinations of the above.

Other text-to-text transformations involving parsing the text and pretty-printing the parse trees are also possible and may provide useful match information. Baker's approach [1] will not work without modification if done before substrings are generated.

3.2 Generation of candidate substrings

The second phase of the process involves producing a collection of substrings that will be checked for matches. This must be done carefully, since producing too many substrings causes much more processing to be required and producing too few will result in matches being missed completely.

It is important that the rate at which substrings are generated can be controlled by the user of the system. Average length measured in number of characters or number of lines is also a necessary parameter.

There is some subtlety in how possible substrings are sampled when it is not feasible to produce all substrings of the requested length. A large match should not be missed because of the sampling strategy. Thus, the sampling strategy can depend only on the string itself and a bounded number of characters before and after the string. Details of this approach are described elsewhere [6].

For the purpose of this investigation, no sampling was done and all sequences of 50 lines were considered for matches.

3.3 Identification of raw substring matches

This is a straightforward sorting of a file containing the content of the substring and an indication of its origin.

However, a file containing all 50-line chunks of text would be about 50 times as large as the original text (for the case study in Section 4, 2 gigabytes). Since a 40-megabyte source tree is considered to be well below the target problem size, this is unacceptable.

As a result, a fingerprinting scheme based on the Karp and Rabin string matching approach [7,8] is used. With this approach a 16-byte fingerprint stands in for the

content of the substring. Fingerprints agree if the underlying substrings agree and disagree with high probability if the underlying substrings disagree. The probability of false matches is kept very low through careful design and a long (128-bit) fingerprint. Details appear elsewhere [6].

3.4 Information-preserving simplification of match database

The raw substrings used for matching purposes have more or less the same length and overlap. Matches larger than the substring length will be represented by many raw matches. To simplify the database of matches and facilitate later processing, it is important to replace the raw substring and match information by a new minimal set of non-overlapping substrings and matches that preserves the information obtained in the matching phase.

This set of substrings has the minimum number of substrings and each substring is of maximum length. Reducing the number of substrings or lengthening any of the substrings would result either in overlapping substrings or in a loss of match information.

How this is achieved as well as a formal proof of the uniqueness of the answer will be discussed elsewhere.

3.5 Data reduction

Another way of looking at a set of matches on non-overlapping substrings is an association between a substring content (sequence of characters) and a set of places such that the set of places collectively constitutes the total source.

Since each place is an offset in a file, this information can be naturally reduced to an association between a substring content and a set of files in which it occurs. For each subset of files one can then total the lengths of substring contents that occur in exactly that set of files.

A problem occurs with matches within files. How many times should such matches be counted? One obvious answer is the combinatorial counting of the number of binary matches but this leads to such matches dominating other more interesting (for the present application) matches. Thus, it was decided for the purposes of this study to ignore such matches and reduce the sizes of files in a corresponding fashion.

This type of data reduction is particular to the application just discussed. Other applications of the basic technology would result in different data reduction strategies. The simplification of Section 3.4 loses no significant information whereas this step does result in a loss of information that in some another context might be important.

3.6 Presentation of file clusters and multi-file matches

If one considers a graph whose nodes are files and where arcs have been added whenever a match involves a set of files containing the given pair, then one can define clusters of files corresponding to the connected components of this graph. Intuitively it might seem that these "clusters" might be large and uninformative but in the case study of Section 4 an example is demonstrated where this is not the case. Each cluster corresponds to a sharing of text that is comprehensible to the maintainer without almost no noise.

A report is produced that groups the associations by cluster. Both associations and clusters are sorted in descending order by size so that the more important clusters and associations within clusters appear first.

4 Study of GNU gcc compiler v. 2.3.3 and v. 2.5.8

This system is being developed for the analysis of legacy systems. These tend to be large, proprietary, and written using languages and conventions that are not well known outside their organizations.

The GNU C compiler *gcc.* was chosen as an example that is generally known, accessible, and understood and conforms to a common source organization. The source, in C, is publicly available from a number of Internet archives.

The gcc compiler is tailored to a wide spectrum of platforms through a system of configuration files. One expects to see textual similarities among configuration files that request the same behaviour from different environments and between pairs of releases.

Copies of the current release (2.5.8), subsequently referred to as "release B", and that of one year ago (2.3.3), "release A", are the basis of this study. The contents of gcc-2.3.3.tar.Z and gcc-2.5.8.tar.gz were unpacked and, without further processing of any kind, subjected to text analysis.

