A Theory of Parameterized Pattern Matching: Algorithms and Applications (Extended Abstract)

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

ABSTRACT

This paper develops a theory and algorithms for an application problem arising in software maintenance. The application is to track down duplication in a large software system. We want to find not only exact matches between sections of code, but parameterized matches, where a parameterized match between two sections of code means that one section can be transformed into the other by replacing the parameter names (*e.g.* identifiers and constants) of one section by the parameter names of the other via a one-to-one function.

This paper formalizes this problem in terms of *parameterized strings* and *parameterized pattern matching* and defines a new data structure (*parameterized suffix tree*) suitable for parameterized pattern matching. It gives efficient algorithms for constructing this data structure, efficient algorithms for parameterized pattern matching, and an efficient algorithm for finding all maximal parameterized matches over a threshold length in a parameterized string.

The algorithms for constructing parameterized suffix trees and for reporting duplication over a threshold length have been implemented. Tests on C code indicate that these algorithms should perform well in the application.

1. Introduction

This paper takes an application problem arising in software maintenance, derives for it a theory of a new kind of pattern matching, develops techniques leading to a new data structure, derives efficient algorithms for this problem, and shows that they should work well in practice. The new pattern matching theory is aimed at matching corresponding repetitions of different symbols and thus departs from the traditional pattern matching domain of matching strings or regular expressions.

The software maintenance problem is to track down duplication in a large software system. The work was motivated by the observations that (1) a 10 million line switching system appeared to contain much duplication, apparently created when programmers made modifications using an editor to make copies, and (2) duplication is undesirable because of its well-known association with bugs. Much of the duplication is not exact, but rather what the author terms "parameterized matches", namely sections of code that match except for renaming of the parameters (*e.g.* identifiers and constants). Conceptually, two sections of code with a parameterized match are like expansions of a macro f, where $f(b_1, b_2, \cdots, b_p)$ gives the first section of code, $f(c_1, c_2, \cdots, c_p)$ gives the second section of code, the b_i's are all distinct, the c_i's are all distinct, and each b_i or c_i may occur multiple times in the macro definition. Excerpts of two 34-line sections of code that are a parameterized match in the X Window System [SchG] source are shown in Figure 1, where the two sections of code are the same except for correspondences between pfi and pfh, lbearing and left, and rbearing and 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++ = ',';
```

Figure 1. A parameterized match in code from the X Window System [SchG].

Two types of tools are needed:

(A) a tool that will locate all "interesting" parameterized duplication. Our definition of "interesting" is all pairs of code sections that are maximal parameterized matches over a threshold length specified by the user, where "maximal" means that the match cannot be extended by additional lines forward or backward in the code.

(B) a tool that will search for parameterized matches to a specific section of code.

The author wrote a program called **dup** [Bak1] for (A). Experiments with **dup** have found about 22% of the lines in a switching subsystem of over a million lines to be involved in parameterized

25th ACM STOC '93-5/93/CA,USA

duplication of at least 30 lines. **Dup** has been used to find obsolete files, instances of extensive parameterized duplication (*e.g.* 900 lines), and places where bugs were not fixed in all copies of some code.

In **dup** parameterized duplication is found as follows: all tokens that might be reasonable parameters (such as identifiers and constants) are transformed into "P", an integer "symbol" is obtained to represent each resulting line, all maximal exact matches are found for these transformed lines, and the exact matches are then checked to see which parts of them correspond to parameterized matches. While this approach produces correct output, it is possibly inefficient in that many (sometimes 99%) of the exact matches found do not correspond to parameterized matches. This possible inefficiency led to developing the theory in this paper. The author intends to use the algorithms from this paper to improve **dup** and to implement a new tool for (B).

Overview.

Parameterized strings, or *p-strings*, are strings that contain both ordinary symbols from an alphabet Σ and parameter symbols from an alphabet Π. Two p-strings are a parameterized match, or *p-match*, if one p-string can be transformed into the other by applying a one-to-one function that renames the parameter symbols. With respect to the application, each parameter symbol corresponds to an input token, obtained by a lexical analyzer, representing an item such as an identifier or constant.

In Section 2, we define a new data structure called the *parameterized suffix tree* (*p-suffix tree*), which is a generalization of suffix trees for strings [McC]. The key idea is to chain together occurrences of the same parameter so that matching parameters correspond to matching chains, and to encode these chains in the p-suffix tree. To get a linear space encoding useful for pattern matching, each symbol is interpreted dynamically in each access according to its depth in the tree. To build a p-suffix tree, the original McCreight algorithm [McC] for building suffix trees cannot be used because of the dynamic way in which p-suffix trees treat input and the fact that a key property of ordinary strings fails for the encodings of p-strings. Through analysis of the structure of p-suffix trees, we derive an algorithm *eager* in Section 4 that constructs a p-suffix tree in time and space linear in input length for fixed alphabets. For variable alphabets Σ and Π, the worst-case running time is $O(n(|\Pi|+\log(|\Sigma|+|\Pi|))$ for input of length n. In the case of variable alphabets, by augmenting the p-suffix tree with auxiliary data structures, including concatenable queues [AHU] and Sleator-Tarjan dynamic trees [ST], it is possible to improve the worst-case bounds for constructing p-suffix trees to $O(n \log n)$ time and $O(n)$ space for input of length n.

For the searching problem (B), two versions are given in Sections 2 and 5. To find p-matches of a pattern string P in a text p-string T over fixed alphabets, we can search a p-suffix tree for T using space $O(|T|)$ and time $O(|P|+k)$, where k is the number of p-matches found, or we can search a p-suffix tree for P using space $O(|P|)$ and time $O(|T|)$. The former algorithm is straightforward based on the definition of p-suffix trees; the latter algorithm adapts the corresponding algorithm for strings [CL] using ideas similar to those for constructing p-suffix trees. For variable alphabets, the worst-case running times of the above algorithms are $O(|P|\log(|\Sigma|+|\Pi|)+k)$ and $O(|T|(|\Pi|+\log(|\Sigma|+|\Pi|)))$, respectively. For the latter algorithm, where the factor of $|\Pi|$ is undesirable, we improve the running time to $O(|T|\log(|\Sigma|+|\Pi|))$ by adding auxiliary data structures for computing lowest common ancestors [HT,SV].

Section 6 describes an algorithm *pdup* that solves problem (A): given a threshold length t and a p-suffix tree for a p-string T, *pdup* finds all maximal p-matches of at least t symbols in T in time linear in the sum of the input length and number of p-matches reported and in space linear in input length. These bounds hold even if the alphabets may vary.

Section 7 describes implementations of the basic algorithm *eager* (without auxiliary data structures) for constructing p-suffix trees and the algorithm *pdup* for finding all maximal p-matches over a threshold length. Experiments indicate that these algorithms are quite practical.

