to appear in *J. Computer and System Sciences*

Parameterized Pattern Matching: Algorithms and Applications

Brenda S. Baker

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

ABSTRACT

The problem of finding sections of code that either are identical or are related by the systematic renaming of variables or constants can be modeled in terms of *parameterized strings* (*p-strings*) and *parameterized matches* (*p-matches*) [Baker93a]. P-strings are strings over two alphabets, one of which represents parameters. Two p-strings are a *parameterized match* (*p-match*) if one p-string is obtained by renaming the parameters of the other by a one-to-one function.

In this paper, we investigate parameterized pattern matching via parameterized suffix trees (p-suffix trees), defined in [Baker93a]. We give two algorithms for constructing p-suffix trees: one (*eager*) that runs in linear time for fixed alphabets, and another that uses auxiliary data structures and runs in $O(n \log (n))$ time for variable alphabets, where n is input length. We show that using a p-suffix tree for a pattern p-string P, it is possible to search for all p-matches of P within a text p-string T in space linear in $|P|$ and time linear in $|T|$ for fixed alphabets, or $O(|T| \log (\min(|P|, \sigma))$ time and $O(|P|)$ space for variable alphabets, where σ is the sum of the alphabet sizes.

The simpler p-suffix tree construction algorithm *eager* has been implemented, and experiments show it to be practical. Since it runs faster than predicted by the above worst-case bound, we reanalyze the algorithm, and show that *eager* runs in time $O(\min(t|S| + m(t,S) | t > 0) \log \sigma))$, where for an input p-string S, $m(t,S)$ is the number of maximal p-matches of length at least t that occur within S, and σ is the sum of the alphabet sizes. Experiments with the author's program **dup** [Baker92] for finding all maximal p-matches within a p-string have found $m(t,S)$ to be less than $|S|$ in practice unless t is small.

The results in this paper were announced at the 25th Annual ACM Symposium on Theory of Computing, May, 1993 [Baker93b]; some related results announced at that conference appear in a separate paper [Baker93a].
Sept. 30, 1993; Revised March 30, 1994

Parameterized Pattern Matching: Algorithms and Applications

Brenda S. Baker

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

1. Introduction

It is well known that copying code in the course of fixing bugs or adding new features to a software system can lead to the presence of bugs. Nevertheless, this method is still used to avoid breaking working code, and over time, the amount of duplication can become substantial.

The author implemented a program **dup** [Baker92] to find the duplication over a threshold length in a software system, using a text-based approach. In addition to finding identical sections of code, **dup** finds *parameterized matches*, namely, pairs of code segments that match except for a systematic change of parameters. An example of the latter is given in Figure 1, which contains fragments of two 34-line sections of source code from the X Window System [SG86] that are identical except for the correspondence between the variable names pfi/pfh and the pairs of structure member names lbearing/left and rbearing/right. The software engineering aspects of **dup** are described in [Baker92]. The amount of duplication found can be substantial. For example, in a million lines of code, approximately 21% of the lines were found to be involved in parameterized matches of at least 30 lines, and approximately 38% of the lines were found to be involved in parameterized matches of at least 15 lines.

```
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. Two fragments of code from source for the X Window System.

When a programmer needs to modify a section of code, it would be helpful to be able to find all related sections without finding all the duplication in the system. Therefore, algorithms for finding all parameterized matches to a particular section of code would also be useful.

The notion of parameterized matches for code was formalized in [Baker93a]. A *parameterized string*, or *p-string*, is a string that can 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. For example, if $\Pi = \{x,y,v\}$, and $\Sigma = \{a,b,c\}$, then $S_1 = axaybxycby$ and $S_2 = ayavbyvcbv$ are a p-match, where x and y of S_1 are renamed as y and v, respectively, in S_2. An encoding *prev* of p-strings was defined such that $prev(P)$ has a p-match at position i of a p-string T iff $prev(P)$ is a prefix of $prev(T_i)$, where T_i is the suffix of T starting at position i.

