On Finding Duplication in Strings and Software

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

ABSTRACT

This paper investigates finding duplication within a string. The results are phrased in terms of maximal matches, which are pairs of substrings that match but for which the match cannot be extended to the left or right in the input. An algorithm is given to find all maximal exact matches over a threshold length. For a finite alphabet Σ, it runs in time $O(n \log |\Sigma| + m)$, where n is the input length and m is the number of matches reported. The algorithm has been implemented and has been used as the basis of a program for finding code duplication in large software systems.

February 15, 1993

On Finding Duplication in Strings and Software

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

1. Introduction

This paper describes an algorithm for finding duplication in strings and its application to finding duplication in software. The basic problem is to find all maximal matches, which are pairs of substrings that match but for which the match cannot be extended to the left or right; a threshold length for reporting may be specified. We show that for a string of length *n*, all maximal matches over the threshold length can be found in time $O(n \log |\Sigma| + m)$, where *n* is the input length and *m* is the number of such matches. For contrast, we show that if all the longest matches do not need to be listed, and it is just desired to identify the distinct substrings for which a maximal match exists, time $O(n \log |\Sigma|)$ time is sufficient. The algorithms are based on string pattern matching techniques, notably the suffix tree data structure [McC], but the problem of finding maximal matches differs from pattern matching in that the patterns to be found are not explicitly specified in a query; they must be deduced from the input itself.

The algorithm for reporting all maximal matches over threshold length has been implemented and forms the basis of a program **dup** [Bak1] for finding duplication in software. It is well known that copying code is a poor programming technique because of the danger of bug fixes being applied to one copy but not all the others. Yet copying code may be used in order to avoid touching working sections of code. **Dup** is intended to help track down duplication that has been introduced over time into a large system, and to aid managers in understanding the extent of duplication. It could also be used to detect plagiarism of code, *e.g.* by students. **Dup** finds all maximal matches over a threshold length under two definitions: sections of code that are exactly the same, or sections of code that are the same except for a systematic change of parameters. **Dup** and some results of applying it to production software systems are described in Section 4.

The algorithm for finding maximal exact matches in strings could have many other applications in the areas of text processing and natural language processing. It could be more helpful than the UNIX diff program [KP] for comparing files that differ by global rearrangements of blocks of text, since diff is most useful when the files are almost the same; scatter plots could be used to aid in visualization. Detection of plagiarism would be another obvious application in the area of text processing. Algorithms for finding maximal exact matches could also be useful for machine translation of technical manuals, which often tend to be repetitious, since an interactive program could aid a translator by providing translations of sections previously translated. Applied to documents with the aid of a parts-of-speech analyzer, it could find repeated sentence structures specified in terms of parts of speech, for use in analyzing writing style or determining authorship of documents.

Little previous work has been done on finding all duplication in a string. An obvious algorithm [Ker] for finding duplication in a string is to calculate a matrix *A* such that $A[i,j]$ is 1 if line *i* matches line *j*, and 0 otherwise, and to search the diagonals for maximal matches; this algorithm would take linear space and quadratic time (in fact, quadratic time for every input). An early algorithm [KMR] to find duplication of length *d* in a string of length *n*, given an alphabet of size *s*, was to construct integers mod s^d representing the $n-d$ substrings of length *d* into s^d buckets in time $O(n)$ and space s^d. The initial implementation of **dup** used a duplication algorithm based on chaining together identical symbols that ran in time $O(n+p)$, where n is the input length and p is the number of pairs of distinct occurrences of identical symbols (where the alphabet size was potentially linear in n) [Bak1].

The problem of finding duplication in a string is closely related to string pattern matching and to data compression. Suffix trees have been used as the basis for algorithms for both problems [McC, RPE]. For string pattern matching, however, the basic problem is to determine whether one string occurs within another, rather than to find all maximal matches. The problems closest to ours studied previously have been (1) to find the longest substring common to two strings [Cr87], (2) to construct a minimal transducer that outputs the position of first occurrence for each substring of a string [Cr86] and (3) to find squares in a string, or to determine whether as string is square-free, where a square is an occurrence of xx for a substring x [AP,Cr86,Gu,ML,Rab,SeiG]. Problem (3) has also been called the problem of finding repetition in a string. In the [RPE] algorithm for Lempel-Ziv data compression [ZL], the algorithm finds some maximal matches in the course of building a suffix tree, but does not find all maximal matches.

The goal of reporting all pairs of matches is quite different from the goal of finding the minimum edit distance between two strings [HS], *i.e.* the number of insertions, deletions, and changes needed to transform one string into the other. Finding the minimum edit distance involves finding corresponding subsequences in the two strings that match, whereas there is no notion of sequencing in reporting all pairs of maximal matches.