In fact, experiments on p-strings of length up to about 1.9 million symbols obtained from C code indicate that the running time of *eager* may be linear in input length in the application, despite large alphabet sizes; the part of the algorithm leading to the factor of Π in the theoretical worst-case bound stated above does not exhibit this blowup in the experiments. A possible reason is that as long as the threshold length is not too small, the number of maximal p-matches over threshold length has been found to be less than the input length for p-strings obtained from code. A reanalysis of the running time of *eager* relates running time to the number of maximal p-matches over threshold length as follows: for any constant t, the algorithm runs in time $O((t|T|+m(t,|T|))\log(|\Sigma|+|\Pi|))$, where $m(t,T)$ is the number of maximal p-matches of length at least t in input T. Since the worst-case bound for *pdup* is linear in the sum of input length and amount of output, the observation that the number of maximal p-matches is less than input length unless the threshold length is small also suggests that running times for *pdup* should be close to linear in input length in practice, as long as the threshold is not too small, and experiments bear this out.

Relationship to other work.

The problem (A) of finding all maximal matches over a threshold length was first proposed in [Bak1], and the parameterized duplication algorithm *pdup* described in this paper is a generalization of the algorithm for finding all maximal exact matches over a threshold length implemented in **dup** [Bak1,Bak2]. For comparison, a brute-force approach, at least for exact matches [Ker], would be to find matches by scanning the diagonals of a matrix M, where $M[i,j]$ represents whether positions i and j match; this algorithm would take time quadratic in input length for every input.

To solve problem (A), the patterns matched must be discovered in the code itself, rather than being specified by the user, as has been standard in string pattern matching. The closest previous results have been finding the maximal common substring of two strings [Cr87], producing the minimal transducer that outputs the position of first occurrence of every distinct substring [Cr86] , Lempel-Ziv data compression [RPE,ZL], which finds some but not all maximal matches, and finding occurrences of specific patterns such as xx, where x is any substring, as in [AP,Cr86,ML,Ra].

Parameterized matching is reminiscent of unification [GeN], where the goal is to determine whether two expressions can be made equivalent via substitutions for variables, but unification differs in three ways from our problem: the domain is expressions (terms) rather than strings, terms (rather than just variables) are substituted for variables, and there is no notion of matching just part of a term.

A parameterized match is a kind of approximate match, but is very different from the standard definition in which two strings are an approximate match if they are within a specified edit distance (number of insertions, deletions, or substitutions) from each other, as studied, for example, in [CL,GG]. Even exact or approximate matches to regular expressions [Aho,MM,WM] do not involve any notion of relating repeated occurrences of corresponding (but different) symbols. A pattern p-string can be encoded as a regular expression by taking the union of patterns obtained by all possible renamings of parameters, but this gives a blowup in the size of the pattern that is exponential in the number of parameters.

The UNIX grep pattern matching program and ed editor [KP] allow a pattern to be a restricted regular expression with backreferencing to refer to parts of the text matching earlier parts of the pattern, as described in [Aho]; this problem is NP-complete [Aho], and the algorithms implemented are undocumented but are based on backtracking [Hu92]. The grep/ed usage of backreferencing is incomparable with our definitions: in our terminology, they have no way of requiring that distinct parameters should match different substrings, while we have no way of allowing them to match the same substrings.

The techniques used in this paper are related to the construction of L-suffix trees [Gi] as a search data structure for square arrays. The encodings for p-strings and arrays both exhibit the failure of the same key property of strings used by McCreight to construct suffix trees in linear time [McC], but are otherwise quite dissimilar. P-strings obey a restricted form of this property that makes possible the construction of p-suffix trees in linear time for fixed alphabets; there is no analog to this for L-suffix trees. For variable alphabets, our use of dynamic trees and concatenable queues to speed up the construction algorithm is like that of [Gi], but the auxiliary data structures that we use to speed up pattern matching cannot be used for L-suffix trees.

Four other methods have been attempted for finding duplication in code: (1) string pattern matching has been applied to strings encoding the call graph and statistics about characteristics such as use of operators to detect student plagiarism [Ja]; (2) signal processing techniques combined with a graphical user interface have been used to find approximate duplication by eye [CH]; and (3) exhaustive search was used on parse trees to identify identical subtrees or subtrees related by change of parameter but was found to be unsuccessful because of time and space usage [Jo]; and (4) data flow analysis and safe approximation techniques have been proposed as a basis for comparing program components in a restricted programming language [Ho].

2. Parameterized suffix trees.

In this section, we introduce the concepts needed to define parameterized suffix trees (p-suffix trees). We assume a RAM model of computation with the uniform cost criterion [AHU].

Throughout, Σ will be a finite alphabet of ordinary symbols and Π will be a finite alphabet of parameter symbols. We assume that Σ and Π are disjoint from each other and the set of nonnegative integers, that symbols are ordered and can be compared in constant time, and that symbols of Π can be used to index into an array in constant time. A string of symbols over $(\Sigma \cup \Pi)^*$ is called a *parameterized string* or *p-string*.

Two p-strings are a *parameterized match*, or *p-match*, if one p-string can be transformed into the other by applying a one-to-one function that renames the parameter symbols. For example, if $\Pi = \{x,y,v\}$, and $\Sigma = \{a,b,c\}$, then $S = axaybxycby$ and $T = ayavbyvcbv$ are a p-match, where x and y of S are renamed as y and v, respectively, in T.

Determining whether two entire p-strings are a p-match is straightforward, as follows. Scan the two p-strings left to right, while constructing a table giving the one-to-one correspondence, to see if any mismatches are found between symbols. A mismatch is either in length or between different non-parameters, between a parameter and a non-parameter, or between two parameters, at least one of which has already been made to correspond

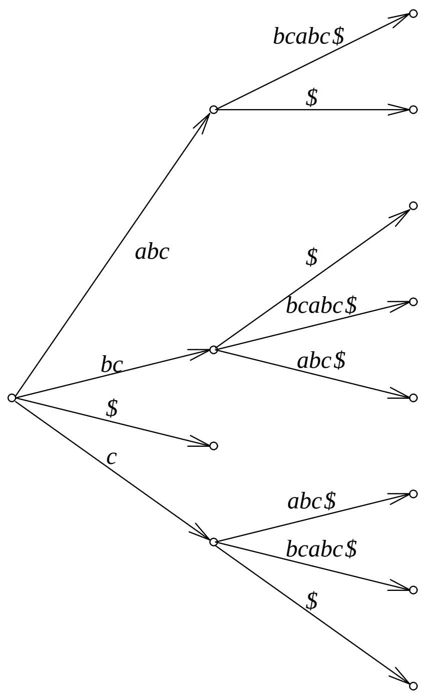

Figure 2. A suffix tree for the string *abcbcabc$*.

to a different parameter. Given our definitions, this can be done in time linear in input length n and space $O(n + |\Pi|)$. Unfortunately, it does not generalize conveniently to pattern matching.

Instead, we use a procedure *prev*, that chains together occurrences of the same parameter, to obtain a string in $(\Sigma \cup \mathrm{N})^*$, where N is the set of nonnegative integers. For each parameter, the leftmost occurrence is represented by a 0, and each successive occurrence is represented by the difference in position compared to the previous occurrence. A number representing such a difference in position is called a *parameter pointer*.

For example, if $\Sigma = \{a,b\}$ and $\Pi = \{u,v,x,y\}$, then $prev(abuvabuvu) = ab00ab442 = prev(abxyabxyx)$. (Each parameter pointer is a single digit here.)

Proposition 1. P-strings S and $\overline{S}$ are a p-match if and only if $prev(S) = prev(\overline{S})$.

