On Finding Duplication and Near-Duplication in Large Software Systems

Brenda S. Baker
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
bsb@research.att.com

Abstract

*This paper describes how a program called **dup** can be used to locate instances of duplication or near-duplication in a software system. **Dup** reports both textually identical sections of code and sections that are the same textually except for systematic substitution of one set of variable names and constants for another. Further processing locates longer sections of code that are the same except for other small modifications. Experimental results from running **dup** on millions of lines from two large software systems show **dup** to be both effective at locating duplication and fast. Applications could include identifying sections of code that should be replaced by procedures, elimination of duplication during reengineering of the system, redocumentation to include references to copies, and debugging.*

1 Introduction.

This paper focuses on locating duplication or near-duplication in a large software system as an aid in maintenance and reengineering. Duplication can become a problem within large software systems if programmers make modifications by copying and modifying sections of code. It has long been known that copying can make the code larger, more complex, and more difficult to maintain. In particular, when a bug has been found in one copy, a bug fix may be made in the copy where the bug was found, but not in the other copies. Nevertheless, copying and modifying code may occur for several reasons. First, making a copy and modifying it may be simpler than more major revisions and therefore less likely to introduce new bugs immediately, especially when the programmer making the bug fixes is not the one who wrote the original code. Second, if multiple versions are created, the interactions between the versions may become intractable as the versions grow apart over time, and eventually it may seem simpler to maintain some of the code separately. Third, process management may encourage duplication, *e.g.* if evaluation of programmers' performance is based in part on how much new code they write, so that programmers have little incentive to rewrite old code. Fourth, copies may be required because of the need to avoid the overhead of a procedure call for efficiency considerations.

This paper addresses the problem of locating exact or near-duplication of code that was created by copying and modifying code with an editor. When code is copied and modified via an editor, the types of changes made may include insertions and deletions of lines, modifications within lines, and global substitutions. The goal is to find copies that are substantially the same line by line except for global substitutions, so that one copy is a variant of the other, rather than sections of code that have evolved to be mostly different. In software reuse terminology, the problem is to locate instances of ad-hoc black-box or white-box software reuse [16] within a software system. Thus, this is a problem in reverse engineering. Moreover, the systems to be examined may be legacy systems running to millions of lines of code.

The approach of this paper is to find maximal sections of code over a threshold length that are either exactly the same, or the same except for a global substitution of names of parameters such as variables and constants, *e.g.* all occurrences of x changed to y and all occurrences of pchar changed to pc. In the former case, we call the two sections of code an *exact match*, and in the latter case, a *parameterized match* (*p-match*). Thus, the approach is text-based and line-based. Comments and white space are ignored. The tool to find maximal exact or parameterized matches is a program called *dup*. To find longer sections of code that were copied and then changed locally in the middle, the exact or parameterized matches can be further analyzed to locate pairs or sequences of matches that match sections of code separated by small gaps; alternatively, such regions can be found by examining scatter plots.

An example of a p-match is given in Figure 1, which contains two code fragments taken from the X Window System [18] source code. The fragments are identical except for the differing indentation (which is ignored by **dup**) and the correspondence between the variable names pfi/pfh and the pairs of structure member names lbearing/left and rbearing/right. These fragments are excerpted from two 34-line sections of code that are a p-match with these parameter correspondences.

Fragment 1:

```
copy_number(&pmin, &pmax,
   pfi->min_bounds.lbearing,
   pfi->max_bounds.lbearing);
*pmin++ = *pmax++ = ',';
copy_number(&pmin, &pmax,
   pfi->min_bounds.rbearing,
   pfi->max_bounds.rbearing);
*pmin++ = *pmax++ = ',';
```

Fragment 2:

```
copy_number(&pmin, &pmax,
      pfh->min_bounds.left,
      pfh->max_bounds.left);
*pmin++ = *pmax++ = ',';
copy_number(&pmin, &pmax,
      pfh->min_bounds.right,
      pfh->max_bounds.right);
*pmin++ = *pmax++ = ',';
```

Figure 1: Two fragments of code from source for the X Window System.

In addition to finding possibly distant sections of code that match, **dup** finds locally repetitive sections of code where the same short section is repeated immediately with different parameters, typically with names ending in a number; if an array were used instead of the numbered parameters, the repetitive code could be replaced by a loop. Such sections could have been generated automatically by a program generator, but instances have been found that were created by hand from a specification for which the specification language lacked arrays within structures.

