A Program for Identifying Duplicated Code

Brenda S. Baker
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes a program called **dup** that finds occurrences of duplicated or related code in large software systems. The motivation is that duplication may be introduced into a large system as modifications are made to add new features or to fix bugs; rather than rewrite working sections of code, programmers may copy and modify sections of code. Over time, proliferation of copies can make the code more complex and more difficult to maintain. **Dup** searches such code for all pairs of duplicated sections. The user may choose to search either for identical sections of code, or for sections that match except for substitution of one set of variable names and constants for another as if they were corresponding procedure parameters. Applications of **dup** could include visualization of the structural complexity of the whole system, identifying unusually complex files, identifying sections of code that should be replaced by procedures, and debugging.

Introduction.

This paper describes a new software tool, a program called **dup**, that finds occurrences of duplicated or related code in large software systems to aid in software maintenance and debugging. The program generates descriptions of the related code sections and statistics about the extent of duplication found. For visualizing the output, the output can be plotted in scatter plots and profiles can be produced to show how many times each line occurs in matching sections of code.

This work is part of the emerging field of *software visualization* [E], whose goal is to display characteristics of large software systems visually, as an aid in dealing with the complexity arising in systems of hundreds of thousands or millions of lines of code created by hundreds or thousands of programmers. Other examples of software visualization are the graphical user interfaces **seesoft** [E] and **dotplot** [CH]. **Seesoft** interactively displays data such as the age or programmer for each line of code; **dotplot** allows interactive manipulation of scatter plots to compare sections of code.

Dup was motivated by the observation that duplication may be introduced into a large system as modifications are made to add new features or to fix bugs. Rather than rewrite working sections of code, programmers may copy and modify sections of code. It has long been known that copying sections of code may 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 to the place where the bug was found, but not to the corresponding parts of other copies. Nevertheless, making a copy and modifying it may be much simpler than more major revisions and therefore less likely to introduce new bugs immediately, and hence copying code may seem preferable to changing a working section of code. This may especially be true when the programmer making the bug fixes is not the one who wrote the original code.

The premise underlying **dup** is that copying is most often accomplished by means of an editor. Therefore, the resulting copies will be largely the same line-for-line, or will be related in some systematic way such as a change of variables. White space and comments may be ignored since they do not affect the functionality of the code.

Given these assumptions, the approach taken in **dup** is line-based. Two lines of code are considered to be identical if they contain the same sequence of characters after removing comments and white space; the semantics of the program statements are not analyzed. Data structures are maintained with regard to lines rather than individual characters to reduce the space requirements.

The output of **dup** is a set of pairs of *longest matches* of sections of code. That is, two sections are a longest match if they match but the preceding lines do not match and the following lines do not match. To provide an example, we rephrase the definition of longest matches in terms of strings. Two identical substrings are a longest match if the preceding characters do not match and the following characters do not match. Thus, in the string *axyzbxyzc*, the *xyz*'s are a longest match, but the *xy*'s and the *yz*'s are not. Note that the longest match relation is not transitive. That is, if section A is a longest match for section B, and section B is a longest match for section C, it could be that section A is not a longest match for section C,