Definitions are given in Section 2. Section 3 gives the algorithm for finding all maximal matches over a threshold length, and for contrast gives another algorithm that merely identifies the substrings that are involved in maximal matches, and finds one maximal match for each. The implementation of the first algorithm in **dup**, for application to finding duplication in software, is described in Section 4. Directions for further research are discussed in Section 5.

2. Maximal exact matches and suffix trees

In this section, we define and explore the notion of maximal exact matches and review the definition of suffix trees. We assume a RAM model of computation with the uniform cost criterion [AHU].

We assume that all strings are over a finite alphabet Σ. If S is a string of length n, it will be convenient to denote the ith symbol of S by S_i and the substring from positions i to j (inclusive) by $S_{i,j}$, for $1 \le i,j \le n$.

DEFINITION 1: If $S_{i,i+k} = S_{j,j+k}$, where $1 \le k \le n-1$ and $1 \le i,j \le n-k$, then we say $S_{i,i+k}$ and $S_{j,j+k}$ *match*. This match is represented by $(i,j,k+1)$ or equivalently, $(j,i,k+1)$; *i.e.*, by the two starting positions of the match and the length of the match.

DEFINITION 2: A match (i,j,k) in a string S is *left-extensible* if $i,j > 1$ and $S_{i-1} = S_{j-1}$, and *right-extensible* if $i+k, j+k < n$ and $S_{i+k+1} = S_{j+k+1}$. A match is *maximal* if it is neither left-extensible nor right-extensible and it is not the trivial match $(1,1,n)$, where n is the length of the string.

The maximal match relation is not transitive, and hence not an equivalence relation. For example, consider the string *bdecdbde*. The triple $(2,5,1)$ represents the maximal match between the first two d's. Similarly, the triple $(5,7,1)$ represents the maximal match between the last two d's. However, the first and last d's are not a maximal match; they are part of the maximal match $(1,6,3)$.

The last example also shows that an interval appearing in one maximal match may be a subinterval of another maximal match. Similarly, maximal match intervals can overlap. For example, in *abcabdbc*, $(1,4,2)$ is a maximal match, and $(2,7,2)$ is also a maximal match; the first and second symbols of the string occur in the first match, while the second and third symbols occur in the second match.

Because the maximal match relation is not an equivalence relation, maximal matches must be reported in general as pairs of substrings that are maximal matches rather than sets of substrings that are maximal matches.

In general, the number of maximal matches can be quadratic in the length of the string. For example, consider strings of the form $(0a1a)^i 0$, of length $4i+1$. In this case, the i substrings a between 0 and 1 are maximal matches for the i substrings a between 1 and 0, contributing i^2 maximal matches to the total.

DEFINITION 3: If S is a string of length n, the ith *suffix* of S is $S_{i,n}$, for $1 \le i \le n$.

DEFINITION 4: A *suffix tree* for a string S is a compacted trie (multiway Patricia trie) representing the suffixes of S [McC], assuming (without loss of generality) that the final symbol is an endmarker *$* that does not occur elsewhere in S. In other words, each arc of the tree is labelled with a nonempty substring of the input, and at each internal (non-leaf) vertex, each arc to a child begins with a distinct symbol. Moreover, it represents the suffixes of S in the sense that for each leaf, the sequence of labels from the root to the leaf is a suffix of S, and each suffix of S is the sequence of labels on a path in the tree starting at the root. Finally, the trie is compacted in the sense that there are no vertices of degree one; this is possible because arc labels are strings rather than just symbols. A suffix tree is shown in Figure 1.

Because of the constraint that the final symbol is unique, no suffix is a substring of another suffix, and every suffix in fact is the sequence of labels on a path from the root to a leaf. Thus there is a one-to-one correspondence between suffixes of S and leaves.

Since there are no vertices of degree one, the number of vertices in a suffix tree is linear in the size of the input. To store the tree in linear space, the label for each arc into a vertex v (other than the root) is represented as a starting index into the input string, called $position(v)$, plus a length, called $arclen(v)$. Note that an arc leading to a leaf represents a unique substring of the input, but an arc leading to an internal vertex can represent a substring occurring in more than one position of the input; however, an index for only one of these positions is stored.

McCreight [McC] showed that a suffix tree can be built in time and space linear in the size of the input for a fixed alphabet, or $O(n \log |\Sigma|)$ time and $O(n)$ space for a finite alphabet Σ and input of length n. To achieve these bounds, the arcs out of each vertex may be stored in a linked list for a fixed alphabet, and in a balanced tree scheme with $O(\log |\Sigma|)$ access time for variable alphabets.

DEFINITION 5: For each vertex v, define the *pathstring* of v, denoted by $pathstr(v)$, to be the concatenation of the labels on the path from the root to v, and the *pathlength* for v, denoted by $pathlen(v)$, to be the length of $pathstr(v)$. (In the terminology of McCreight [McC], each node v is the locus of its pathstring.) For each vertex v, let $T(v)$ be the subtree rooted at v.