For programmers, **dup** describes the matching sections of code and the correspondence between the parameter names in the two sections. If the programmer wants to turn the multiple copies of the code into calls to a new procedure, the correspondences between the parameter names in the two sections suggest what the formal parameters should be for the procedure. On the other hand, if it seems better to leave the duplication (*e.g.* to avoid the overhead of a procedure call or the time for rewriting), a profile can be generated that shows for each line of code where other copies occur in the system, based on the maximal exact or parameterized matches, so that when a bug occurs in one copy of some code, the programmer can fix it in the other copies as well. Comments about the location of other copies of code could also be added to redocument the code.

For managers, the postprocessor computes how much duplication is present in the system, estimates how much code could be saved if the duplication were eliminated, and computes which files or pairs of files contain the most duplication. This information provides a new measure of software quality and if the system is reengineered, the information could guide in eliminating the duplication. In the case of repetitive code, the information from **dup** identifies code that could be rewritten using arrays and loops. For visualization, a scatter plot of the output makes apparent which sections of code contain large amounts of duplication, which sections of code are similar except for small gaps, and whether duplication is local or distant.

Dup and the postprocessor have been applied to millions of lines of code from two large software systems. In the complete source of the X Window System (minus some tables), including 714479 lines of code, **dup** located 2487 matches of at least 30 lines and these matches involved 19% of the code; **dup** estimated that 12% of the input was duplication that could be eliminated by rewriting. These matches can be divided into 976 groups, each of which apparently represents an instance of copying and editing of code. **Dup** has also been run on subsystems of a 10-million line production system. For a production subsystem with 1.1M lines, the 5550 parameterized matches of length at least 30 lines included 20% of the code; **dup** estimated that 13% of the subsystem was duplication that could be eliminated by rewriting. These matches can be divided into 2180 groups, each apparently representing an instance of copying and editing of code. Some interesting anomalies have been found in this production system via **dup**. These have included unusually complex files, an obsolete file, and a place where a bug fix was apparently applied to one copy of some code but not to another other copy. Two whole directories of 800 lines were found to be the same except for a systematic change of parameter names and a line break. One subsystem contained two 40-line procedures for date calculations that were identical except that one used shorter identifiers than the other did.

In dealing with large systems of millions of lines of code, it is essential for a tool to use efficient techniques to attain a reasonable processing speed. **Dup** runs very fast; using one R3000 40MHZ processor, it can process a million lines of code in seven minutes. The speed comes partly from the choice to make it a text-based, line-based tool and partly from efficient algorithms based on a new data structure, called a parameterized suffix tree [2, 3]. **Dup** and the postprocessor are implemented in about 2300 non-commentary lines of C and Lex [11] and run under UNIX.

Experiments on several million lines of production code suggest that in practice, for thresholds of more than about fifteen lines, the running time of **dup** on C code (excluding tables) is linear in input size, although it could be quadratic in the worst case. (On tables, depending on the values of the data, the number of matches to be reported might be quadratic in table size. Locally repetitive code can also lead locally to a quadratic amount of output but this has not been found to be a dominant effect over a whole system.)

Overall, the data show that production systems can contain a large amount of duplication that was apparently created by copying and editing code. The concept of maximal p-matches appears to be more useful than just exact matches in locating such duplication. **Dup** runs fast enough to be useful for systems with millions of lines of code. Finally, it appears that the duplication information should be useful in practice for finding previously unknown features of the code and for maintenance and reengineering of large systems.

Other researchers have taken different approaches to finding commonality in code. These approaches have included finding common style or complexity measures [5, 8, 14, 12], common parse trees [10], common data flow [1, 7], fingerprints for files [9, 13], the UNIX **diff** command [11], data compression [17, 19], and graphical user interfaces (GUIs) [6]. These methods have been deficient for various reasons. Approaches based on common style or complexity characteristics have no guarantees about exactly how the code is related. The parse tree method used exhaustive search and was slow [10]. The data flow methods have only been applied to toy programming languages. The fingerprint approaches were aimed at finding similar files rather than copies of parts of the files. **Diff** and other approaches based on edit distance can take quadratic time, are only designed for comparing whole files, and are too slow for millions of lines of code. Data compression methods find some cases of exact duplication but not all maximal matches and certainly not parameterized matches or local editing changes. Church and Helfman's GUI, **Dotplot**, requires that the user pick out patterns of similarity by eye, and the patterns are often dominated by repetitive code structure.

Section 2 describes how the definition of maximal parameterized matches in code leads to the design of a useful tool for finding duplication. Section 3 describes the data structure used in **dup**. Section 4 discusses the results of applying **dup** to two software systems. The last section contains further discussion and directions for further work.