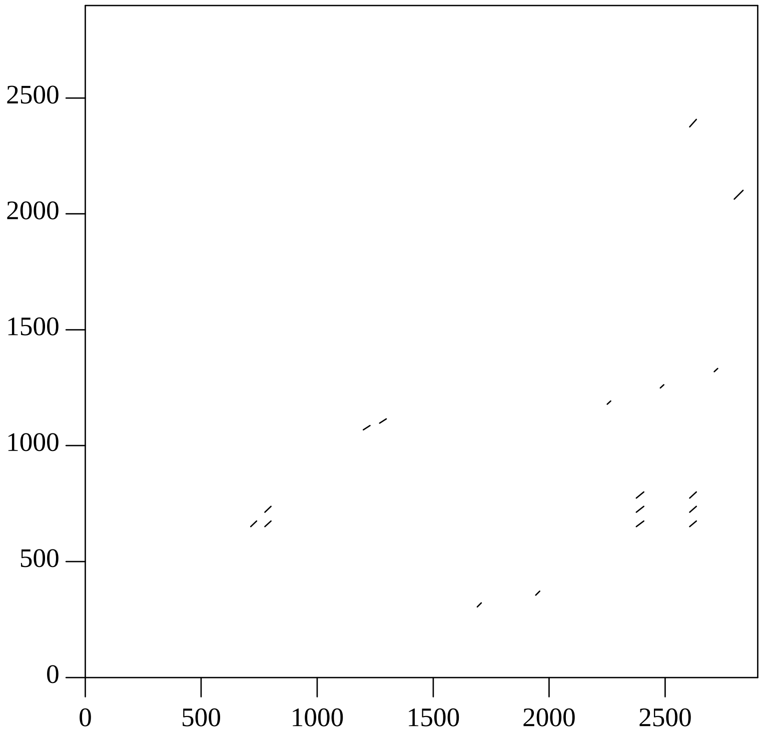

Figure 1. Exact matches for a C file.

because these sections are both contained in a longer match. In practice, when there are several related sections of code, the various pairs do differ in exactly how much matches. Therefore, the program reports the longest matches in pairs rather than looking for larger sets, and the scatter plots make evident that some of these matches overlap.

Since very short matches may not be interesting, the user may specify a minimum length match to report. Figure 1 shows a scatter plot produced by the program for a file of 2846 lines, or 1761 lines after pruning white space and comments, with a minimum match length of 15 lines. Only line segments below the main diagonal are plotted in this paper, because in plots of large amounts of code, most of the line segments are very close to the main diagonal, even though no line is matched with itself. Thus, each longest match is represented by exactly one roughly diagonal line in the plot; 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. In this case, the program found 18 matches involving 419 lines, or 24% of the file.

Rather than looking for just exact matches, the program can look for *parameterized* matches, where the code sections match except for a one-to-one correspondence between candidates for parameters such as variables, constants, macro names, and structure member names. (Keywords and operators are not candidates to be parameters.) For example, the following two code fragments taken (with some editing in order to fit) from the X Window [SG] source code are identical except for the correspondence between the variable names `pfi`/ `pfh` and the pairs of structure member names `lbearing`/ `left` and `rbearing`/ `right`.

```
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++ = ',';
```

```
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++ = ',';
```

For parameterized matches, matching sections are like expansions of the same macro with different parameters, for

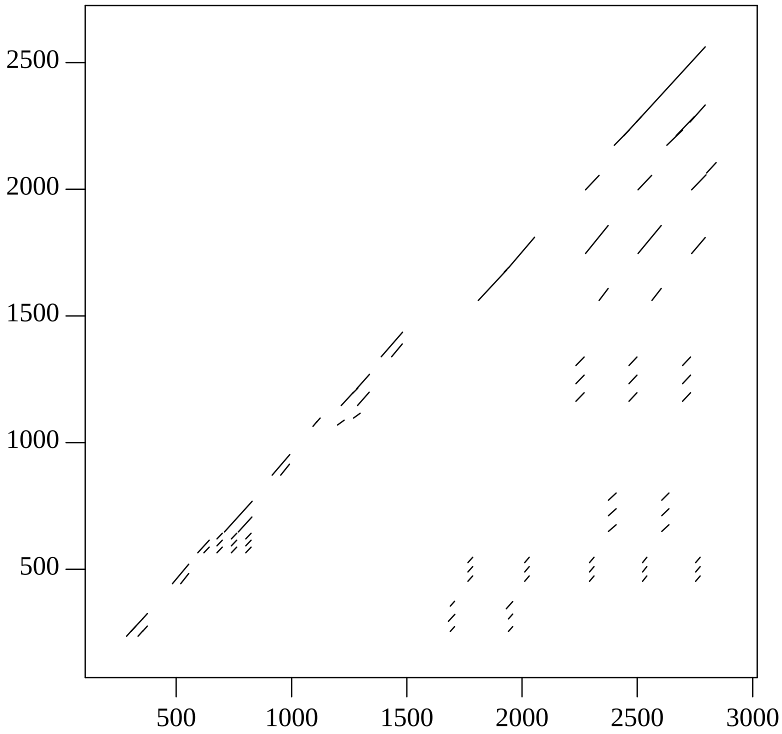

Fig. 2. Parameterized matches for the same file as Figure 1.