As described above, there is a one-to-one correspondence between suffixes in the input and leaves of a suffix tree. It will be convenient to have notation specifying this correspondence.

DEFINITION 6: For a leaf L of a suffix tree for a string S, define the starting position of (the suffix corresponding to) L to be $start(L) = position(L) + arclen(L) - pathlen(L)$, and the left context of L to be $LC(L) = S_{start(L)-1}$, *i.e.* the left context of $start(L)$. Define $leaf(i)$ to be the leaf whose pathstring is the ith suffix, $S_{i,n}$,

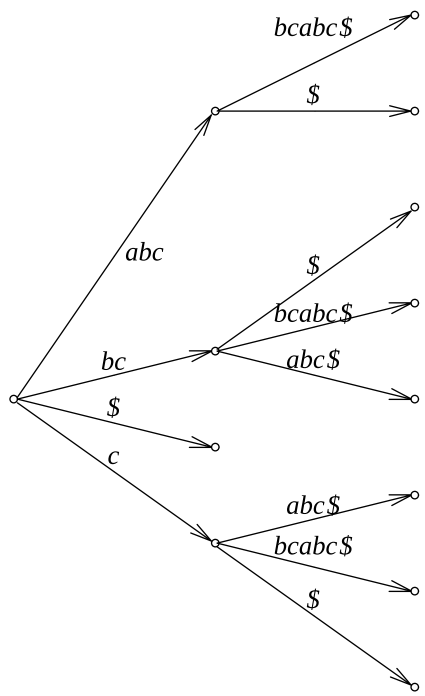

Figure 1. A suffix tree for the string *abcbcabc$*.

3. Finding duplication in a string.

This section gives two algorithms for finding duplication in a string. The first reports all pairs of maximal matches over a threshold length. The second reports all substrings that occur in maximal matches, and for each such substring gives one maximal match.

DEFINITION 7: Suppose $S = S_1 S_2 \cdots S_n$ is a string of length *n*. Let S_{n+1} be an endmarker, say *$*, that does not occur in *S*, and let S_0 be a beginning marker that does not occur in *S*, $S_0 \neq S_{n+1}$. Define the *left context* at position *j*, $1 \leq j \leq n+1$, to be S_{j-1}, and the *right context* at position *j*, $0 \leq j \leq n$, to be S_{j+1}.

To derive the algorithms, we start by considering how suffix trees relate to longest matches. Suppose *S* has length *n*. A substring $\alpha \neq S$ occurs in a match that is not right-extensible iff α is a pathstring for an internal vertex *v* in the suffix tree for *S$*, because the distinct symbols following two occurrences of α correspond to the distinct initial symbols on arcs from *v* to its children. However, α may not correspond to a maximal match, because the descendant leaves of this vertex could possibly all have the same left context. More precisely, a substring $\alpha \neq S$ corresponds to a maximal match (i, j, k), where $k = |\alpha|$, iff the leaves representing $S_{i,n}$ and $S_{j,n}$ are descendants of different children of an internal vertex *v* and the left contexts of *i* and *j* are different. Therefore, the algorithm must discover for each internal vertex *v* which descendant leaves reached by distinct arcs from *v* have different left contexts.

Thus, we essentially want to accomplish a bucket sort of the descendant leaves of each internal vertex, so as to be able to report which pairs of these leaves have different left contexts and consequently represent maximal matches. For efficiency, it is clearly advantageous to take a recursive approach, so as to be able to take advantage of the information computed for the descendants of each internal vertex. To keep track of the necessary information, we augment the suffix tree data structure with auxiliary lists as follows.

DEFINITION 8: A *plist* is a list of integers. A *clist* is a list of plists. The *left context* for a plist *pl*, $LC(pl)$, is S_{i-1} if the first element of *pl* is *i*.

In our algorithm, a plist *pl* will represent positions in the input with the same left context, $LC(pl)$; hence $LC(pl)$ represents the left context of all positions in *pl*. A clist will be a list of plists with distinct left contexts.