2 Exact and parameterized matches and the design of dup.

The basic tool in identifying duplication in software is the program **dup** for finding maximal exact or parameterized matches over a threshold length specified by the user. A postprocessor analyzes the matches further. Currently, **dup** processes code written in C, but front ends could be easily written for other input languages. This section defines maximal exact and parameterized matches and how these definitions are adapted in **dup** to the task of finding interesting duplication or near-duplication in code.

Two sections of code are said to be a *maximal exact match* if their lines match exactly character by character but the preceding lines do not match and the following lines do not match. (White space and comments are ignored.)

A scatter plot helps to visualize maximal exact matches. Figure 2 shows a scatter plot of exact matches in a production system file of 2846 lines, or 1761 lines after pruning white space and comments, with a minimum match length of 15 lines. Each (approximately) diagonal line from (n_1, n_2) to (n_3, n_4) represents a match between lines n_1 to n_3 and n_2 to n_4; the lines are not strictly diagonal because the white space and comments have been ignored, while the line numbers are the original line numbers in the file. Only the part of the plot below the main diagonal is shown, so that each match corresponds to exactly one line segment. The full plot would be symmetric around the main diagonal and contain two line segments for each match. In this case, there are 18 exact matches involving 419 lines, or 24% of the file.

Two sections of code are a *parameterized match* (*p-match*) if there is a one-to-one function that maps the set of parameters in one section onto the set of param-

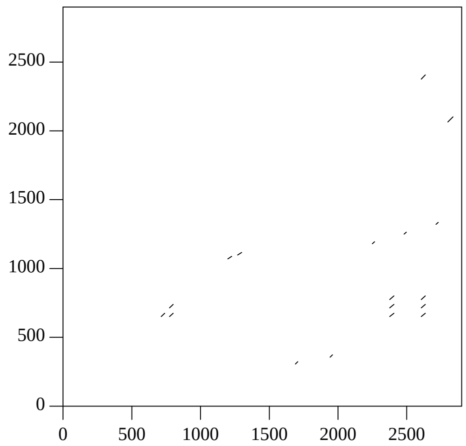

Figure 2: Exact matches for a C file.

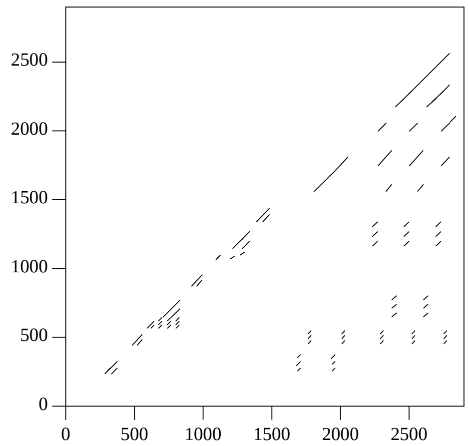

Figure 3: P-matches for the same file as Figure 2.

eters in the second section, such that the text of the first section is transformed into the text of the second by textually substituting $f(p)$ for p everywhere that p occurs in the first section. (Comments and white space are ignored.) For example, in the code of Figure 1, the one-to-one function maps lbearing into left, rbearing into right, and pfi into pfh, but is the identity on other parameter candidates such as copy_number and pmin. Parameters in **dup** are currently defined to include identifiers, constants, field names of structures, and macro names. Keywords such as "while" or "if" are not candidates for parameters.

Two sections of code are a *maximal p-match* if they are a p-match and the p-match cannot be extended to the preceding lines or the following lines.

Figure 3 shows a scatter plot of the maximal p-matches for the same file whose exact matches are plotted in Figure 2. With a threshold of 15 lines, there are 87 maximal parameterized matches involving 85% of the file, compared to 18 exact matches involving 24% of the file. The maximal parameterized match found is 182 lines, compared to 37 lines for the exact matches.

Sections of code that are a p-match generally look related. In certain circumstances, such as sequences of lines consisting of C "case variable:" statements, matches are found between sections of code that don't appear to be related in that arbitrary variable names are paired line after line. Experiments have shown that an effective way of avoiding such output is to report only p-matches where the number of non-identical parameter pairs is at most half the number of non-commentary lines in the match; more generally, this could be turned into a percentage to be set by the user.

The quality of the output is also improved by pruning off closing braces at the start of a match. Because of the definition of maximality and the frequency of lines containing just a closing brace, maximal matches often begin with one or more closing braces, but the closing braces usually belong to code preceding the interesting part of the match.

Input code can be provided to **dup** either via the standard input or via a list of file names. In the latter case, **dup** does not allow matches to cross file boundaries. It does, however, allow matches to cross procedure boundaries, so that whole files can be found to match. An option to restrict matches from crossing procedure boundaries may be added in the future.