example, `f(p1,...,pn)` and `f(q1,...,qn)`. Only pairs of parameters that are not identical need be reported. When run on the original X source containing the above code fragments, the output produced by **dup** is the following. (The line numbers given are the original line numbers; "pruned" lines are the ones remaining after stripping off white space and comments.)

```
xlsfonts.c:274-309,
fslsfonts.c:384-419,
34 pruned lines match, with parameters:
1: pfi, pfh
2: lbearing, left
3: rbearing, right
```

Commonly, code will have more parameterized matches than exact matches. For example, Figure 2 shows the parameterized matches for the same file as Figure 1. In this case, the program finds 87 longest parameterized matches of at least 15 lines, involving 85% of the file. The longest match found is 182 lines, compared to 37 lines for the exact matches.

The program makes an estimate of how much more succinctly the code could have been written, if alternative programming methods such as procedures had been used instead of copying. The estimate is based on 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 the file of Figures 1 and 2, the program estimates that the code could have been shrunk by 14% based on exact matches, but 61% based on parameterized matches.

The program can also provide other aids such as a profile of the code showing how many copies of each line occurred in the matches found.

Other researchers have taken different approaches to finding common code. Programs aimed at detecting student plagiarism have typically used statistical comparisons of style characteristics such as the use of operators, use of special symbols, frequency of occurrences of references to variables, or the order in which procedures are referenced [H,Ja]. Johnson [Jo] has taken a parse-tree based approach to finding duplicated code. However, a line-based approach has two advantages compared to a tree-based approach. First, the line-based approach allows for use of data structures and efficient algorithms developed for string pattern matching, notably the suffix tree [McC]. Moreover, if the code contains macros, and some of these have, for example, unbalanced parentheses, it may be necessary to expand the macros to obtain a valid parse tree for the tree-based approach. However, the macros need not be expanded for a line-based approach; thus, there is no need to rematch expansions of the same macros, and the resulting output is more easily related to the original code.

Church and Helfman [CH] have combined signal processing techniques with a graphical user interface **dotplot** enabling easy manipulation of scatter plots to aid in visually scanning for similarities between sections of code. By running **dup** and **dotplot** on the same input and overlaying the plots, it was found that some duplication may be found by means of both programs, while each program finds features that are not found by the other. **Dup** finds parameterized matches not noticeable in **dotplot** and eliminates the noise that appears in **dotplot** plots, while patterns prominent in **dotplot** plots may reflect characteristics such as repetitive control flow structure rather than duplication in the sense of **dup**.

Other related work has been done in the areas of file comparison, genome sequencing, and data compression. The UNIX **diff** program for file comparison [KP] and algorithms for genome sequencing [SK,LMW] have been based on finding a best match (*longest common subsequence* or *smallest edit distance*) between two sets of data, from start to end, whereas **dup** looks for pairs of shorter related sections that could appear in any order. Data compression algorithms such as [RPE,ZL] have been based on finding copies of substrings, but for data compression, one copy is sufficient, as opposed to this work, in which the goal is to find all sufficiently long copies.

Exact Matching

This section describes how **dup** finds the set of pairs that are longest exact matches.

First, a lexical analysis phase hashes the lines in order to assign each line an identifying integer such that two lines are the same if and only if their ids are the same. The output from the lexical analysis phase can be considered a string of symbols over an alphabet of integers.

Given this string, a brute force method of finding longest matches would be to do the equivalent of scanning along diagonals of a scatter plot for longest matches, using time $O(n^2)$ and space $O(n)$, where n is the number of lines of input. No algorithm can avoid quadratic behavior in the worst case, since the number of longest exact matches can be quadratic in the size of the input. However, an exact matching algorithm need not exhibit quadratic behavior for every input.

In particular, two algorithms have been implemented, with a time-space tradeoff. The first algorithm runs in time $O(n+p)$, where p is the number of distinct pairs of identical lines. (For typical C code, p is dominated by the closing braces that usually appear alone on a line.) The second algorithm, based on the suffix tree data structure [McC], runs much faster; on a fixed alphabet, it runs in time $O(n+m)$, where m is the number of matches found (ordinarily small compared to the size of the input). In practice, the alphabet is very large, but hashing is used to avoid a blowup in running time. Both algorithms use space linear in the size of the input, but the suffix-tree-based algorithm currently consumes three times as much space as the slower one, although further tuning may bring this down. Since the space consumption for millions of lines of code may be in the vicinity of main memory sizes, both algorithms are potentially of interest.