A data structure called a *parameterized suffix tree* (*p-suffix tree*) was defined [Baker93a] as a generalization of suffix trees for strings [McC]. Whereas a suffix tree is a compacted trie for suffixes of a string, a p-suffix tree is a compacted trie for encoded suffixes of a p-string. In order to store the p-suffix tree in linear space, the labels are stored as indices into the input, but whenever an input symbol is used, it is interpreted dynamically in terms of how deep it is in the p-suffix tree. Unfortunately, p-strings (under the *prev* encoding) do not satisfy a key property for strings called the Distinct Right Context Property [Baker93a]. (The distinct right context property says that if the first k symbols of two strings are followed by distinct symbols, then when the first symbol is removed from each string, the first $k-1$ symbols are still followed by distinct symbols.) Consequently, it is not possible to imitate McCreight's linear time construction of suffix trees [McC] directly. The problem is that his algorithm depends on *suffix links*, which are defined for each internal vertex in the tree, and point from the vertex representing a substring aS of the input to the vertex for the substring S, and the failure of the Distinct Right Context Property means that in a p-suffix tree there may be no vertex for a suffix link to point to.

In [Baker93a], this problem was solved by allowing "bad" suffix links, that point to a vertex just above (*i.e.* closer to the root than) the desired place, and by updating them when they are used (lazy evaluation). An algorithm was given to construct a p-suffix tree in linear time for fixed alphabets; for variable alphabets Σ and Π, the bound obtained was $O(n(|\Pi|\log\sigma))$ time and $O(n)$ space for a p-string of length n, where σ is the sum of the alphabet sizes. It was shown that given a p-suffix tree for a p-string S, all maximal p-matches within S over a threshold length t can be reported in $O(|S| + m(t,S))$ time and $O(|S|)$ space, where $m(t,S)$ is the number of p-matches reported.

In this paper, we begin by defining "bad" suffix links to point to a slightly different place (just below the desired position rather than just above). We obtain a key lemma that shows that with this definition, all vertices with "bad" suffix links pointing to the same vertex lie in a chain of vertices connected by arcs whose labels begin with 0. This characterization is the basis for the algorithms in the paper.

We give another algorithm *eager* for building p-suffix trees that uses "eager" evaluation of the "bad" suffix links, *i.e.* it keeps all "bad" suffix links up-to-date in terms of the current tree structure. Because of the lemma, all bad suffix links pointing to the same vertex can be located by a pointer to the top of the chain, and the updating can be done efficiently. The algorithm runs in $O(n(|\Pi| + \log\sigma))$ time and $O(n)$ space, where n is input length and σ is the sum of the alphabet sizes. In addition, using the eager evaluation approach, we can augment the p-suffix trees with auxiliary data structures, including concatenable queues [AHU] and the dynamic tree operations of Sleator and Tarjan [ST], to attain an improved bound of $O(n\log n)$ time and $O(n)$ space for constructing a p-suffix tree for a p-string of length n for variable alphabets.

Moreover, when the eager evaluation approach is used, the resulting p-suffix tree has up-to-date suffix links that can be used for parameterized pattern matching. Without suffix links, a p-suffix tree for a p-string T can be searched for p-matches to a pattern p-string P in space $O(T)$ and time $O(|P|\log\sigma + tocc)$, where *tocc* is the number of p-matches reported and σ is the sum of the alphabet sizes [Baker93a]. We show here that it is possible to find all parameterized matches in T in time $O(|T|(|\Pi| + \log\sigma))$ time and $O(|P|)$ space by generalizing a pattern matching algorithm for string suffix trees [CL90] and techniques for suffix automata [Croch87] to p-suffix trees with up-to-date suffix links. By taking advantage of the chain characterization of bad suffix links and augmenting the p-suffix tree with additional data structures, we improve the time bound to $O(|T|\log(\min(|P|,\sigma))$. In this case, the auxiliary data structures include those of Harel and Tarjan [HT84] or Schieber and Vishkin [SV88] for finding lowest common ancestors.