The algorithm will recursively construct a clist for each vertex to represent the descendant leaves sorted into plists by left context. Suppose we represent a plist *pl* with positions $p_1, p_2, \cdots, p_k$ by $LC(pl): p_1, p_2, \cdots, p_k$. The recursion is based on combining the plists of the children to obtain the plist of the parent. For example, if the first child of vertex *v* has a clist containing plists $a:77,80,13$ and $b:43,72$, the second child has plists $a:27,39$ and $c:21,88,40$, and the third child has plists $a:31$, $b:29,10$, and $c:93$, then *v* will have plists $a:77,80,13,27,39,31$, $b:43,72,29,10$, and $c:21,88,93$. The sorting will be done in such a way that the time for sorting is linear in the input length plus the number of matches reported.

```
clist sdup(vertex v, int len, int t) {
   clist cl; vertex s;
   if (v is a leaf) return a clist containing one plist
      consisting of (start(v));
   cl = NULL;
   for each child s of v
      cl = combine(cl,sdup(s,len+arclen(s),t),len,t);
   return cl;
}

clist combine(clist cl1, clist cl2, int len, int t) {
   plist pl1, pl2;
   clist outputlist;
   if (len < t) return (NULL);
   for each plist pl1 of cl1 and each plist pl2 of cl2
      if (LC(pl1) ≠ LC(pl2))
         for every p1 in pl1 and every p2 in pl2
            report a maximal match (p1,p2,len);
   /*construct outputlist */
   for each plist pl1 of cl1 and each plist pl2 of cl2
      if (LC(pl1) == LC(pl2)) {
         include in outputlist the plist obtained by appending
            pl1 and pl2;
         mark pl1 and pl2 as used;
      }
   for each plist pl1 of cl1 that is not marked used
      include pl1 in outputlist;
   for each plist pl2 of cl2 that is not marked used
      include pl2 in outputlist;
   remove marks from all plists in outputlist;
   return outputlist;
}
```

Figure 2. The algorithm for reporting all maximal matches of length at least *t*.

THEOREM 1. Let t be a positive integer. Given a string S of length n, all maximal matches in S of length at least t can be reported in space $O(n)$ and time $O(n\log|\Sigma|+m)$, where m is the number of matches reported.

Proof. Build a suffix tree for $S\$$ in $O(n\log|\Sigma|)$ time using McCreight's algorithm [McC]. Then call $sdup(root(T),0,t)$, where *sdup* is given in Figure 2.

The basic strategy is as follows. The procedures *sdup* and *combine* recurse over *T*, building plists and clists bottom up, and *combine* reports the maximal matches corresponding to each vertex of sufficient pathlength. The appending of plists and clists, which is not described explicitly in the code, is done via pointers rather than by actual copying of elements; by maintaining pointers to the beginning and end of each list, the algorithm accomplishes each append in $O(1)$ time.

We need to prove that the algorithm carries out this strategy correctly and runs in time $O(n+m)$.

First, note that when $sdup(v,len,t)$ is called, $len = pathlen(v)$. This follows because it is called on the root with $len=0$, and for each vertex *v*, $arclen(s)$ is always added to *len* when *sdup* is called on a child *s* of *v*.

We claim that the following statements hold:

(1) for each vertex *v* of pathlength at least *t*, the clist *cl* returned when *sdup* finishes processing *v* contains at most one plist for any left context; moreover, for each left context *c*, this clist contains a plist with left context *c* iff some leaf of $T(v)$ has left context *c*, and the elements of this plist are precisely the starting positions of suffixes corresponding to leaves of $T(v)$ with left context *c*.

(2) for each child *s* of a vertex *v* of pathlength at least *t*, when the loop of $sdup(v,len,t)$ begins to process *s*, the clist *cl* contains at most one plist for any left context; moreover, it contains a plist with left context *c* iff the subtree rooted at a child of *v* to the left of *c* contains a leaf with left context *c*, and the elements of this plist are precisely the starting positions of suffixes corresponding to leaves with this left context in the subtrees of children of *v* to the left of *s*.

The proof is by double induction. If *v* is a leaf, the clist returned by *sdup* contains just a plist consisting of $start(v)$, satisfying (1); (2) does not apply. So suppose *v* is not a leaf, the pathlength of *v* is at least *t*, and (1) holds for the clist returned by $sdup(u,len+arclen(u),t)$ for each child *u* of *v*. We show inductively that (2) holds for all children of *v*. If *s* is the leftmost child of *v*, *cl* is NULL, and (2) is vacuously true. Otherwise, assume that (2) holds for *s*. We will show (2) holds for the right sibling of *s*. Now, in *combine*, $cl1$ is either NULL (if *s* has no sibling to the left) or the clist produced for the children of *v* to the left of *s*, while $cl2$ is the clist produced by *sdup* for *s*. The second loop of *combine* finds plists of $cl1$ and $cl2$ with the same left context and combines them, while the two loops searching $cl1$ and $cl2$ adds in plists for left contexts that occur in exactly one of $cl1$ and $cl2$. Obviously, at most one plist is placed in *outputlist* for each distinct left context. Moreover, a plist is placed in *outputlist* iff it occurs in at least one of $cl1$ and $cl2$ iff it occurs in a subtree rooted at *s* or one of its siblings to the left by the induction hypotheses, and this plist contains exactly the leaves with this left context in the subtrees of *s* or the siblings to the left, by the induction hypothesis. Thus, (2) holds for the right sibling of *s* (if any). Finally, we complete the double induction by observing that if *s* has no right sibling, then the resulting clist is returned by $sdup(v,len,t)$, satisfying (1) for *v*.