The first, more space-efficient, algorithm creates an array in which all identical lines are linked in lists, from the end of the file toward the front. Starting at the last line of the file, and working through the preceding lines, it "grows" matches through the file by following the linked lists and accumulating longer and longer matches. Two additional arrays are needed to keep track of the current lengths and the previous lengths computed. Thus, the algorithm requires three words per line of input.

The suffix-tree-based algorithm is more interesting. A suffix tree is a multiway Patricia trie; an example is given in Figure 3. Suppose the input is $b_1 b_2 \cdots b_n$; without loss of generality, we assume that the last symbol, b_n, occurs only once in the input. Each leaf of the suffix tree represents a distinct suffix of the input, where a suffix is a substring $b_i \cdots b_n$ for some i. Thus, there are exactly n leaves. Each arc of the tree is labelled with a nonempty substring of the input; the sequence of arcs from the root to a leaf yields the suffix represented by the leaf. For readability, in Figure 3, each leaf is labelled with the position of the start of the corresponding suffix and the substring representing the suffix, and each branching node is labelled with the sequence of labels on the path from the root to the branching node. Each node other than a leaf has at least two children; hence the number of nodes is linear in n. To make the size of the whole tree linear in the size of the input, the arc labels are stored as pointers into the input string. (Node labels need not be stored.) A suffix tree can be built in time linear in the size of the input, for a fixed alphabet [McC].

Now, two substrings of the input are a longest match if they are identical, but the preceding characters are different and the following characters are different. If two leaves are reached by different branches from a common ancestor B, their suffixes agree on the path from the root to B and then diverge. For example, the suffixes *bc*% and *bcbcabc*% agree on the initial *bc*, which is the label on the edge from the root to their lowest common ancestor, labelled *bc*. Thus, the labels on the path from the root to a