Finally, we turn to the application and describe an implementation of the basic algorithm for constructing p-suffix trees. The basic algorithm performs well on input derived from code; in fact, it appears to run in linear time. The log factor due to arc access time in the theoretical worst case bound is avoided by hashing in the implementation. More importantly, the two operations (rescanning and updating) that led to the $O(|\Pi|n)$ in the theoretical worst-case bound have in experiments never contributed more than n steps each, for inputs ranging from hundreds of lines to over a million lines. We reanalyze the algorithm in an effort to explain why this should be true. The reanalysis relates the running time of *eager* to the amount of parameterized duplication in the input. For a p-string S and any positive integer t, recall that $m(t,S)$ is the number of maximal parameterized matches of length at least t within S. Thus, $m(t,S)$ measures all the parameterized duplication in S over the threshold length t. For our intended application, experiments with **dup** have found $m(t,S)$ to be less than $|S|$unless t is small. We show that the worst-case running time of *eager* on input S is bounded by $O(\min(t|S|+m(t,S)|t>0)\log\sigma)$, which combined with the experimental observations on $m(t,S)$ suggests why the algorithm runs fast in the application. The proof is by showing that each rescanning and updating step corresponds to a distinct maximal p-match between two substrings of the input, where a maximal p-match is a p-match that cannot be extended to the left or right.

The relationship between p-suffix trees and the Giancarlo's L-suffix trees [Gian93] for two-dimensional patterns is interesting. Both are generalizations of suffix trees to deal with an input type for which the Distinct Right Context Property fails. For both, dynamic tree operations of [ST83] and concatenable queues [AHU74] can be used as auxiliary data structures to obtain an $O(n\log(n))$ time bound for constructing the tree, where n is the input size. However, the L-suffix trees have no analog to our chain characterization for bad suffix links. Therefore, there is no simple construction algorithm for L-suffix trees that corresponds to the practical algorithm *eager* of this paper, and the techniques used here for pattern-matching for variable alphabets cannot be used for L-suffix trees.

We conclude this section with some comments on how parameterized pattern matching relates to previous work on string pattern matching. 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 [Aho90]; 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.

2. P-strings and p-suffix trees

This section defines p-strings, p-matches, and p-suffix trees, as in [Baker93a].

DEFINITION: Let Σ and Π be disjoint finite alphabets. Symbols in Σ are called *constants*, while symbols in Π are called *parameters*. A string in $(\Sigma \cup \Pi)^*$ is called a *parameterized string* or *p-string*. Two p-strings S and $\overline{S}$ are a *parameterized match*, or *p-match*, if S can be transformed into $\overline{S}$ by applying a "renaming" bijection g from the symbols of S to the symbols of $\overline{S}$, such that g

is the identity on the constant symbols.

For example, if $\Sigma = \{a, b\}$ and $\Pi = \{u, v, x, y\}$, then $S = abuvabuvu$ and $\overline{S} = abxyabxyx$ are a p-match, with g as the renaming function, where $g(u) = x$ and $g(v) = y$.

We assume a RAM model of computation with the uniform cost criterion [AHU74]. It is convenient to assume that Σ is disjoint from the set N of nonnegative integers; if not, the encodings used below can be modified without changing the time or space bounds. We assume that the alphabets are ordered, symbols can be compared in $O(1)$ time, and whether a symbol belongs to Σ or Π can be determined in $O(1)$ time. Define $\sigma = |\Sigma| + |\Pi|$.

It is straightforward to determine whether two p-strings p-match by checking whether a renaming function exists for them. However, an encoding of p-strings in terms of relative positions of parameter symbols will be useful in searching for p-matches of a p-string pattern in a p-string text.

DEFINITION: Let N be the set of nonnegative integers. Define $prev: (\Sigma \cup \Pi)^* \rightarrow (\Sigma \cup N)^*$ to be the function such that for $S = b_1 b_2 \cdots b_n$, $prev(S) = c_1 c_2 \cdots c_n$, where the value of c_i, $1 \le i \le n$, is determined as follows. If $b_i \in \Sigma$, then $c_i = b_i$; if $b_i \in \Pi$, then $c_i = 0$ if i is the leftmost position in S at which b_i occurs, and $c_i = i - k$ if k is the previous position to the left at which b_i occurs. If $c_i \notin \Sigma$ it is called a *parameter pointer*.

Returning to our example, and enclosing integers by parentheses to prevent ambiguity, we have $prev(abuvabuvu) = ab(0)(0)ab(4)(4)(2) = prev(abxyabxyx)$.