A postprocessor analyzes the p-matches and generates statistics and plots. A number of kinds of output are available from the postprocessor.

For each p-match, the program outputs the number of matching non-commentary lines, the pairs of matching intervals, and a list of the nonidentical parameter correspondences for each p-match. Figure 4 gives an example from the X Window System [18]; the match is the one from which the fragments of Figure 1 were extracted. The intervals are described as a file number, path name, and range of line numbers. (The file number is useful visually when path names are long and differ by only a character or two.) The match length is specified by "34 ncsl", which means "34 non-commentary source lines", *i.e.* the number of lines in the match excluding comments and blank lines.

The postprocessor calculates summary information including the number of matches, number of non-

```
34 ncsl
1552,mit/clients/xlsfonts/xlsfonts.c:274,309
327,mit/fonts/clients/fslsfonts/fslsfonts.c:384,419
3 parameters
1: pfi, pfh
2: lbearing, left
3: rbearing, right
```

Figure 4: Output for the parameterized match for which Figure 1 is an excerpt.

commentary lines in the whole system involved in the matches, percentage of non-commentary lines in the system involved in the matches, and distribution of match lengths. These calculations are straightforward.

The postprocessor computes an estimate of the percentage of lines that could be eliminated if the code were rewritten using alternative methods such as procedures instead of copying. The estimate is derived using the simple assumption that if the same line appears in k sufficiently long matching sections of code, then $k-1$ of these occurrences could have been avoided. For example, for the file whose p-matches are plotted in Figure 3, the postprocessor estimates a potential shrinkage of 61% if the code were rewritten to avoid parameterized duplication. The computation of the estimate is complicated by matches that pair up the same lines of code because they overlap in both intervals. For example, it would be possible for lines 30-60 and 130-160 to be a maximal p-match and for lines 40-70 and 140-170 to be another maximal p-match, where a longer p-match is not possible because a correspondence of x and y in lines 39 and 139 conflicts with a correspondence of x and z in lines 61 and 161. In this example, both p-matches match lines 40-60 with lines 140-160. The calculations of redundancy handle this situation by counting this as one extra copy of each of the lines in these ranges, rather than two. Such situations are caused by conflicting pairings of values, often pairings of small integer constants (especially zero) that may be used as values for more than one variable in one section of code but not the other.

As an option, the postprocessor prints out a profile of the code showing how much duplication occurs where. In particular, it identifies intervals (sequences of lines) in the input that are involved in exactly the same set of matches. For each such sequence of lines, it prints out the range of line numbers, the number of distinct matches, and a list of the match numbers. In our above example, lines 30-60 and 130-160 were a p-match and lines 40-70 and 140-170 were a p-match, and both p-matches match lines 40-60 with lines 140-160. In this situation, the postprocessor will identify intervals 30-39, 40-60, 61-70, 130-139, 140-160, and 161-170 as sequences of line numbers within which the lines are involved in the same matches. However, it does not count the two p-matches as distinct matches for the intervals 40-60 and 140-160 in which they overlap, since they pair up the same lines.

Since a system can contain thousands of files, and the duplication may be unevenly distributed among them, another postprocessor option is to calculate the percentage redundancy and number of redundant lines within each file and between each pair of files in the input. For efficiency, these calculations are done by intervals participating in the same matches, as defined in the preceding paragraph, rather than by individual lines. Sorting can be used to identify the files or file pairs with the most duplication.

Further processing of matches can be done to group matches that appear to be related, in the sense that together they represent a region of code that was copied and then edited. Two classes of these matches arise as follows.

First, there is the case described above of two matches that would be one match if not for a parameter conflict in the middle of the code. This is detected by overlaps in both intervals and identical distances between the first and second intervals in the two matches. Pairs or sequences of successive p-matches with this relationship can be detected and labeled as part of a longer match with a conflict in parameters.

Second, if some code was copied and then modified in the middle, what would be detected by **dup** would be a pair of matches pairing up sections of code that are close together but not overlapping, *e.g.* one match pairing up lines 30-50 and 500-520, and another match pairing up lines 55-75 and lines 530-550. Such pairs (or more) of matches can be identified by sorting the matches by endpoint and looking for pairs of matches whose intervals are within δ lines of each other, for a δ specified by the user. They can be grouped and labeled as apparently due to copying and modifying a section of code.

Finally, matches that arise from a locally repetitive region of code can be identified, grouped, and labeled. A repetitive region of code contains short code segments that are immediately repeated with a change of parameter names. Such repetitions may be generated automatically by program generators, but they may also be generated by hand as a result of system conventions. An example is code generated by hand from a specification language that does not permit fields in structures to include arrays. Repetitions can result locally in a number of matches quadratic in the number of repetitions, and similarly for matches between two areas of repetitive code. Thus, it is especially helpful to group these matches for the user.