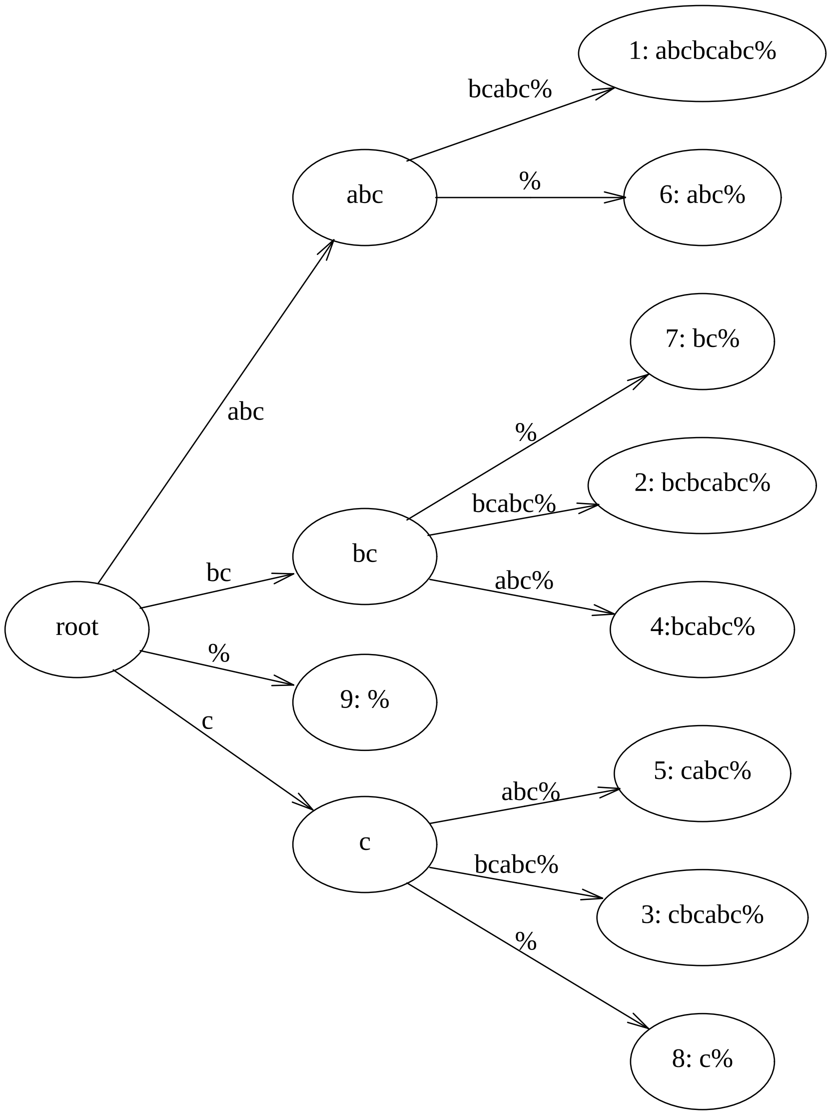

Fig 3. A suffix tree for the string *abcbcabc*%.

branching node B in the suffix tree represent a substring that occurs in at least two places in the input, and their *right contexts*, *i.e.* the characters following the two occurrences, are different. However, the *left contexts*, *i.e.* the characters preceding the two occurrences, may be the same. Therefore, the program must do more work to determine which such pairs also have distinct left contexts, resulting in a longest match.

To do this, the program recurses over the suffix tree. It has two jobs to do: to build up lists of suffixes grouped by left context, and to compare the lists found for its subtrees to identify longest matches. Thus, for a node N and a symbol c, a list $c{:}p_1, p_2, ..., p_k$ will be created, where $p_1, p_2, ..., p_k$ are the starting positions for suffixes with left context c that correspond to leaves in the subtree for N. Identification of longest matches is by checking each pair of elements in the cross-product of each pair of lists from distinct subtrees with distinct left contexts. For example, at the branching node corresponding to *bc*, the occurrence of *bc* starting at position 4 (with left context *c*) has a longest match with the occurrences of *bc* starting at positions 2 and 7, because each of them has left context *a*. In order to get the desired $O(n+m)$ running time, the above computations are only performed for branching nodes representing substrings that are long enough to be reportable.

Parameterized Matching.

The program to find parameterized matches has three phases. First, the lexical analyzer copies each line while replacing each token name that is a candidate for a parameter into a `P`, while creating a separate list of all such token names. For example, `x=3*y;` is turned into `P=P*P;`. Next, the transformed lines are passed to one of the exact matching algorithms described in the previous section. Exact matches that are sufficiently long are then analyzed for parameterized matches.

In such an exact match of transformed lines, the corresponding lines match exactly, including the P's that replaced parameter candidates. Therefore, the *i*th lines of the two segments must have the same number of parameter candidates in the same positions, for all *i*. If the original lines of the entire exact match have a parameterized match, then a one-to-one correspondence can be found that pairs the *j*th parameter name in the *i*th line of the first code segment with the *j*th parameter name in the *i*th line of the second code segment, for all *i* and *j*.

In general, it will not be possible to find a one-to-one correspondence for the entire exact match. Nevertheless, there may be parts of the exact match that have a sufficiently long parameterized match to report. For example, consider the following two code fragments.

```
x=y-z;
if (y>z)
        m=1;
h=f(x);
y=x;

x=b-c;
if (b>c)
        n=1;
h=f(x);
c=x;
```

Pairings `y=b`, `z=c`, and `m=n` (and the identity on other parameter candidates) yield a parameterized match for the first four lines. However, the fifth line requires a pairing `y=c`, which conflicts with both `y=b` and `z=c`. Pairings `y=c` and `m=n` (and the identity on other parameter candidates) yield a parameterized match for the last three lines.

Therefore, the algorithm scans the exact match line by line, keeping track of the start of the current match. If a conflict occurs between the one-to-one correspondence needed for a new pair of lines and the one-to-one correspondence that has already been established, then the current parameterized match is terminated, and if it is sufficiently long, reported as a longest parameterized match. The conflicting pairs are removed from the old one-to-one correspondence, the pairs from the new line are added to it, and the line after the last conflict with the new one-to-one correspondence is taken as the beginning of a new parameterized match. When all the lines have been processed, the last parameterized match is reported if it is sufficiently long. The algorithm runs in time linear in the length of the exact match.