DEFINITION: If $S = b_1 \cdots b_n$ is a p-string, define $psub(S, i, j) = prev(b_i \cdots b_j)$ for $i \le j$, and $psub(S, i, j)$ to be the empty string for $j < i$. For $1 \le i \le n$, define the ith *p-suffix* of S to be $psuffix(S, i) = psub(S, i, n)$.

For the above example, with $S =$ $abuvabuvu$, we have $psub(S, 3, 7)$ = $(0)(0)ab(4)$, $psuffix(S, 3)$ = $(0)(0)ab(4)(4)(2)$, $psuffix(S, 4)$ = $(0)ab(0)(4)(2)$, and $psuffix(S, 5)$ = $ab(0)(0)(2)$.

The *prev* encoding is related to finding p-matches as follows [Baker93a]. First, p-strings S and $\overline{S}$ are a p-match if and only if $prev(S) = prev(\overline{S})$. Second, if P is a p-string pattern and T is a p-string text, P has a p-match starting at position i of T iff $prev(P)$ is a prefix of $psuffix(T, i)$.

For a p-string S, $prev(S)$ can be constructed in $O(|S| \min(\log |S|, \log |\Pi|))$ time and $O(|S|)$ space by means of a balanced tree structure. Under stronger assumptions ($|\Pi|$ is known, symbols of Π can be used to index into a table, and a table can be allocated in $O(1)$ time), $prev(S)$ can be constructed in $O(|S|)$ time and $O(|\Pi|)$ space by means of a table of size $|\Pi|$ that is initialized to 0 for those symbols of Π occurring in S and then used in a second scan of S to contain the positions of last occurrence for these symbols.

The value of the jth symbol of $psuffix(S, i)$ can be computed in constant time from j and the $(j + i - 1)$st symbol of $prev(S)$ by the following "transform" function f.

DEFINITION: For $b \in \Sigma \cup N$, $f(b, j) = 0$ if b is a nonnegative integer larger than $j - 1$, and otherwise, $f(b, j) = b$.

DEFINITION: If S is a p-string terminated by an endmarker \$ that appears nowhere else in S, a *p-*

suffix tree for S is a compacted trie for the p-suffixes of S.

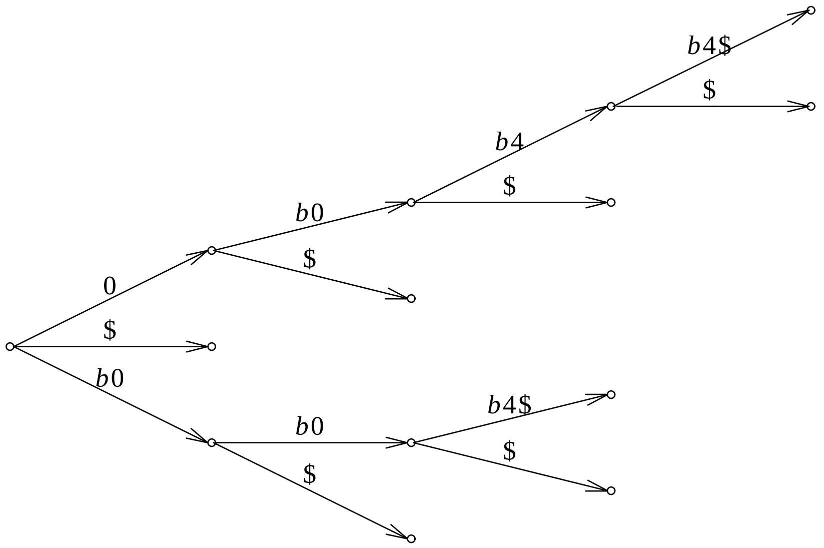

Figure 2. A p-suffix tree for the p-string $S = ubvbubv\$$, with $prev(S) = 0b0b4b4\$$.

Example. Let $S = ubvbubv\$$, where $\Sigma = \{a, b, \$\}$, and $\Pi = \{u, v\}$. The p-suffix tree for S is shown in Figure 2. The p-suffixes stored are $0b0b4b4\$$, $b0b0b4\$$, $0b0b4\$$, $b0b0\$$, *etc.* Notice that the parameter pointers change to 0 as the preceding part of the string is shortened.