For a p-string S, let S_i denote the ith symbol in S. The ith *p-suffix* of a p-string S of length n is $psuffix(S,i) = prev(S_i S_{i+1} \cdots S_n)$. Note that a symbol of $prev(S)$ corresponds to a different value in $psuffix(S,i)$ if it is a parameter pointer that points to a position before i.

Proposition 2. If P is a p-string pattern and T is a p-string text, P p-matches at position i of T iff $prev(P)$ is a prefix of $psuffix(T,i)$.

We also note that the value of the jth symbol of $psuffix(S,i)$ can be computed in constant time from j and the corresponding symbol $b = S_{j+i-1}$ of $prev(S)$. We let f be this function: if b is a parameter pointer larger than $j-1$, $f(b,j) = 0$; otherwise $f(b,j) = b$.

Our strategy is to generalize suffix trees for strings based on proposition 2 and the function f. First, we briefly review suffix trees.

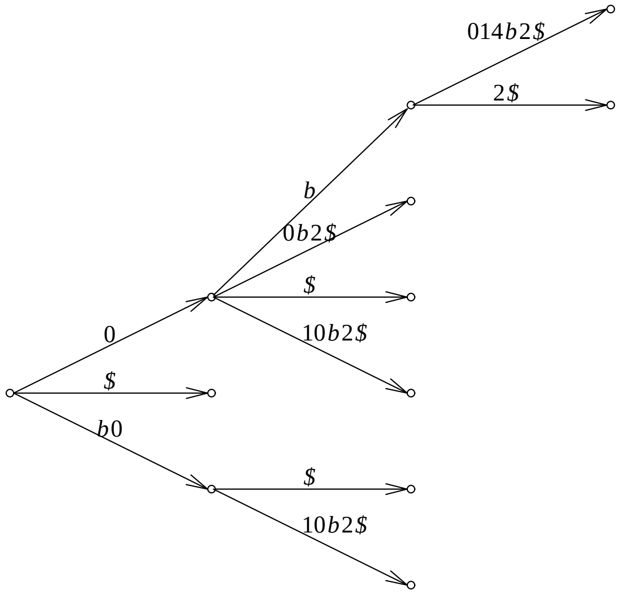

Figure 3. A p-suffix tree for the p-string $S = xbyyxbx\$$, where $\Sigma = \{b,\$\}$ and $\Pi = \{x,y\}$.

Suppose that Σ is an alphabet and $S = S_1 S_2 \cdots S_n$ is a string over Σ. For each i, $1 \le i \le n$, the substring $S_i \cdots S_n$ is a *suffix* of *S*. We assume that Σ contains a special endmarker *$* that is used only to mark the end of strings. A suffix tree for a string *S* is a compacted trie (multiway Patricia trie) representing the suffixes of *S* [McC], provided that *S* ends in *$* (if not, append it and use *S$*). A suffix tree is shown in Figure 2. Each arc of the tree is labelled with a nonempty substring of the input. For each leaf, the sequence of arcs from the root to the leaf is a suffix of *S*. Because of the endmarker, no suffix is a prefix of another suffix, and there is a one-to-one correspondence between the leaves and the suffixes. Each node other than a leaf has at least two children; hence the number of nodes is linear in *n*.

Similarly, we define a p-suffix tree for a p-string *S* to be a compacted trie that stores the p-suffixes of *S*, assuming that *S* ends in *$* (if not, append it and use *S$*). Each arc represents a nonempty substring of a p-suffix, each nonterminal node has degree at least two, and each arc out of a node has a distinct starting symbol. For each node, let the *pathstring* be the sequence of labels on the path from the root to that node; the length of the pathstring is the *pathlength*. For each leaf, the pathstring is a distinct p-suffix of *S*. Thus, for a p-string of length *n*, the number of leaves is *n*, and the number of nodes in the p-suffix tree is linear in *n*.

Example. Let $S = xbyyxbx\$$, where $\Sigma = \{b,\$\}$, $\Pi = \{x,y\}$, and $prev(S) = 0b014b2\$$. (All parameter pointers are single digits here.) 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 3.

Arc labels are calculated dynamically as follows. We assume that $prev(S)$ has been computed and stored in an array. For each node other than the root, we store the pathlength and indices into the input representing the starting and ending positions of the label of the arc from its parent. If a label symbol is at pathlength *j* from the root and corresponds to index *k* into the input, its value is $f(b,j)$, where *b* is the *k*th symbol of $prev(S)$, and *f* is as defined after the description of *prev*; extracting each actual arc symbol takes constant time using the stored information.

The number of children of each nonterminal is bounded by $|\Sigma| + |\Pi|$, for the following reason. The pathstring of any node contains parameter pointers representing at most $|\Pi|$ distinct chains of parameters. The first symbol of an arc to a child must either be a symbol of Σ, a 0 (implying that at most $|\Pi| - 1$ chains appear in the pathstring) or a nonnegative integer pointing to the last element of one of these chains.

For each node, the arcs to its children (with one exception) are stored in a balanced tree scheme [AHU] so that if there are *k* arcs, an arc beginning with a particular symbol can be accessed, inserted, or deleted in $O(\log k)$ time, and $O(k)$ time is sufficient to access the *k* children in succession. The exception is that the arc (if any) whose label begins with 0 is stored so that it can be accessed in $O(1)$ time.

Thanks to Proposition 2, searching the p-suffix tree of a text p-string *T* for a pattern p-string *P* is straightforward: follow the path determined by successive symbols of $prev(P)$ from the root downward in the p-suffix tree for *T* to see if $prev(P)$ is identical to the first part of some p-suffix of *T*. This search can be accomplished in time $O(|P|\log(|\Sigma| + |\Pi|))$. The actual matching positions can be calculated from the descendant leaves. Thus, we obtain the following result.

Lemma 1. Given p-strings *P* and *T* and a p-suffix tree for *T*, whether *P* has a p-match with a substring of *T* can be determined in time $O(|P|\log(|\Sigma| + |\Pi|))$ and space $O(T)$, and all positions of *T* at which *P* has a p-match can be found in time $O(|P|\log(|\Sigma| + |\Pi|) + k)$ and space $O(T)$, if there are *k* such positions.

3. Augmenting a p-suffix tree with suffix links.

Strings have the following two trivial properties, where *S* and *T* are strings and a,b,c,d are symbols:

(1) (Common Prefix Property) if $aS = bT$, then $S = T$, and

(2) (Distinct Right Context Property) if $aS = bT$ and $aSc \ne bTd$, then $Sc \ne Td$.

Because of these two properties, it is possible to augment a suffix tree with pointers called *suffix links* as follows. If an internal node has pathstring *aT*, where *a* is a symbol and *T* is a string, then its suffix link points to an internal node with pathstring *T*; in addition, the suffix link for the root points to the root. Suffix links are useful both for building a suffix tree [McC] and for pattern matching in space proportional to the size of the pattern [CL].

The Common Prefix Property generalizes to p-strings, but unfortunately the Distinct Right Context Property does not.

Proposition 3. (Common Prefix Property for p-strings) Let *S* and *T* be p-strings and $a,b \in \Sigma \cup \Pi$. If $prev(aS) = prev(bT)$, then $prev(S) = prev(T)$.