To see the kinds of maximal matches to expect in a repetitive region, consider the exact duplication case of four successive occurrences of lines abc, where a, b, and c represent lines of code. With a threshold of one line, **dup** will report three matches: one of length 9 between positions 1-9 and 4-12 representing an overlapping match of $abcabcabc$ with itself, one of length 6 between positions 1-6 and 7-12 representing a match of $abcabc$ with itself, and one of length 3 between positions 1-3 and 9-12 representing a match between the first copy of abc and the last copy of abc. This is in contrast to the natural factorization of $(abc)^4$ that would be desirable.

From overlapping as in the $abcabcabc$ of the first match in the last example, **dup** deduces that the region contains repetitive code, but it does not as yet have an algorithm to replace the multiple matches by an $(abc)^4$-type factorization. However, the matches within a repetitive region or between two repetitive regions can be grouped and labeled as matches in a repetitive region.

3 How dup finds maximal p-matches fast.

Dup finds p-matches by means of an efficient algorithm based on a new data structure called the *parameterized suffix tree*. This section gives an overview of the algorithm; details and proofs of correctness are in [2, 3].

For each line of input, the lexical analyzer generates a string consisting of one "non-parameter symbol" and zero or more "parameter symbols". (Non-parameter symbols and parameter symbols are represented by disjoint sets of integers.) A line such as $x = x + y$ is first transformed into $P = P + P$ and a list x, x, y; then a non-parameter symbol is generated to represent the $P = P + P$ and three parameter symbols are generated to represent x, x, y. In this manner, both the parameter candidates and their positions are recorded in the resulting string.

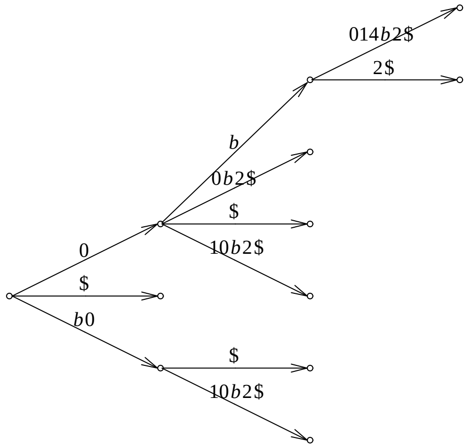

Figure 5: A p-suffix tree for the p-string $S = xbyyxbx\$$.

The input to the parameterized matching algorithm is the string resulting from concatenating the individual strings of non-parameter and parameter symbols from all the lines. Such a string is called a *parameterized string* or *p-string*.

The following encoding of p-strings underlies our algorithm. The first occurrence of each parameter symbol is replaced by a 0. Each later occurrence of a parameter symbol is replaced by the distance in the string since the previous occurrence of the same parameter symbol. Non-parameter symbols are left unchanged. For example, if a and b represent non-parameter symbols, and x and y represent parameter symbols, a p-string $axybxay$ would be encoded as $a00b3a4$. If S is a p-string, the resulting encoding of S is called $prev(S)$.

The significance of this encoding is that a p-string S is a p-match for a p-string T if and only if $prev(S) = prev(T)$. For example, the p-string $T = auvbuav$, with parameter symbols u and v, has the same encoding $prev(T) = a00b3a4$ as the p-string S of the preceding paragraph, and T and S are a p-match. Therefore, the encoding can be used to test for p-matches.

After the lexical analysis, **dup** builds a data structure called a *parameterized suffix tree* (*p-suffix tree*) for the p-string $S = b_1b_2...b_n$ representing the input. Call $S_i = b_ib_{i+1}...b_n$ the ith *suffix* of S. A p-suffix tree is a generalization of the suffix tree data structure [15]. Whereas a suffix tree is a compacted trie containing the suffixes S_i of a string S, $1 <= i <= n$, a p-suffix tree for S is a compacted trie containing $prev(S_i)$ for $1 <= i <= n$. Because it is a trie, the labels on arcs to the children of a node all begin with distinct symbols; the trie is compacted in the sense that labels are allowed to be strings and each non-leaf has at least two children. Each leaf in the p-suffix tree represents $prev(S_i)$ for some i, $1 <= i <= n$, in the sense that concatenating the labels of the arcs on the path from the root to the leaf yields $prev(S_i)$. The p-suffix tree can be represented in space linear in input size provided that at each access, the value of a symbol of $prev(S_i)$ is calculated dynamically from its value in $prev(S)$ and the access depth in the tree.