Discussion

Dup was implemented in about 2500 lines of C and **lex** [LS], and runs under *UNIX*^{TM}. It has been applied to the source for the X Window System [SG] (minus some table initializations) and part of a larger AT&T software system. The parameterized matches for the X source and some of the AT&T system are plotted in Figures 4 and 5, respectively, where the minimum match length is 30 lines.

The parameterized matches include a substantial amount of the code in each case, namely 15% (1136 matches) for the X system, and 23% (596 matches), for the AT&T system. These numbers increase dramatically for smaller minimum match lengths; for example, if the minimum match length is reduced to 15 lines, the number of matches for the AT&T system increases to 4174 and the percentage of lines involved increases to 38%.

The plots are dense near the main diagonal, implying that most copies tend to occur fairly locally, e.g. within the same file or module. However, certain line segments occur away from the main diagonal; it would be interesting to investigate why the corresponding sections of code are duplicated.

When individual matches are examined, the matches for C code usually look reasonable. In particular, the one-to-one correspondence between parameters usually finds pairings of similar tokens, *i.e.* small integers with small integers or variables with other variables with related names.

When conflicts are found by the algorithm that generates parameterized matches from the exact matches for transformed lines, the conflicts frequently involve small integer constants, especially zero, used differently in two places.

There are two situations in which the program has produced output that did not appear reasonable. One situation is the initializations of large tables, with one value per line, where many matches may be found even for random values. These should probably be filtered out automatically. The other situation involves `case` statements, where a succession of statements of the form `case` *name*: allow a parameterized match to be found for a long list of probably unrelated cases. A reasonable solution