The Distinct Right Context property fails to generalize to p-strings, because *prev* turns nonnegative integers into 0's as the front end of the string is chopped off. For example, suppose $S = xabxyabz$, with $\Sigma = \{a,b\}$ and $\Pi = \{x,y,z\}$. Then $prev(xabx) = 0ab3$ and $prev(yabz) = 0ab0$, which have a common prefix of $0ab$, but $prev(abx) = ab0 = prev(abz)$, and the distinctness of the right contexts of $0ab$ is lost.

The best we can do is the following Restricted Distinct Right Context Property, which encapsulates exactly where the Distinct Right Context property fails for p-strings.

PROPOSITION 4. (Restricted Distinct Right Context Property for p-strings) Let S and T be p-strings and $a,b,c,d \in \Sigma \cup \Pi$. If $prev(aS) = prev(bT)$ and $prev(aSc) \neq prev(bTd)$, then $prev(Sc) \neq prev(Td)$ unless the last symbol of one of $prev(aSc)$,$prev(bTd)$ is k while the other is 0, where $k = |aS| = |bT|$.

The consequence of the failure of the Distinct Right Context Property to generalize is that we need to allow more leeway in the definition of suffix links. Suppose $S = S_1 \cdots S_n$ is a p-string. Let N be an internal node with pathstring $prev(S_i \cdots S_j)$. Whereas we wish we could define the suffix link $SL(N)$ for N to point to a node with pathstring $prev(S_{i+1} \cdots S_j)$, the best we can do is to define $SL(N)$ to point to the node for which the pathstring is the shortest extension of $prev(S_{i+1} \cdots S_j)$ that exists as a pathstring in the tree. Fortunately, such a node always exists, as a consequence of the Common Prefix Property for p-strings and the unique endmarker.

If $SL(N)$ of the previous paragraph points to a node with pathstring $prev(S_{i+1} \cdots S_j)$ as desired, we say it is a *good* suffix link; otherwise we say it is *bad*. (By this terminology, all suffix links as originally defined for suffix trees are good.)

Example. The suffix link for the node with pathstring $0b$ in Figure 3 will be bad, because there is no node with pathstring b; the suffix link will point to the node with pathstring $b0$.

The algorithms in this paper are designed around the following characterization of where bad suffix links occur in a p-suffix tree. This characterization is derived from the Restricted Distinct Right Context Property for p-strings and the fact that there cannot be two distinct parameter pointers in a p-suffix pointing directly to the first symbol in the p-suffix.

LEMMA 2. Let N be a node. In the p-suffix tree for S, if $N_1, N_2, \cdots, N_k$ are all the nodes N_j with bad suffix links for which $SL(N_j) = N$, then they occur in a chain of nodes connected by arcs whose labels all begin with 0.

4. Constructing a p-suffix tree.

We would like to follow the McCreight algorithm [McC] for constructing suffix trees as much as possible, since it runs in linear time for a fixed alphabet. To build a suffix tree for a string S, McCreight's algorithm inserts suffixes in stages, where Stage i inserts suffix i (the suffix starting at position i of the input). Define $head_i$ to be the longest prefix of the ith suffix of S that is also a prefix of the jth suffix of S for some $j < i$. The path for suffix i in the tree coincides with an existing path up to $|head_i|$; the algorithm inserts a new node hd_i at that point, if no node exists there already, and gives it a child that is a leaf whose pathstring is suffix i.

Suffix links are McCreight's key to turning this idea into an efficient algorithm. Suppose Stage $i-1$ found that $head_{i-1}$ was $a\alpha$, where a is a symbol and α a string, and this is the common prefix of suffix $i-1$ and suffix j, $j < i-1$. By the Common Prefix Property, α will be a common prefix of suffix i and suffix $j+1$, implying that $head_i$ will be at least as long as $|\alpha|$. In Stage i, if the suffix link is already defined for the node with pathstring $a\alpha$, there is no need to trace the prefix α of $head_i$ from the root; the processing can jump directly from the node with pathstring $a\alpha$ to the node with pathstring α via a suffix link, and can continue downward from there looking for the right place to put hd_i.

Unfortunately, the suffix links are constructed dynamically and the desired suffix link may not actually be defined until after it would be most useful in Stage i. McCreight's algorithm gets around this obstacle as follows. In Stage i, it uses the best suffix link available, namely that of the parent of the node with pathstring $a\alpha$, and follows a path downward in the tree while *rescanning* part of the input, up to a pathlength of $|\alpha|$, which is the point that the undefined suffix link needs to be set to. Fortunately, we know what input symbols were already scanned, just not where to go in the tree. Thus, only the first symbol of each arc needs to be rescanned; this fact is crucial for the analysis of running time.

In generalizing this algorithm to p-suffix trees, there are three difficulties resulting from bad suffix links and the failure of the Distinct Right Context Property:

(1) suffix links are defined in terms of current tree structure, which changes as nodes are added,

(2) bad suffix links don't point exactly where we want them to, and

(3) when we use the "best available" suffix link, as described above for McCreight's algorithm, we may have to jump farther back in the input than we would have to if the Distinct Right Context Property held, and this results in rescanning more symbols.

For (1), there is a choice between lazy evaluation and eager evaluation (*i.e.* keeping suffix links up-to-date). We choose the latter approach here. Lazy evaluation results in an algorithm that runs in worst-case linear time for fixed alphabets [Bak3], but eager evaluation has the advantages of being amenable to improvements in worst-case running time for variable alphabets and yielding up-to-date suffix links that can be used in pattern-matching (Section 5). If suffix links are kept up-to-date, (2) is not a problem, because if a bad suffix link points to a node N, the desired point lies on the arc from the parent of N to N, and no searching is needed in the tree. (3) is not a problem in terms of deriving a correct algorithm, but is potentially a problem in terms of running time; the running time issue is addressed after the description of the algorithm.

Let S be a p-string. To construct a p-suffix tree for S, our algorithm imitates McCreight's algorithm by inserting the ith p-suffix in Stage i, for $1 \leq i \leq n$. Define $head_i$ to be the longest prefix of the ith p-suffix that is also a prefix of p-suffix j, for some $j < i$. For each i, exactly one leaf is created in stage i, with pathstring $psuffix(i)$, and if no internal node with pathstring $head_i$ already exists, it is created in Stage i. The following properties will hold at the end of Stage i. (1) The p-suffix tree is a compacted trie for the p-suffixes from 1 to i. (2) Except for the (at most one) new internal node created in Stage i, each internal node N has a suffix link defined in terms of the nodes currently existing in the tree. That is, if N has pathstring $prev(S_i \cdots S_j)$, then $SL(N)$ points to the node for which the pathstring is the shortest extension of $prev(S_{i+1} \cdots S_j)$ that exists as a pathstring in the current tree. (3) For each node N other than the root, there is a pointer $min(N)$ that points to the node of smallest pathlength with a bad suffix link pointing to N, if such a node exists, and otherwise either points to a node with a good suffix link to N or is undefined.