Next, we show that the following statement holds:

(3) the algorithm reports exactly the maximal matches of length at least *t*, and for each vertex *v*, the matches reported when *combine* is called in the loop of $sdup(v,len,t)$ are precisely those for which the pathstring is $pathstr(v)$.

Suppose (i,j,k) is a maximal match, where $k \geq t$. If $i = j$, the match is all of *S*, which was reported separately. So assume $i \neq j$. Since the strings of length *k* starting at *i* and *j* are the same, say α, but their right contexts are different, there is a vertex *v* for which the pathstring is α, and the suffixes starting at *i* and *j* correspond to the pathstrings for leaves $leaf(i)$ and $leaf(j)$, respectively, in subtrees of different children of *v*. Without loss of generality (since the maximal match (i,j,k) is equivalent to (j,i,k)), suppose the subtree containing $leaf(i)$ is to the left of the one containing $leaf(j)$. By (2) above, when *sdup* is ready to process the subtree containing $leaf(j)$, *i* will be in the plist of *cl* for the left context S_{i-1}. From (1) above, the clist produced by *sdup* from the child of *v* whose subtree contains $leaf(j)$ will include *j* in the plist for its left context S_{j-1}. Since $S_{i-1} \neq S_{j-1}$, *combine* will report matches for the cross-product of their plists, including (i,j,len). From above, $len = pathlen(v) = |\alpha| = k$.

Conversely, suppose that in the loop of $sdup(v,len,t)$, *combine* is called for a child *s* of *v* and *combine* reports (i,j,len). From above, $len = pathlen(v)$, and by inspection of *combine*, $len \geq t$. Then *i* is in a plist $pl1$ of $cl1$ and *j* is in a plist $pl2$ of $cl2$, and $pl1$ and $pl2$ have different left contexts. By (1) and (2) above, $leaf(i)$ is in the subtree of a sibling of *s* to the left of *s*, and $leaf(j)$ is in the subtree of *s*. By definition of suffix trees, the fact that $leaf(i)$ and $leaf(j)$ are in subtrees of different children of *v* means that the substrings starting at *i* and *j* agree with $pathstr(v)$ for length $pathlen(v)$ and then have different right contexts. Since in addition, the left contexts of *i* and *j* are different, $(i,j,pathlen(v))$ is a maximal match. Furthermore, *combine* only reports a match if $pathlen(v) = len \geq t$.

During the recursion, *sdup* is called on each vertex *v* of the tree exactly once. For each non-leaf child *s* of *v*, *combine* is called on the clist *cl* and the clist resulting from $sdup(s, pathlen(s), t)$. Now, *combine* examines the cross-product of the two clists twice and searches $cl1$, $cl2$, and *outputlist* once each. We will show that over all vertices, the amount of time spent in cross-product loops on processing pairs with distinct left contexts is linear in *m*, the amount of time spent on processing pairs with the same left contexts is linear in *n*, and the amount of time spent in the last three clist searches at each vertex is linear in $n + m$.

In the first clist cross-product loop at a vertex *v*, whenever the left contexts are different for the plists, a match is reported by *combine* for each element in the cross-product of the plists. Therefore, the amount of time spent at this vertex on processing the number of pairs with different left contexts is linear in the number of matches reported at this vertex. From this fact and (3), we see that the amount of time spent at all vertices on processing clist pairs with different left contexts is linear in *m*. Similarly, over all vertices, the amount of time spent in the second cross-product loop checking pairs of plists with different left contexts is linear in *m*.

In the second clist cross-product loop, whenever the left contexts are the same, *combine* appends one plist to the other using constant time. The number of appends to form a plist with *k* elements is easily shown by induction to be at most $k - 1$. By (1), the clist produced for the root by *sdup* contains a plist for each left context occurring for a suffix, and each plist contains exactly the starting positions of the suffixes with that left context. Therefore, the total amount of time spent at all vertices by *combine* on appending plists with the same left context is linear in the number of leaves with that left context, and the total amount of time spent by *combine* on appending plists is linear in *n*. Similarly, in the first clist cross-product loop, the time spent at all vertices in finding that the left contexts are the same is $O(n)$. Thus, the total amount of time spent in the two clist cross-product loops on plists with the same left context is $O(n)$.

At each vertex, each of the two loops that merely search the clists $cl1$ and $cl2$ takes time linear in the length of the clist, which is linear in the time for processing the cross-product of the two clists. Therefore, over all vertices, these loops take time $O(n + m)$. Similarly, the unmarking of *outputlist* takes time linear in the lengths of $cl1$ and $cl2$ and thus time $O(n + m)$. □

Finally, we note that not as much work needs to be done to identify just the substrings for which maximal matches occur, if the list of all maximal matches does not need to be generated. To show this, first we need a simple characterization of which substrings can occur in a maximal match.