DEFINITION: For each vertex v of the suffix tree, define $path(v)$ to be the sequence of arcs leading from the root to v. Define the *pathstring* of v, denoted by $pathstr(v)$, to be the concatenation of the labels on the path for v, and the *pathlength* for v, denoted by $pathlen(v)$, to be the length of $pathstr(v)$.

Since the p-suffix tree for a p-string S is a trie, each arc in a p-suffix tree represents a nonempty substring and each arc from a vertex to one of its children has a distinct starting symbol. Since each pathstring from the root to a leaf is a distinct p-suffix of S, the number of leaves is $|S|$. Since the trie is compacted, each internal (non-leaf) vertex has at least two children, and the number of vertices in the p-suffix tree is linear in $|S|$.

For each internal vertex v, there are (at least one pair) i, j such that the arc label of the arc into v from its parent is $psub(S, i, j)$. In order to store the arc labels in linear space, we represent the label for v by one such starting position i, called $firstpos(v)$, and the length, $j - i + 1$, called $arclen(v)$. The actual symbols of the label are obtained by applying the transform function f to $prev(S)$. If the symbol b to be read from $prev(S)$ is the kth symbol in the current path from the root, it is treated as if it is $f(b, k)$. This can be done in $O(1)$ time if the pathlength is stored at each vertex.

While the value of a pointer may be as large as one less than the p-string length, at any vertex v there are at most $|\Pi|$ distinct values of pointers that may begin the substrings labeling the arcs out of the vertex. This follows because each nonzero pointer refers to the previous position of a parameter symbol, and there are at most Π distinct parameters. Thus, the number of children of each non-leaf vertex is bounded by $\sigma = |\Sigma| + |\Pi|$.

We assume that for each vertex, the arc (if any) whose label begins with 0 is stored so that it can be accessed in $O(1)$ time, and that the remaining arcs to children are accessed via a balanced tree scheme [AHU74] so that an arc beginning with a particular symbol can be accessed,inserted, or deleted in $O(\log\sigma)$ time.

3. Augmenting a p-suffix tree with suffix links

For strings, McCreight's linear time construction algorithm for suffix trees depends on the ability to define *suffix links*: a suffix link for a vertex with pathstring *aS* points to the vertex with pathstring *S*. The existence of a vertex to point to is guaranteed by the *distinct right context* property of strings: if a,b,c,d are symbols and *S* and *T* are strings, and $aS = bT$ but $aSc \neq bTd$, then $Sc \neq Td$. In particular, since a suffix tree is a compacted trie, the existence of a vertex with pathstring *aS* means there are input suffixes beginning with *aSc* and *aSd*, $c \neq d$, and therefore there are input suffixes beginning with *Sc* and *Sd*, implying that a vertex exists with pathstring *S*.

Unfortunately, the distinct right-context property fails for p-strings, because nonzero parameter points can turn into 0's as the left end of the string is chopped off. For example, suppose $prev(S) = 0ab30ab0$. Let $x = 0$, $\alpha = ab$, $c = 3$, and $d = 0$. Then $x\alpha c = 0ab3 = psub(S,1,4)$ and $x\alpha d = 0ab0 = psub(S,5,8)$, and the right context is different for the two occurrences of $x\alpha$. However, $psub(S,2,4) = ab0$ while $psub(S,6,8) = ab0$, so that the distinctness of the right contexts of the two occurrences of α is lost.

Thus, we need to allow more leeway in the definition of suffix links. Suppose the p-suffix tree is constructed for a p-string *S*. Let *v* be an internal vertex with pathstring $psub(S,i,j)$. Whereas for strings the suffix link $SL(v)$ would point to a vertex with pathstring $psub(S,i+1,j)$, the best we can do is to define $SL(v)$ to point to a vertex for which the pathstring is close to $psub(S,i+1,j)$. In [Baker93a], a suffix link pointed to the vertex whose pathstring is the longest prefix of $psub(S,i+1,j)$ in the tree. Here, we make the opposite choice.