The main procedure *eager* for building a p-suffix tree operates as follows, assuming that $prev(S)$ has already been computed. The p-suffix tree is initialized to a root with a suffix link to itself, *oldhd* is initialized to the root, and *oldchild* is undefined. Stage i, $i = 1,2,...,n$, has six phases. *Phase 1*: This phase merely sets (a possibly temporary) value for $SL(oldhd)$ if it is undefined. In this case, *oldhd* is not the root, and the pathstring of *oldhd* is $head_{i-1} = prev(S_{i-1} \cdots S_k)$ for some $k \geq i-1$. Starting at $SL(parent(oldhd))$, *eager* searches downward in the tree to find the node closest to the root whose pathstring is an extension of $prev(S_i \cdots S_k)$, and sets $SL(oldhd)$ (at least temporarily) to point to this node; only the first symbol of each arc needs to be checked to determine the path to follow downward in the tree, as in the rescanning phase of McCreight's algorithm. *Phase 2*: *eager* scans downward symbol by symbol, starting at the root if *oldhd* is the root, and otherwise starting at the point on the arc entering $SL(oldhd)$ whose pathlength is one less than the pathlength of *oldhd*, then following the path in the tree determined by successive transformed input symbols, and continuing until the next transformed input symbol $f(b,p)$, where b is a symbol of $prev(S)$ and p is the pathlength, is not available as the next symbol on an arc. This corresponds to the scanning phase of McCreight's algorithm. *Phase 3*: Let g denote the node entered by the last arc for which a transformed input symbol was matched to a symbol in the label in Phase 2. If the entire label was matched, the pathstring of g is $head_i$, so *newhd* is set to g and no new internal node is created. Otherwise, the arc from the parent of g to g is split by inserting a new node *newhd* numbered i as the parent of g, such that the pathstring of *newhd* ends at the last matched symbol; the pathstring of *newhd* is $head_i$. In the latter case, the procedure $update(g, newhd, oldchild, i)$ described in the next two paragraphs is called to update the suffix links pointing to g that should now point to *newhd* and to set $min(newhd)$ and $min(g)$. If the pathlength of *newhd* is less than that of $SL(oldhd)$, $SL(oldhd)$ is reset to *newhd*. ($SL(oldhd)$ must be reset separately since *oldhd* may not be in the chain of nodes with bad suffix links to g, because of pathlength or because *oldhd* has a good suffix link). *Phase 4*: If $min(SL(oldhd))$ is undefined or its pathlength is greater than that of *oldhd*, set $min(SL(oldhd))$ to *oldhd*. *Phase 5*: *newhd* is given a new child that is a leaf with pathstring $psuffix(S,i)$. *Phase 6*: *oldhd* is reset to *newhd* and *oldchild* is set to g.

To obtain our procedure *update*, we would like to make use of Lemma 2. If no new internal node was created in Stage $i-1$, Lemma 2 holds at the start of Stage i. Unfortunately, if Stage $i-1$ created a new internal node, Lemma 2 may not hold at the start of Stage i. In particular, if Stage $i-1$ created an internal node *oldhd* with pathstring $prev(S_{i-1} \cdots S_j)$, Lemma 2 holds at the start of Stage i for all nodes N except for the node N_e that is the shortest extension of $prev(S_i \cdots S_j)$; node N_e is the node that the suffix link of *oldhd* should point to in terms of the tree structure at the start of Stage i. For N_e, the nodes with bad suffix links pointing to N_e are still in a chain as described by Lemma 2, except that an extra node *oldhd* may lie in the middle. The node *oldhd* is identifiable if it exists because it is numbered $i-1$. The next node below *oldhd* in the chain is the node *oldchild* saved at the end of Stage $i-1$.

Consequently, for nodes N, *newhd*, and *oldhd* and integer j, the procedure $update(N, newhd, oldchild, i)$ operates as follows to update suffix links and set $min(N)$ and $min(newhd)$. If $min(N)$ is undefined or the pathlength of $min(N)$ is greater than one more than the pathlength of *newhd*, $min(newhd)$ is left undefined and $min(N)$ is unchanged. Otherwise, starting at $min(N)$, as long as the pathlength of the current node is at most one more than the pathlength of *newhd*, *update* searches downward, following arcs whose labels begin with 0, except that if it reaches an internal node numbered $i-1$, it goes to *oldchild* instead; at each internal node along the way, it resets the suffix link to *newhd*. After the loop, $min(N)$ is reset to point to the last node reached (of pathlength greater than one more than the pathlength of *newhd*) if it is defined and not a leaf and its suffix link points to N, and to undefined otherwise.

For lack of space, further details and a proof of correctness are omitted. We give a brief analysis of the running time as follows. For Phase 1 of Stage i, except for the first input symbol rescanned, only input symbols transformed into 0's by the transform function f in Stage i can have been previously rescanned. The proof is by applying the Restricted Distinct Right Context Property for p-strings inductively to show that for other symbols, a good suffix link would have existed contradicting the need for rescanning the symbol. Since the number of 0's in a pathstring is at most $|\Pi|$, at most $|\Pi|$ symbols transformed into 0's are rescanned in each stage, with $O(1)$ arc access time for each one, plus at most one additional symbol with $O(\log|\Sigma|+|\Pi|)$ arc access time. Thus the total time spent over all stages on rescanning is $O(n(|\Pi|+\log(|\Sigma|+|\Pi|)))$, where n is the input length. The time used in the second phase of each stage, scanning, is $O(n\log(|\Sigma|+|\Pi|))$ over all stages. For the third phase, involving updating bad suffix links, we note that the length of each chain is at most $|\Pi|+1$ and accessing each arc whose label begins with 0 takes time $O(1)$, and conclude that at most $O(|\Pi|)$ steps are used in each stage to update suffix links. Finally, the remaining phases of each stage take $O(1)$ time. Combining these observations yields the following theorem.

THEOREM 1. Given disjoint alphabets Σ and Π and a p-string S of length n over these alphabets, the algorithm *eager* constructs a p-suffix tree for S in $O(n(|\Pi|+\log(|\Sigma|+|\Pi|)))$ time and $O(n)$ space.

These bounds are linear in input length for fixed alphabets. For variable alphabets, we would like to avoid the factor of $|\Pi|$. The running time can be improved by auxiliary data structures: Sleator-Tarjan dynamic trees [ST] and concatenable queues [AHU].

First, we describe the use of concatenable queues. Suffix links pointing directly to a node in the original construction will be replaced by pointers pointing indirectly through a concatenable queue. That is, for each node N, there will be a concatenable queue whose root has an associated pointer to N. Wherever the original construction would set $SL(\overline{N})=N$ for a node $\overline{N}$, the new construction inserts a new leaf into the concatenable queue for N in time $O(\log n)$, and $\overline{N}$ gets a pointer to the leaf instead. When the suffix link is to be used, $O(\log n)$ time is sufficient to find the root of the concatenable queue and the associated pointer. The procedure *update* called when node *newhd* is made the parent of N is implemented in time $O(\log n)$ by splitting the concatenable queue for N based on the pathlength of *newhd*. Finally, in addition to maintaining these indirect suffix links for internal nodes, the new construction will maintain an indirect suffix link for each leaf; the suffix link for the leaf with pathstring $psuffix(i)$ will point indirectly to the leaf with pathstring $psuffix(i+1)$, and is set when the latter leaf is created. Thus, over all stages, the time required to maintain the concatenable queues is $O(n\log n)$.