The prototype implementation does not allow for the selection of files for analysis. Thus, all files in both directories were analysed. In total this corresponds to 1440 files and 40 megabytes.

4.1 Exact matches of 50 lines or more

As discussed in Section 3, all substrings of length 50 lines were generated and all matches of substring content produced. The raw matches were simplified to involve a minimal set of non-overlapping substrings, as discussed

above and data reduction and clustering was also performed. The result was 988 clusters each of which indicates something useful about the source.

These clusters fall naturally into categories depending on their size and which of the two releases are represented:

- Any cluster containing three or more files will be considered Complex.
- A cluster containing one file from the A release and one file from the B release where the file names are different will be considered of type ABX.
- A cluster containing one file from each of the A and B releases with the same name will be considered of type AB=.
- A cluster containing two files from the A release will be considered type AA.
- A cluster containing two files from the B release will be considered type BB.
- A cluster containing one file from the A release will be considered type A.
- A cluster containing one file from the B release will be considered type B.

Table 4.1 shows a breakout of the 988 clusters according to these types.

	Clusters	Files	Characters	Unique Characters
Cplx	27	155	10 019 972	7 069 451
ABX	122	244	8 614 753	5 891 012
AB=	193	386	20 354 815	12 931 221
AA	8	16	43 397	28 353
BB	1	2	458	229
A	240	240	777 667	777 667
B	397	397	2 510 274	2 510 274
Total	988	1440	42 321 336	29 208 207

Table 4.1 Exact Matching Cluster Breakout

As might be expected, the bulk of the clusters, the files, and the characters are associated with ABX and AB= type clusters. Each of the files involved in this type of cluster is rather unique within its own release but similar to a file in the other release. This is a recognition of the fact that in one year a large part of the code will remain unaffected even when very intense development is underway.

It is also interesting that a substantial number of these clusters correspond to files that have changed names between releases. On closer investigation this is seen to be primarily the result of a reorganization of the configuration subdirectory so that 'config.rs6000.md' becomes 'config/rs6000/rs6000.md'. A file 'objc-actions.c' becomes 'objc-act.c', probably to address a file system naming restriction. In addition there is a reorganization of gcc.info files so that, for example, 'A/gcc.info-13' has a large match with 'B/gcc.info-16'. The analysis has revealed a useful piece of information about change.

The file similarities have been computed from the file sizes and the extent of match. This appears to be an indication of the extent and location of change for clusters of types ABX and AB=.

Clusters of type A represent source that was changed in a massive way or discarded in favour of new versions. Clusters of type B correspondingly represent source that was created between the releases. As might be expected, there are many more new files created than old files discarded. Many in both of these categories are config files, further revealing that massive change has gone on in that subdirectory.

The AA and BB clusters identify a number of files that have two names either through copying or by means of a link. For example, 'A/objc/Object.h' and 'A/objc/object.h' are identical and are in fact 'hard-linked'. The files 'A/config/t-decstatn' and 'A/config/t-mips' are identical but copied. The file 'A/config/pa-hpux.h' is properly contained in 'A/config/pa-hpux7.h'. This type of cluster contains only a small part of the source, however.

Analysis of the complex clusters requires more sophisticated tools and is the subject of current work. However, the files participating in each of these identifies some part of the source where there is much sharing of file contents within and between releases.

The largest 10 of the 27 complex clusters can be summarized as follows:

(1) c-parse.{c,h}, cexp.{c,y}, cp-parse.{c,h,y}, objc-parse.c, and B/bi-parser.{c,y}
(2) c-decl.c and cp-decl.c
(3) A/ChangeLog, B/ChangeLog.6, and B/ChangeLog.7
(4) c-common.c, c-typeck.c, cp-type2.c, and cp-typeck.c
(5) calls.c and expr.c
(6) INSTALL, gcc.info-2, gcc.info-3, gcc.info-4, gcc.info-5, gcc.info-6, and B/gcc.info-7
(7) The config files: 3b1.h, crds.h, hp320.h, m68k.h, mot3300.h, news.h, tower-as.h, fx80.h, and B/config/m68k/dpx2.h
(8) The config files: fx80.md and m68k.md
(9) dbxout.c and protoize.c
(10) A/gcc.info-8, A/gcc.info-9, gcc.info-10, gcc.info-11, B/gcc.info-12, B/gcc.info-13, and B/gcc.info-14

Table 4.2 shows the sizes of these 10 clusters.