DEFINITION: For $i,j > 0$, if the pathstring of a vertex *v* is $psub(S,i,j)$, then $SL(v)$, the suffix link for *v*, points to the vertex whose pathstring is the shortest extension of $psub(S,i+1,j)$ in the tree, *i.e.* the vertex whose pathstring is the shortest of all those for which $psub(S,i+1,j)$ is a prefix.

Since $psub(S,i,j)$ is the empty string for $i > j$, this definition sets the suffix link of the root to point to the root. Note that a vertex must exist for $SL(v)$ to point to, since $psuffix(S,i+1)$ is a pathstring in the tree. We claim that our suffix links are well-defined, in the sense that (1) $SL(v)$ is consistently defined for any i,j such that the pathstring of *v* is $psub(S,i,j)$ and (2) the vertex pointed to is unique. Well-definedness follows from the following property of p-strings.

Common Prefix Property: If *S* and *T* are p-strings, $a,b \in \Sigma \cup \Pi$, and $prev(aS) = prev(bT)$, then $prev(S) = prev(T)$ [Baker93a].

This property is called the Common Prefix Property because it implies that if the *i*th and *j*th p-suffixes share a prefix of length *k*, the $(i+1)$st and $(j+1)$st p-suffixes will share a prefix of length (at least) $k-1$.

DEFINITION: Suppose *v* has pathstring $psub(S,i-1,j)$. If $SL(v)$ points to a vertex with pathstring $psub(S,i,j)$, we say it is a *good* suffix link; otherwise we say it is *bad*.

Although the Distinct Right Context Property fails for p-strings, the following restricted form holds.

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 $k=|aS|=|bT|$ [Baker93a].

We make use of the Restricted Distinct Right Context Property to characterize bad suffix links pointing to the same vertex as belonging to a chain of vertices connected by arcs whose labels begin with 0. This characterization forms the basis of the algorithms in this paper.

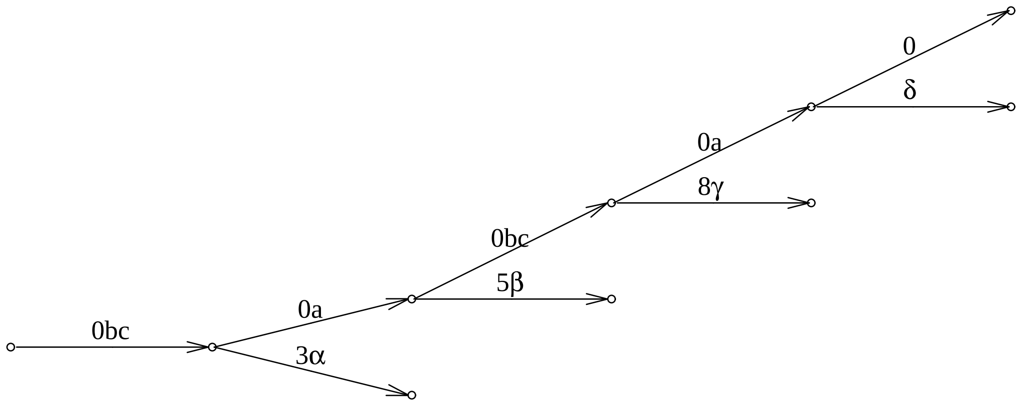

Figure 3. A chain of arcs beginning with 0's.

Figure 3 shows such a chain. For all vertices in the chain except the last, there are exactly two arcs, one with a label beginning with 0, and the other with a label beginning with a parameter pointer pointing to the first symbol in the pathstring. When the first symbol is stripped off for the next p-suffix, that parameter pointer turns into a 0, the distinct right-context property fails, and a bad suffix link may be created for that vertex.

Consider the vertex v with pathstring $0bc3\alpha$ in Figure 3. Suppose there are two arcs out of v, one beginning with a 0 and one beginning with an integer k. Can the distinct right-context property fail at v? No; k cannot point to the parameter corresponding to the first symbol in the pathstring, because the 3 points to that one, and represents a closer occurrence of the same parameter. This idea leads to the following characterization.