Example. Let $S = xbyyxbx\$$, where b and \$ are non-parameter symbols and x and y are parameter symbols. The p-suffixes to be encoded in the tree are $0b014b2\$$, $b010b2\$$, $010b2\$$, $00b2\$$, $0b2\$$, $b0\$$, $0\$$, and \$. Notice that the parameter pointers change to 0 as the preceding part of the string is shortened. The p-suffix tree for S is shown in Figure 5.

The algorithms to construct a p-suffix tree and recurse over it to find the parameterized duplication are complicated; for a detailed description, proof of correctness, and analysis of worst-case performance, see [2, 3]. In practice, the overall running time of **dup** has been found experimentally to be linear in input length, although in a worst-case scenario it could be quadratic.

4 Experiments on two production systems

The experiments described in this section were run on the source for the X Window System [18] and a subsystem we will call SS from a production system. In each case, first **dup** found all maximal p-matches over a threshold length and the postprocessor calculated statistics about the amount of duplication. Then the matches were grouped as described in Section 2, *i.e.* two matches were placed in the same group if either (1) the two matches overlap and pair the same lines, but were separate matches because of a conflict of parameters, or (2) the two matches do not overlap, and the first intervals are within 20 lines of each other and the second intervals are within 20 lines of each other, suggesting that the entire region was copied and then edited in the middle. Finally, the overlapping matches indicating repetitive regions were counted and the number of repetitive regions was computed.

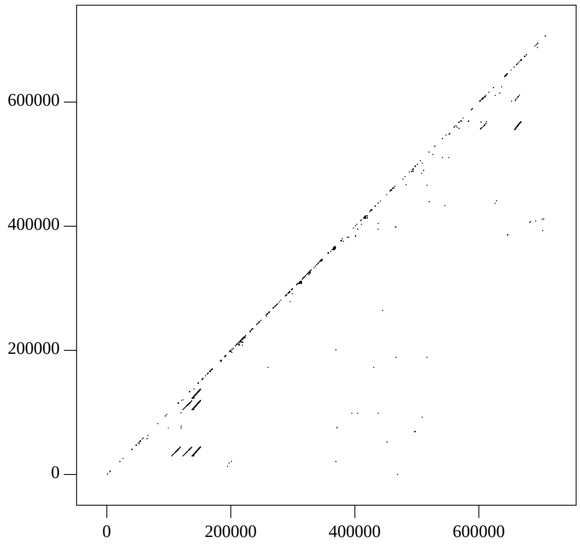

Figure 6: Parameterized matches for X source (minus some tables)

4.1 Extent of Duplication

The parameterized matches include a substantial amount of the code in each case.

In the complete source of the X Window System (minus 13 files of tables), including 714479 lines of code (514579 lines after removing white space and comments), **dup** located 2487 matches of at least 30 lines; these included 19% of the code, and the program estimated that 12% of the program was duplication that could be eliminated by rewriting. The matches are plotted in Figure 6. The longest match was 2585 non-commentary lines and 60 matches had at least 200 non-commentary lines. Grouping the 2487 matches resulted in 976 groups. There were 463 overlapping matches indicating repetitive regions of code and these were distributed over 148 repetitive regions.

In SS, which has almost 1.1M lines, or 605k lines after removing white space and comments, **dup** found 5550 matches of length at least 30 lines and these included 20% of the code[1]. **Dup** estimated that 13% of the program was duplication that could be eliminated by rewriting. The matches are plotted in Figure 7. The longest match was 553 non-commentary lines; there were 51 matches of at least 200 non-commentary lines. Grouping the 5550 matches resulted in 2180 groups. There are 775 overlapping matches indicating repetitive code, and these were distributed among 337

[1]This subsystem did not include machine-generated code, which might be expected to have large amounts of duplication.

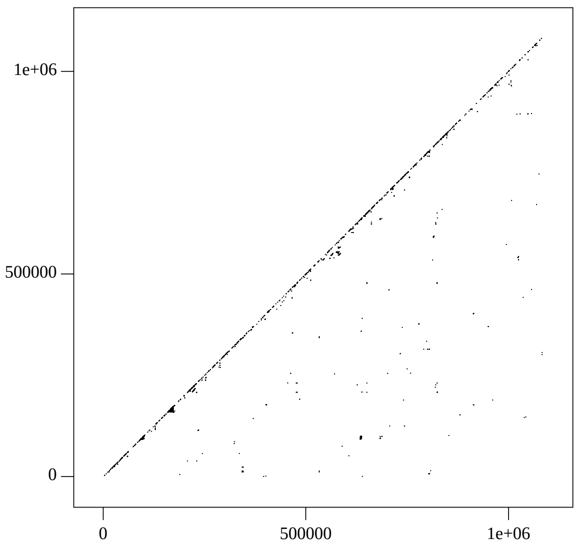

Figure 7: Parameterized matches for some code from a production system.

regions of repetitive code.