Next, we describe how dynamic trees are used to avoid rescanning. A dynamic tree has edges labeled as either dashed or solid; solid edges occur only in paths, and each solid edge path is maintained as a locally biased binary tree. A dynamic tree permits m operations of *insert*, *link*, *cut*, and *expose* in $O(m\log k)$ time, where k is the number of nodes in the trees. Except for briefly when a new node is being inserted into the p-suffix tree, our forest of dynamic trees will consist of a single tree. The *insert*, *link*, and *cut* operations allow a new node to be inserted as a parent of an existing node; a new leaf can be inserted by means of a *link* operation. The key property of dynamic trees for our purposes is the *expose* operation; $expose(N)$ produces a locally biased binary tree that represents the path from N to the root. Where the original construction follows a suffix link of the parent of *oldhd* and rescans to find the node that should become $SL(oldhd)$, the new construction finds a child of *oldhd* that already has an indirect suffix link defined (such a child must exist since *oldhd* has at least two children, and at most one was created in the last stage), follows the indirect suffix link to node N, calls $expose(N)$, and then does a binary search on the resulting locally biased binary tree based on the pathlength of *oldhd* to find the node to which *oldhd* should get an indirect suffix link.

Over all stages, there are $O(n)$ calls to *insert*, *link*, *cut*, and *expose*, and the tree has at most $2n$ nodes. Hence, the dynamic tree operations take $O(n \log n)$ time.

These two improvements yield the following theorem.

THEOREM 2. Given finite disjoint alphabets Σ and Π, a p-suffix tree can be built for a p-string of length n in space $O(n)$ and time $O(n \log n)$.

5. More on searching for a pattern.

In this section, we reconsider how to search for p-matches of a pattern p-string P in a text p-string T. In contrast to Lemma 1, which uses a p-suffix tree for T, and hence $O(|T|)$ space, we show here how to search using a p-suffix tree for P and $O(|P|)$ space, an important difference in practice.

Our algorithm generalizes the algorithm of [CL] that finds all occurrences of a pattern P in a text string T by means of a suffix tree for P (technically, $P\$$) augmented with suffix links. The basic idea for strings is to follow the path determined by symbols of T through the tree, starting at the root and the first symbol of T. Whenever the next symbol of T is not available in the tree, follow the suffix link from the last node reached, rescan downward using arc lengths as in McCreight's algorithm to catch up to the last symbol of T already matched, and continue following the path downward determined by symbols of T. Whenever the current position in the tree corresponds to matching the last symbol of P before the endmarker, report a match.

In generalizing this algorithm to p-strings, the problems to be solved are analogous to those solved for building the p-suffix tree, without the need to update suffix links. Let $T = T_1 T_2 \cdots T_n$ be the text p-string and P the pattern p-string. Assume that the p-suffix tree has been built for the p-string P (technically, $P\$$), and that suffix links are available. By the Common Prefix Property for p-strings, if $T_i T_{i+1} \cdots T_q$ is a prefix of a pathstring in the p-suffix tree for P, so is $T_{i+1} \cdots T_q$, and we can use the existing suffix links when processing T.

In the ith stage, $1 \le i \le |T|$, the algorithm compares the ith p-suffix of T to the p-suffixes of S. For $i = 1$, initialize the algorithm with $N = root$, and $len = 1$; in general, at the start of a stage, N represents the node to begin searching from, and len represents the pathlength corresponding to the next text symbol that needs matching. In Stage i, this will be symbol $T_{i+len-1}$; the algorithm will guarantee that the pathstring $prev(T_i \cdots T_{i+len-2})$ terminates either on the arc from $parent(N)$ to N or below N. In the latter case, rescanning of $T_i \cdots T_{i+len-2}$ by comparing the first symbols of arcs to the text symbols (transformed by f) is used to get to a point in the tree corresponding to pathstring $prev(T_i \cdots T_{i+len-2})$ and pathlength $len - 1$. Starting with $j = len$, the algorithm tries for successive values of j to match the transformed jth symbol of the ith p-suffix of T to a symbol at pathlength j in the tree; the transformed jth symbol is $f(prev(T)[i+j-1], j)$, where f is as defined in Section 2. Thus, the algorithm follows a path downward in the tree, matching transformed input symbols against symbols in the pathstring. If it passes through the symbol corresponding to the last symbol of P (not counting the endmarker), a p-match is reported. When the jth symbol of the ith p-suffix of T is not available in the tree, no longer prefix of this p-suffix matches any prefix of a p-suffix of P. In this case, if the algorithm has just reached a node at pathlength $j-1$, call it $\bar{N}$; otherwise let $\bar{N}$ be the parent of the node reached by the arc currently being examined. Set $N = SL(\bar{N})$ and increment i; if $j = 1$, set $len = j$, and otherwise, set $len = j - 1$.

For lack of space, we omit a proof that this algorithm is correct, and only give a brief analysis of running time. Each symbol is first scanned or first rescanned at most once each, with $O(\log n)$ access time for each arc. Other than at the first rescanning step in each stage, a symbol can be rescanned an additional time only if the corresponding symbol in an arc label is a 0, for the same reason as in the construction of p-suffix trees (inductive application of the Restricted Distinct Right Context Property for p-strings). In each stage, at most $|\Pi|$ such symbols can be rescanned with $O(1)$ access time for the corresponding arc, plus at most one symbol with $O(1)$ access time for the corresponding arc.

THEOREM 3. Let Σ and Π be finite disjoint alphabets, and let P be a pattern p-string and T a text p-string. Given a p-suffix tree for P with suffix links, the positions in T where P has p-matches can be determined in $O(|T|(|\Pi| + \log(|\Sigma| + |\Pi|)))$ time and $O(|P|)$ space.

The running time is linear in $|T|$ if the alphabets are fixed. We would like to eliminate the factor of $|\Pi|$ from the running time for variable alphabets. Since the p-suffix tree is static for this searching problem, we can take advantage of the chain structure for bad suffix links and lowest common ancestor algorithms on static trees to do even better than we could if we used dynamic trees again. From [HT] or [SV], it is possible to construct auxiliary data structures in $O(|P|)$ preprocessing time, so that the lowest common ancestor (LCA) of two nodes can be computed in $O(1)$ time. We use their auxiliary data structures for the p-suffix tree, and in addition, we create an array for each chain of arcs whose labels begin with 0. This preprocessing takes time $O(|P| \log(|\Sigma| + |\Pi|))$.

For the new algorithm, we assume that suffix links have been created for leaves as well as for internal nodes, which is easily done since the suffix link for the leaf created in Stage i must point to the leaf created in Stage $i+1$. The new algorithm is the same as before, except that during rescanning, whenever we reach a chain of 0's, we skip directly to the last point in this chain that is an ancestor of the node we are seeking. In particular, when we follow the suffix link for an internal node N preparatory to rescanning, we also remember a node $\bar{N}$ pointed to by a suffix link for a child of N, because we want to rescan to a node that is an ancestor of $\bar{N}$. In rescanning, whenever we reach a chain of 0's, we find the LCA of the bottom of the chain and $\bar{N}$, and if the desired pathlength is below the LCA, we jump right to the LCA, and if not, we use binary search on the array representing the chain to find the desired node. Thus, $O(\log|\Pi|)$ time is used at each chain, and $O(\log(|\Sigma| + |\Pi|)$ time at each intervening arc. As before, over all stages, the total number of arcs accessed for rescanning whose labels do not begin with 0 is linear in input length. This yields the following result.