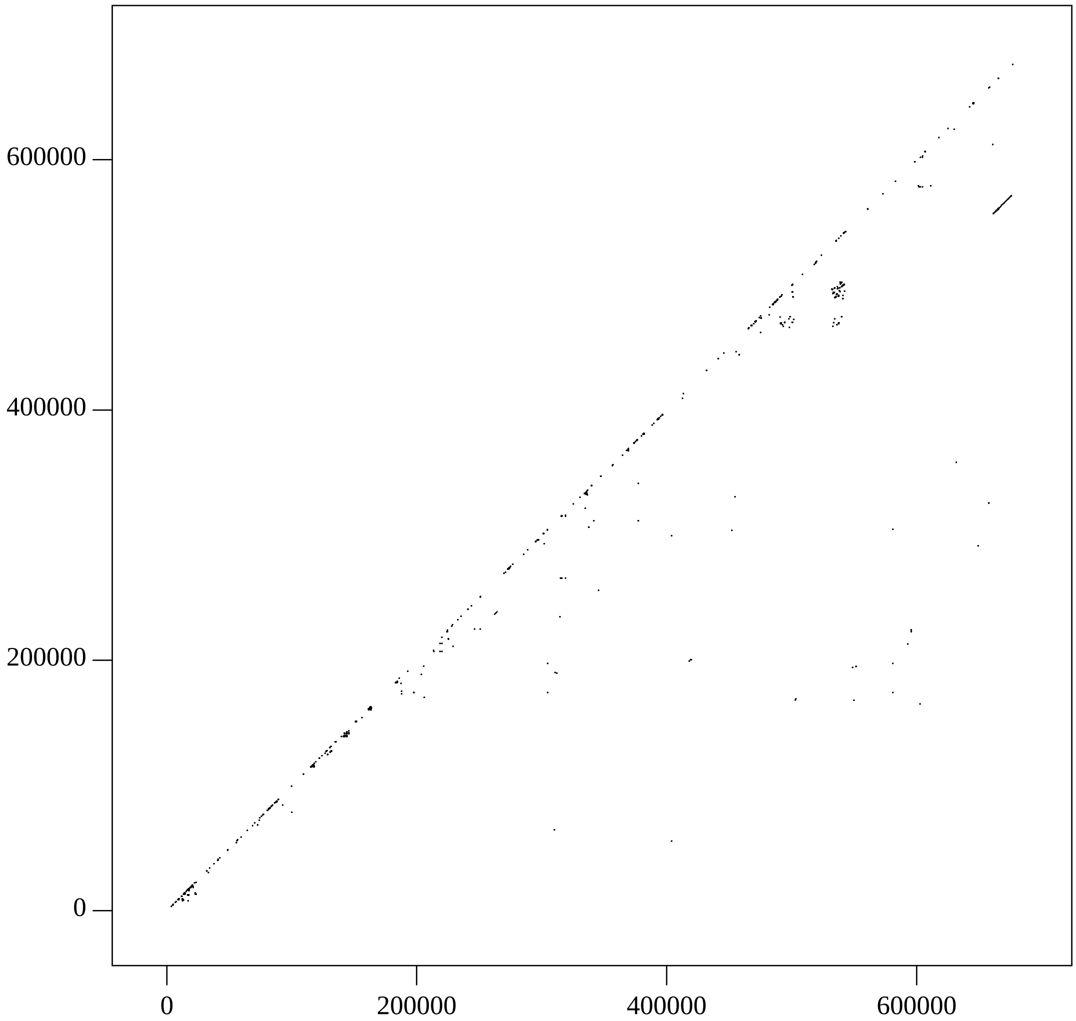

Fig. 4. Parameterized matches for X source (minus some tables).

for avoiding such output was found to be to limit parameterized matches so that the number of parameters is restricted to be at most half the number of lines.

In practice, many of the parameters found are related to error checking and handling, in both of the systems that were studied. The existence of unique constants used for error handling is one reason that many more parameterized matches are found than exact matches.

An obvious question is whether the output could be used to generate parameterized procedures automatically from the input to reduce the size of the code. Regretfully, 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 most files by hand, although the output from **dup** would undoubtedly be helpful.

However, the plots of large amounts of code should be useful to managers in visualizing the complexity and interrelationships of a whole software system. This paper includes plots of hundreds of thousands of lines of code. However, the program has been applied to a subsystem of over a million lines of code, and could be applied to still larger amounts of code.

Moreover, zooming in on smaller sections can be useful for finding anomalies in the scatter plot that reflect anomalies in the code. Three types of interesting features that have been found in casual browsing have been 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. The obsolete file was found by noticing rather extensive duplication between two files in a module. Figure 6 shows a scatter plot with a gap between