The following conclusions may be drawn from the above data. The systems contained a substantial amount of duplication. Grouping matches into regions that were apparently copied by an editor and modified in the middle made a significant reduction in the number of separate pieces of information to be examined by the user. There were repetitive regions in the code, but the number of matches within each region was small on average. Overall, the approach of viewing the duplication problem as that of finding code that was copied by an editor and modified, rather than just as finding p-matches, appears justified by the grouping and repetitive code data.

4.2 Running times

Dup is quite fast. Processing the almost 1.1M lines of the subsystem SS with a threshold of 30 lines took 7.2 minutes, with an additional time of 7 seconds for the postprocessor. These times represent cpu time on one 40MHz R3000 processor (primary I and D cache 64KB, secondary 1MB, main 256MB, SGI IRIX 4.1).

4.3 Effect of Threshold Lengths

The amount of duplication reported increases dramatically as the match lengths become small. If a very small threshold is used, say less than 8 lines, half of the lines in a match may be closing braces, which have little information value. When the match length is reduced to one line, or even several lines, the number of matches generated grows quadratically in input length due to lines consisting of closing braces, and the amount of output becomes huge and mostly meaningless. For this reason, threshold lengths of 20 or above appear to be most meaningful. (An alternative would be to filter out closing braces, in which case smaller thresholds might be reasonable.)

When individual matches of at least 20 lines are examined, the matches for C code usually look reasonable. In particular, the one-to-one correspondence between parameters usually pairs similar tokens, *i.e.* small integers with small integers, strings with strings, or variables with other variables with related names. (Recall that the parameterization algorithm treats all parameter candidates the same, so that in principle it could pair an integer and a string, or structure member names with a floating point number.) Figure 4 gives an example of match of 34 non-commentary lines from the X source, with the pairings pfi/pfh, lbearing/left, rbearing/right. In SS, the following pairs were found in a match that turned out to be two 40-line procedures for converting a date to a Julian date: AMydays/AMyeardays, AMmdays/AMmodays. Other examples from SS are a match of 31 non-commentary lines with the pairings b2/b1, 0x9000/9x8000, and “bfcp-i3”/“bfcp-i2”, and 31 non-commentary lines with the pairings 0/1, “u46”/“i11”, RCERR14/RCERR10, and “u47”/“i12”.

In practice, many of the parameters found are related to error checking and handling, in both of the systems that were studied. An example from SS is a match of 50 non-commentary lines with five pairs of parameters, of which only the first pair is not involved in error-handling. The remainder are either error code numbers or unique strings representing the current point of execution for defensive programming in case of a fault.

4.4 Visualization and anomalies

Plots of large amounts of code as in Figures 6 and 7 should be useful to managers in visualizing the complexity and interrelationships of a whole software system. The scatter plots of Figures 6 and 7 appear dense near the main diagonal and sparse elsewhere. No line segments are plotted exactly on the main diagonal; at this scale, they merely appear to be on the main diagonal. In fact, the line segments near the main diagonal represent matches of code segments that are near each other in the system, *e.g.* in the same file or directory. The apparent density near the main diagonal is somewhat misleading because line segments are plotted with a minimum length, so that every match corresponds to a visible mark in the plot. Away from the main diagonal, the matches are quite scattered; these would be good candidates to investigate as to the origin of the duplication.

Three types of interesting features that have been found in casual browsing of scatter plots of smaller amounts of code have been unusually complex files, an obsolete file that had not been deleted, and a place where a bug fix was apparently applied to one copy of some code but not to another other copy. The obsolete file was found by noticing rather extensive duplication between two files in a module. Figure 8 shows a scatter plot with a gap between two collinear line segments representing two long matches. The gap corresponds to two sections of code, one of which has a loop that runs off the end of an array, and the other of which has a correct loop with a comment describing the correction; the latter apparently has a bug fix that was not applied to the other copy.

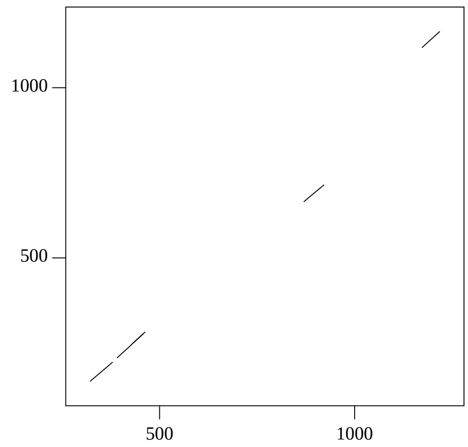

Figure 8: The plot for a file with an anomaly: a small gap between two matches.