THEOREM 4. Let Σ and Π be finite disjoint alphabets and let P be a pattern p-string and T a text p-string. Given a p-suffix tree for P with suffix links, all positions of T where P has p-matches can be reported in space $O(|P|)$ and time $O(|T| \log(|\Sigma| + |\Pi|))$.

6. Finding all parameterized duplication

This section gives an algorithm *pdup* for finding all pairs of maximal p-matches in an p-string by recursing over a p-suffix tree.

We say that two substrings of a p-string are a *maximal p-match* if they are a p-match and the p-match cannot be extended in either or both directions. In fact, it is sufficient to check whether the p-match can be extended by one symbol in either direction; the proof is omitted for lack of space. If the two substrings start at positions i and j, respectively, $i \neq j$, and have length k, we denote this maximal p-match by (i, j, k).

Let us consider how p-suffix trees relate to maximal p-matches. The existence of an internal node N in a p-suffix tree means that its pathstring can be followed by distinct symbols in at least one pair of distinct p-suffixes, implying the existence of at least one p-match that cannot be extended to the right. If we dealt only with symbols of Σ, such a p-match would maximal iff the preceding symbols are also different.

However, for p-strings, it is not enough just to compare the preceding symbols, because the value of a parameter pointer depends on how much preceding context there is. If, for example, a 0 occurs in the common prefix, it could mean different things in the two p-suffixes. In one, it might correspond to the next occurrence of a parameter immediately preceding this p-suffix, while in the other, it might refer to an earlier parameter or represent the first occurrence of a parameter. In this situation, checking just the symbol to the left is not enough to determine whether the two p-strings are a maximal p-match. It is, however, sufficient to check these two symbols, and if they are both parameters, check whether the next occurrences, if any, of these symbols in the common prefix are in corresponding positions.

To do this efficiently, we construct another string, $(prev(S^r))^r$, where the superscript r represents reversal; here the parameter pointers point to the next symbol to the right rather than to the next symbol to the left. This string can be constructed in linear time from $prev(S)$. By applying a transform function analogous to f of Section 2 to each left context taken from this string, we need check only the left context positions for equality.

The strategy of *pdup* is to recurse over the p-suffix tree, building up lists of p-suffixes categorized by this transformed left context. A p-list is a list of starting positions of p-suffixes with the same transformed left context; a c-list is a list of p-lists. The algorithm recursively constructs a c-list for each node of pathlength at least threshold length, such that the elements of the p-lists correspond to the leaves in the subtree rooted at that node. For each node N, the strategy is to accumulate the c-lists of the children of N from left to right; at each child c, two cross-products are taken of the c-list representing all children to the left of c and the c-list for c. In the first cross-product, maximal matches are reported for the cross-product of any pair of p-lists with different transformed left contexts; the fact that the cross-product of pairs of p-lists with different transformed left contexts corresponds to maximal matches is the key to the running time analysis. In the second cross-product, the c-list for c is merged with the c-list for the siblings to the left. Since the left contexts of the plists are obtained by a transformation function applied to the input and pathlength, the values of the left contexts may be different when evaluated at N and at the parent of N; hence, before a c-list for N is used in processing its parent, p-lists whose left contexts become 0 for the parent must be merged.

THEOREM 5. Given disjoint alphabets Σ and Π, a constant t, a p-string S of length n, and a p-suffix tree for S, *pdup* finds all pairs of parameterized matches of length at least t in time $O(n + m(t, S))$ and space $O(n)$, where $m(t, S)$ is the number of p-matches reported.

7. The software maintenance application

The p-suffix tree construction algorithm *eager* (with no auxiliary data structures) of Section 4 and the duplication algorithm *pdup* of Section 6 have been implemented. This section describes the implementation, shows the program to be surprisingly fast in the application, and explores the reasons in terms of observations about the input and another theorem reanalyzing the running time of *eager*.

The program is implemented in C and runs under UNIX (TM). The implementation uses a semi-line-based approach as follows. Comments and white space are ignored. Tokens such as identifiers and constants are turned into parameter symbols by the lexical analyzer via hashing. In addition, each line, with parameter tokens replaced by a single symbol P, is transformed by the lexical analyzer into a symbol of Σ via hashing. Thus, from each line, the lexical analyzer will generate a parameterized string consisting of one symbol of Σ followed by a (possibly empty) sequence of parameter symbols corresponding to the parameter tokens in the order in which they occur in the original line. Thus, the parameterized string for the entire program will have n symbols of Σ and p symbols of Π if the entire program had n lines other than comments and white space and p parameter tokens. Accessing arcs is done by hashing (as suggested by McCreight [McC] for ordinary suffix trees) rather than by the balanced tree scheme used in the theoretical analysis of worst-case running time.

Experiments were done on various samples of C code taken from four subsystems of a 10-million-line switching system, developed by thousands of programmers. The performance is quite practical. For example, lexical analysis of a complete subsystem of 1085432 lines (604532 lines after removing comments and white space), resulted in a p-string of length 1931508, with $|\Sigma| = 18767$ and $|\Pi| = 59827$, and took about 190 seconds. On this p-string, the time for building a p-suffix tree using *eager* was about 100 seconds. With a threshold of 150, *pdup* found 6620 maximal p-matches and took about 110 seconds, for a total time of about 400 seconds for the program (including lexical analysis and building the p-suffix tree); with a threshold of 30, *pdup* found 537928 maximal p-matches and took about 130 seconds, for a total time of about 420 seconds. These experiments were run on one processor of an SGI machine with four 40 MHz R3000 processors, data and instruction caches of 64 Kbytes, a secondary data cache size of 1 Mbyte, and main memory size of 256 Mbytes.

Theorem 1 gives a bound of $O(n(|\Pi| + \log(|\Sigma| + |\Pi|)))$ time for *eager*, where n is input length. The $n|\Pi|$ comes from two places in the algorithm: rescanning arcs whose labels begin with 0 and updating suffix links. However, experiments have found the number of such rescanning steps and the number of update steps in *eager* to be less than n for all tests on segments of code of varying sizes from the 10-million line switching system. For example, on the complete subsystem mentioned above, resulting in an input p-string of length 1931508, the total number of rescanning steps on arcs whose labels begin with 0 was 220587, with a maximum of 18 within any stage, and the total number of update steps was 9543, with a maximum of 4 within any one stage. Since the log factor arising from arc access is avoided by hashing, it appears that *eager* actually runs in time linear in input length on C code in this application, despite the large alphabet sizes.

In an attempt to explain why the observed running time is faster in this application than predicted by Theorem 1, we reanalyze the running time of *eager* as follows.

THEOREM 6. Given disjoint alphabets Σ and Π, input T of length n, and an integer t, the algorithm *eager* constructs a p-suffix tree for T in worst-case time $O((tn + m(t, T)) \log(|\Sigma| + |\Pi|))$, where $m(t, T)$ is the number of maximal p-matches of length at least t in T.