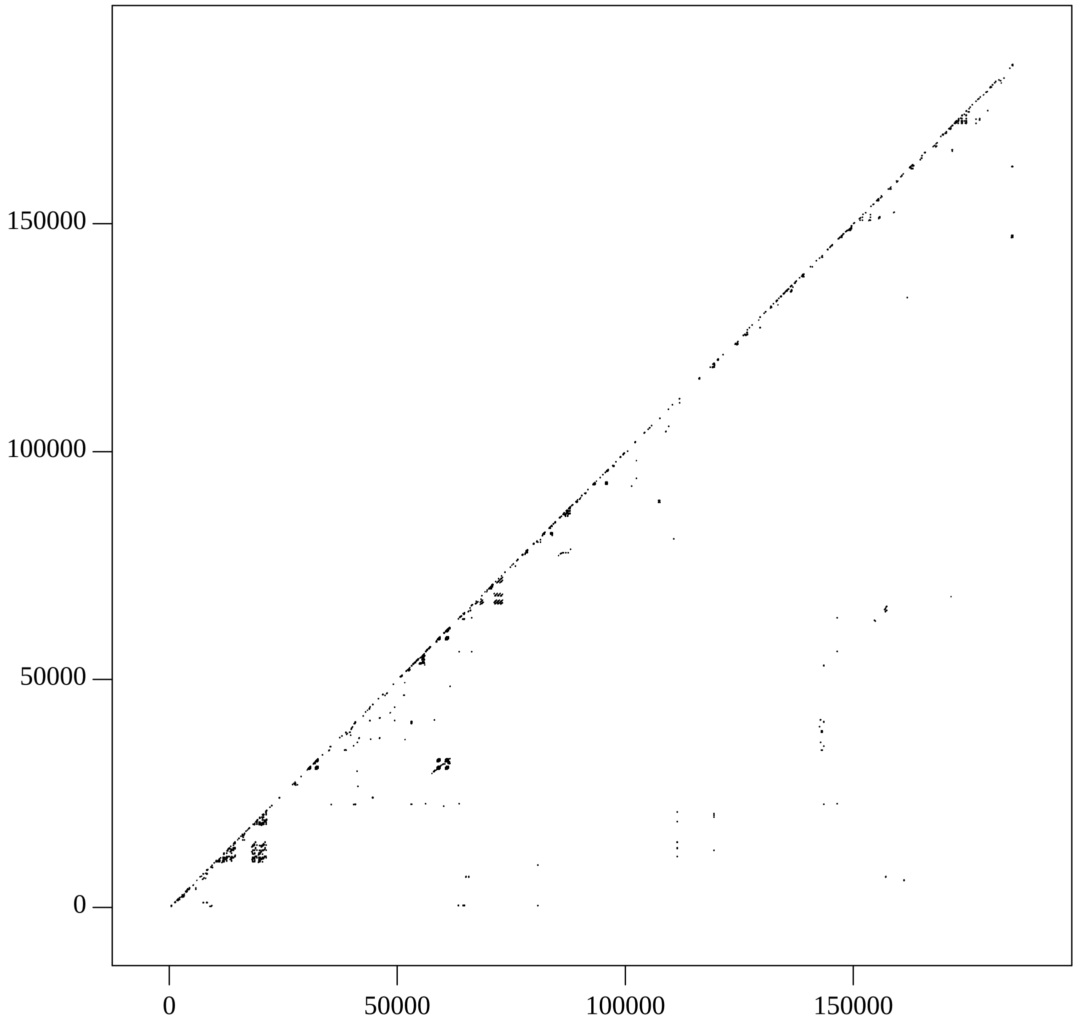

Fig. 5. Parameterized matches for some code from a production system.

two collinear line segments representing two long matches. The gap corresponds to two sections of code, one of which is 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 is apparently a bug fix that was applied in one copy and not the other.

Acknowledgements

I would like to thank Brian Kernighan for calling my attention to the code duplication problem and for suggesting the first lexical analysis part of the parameterization algorithm. I would also like to thank William Chang for providing his code for suffix tree construction. I am 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.

References

[CH] Kenneth W. Church and Jonathan I. Helfman, Dotplot: a program for exploring self-similarity in millions of lines of text and code, *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface* (1992).

[E] Stephen G. Eick, Dynamic Graphics for Software Visualization, *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface* (1992).

[H] M.H. Halstead, *Elements of Software Science*, Elsevier North-Holland, New York (1977).

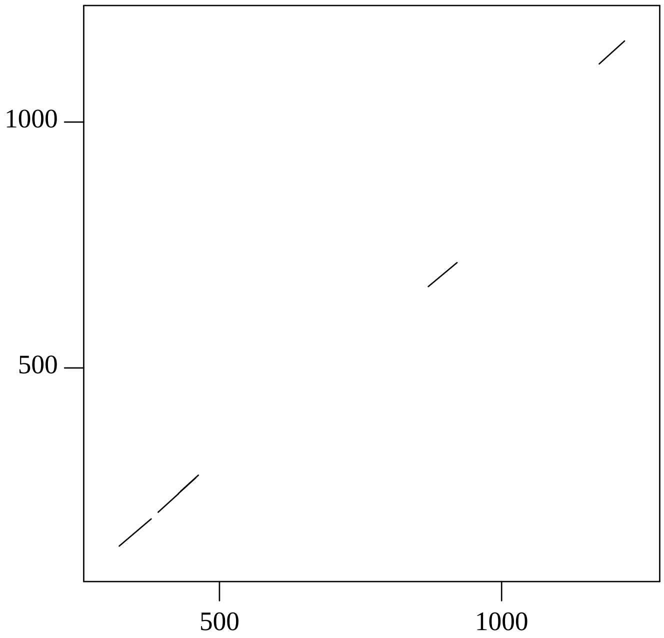

Fig. 6. A plot with an anomaly: a small gap between two matches.

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

[Jo] Ralph Johnson, personal communication (October, 1991).

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

[LS] M.E. Lesk and E. Schmidt, Lex- a lexical analyzer generator, *UNIX Programmer's Manual*, Holt Reinhart and Winston (1983-1984).

[LMW] Richard J. Lipton, Thomas G. Marr, and J. Douglas Welsh, Computational approaches to discovering semantics in molecular biology, *Proc. of the IEEE* 77,7 (1989), pp. 1056-1060.

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

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

[SK] D. Sankoff and J.B. Kruskal (editors), *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison* (1983), Addison-Wesley, Reading, MA.

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

[ZL] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Inf. Theory* IT-23 (1977), pp. 337-343.