LEMMA 1. Let w be a vertex in a p-suffix tree for a p-string S. If $v_1, v_2, \cdots, v_k$ are all the vertices v_j with bad suffix links such that $SL(v_j)=w$, then they occur in a chain of at most $|\Pi|$ vertices connected by arcs whose labels all begin with 0.

Proof. Suppose vertex v has a bad suffix link pointing to vertex w. Then v has exactly two arcs to children: one beginning with a 0 and one beginning with a parameter pointer $pathlen(v)$ pointing to the first position in the pathstring of v. To see this, observe that for any i,j, if $psub(S,i,i+t)=psub(S,j,j+t)$, then $psub(S,i+1,i+t)=psub(S,j+1,j+t)$ by the Common Prefix Property, and then apply the Restricted Distinct Right Context Property.

Now, suppose there are two vertices u and v with bad suffix links to w, and neither is an ancestor of the other. For some i, j, r, s, the pathstring for u is $psub(S, i, r)$ and the pathstring for v is $psub(S, j, s)$. Let k be the length of the common prefix of these pathstrings. Then $psub(S, i, i+k-1) = psub(S, j, j+k-1)$ but $psub(S, i, i+k) \neq psub(S, j, j+k)$. Since u and v have bad suffix links to the same vertex, $psub(S, i+1, i+k) = psub(S, j+1, j+k)$. By the Restricted Distinct Right Context Property, either u or v, say u, is reached via an arc whose label begins with a parameter pointer to the first position in its pathstring. From the previous paragraph, u also has an arc whose label begins with a parameter pointer to the first position in the pathstring, contradicting the definition of parameter pointers. We conclude that if u and v have bad suffix links pointing to the same vertex, one is an ancestor of the other.

Next, suppose v has a bad suffix link pointing to w, and u is a descendant of v reached through the arc of v whose label starts with $pathlen(v)$. If u also has a bad suffix link, then it must have a child arc whose label is $pathlen(u)$, representing a parameter pointer pointing to the first position in the pathstring. But then there would be two parameter pointers in the same pathstring pointing to the same position, contradicting the definition of parameter pointers. Therefore, any descendants of v that have bad suffix links must be reached through the arc of v whose label begins with 0.

Now, suppose that the child of v reached by the arc of v whose label begins with 0 does not have a bad suffix link pointing to w. Then its suffix link points to a vertex z below w, and its descendants cannot have bad suffix links pointing to w, because of the relative pathlengths involved.

Applying this argument inductively to successive vertices with bad suffix links pointing to w shows that they all lie in a chain of vertices connected by arcs whose labels begin with 0.

Finally, we observe that there are at most $|\Pi|$ members of such a chain, since each 0 corresponds to the first occurrence of a different parameter. □

Of course, the vertices in a chain of arcs whose labels begin with 0 do not necessarily have bad suffix links, and if they do have bad suffix links, the bad suffix links may not all point to the same vertex, because of the existence of other vertices to point to. A vertex with a good suffix link pointing to w need not be near the chain of vertices with bad suffix links pointing to w.

4. Construction of a p-suffix tree

In this section, we give two algorithms for constructing a p-suffix tree. It will be helpful to have notation for the point at which the ith p-suffix branches off from all previous p-suffixes in a p-suffix tree.

DEFINITION: For a p-string S, define $head_0(S)$ to be the empty string, and for $i \geq 1$, define $head_i(S)$ to be the longest prefix of $psuffix(S, i)$ that is also a prefix of $psuffix(S, j)$ for some $j < i$. If S is clear from context, we omit the S and write just $head_i$.

The usefulness of this definition lies partly in the following property.

Nondecreasing Head Property: For a p-string S, if $head_{i-1} = psub(S, i-1, r)$, where $0 \leq r$, then $head_i = psub(S, i, s)$ for some $s \geq r$ [Baker93a].