PROPOSITION 1. Let S be a string. For a substring α of S, $\alpha \neq S$, α occurs in a maximal match iff in the suffix tree for $S\$$, α is a pathstring for some vertex v such that there are leaves L_1 and L_2 in the subtree rooted at v with $LC(L_1) \neq LC(L_2)$.

Proof. Suppose α occurs in a maximal match (i,j,k). If $i=j$, $\alpha=S$. So assume $i \neq j$. By definition of maximal, the match is not right-extensible, and $S_{i,i+k-1}=S_{j,j+k-1}$ is the pathstring for an internal vertex v of T. Leaves $leaf(i)$ and $leaf(j)$ are in the subtree rooted at v. Since match (i,j,k) is not left-extensible, the left contexts $S_{i-1}=LC(leaf(i))$ and $S_{j-1}=LC(leaf(j))$ are distinct.

On the other hand, suppose that v has pathstring α and there are two leaves in a subtree rooted at v whose left contexts are distinct. If these two leaves are descended from different children of v, let L_1 and L_2 be these leaves. Otherwise, these two leaves are descended from the same child of v. In the latter case, let L_1 be any leaf descended from a different child of v, and let L_2 be one of the first two leaves that has a left context different from that of L_1; since the first two leaves do not have the same left context, such an L_2 exists. We claim $(start(L_1), start(L_2), |\alpha|)$ is a maximal match corresponding to α. This follows because the prefixes of the suffixes corresponding to L_1 and L_2 both equal α, their left contexts are different by choice of L_1 and L_2, and their right contexts are different because L_1 and L_2 are in subtrees of different children of v. □

This characterization of which substrings can appear in a maximal match leads to the following result.

THEOREM 2. Let t be a positive integer. For a string S of length n over a finite alphabet Σ, $O(n \log |\Sigma|)$ time is sufficient to find all substrings of S that correspond to maximal matches, and to report for each such substring α, one maximal match (i,j,k) for which $S_{i,i+k-1} = S_{j,j+k-1} = \alpha$.

Proof. We use an algorithm based on Proposition 1. Construct the suffix tree T for $S\$$ in $O(n \log |\Sigma|)$ time using McCreight's algorithm [McC]. In order to identify the substrings of S that occur in maximal matches, we recurse over T, and for each vertex v, compute two distinct left context positions for leaves in the subtree rooted at v if at least two exist, or the unique left context position, if only one exists. To calculate this for an internal vertex v, we need merely scan the results from the children of v to determine if there are (at least) two distinct left context values. For a leaf L, the algorithm returns the left context of $start(L)$. This procedure runs in time linear in the size of T. By Proposition 1, the maximal matches correspond to the pathstrings of those vertices for which two distinct left contexts are reported; their pathstrings can be reported in terms of their starting positions and lengths.

Finally, in order to return a maximal match for each substring found to correspond to a maximal match, we extend the algorithm in four ways: (1) we let it calculate the pathlength of the current vertex recursively, (2) at a leaf L, along with the left context, the algorithm returns $start(L)$ (the position of the beginning of the suffix corresponding to L, calculated from $position(L)$, $arclen(L)$, and $pathlen(L)$, by definition), (3) at each internal vertex v, the algorithm returns both a left context and the associated position, and (4) when two distinct left contexts are reported for a vertex v, they are picked from different children of v, as in the proof of Proposition 1. Then, at a vertex v, if the algorithm finds two distinct left context values with positions i and j, respectively, it reports a maximal match $(i, j, pathlen(v))$. □

4. An implementation of *sdup* for application to software systems.

The algorithm *sdup* has been implemented and is the basis of a program **dup** for finding duplication in C code [Bak1, Bak3]. **Dup** is implemented in C and runs under *UNIX*^{TM}. The user may ask for **dup** to find either all maximal exact matches over a threshold length between sections of code, or all maximal parameterized matches over a threshold length. In a parameterized match, two sections of code are the same except for systematic substitution of one set of variables and constants for another, *e.g.* one section of code is transformed into the other section by substituting y for z, f for g, and 1 for 0 everywhere z, g, and 0 occur, respectively.

To find maximal exact matches over a threshold length in C code, **dup** operates as follows. A lexical analyzer scans the input, hashes each line, and obtains an integer to represent each line. (The integers representing two lines are identical iff the two lines are identical.) Then the algorithm in this paper is applied to the resulting string.