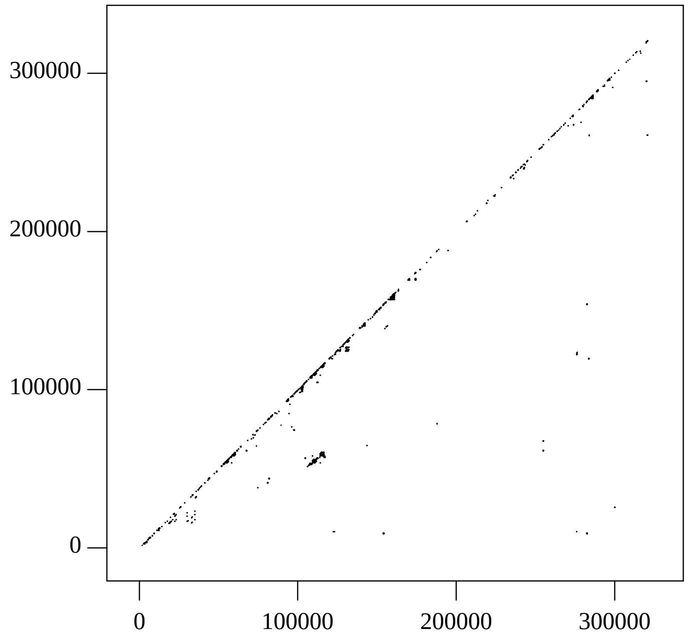

Figure 4. A scatter plot of maximal p-matches of length at least 75 symbols in about 101,000 lines of C code (excluding comments), corresponding to a p-string of about 321,000 symbols.

The basic idea of the proof is as follows. We show that each rescanning of an input symbol, except for the first one and last one in each stage, is caused by the branching due to a different maximal p-match of the current p-suffix being processed and a p-suffix previously processed. Similarly, each update of a suffix link, except for the first and last in each stage, corresponds to a different maximal p-match of the current p-suffix being processed and a p-suffix previously processed. These arguments require reasoning about chains of suffix links, as opposed to chains of arcs whose label begins with 0. Longest p-matches of length less than *t* in this argument account for the *t* in the time bound.

Now, the time bound of Theorem 6 is quadratic in input length in the worst case, even for fixed alphabets, since $m(t,T)$ can grow quadratically in $|T|$in the worst case. However, experiments with *eager* and *pdup* and data from **dup** [Bak1] suggest that in practice, $m(t,T) < |T|$, unless t is small. (Since the definitions of maximal match and threshold length in **dup** are line-based, rather than symbol-based as in this paper, experiments were run with *eager* and *pdup* to determine the value of *t* at which *m* exceeds *n* for a sample of 158579 lines of code, and the experiments revealed that $m(t,T) < |T|$for $t \geq 20$.) Thus, this alternative analysis suggests a reason that the observed running time of *eager* (using hashing for arc access) should be linear.

Now, consider again the duplication algorithm pdup, for which our theoretical worst-case time bound is $O(|T| + m(t,T))$ by Theorem 5, where *T* is the input p-string. To predict actual running times on code, we note that analysis of the algorithm indicates that the constants resulting from the part of the algorithm whose time depends on $m(t,T)$ should not greatly dominate the constants resulting from the part whose time depends on $|T|$. Moreover, pdup always performs $\Omega(n)$ operations on input of length *n*. Therefore, for fixed *t*, the observed running time should be close to linear in input length, rather than showing a dramatic nonlinear blowup, as long as $m(t,T) < |T|$. Experiments for $t = 75$ and samples of code of length up to 158,579 lines (corresponding to a p-string of length 517,415) support this prediction.

Figure 4 shows a scatter plot of 7674 maximal p-matches of at least 75 symbols found in about 101,000 lines (excluding comments and white space) of C code input, corresponding to a p-string of length 320947. Each axis represents position in the input p-string. Each "dot" is really a diagonal line segment from (p_1, p_2) to $(p_1 + k, p_2 + k)$ representing a maximal p-match of length $k + 1$ between substrings starting at positions p_1 and p_2 in the input p-string, for some p_1, p_2, and *k*. The p-match relation is symmetric; only points below the main diagonal are plotted. The plot is dense near the main diagonal because of local matches; no points are plotted on the actual main diagonal.

The appearance of Figure 4 suggests why $m(t,T)$ does not grow faster with respect to $|T|$: the density near the main diagonal and the sparseness away from the main diagonal show that much of the duplication is fairly local. To be fair, the experiments with *eager* and *pdup* and those with **dup** did not include large tables such as font tables, where quadratic growth in the number of matches might be more likely. Moreover, when the threshold is taken to be a single line in **dup**, $m(t,T)$ grows quadratically in $|T|$, due to lines consisting of only a closing brace. Experiments with **dup** have shown that small thresholds, under about 10 lines, lead to reporting many pairs of code segments dominated by closing braces, and these matches are not of much interest to the programmer. Thresholds on the order of 30 lines, on the other hand, report mostly matches that appear to have substantial information content. Thus, intuitively, a threshold low enough for the number of p-matches to exceed the input length will yield p-matches of low information content.

Acknowledgements.

The author would like to thank Raffaele Giancarlo for many discussions relating to this work, and Raffaele Giancarlo and Mihalis Yannikakis for their thoughtful comments on this abstract.

References

[Aho] A.V. Aho, Algorithms for finding patterns in strings, *Handbook of Theoretical Computer Science* (1990), ed. Van Leeuwen, Elsevier Science Publishers B.V., Amsterdam, pp. 255-300.

[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* (March, 1992).

[Bak2] Brenda S. Baker, On finding duplication in strings and software, submitted for publication.

[Bak3] Brenda S. Baker, Parameterized duplication in strings: algorithms and an application to software maintenance, submitted for publication.

[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.

[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* (March, 1992).

[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.

[GG] Z. Galil and R. Giancarlo, Data structures and algorithms for approximate string matching, *J. Complexity 4* (1988), pp. 33-72.

[GeN] Michael R. Genesereth and Nils J. Nilsson, *Logical Foundations of Artificial Intelligence* (1987), Morgan Kaufman Publishers, Los Altos, California.

[Gi] R. Giancarlo, The suffix tree of a square matrix, with applications, *Proc. of 4th Annual ACM-SIAM Conference on Discrete Algorithms* (SODA) (Jan. 1993), pp. 402-411.

[Ho] Susan Horwitz, Identifying the semantic and textual differences between two versions of a program, *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1990), pp. 234-245.

[HT] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. on Computing 13* (1984), pp. 338-355.

[Hu92] Andrew G. Hume, personal communication (November 1992).

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

[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).

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

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

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

[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.

[MM] Eugene W. Myers and Webb Miller, Approximate matching of regular expressions, *Bulletin of Mathematical Biology 51* (1989), pp. 5-37.

[Ra] 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.

[SV] Baruch Schieber and Uzi Vishkin, On finding lowest common ancestors: simplification and parallelization, *SIAM J. on Computing 17* (1988), pp. 1253-1262.

[ST] Daniel D. Sleator and Robert E. Tarjan, A data structure for dynamic trees *J. Comput. Sys. Sci. 26* (1983), pp. 362-391.

[WM] Sun Wu and Udi Manber, Fast text searching allowing errors, *Comm. ACM 35,10* (Oct 1992), pp. 83-91.

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