5 Discussion and directions for further work.

Much duplication has been found using **dup**. In some cases it is very apparent that the copying was done by editor, as when two copies have the same level of indentation, but in one copy this is the wrong amount of indentation for the local control structure. (**Dup**, however, ignores the indentation in any case.) Of course, there is no easy way to measure how many instances are missed of code sections that would be considered duplication by a person but don't get picked up by the maximal p-match search done by **dup**.

Dup's estimate of the percentage by which the code could be shrunk is optimistic in predicting that any number of copies of a section of code could be replaced by one copy the same size, since the copies are not generally complete semantic units that could be directly turned into procedures and in some cases, there may be valid reasons for separate copies, such as efficiency.

An obvious question is whether the output from **dup** could be used to generate procedures with parameters automatically from the input to reduce the size of the code. Regretfully, the answer appears to be negative. Since the code segments identified in matches usually do not correspond exactly to subtrees in the parse tree for the program, or to any obvious semantic unit, a programmer would need to rewrite the code by hand. However, the output from **dup** would be helpful in determining the scope of the procedures and the formal parameters.

A problem that has been only partially solved by **dup** is how to factor duplication into repetitions of short blocks of code, as in factoring $abcabcabc$ into $(abc)^3$, where a, b, and c represent distinct lines of code. (In real code, the copies of abc would have parameter changes and not be exactly the same.) **Dup** can identify repetitive regions of code from overlapping matches, but more work needs to be done on algorithms to find the $(abc)^3$ factorization.

Another challenge is how to aid a programmer in examining the large amount of output generated from a large software system. A graphical user interface would probably be useful in this regard.

Another useful program would be one that searches code for a parameterized match to a particular code segment. A programmer making a bug fix could then search for other related sections of code that might also require the same bug fix, without having to obtain all maximal matches for the whole body of code via **dup**. The author has developed algorithms to do this by a generalization of a Boyer-Moore-type algorithm for parameterized pattern matching [4].

Finally, if sections of code are reused informally by copying, they may be good candidates for more formal reuse via a library. This would be especially true if they have been copied from one system to another. It would be interesting to compare different systems to see if any procedures are found in common.

Acknowledgments

The author would like to thank Brian Kernighan for calling her attention to the code duplication problem and for suggesting part of the encoding done by the lexical analyzer. She would also like to thank William Chang for providing his code for suffix tree construction, which she modified to generate p-suffix trees. She is appreciative of many useful discussions with

Ken Church, Ken Clarkson, Bryan Ewbank, Raffaele Giancarlo, Eric Grosse, Jon Helfman, Andrew Hume, Brian Kernighan, and Eric Sumner, Jr..

References

[1] B. Alpern, M. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *15th ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[2] Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications (extended abstract). In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 71–80, May 1993.

[3] Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *J. Comput. Syst. Sci.*, 1995. To appear.

[4] Brenda S. Baker. Parameterized pattern matching via boyer-moore-type algorithms. In *Proc. Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 541–550, January 1995.

[5] H.L. Berghel and D.L. Sallach. Measurements of program similarity in identical task environments. *SIGPLAN Notices*, 9(8):65–76, August 1984.

[6] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, June 1993.

[7] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, June 1990.

[8] H.T. Jankowitz. Detecting plagiarism in student PASCAL programs. *Computer Journal*, 31(1):1–8, 1988.

[9] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proc. International Conf. on Software Maintenance*, pages 120–126, 1994.

[10] Ralph Johnson, October 1991. personal communication.

[11] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[12] L. Malmi, M. Henrichson, T. Karras, J. Saarhelo, and S. Saerkilahti. Detecting plagiarism in Pascal and C programs. Technical report, Helsinki Univ. of Technology, Espoo (Finland). Lab. of Information Processing Science, 1992.

[13] Udi Manber. Finding similar files in a large file system. In *Proc. 1994 Winter Usenix Technical Conference*, pages 1–10, Jan 1994.

[14] T.J. McCabe. Reverse engineering, reusability, redundancy: the connection. *American Programmer*, 3(10):8–13, Oct. 1990.

[15] E.M. McCreight. A space-economical suffix-tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[16] Ruben Prieto-Diaz. Status report: Software reusability. *IEEE Software*, pages 61–66, May 1993.

[17] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithms for data compression via string matching. *J. ACM*, 28(1):16–24, 1981.

[18] R.W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[19] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23:337–343, 1977.