The algorithm implemented in **dup** for finding maximal parameterized matches is also based on the algorithm *sdup*. First, a lexical analyzer scans the input, and for each line, generates a transformed line in which all parameter candidates such as identifiers and constants are transformed to the same symbol P, and a list of the parameter candidates. For example, if the line is `x=fun(y)+3*x;` the lexical analyzer would generate `P=P(P)+P*P;` and a list `x`, `fun`, `y`, `3`, `x`. Then the lexical analyzer generates an integer representing each transformed line, using hashing as in the lexical analyzer for exact matching. An exact match of two sequences of transformed lines corresponds to a parameterized match if a one-to-one correspondence can be found between the actual parameters used in the two sections of code, *i.e.* between the lists of parameters produced for each line by the lexical analyzer; more generally, portions of an exact match may correspond to zero or more parameterized matches. Thus, the approach is to run *sdup* to find the maximal matches of the transformed lines and then check each maximal exact match to determine which parts of it correspond to maximal parameterized matches. The latter task is accomplished as follows: compare pairs of lines in order and construct a one-to-one correspondence along the way; when a conflict occurs with the one-to-one correspondence already created, report a maximal parameterized match if it is over threshold length, update the line number for the start of the match, update the one-to-one correspondence to reflect the new pairings, and continue.

Dup has proved to produce useful information. For a million-line subsystem of a 10-million line switching system, approximately 21% of the code was found to be involved in parameterized matches of at least 30 lines (not counting comments and white space). Furthermore, examining the output has led to identifying whole (900-line) directories that are identical except for parameters, obsolete files that were not deleted, and a probable bug where a bug fix was apparently applied to one copy of a code segment but not another. Maximal parameterized matches have been found to be more useful than maximal exact matches, partly because error-handling statements occur frequently, and tend to contain integers (error numbers, for example) that occur in only one place in the code, and these unique integers break up what would otherwise be longer exact matches. For further discussion of the software engineering aspects of **dup** and the results of applying it to the X Window System [SchG] source and to the source for a subsystem of a production switching system, see [Bak1].

The implementation of McCreight's algorithm is adapted from an implementation provided by William Chang. In this implementation, for efficiency with a large alphabet, the arcs are hashed as suggested by McCreight [McC] rather than stored as linked lists or a balanced tree structure as specified in the theoretical analysis of worst case results. Unfortunately, the hashing method serves well for building a suffix tree for a large alphabet, but does not provide for convenient recursive search of the children at each vertex. Therefore, after the suffix tree is built, the hashed arcs are transformed into linked lists for use by *sdup*.

Tables 1 and 2 give some data on running times of *sdup* obtained by running **dup** on parts of the switching system mentioned above. Three disjoint samples of code were provided by a programmer for the switching system; these will be referred to in the following as subsystems. **Dup** was run on these three subsystems with varying threshold lengths and two lexical analyzers, one for exact matches and one to make the transformation needed for parameterized matches; these lexical analyzers are referred to as "exact" and "param", respectively, under "lexical analysis" in the tables. The times given for *sdup* include the time for transforming the arc representation from hashing into linked lists and the time for running *sdup*. The experiments were run on one processor of an SGI machine with eight 33 MHz R3000 processors, data and instruction caches of 64 Kbytes, a secondary data cache size of 256 Kbytes, and main memory size of 256 Mbytes.

After discarding white space and comments, subsystem A has 35233 lines, subsystem B has 101874 lines, and subsystem C has 158579 lines; each lexical analyzer generates one symbol from each line as input for building the suffix tree. Using the "exact" lexical analyzer, the alphabet sizes are 12875, 31759, and 54891 for subsystems A, B, and C, respectively. Using the "param" lexical analyzer, the alphabet sizes are 2406, 3992, and 8808 for subsystems A, B, and C, respectively. For comparison with the *sdup* times, building the suffix tree for output from the "exact" lexical analyzer takes about 2.0 seconds for subsystem A, 2.3 seconds for subsystem B, and 3.8 seconds for subsystem C, and building the suffix tree for output from the "param" lexical analyzer takes about 1.2 seconds for subsystem A, 3.4 seconds for subsystem B, and 5.3 seconds for subsystem C.

Threshold Length	Lexical Analysis	Number of Matches	*Sdup* Time in Seconds
32	exact	149	0.64
16	exact	1761	0.78
8	exact	15634	1.27
4	exact	56892	2.72
2	exact	575975	23.84
32	param	9016	1.15
16	param	14008	1.37
8	param	20750	1.71
4	param	78641	4.11
2	param	1264641	49.41

Table 1. Running times of *sdup* on subsystem A, with 35233 lines

Table 1 gives data for running times of *sdup* on subsystem A for each lexical analyzer and varying threshold lengths. It illustrates the growth in the number of maximal matches and running time of *sdup* as the threshold length varies. The number of maximal matches blows up as the threshold length becomes very small, because the matches become dominated by closing braces. These very short matches do not appear to be of much interest in the application; threshold lengths of about 20 lines or more appear to provide useful information [Bak3].