In each case it is possible using only the file names to imagine the types of commonalities. Cluster (1) clearly contains different versions of C grammars ready to be processed by the Bison parser generator or C source files that are generated by the parser generator. Clusters (6) and (10) are a collection of related documentation files.

Clusters (7) and (8) are configuration files that are similar as a result of software cloning.

	Files	Characters	Unique Characters
(1)	18	1 612 891	1 371 865
(2)	4	1 161 780	906 350
(3)	3	1 079 307	705 060
(4)	8	956 978	721 881
(5)	4	698 261	532 288
(6)	13	648 043	518 658
(7)	17	545 952	316 502
(8)	4	450 585	250 075
(9)	4	443 487	252 874
(10)	9	411 383	335 151

Table 4.2 Exact Matching Cluster Breakout: 10 largest Complex Clusters

4.2 Approximate matches of 50 lines or more

The exact matches of the previous subsection required the exact matching of 50 consecutive lines to be detected at all. An attempt was made to find more matches by performing the following transformations on the text before 50 line matches were done:

- All white space characters other than line separators were deleted.
- Each maximal sequence of alphanumeric characters was replaced by the single letter 'i'.

This would have the effect of changing the line " for(k = 1; k <= n; k++) {" by the line "i(i=i;i<=i;i++){" and the line "#define XDEF 234" by "#iii".

This is a rather aggressive loss of information, but keeping the requirement for a 50-line match ensured that there was no explosion of match information. Basically, 135 files moved from simpler clusters to more complex ones and a few of the complex clusters merged. After this process, there were 922 clusters of which 42 were complex. The breakout is summarized in Table 4.3.

	Clusters	Files	Characters	Unique Characters
Cplx	42	236	3 292 792	2 092 223
ABX	111	222	2 081 064	1 430 791
AB=	190	380	5 183 447	3 242 684
AA	14	28	13 198	8 265
BB	9	18	2 020	1 010
A	206	206	127 055	127 055
B	350	350	526 219	526 219
Total	922	1440	11 225 795	7 428 247

Table 4.3 Approximate Matching Cluster Breakout

A brief examination of the new matches indicates that most seem reasonable based on the file names. For example, 'A/cphash.h' and 'B/cphash.h' now have 614 characters matching out of 2103 and 2209 respectively, a 28% match. These are generated by an application 'gperf' from the file 'gplus.gperf,' which is almost identical in the two directories. The outputs are quite different in that the lines are in a different order but the layout is similar.

4.3 Summary of results

The tool identified very clearly the changes between release A and release B and pointed out a number of associations of files within releases that seem to be worth following up on. In addition, very few nonsense associations were seen. Further study of the detailed information will no doubt provide additional insight.

5 Conclusions and current directions

There are many clues in a legacy source that help one understand its structure. Sifting through these clues in an effective way requires tools, but with the right tools the "gold" can be found.

The approach of exact matching of sequences of lines appears to provide much information with comparatively little noise. A preliminary analysis reveals that a good understanding of the high-level structure of two releases of a 20 megabyte source directory can be achieved in a relatively automated fashion.

Work is now focusing on elaborating the technique for understanding the wealth of information that is produced by this process. For example, the ideas of software visualization would be quite useful for understanding complex clusters. Aggregating matches at a level less than a file is also important for some types of applications.

References

[1] B. S. Baker, "A Program for Identifying Duplicated Code", Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, (1992).

[2] T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse", *Computer* **22**(7), pp. 36–49, (July 1989).

[3] E. Buss, R. De Mori, M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. Tilley, J. Troster, and K. Wong, "Investigating Reverse Engineering Technologies: The CAS

Program Understanding Project", *IBM Syst. Journal,* (In press).

[4] E. Buss and J. Henshaw, "Experiences in Program Understanding", *Proceedings of the 1992 CAS Conference,* pp. 157–189 (November 9–12, 1992).

[5] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software* **7**(1), pp. 13–17 (January 1990).

[6] J. H. Johnson, "Identifying Redundancy in Source Code using Fingerprints", *Proceedings of the 1993 CAS Conference,* pp. 171–183 (October 24–28, 1993).

[7] R. M. Karp, "Combinatorics, Complexity, and Randomness", *Communications of the ACM* **29**(2), pp. 98–109, (February 1986).

[8] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms", *IBM J. Res. Develop.* **31**(2), pp. 249–260, (March 1987).