Table 2 shows variations in running times for the three subsystems, sorted by threshold lengths and exact versus parameterized matches. The maximum time for *sdup* is about 13

Number of Lines	Threshold Length	Lexical Analysis	Number of Matches	*Sdup* Time in Seconds
35233	32	exact	149	0.64
101874	32	exact	70	2.68
158579	32	exact	80	3.67
35233	16	exact	1761	0.78
101874	16	exact	564	2.23
158579	16	exact	499	3.87
35233	8	exact	15634	1.27
101874	8	exact	4676	2.70
158579	8	exact	3384	3.99
35233	32	param	9016	1.15
101874	32	param	2245	2.86
158579	32	param	17784	5.06
35233	16	param	14008	1.37
101874	16	param	28938	4.09
158579	16	param	49094	6.56
35233	8	param	20750	1.71
101874	8	param	253967	13.15
158579	8	param	184023	12.57

Table 2. Performance for three subsystems of varying size

seconds. Even in Table 1, where the number of matches is over a million for a threshold length of two lines, the largest time for *sdup* is under a minute. Thus, *sdup* is a practical algorithm.

5. Directions for further research

Suffix trees use an amount of space that is linear in input length, but the constant is rather large. In this implementation, for a string of length n, up to $44n$ bytes are needed for building the suffix tree, which has between $n+1$ and $2n$ vertices: approximately $16n$ bytes to store the starting position and length of the substring corresponding to each arc, and the remainder for hashing and suffix links. (Some of this space can be freed up when the tree arcs are transformed into a linked list representation for *sdup*.) It would be interesting to investigate whether other data structures that have been found useful for string pattern matching could provide a basis for a more space efficient algorithm for finding duplication. One possibility would be to base the duplication algorithm on directed acyclic word graphs [BBEHM], which represent information similar to that in suffix trees in a more compact form. A second possibility would be to base the duplication algorithm on suffix arrays [CL].

While the duplication algorithm in **dup** is based on finding exact matches of a transformed input string first, as described in the previous section, a more elegant approach is under investigation [Bak2]. This approach depends on constructing a new kind of data structure called a parameterized suffix tree (a generalization of suffix trees), and finding parameterized duplication directly from the parameterized suffix tree.

Acknowledgements.

The author would like to thank Brian Kernighan for suggesting the problem of finding duplication in code, Bill Chang for providing an implementation of McCreight's algorithm for building suffix trees, and Raffaele Giancarlo and Mihalis Yannakakis for helpful comments on a draft of this paper.

References

[AHU] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (1974), Addison-Wesley, Reading, Massachusetts.

[AP] A. Apostolico and F. Preparata, Optimal off-line detection of repetitions in a string, *Theoretical Comput. Sci. 22* (83), pp. 297-315.

[Bak1] Brenda S. Baker, A Program for Identifying Duplicated Code, *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface* (1992), to appear.

[Bak2] Brenda S. Baker, Parameterized duplication in strings: algorithms and an application to software maintenance, in preparation.

[Bak3] Brenda S. Baker, A software tool for finding duplication in large software systems, in preparation.

[BBEHM] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., and McConnell, R., Building a complete inverted file for a set of text files in linear time, 16th ACM Symposium on Theory of Computing (1984), pp. 349-358.

[CL] W. I. Chang and E. L. Lawler, Approximate string matching in sublinear expected time, Proc. of 31st Symposium on Foundations of Computer Science (1990), 116-124.

[Cr87] Maxime Crochemore, The Longest Common factor of two words, Proc. of International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Vol. I, *Lecture Notes in Computer Science 249* (1987), ed. G. Goos and J. Hartmanis, Springer-Verlag, Berlin, pp. 26-36.

[Cr86] Maxime Crochemore, Transducers and Repetitions, *Theoretical Computer Science 45* (1986), pp. 63-86.

[Gu] L. Guibas, Periodicities in Strings, *Combinatorial Algorithms on Words* (1984), ed. A. Apostolico and Z. Galil, Springer-Verlag, Berlin, pp. 257-270.

[HS] J.W. Hunt and T.G. Szymanski, A fast algorithm for computing longest common subsequences, *Comm. ACM 20,5* (1977), pp. 350-353.

[KMR] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg, Rapid identification of repeated patterns in strings, trees, and arrays, Proc. of Fourth Annual ACM Symposium on Theory of Computing (1972), pp. 125-136.

[Ker] Brian W. Kernighan, personal communication (August, 1991).

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

[ML] M.G. Main, and R.J. Lorentz, An O(n log n) algorithm for finding all repetitions in a string, *J. Algorithms 5* (1984), 422-432.

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

[Rab] M.O. Rabin, Discovering repetitions in strings, *Combinatorial Algorithms on*

Words (1984), ed. A. Apostolico and Z. Galil, Springer-Verlag, Berlin, pp. 257-270.

[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.

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

[SeiG] Joel Seiferas and Zvi Galil, Real-Time Recognition of Substring Repetition and Reversal, *Math. Systems Theory 11* (1977), pp. 111-146.

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