Now, let us look at the main ideas of the McCreight construction. The suffixes are inserted one at a time into the suffix tree, from longest to shortest. To insert the ith suffix, the algorithm must find or create a vertex whose pathstring is $head_i$, and add an arc to a new leaf representing the remainder of the ith suffix. To do this efficiently, it makes use of information saved from inserting the $(i-1)$st suffix. In particular, when the $(i-1)$st suffix has just been inserted, the algorithm knows the vertex h_{i-1} whose pathstring is $head_{i-1}$. At this point, the suffix link is defined either for h_{i-1} or for its parent. The algorithm follows this suffix link, and searches in the direction away from the root for where $head_i$ should end in the tree; that it must end in this direction is guaranteed by the Nondecreasing Head Property. Along the way, the algorithm also sets the suffix link for h_{i-1}. It is at this point that the distinct right-context property is needed in order to guarantee that there is a branching vertex to point to. Once the vertex with pathstring $head_i$ is found or created, a leaf can be added whose pathstring is the ith suffix.

A generalization of this algorithm to p-suffix trees, with suffix links defined as in Section 3, must overcome three difficulties created by 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 vertices are added,

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

(3) when we use a 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 [Baker93a], 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). Our strategy for keeping suffix links up-to-date is based on Lemma 1: for each vertex v, a pointer $min(v)$ is kept to the vertex of smallest pathlength with a bad suffix link pointing to v, and when the arc into v from its parent is split by inserting a new vertex, the bad suffix links to v can be updated by tracing down the chain of vertices whose labels begin with 0. If suffix links are kept up-to-date, (2) is not a problem, because if a bad suffix link points to a vertex v, the desired point lies on the arc from the parent of v to v, and no searching is needed in the tree. (3) is not a problem in terms of deriving a correct algorithm, but affects the running time.

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 $1 \le i \le n$. For each i, exactly one leaf is created in stage i, with pathstring $psuffix(i)$, and if no internal vertex with pathstring $head_i$ already exists, it is created in Stage i.

The following properties will hold at the end of Stage i.

(P1) The p-suffix tree is a compacted trie for the p-suffixes from 1 to i.

(P2) Except for the (at most one) new internal vertex created in Stage i, for which the suffix link may be undefined, each internal vertex v has a suffix link defined in terms of the vertices currently existing in the tree. That is, if v has pathstring $prev(S_i \cdots S_j)$, then $SL(v)$ points to the vertex 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.

(P3) For each vertex v other than the root, there is a pointer $min(v)$ that points to the vertex of smallest pathlength with a bad suffix link pointing to v, if such a vertex exists, and otherwise either points to a vertex with a good suffix link to v or is undefined.

The algorithm *eager* for building a p-suffix tree operates as follows. First, it computes $P = prev(S)$. 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, $i = 1, 2, \ldots, n$, has six phases.

Phase 1: If $SL(oldhd)$ is undefined, this phase sets a value for it in terms of the current tree structure. In this case, *oldhd* is not the root, and the pathstring of *oldhd* is $head_{i-1} = psub(S, i-1, s)$ for some $s \geq i-1$. Starting at $SL(parent(oldhd))$, *eager* "rescans" in the direction away from the root to find the vertex closest to the root whose pathstring is an extension of $psub(S, i, s)$, and sets $SL(oldhd)$ (at least temporarily) to point to this vertex. In rescanning, only the first symbol of each arc is checked to determine the path to follow in the tree toward a leaf, as in the rescanning phase of McCreight's algorithm.

Phase 2: *eager* scans symbol by symbol in the direction away from the root, 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. When the scanning ends, the value of t has been determined such that $head_i = psub(S, i, t)$. Let g denote the vertex entered by the last arc for which a transformed input symbol was matched to a symbol in the label.

Phase 3: If the entire label was matched for the arc entering g, the pathstring of g is $head_i$, so *newhd* is set to g and no new internal vertex is created. Otherwise, *eager* does the following. (1) It splits the arc from the parent of g to g by inserting a new vertex *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 $head_i$. (2) The procedure *update* described below is called to update the suffix links pointing to g that should now point to *newhd* and to set $min(newhd)$ and $min(g)$. (3) 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 vertices 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: A new leaf is created and made a child of *newhd* via an arc with label $psub(S, t+1, n)$, so that its pathstring is $psub(S, i)$.

Phase 6: *oldhd* is reset to *newhd* and *oldchild* is set to g.

The procedure *update* must reset any bad suffix links pointing to g that now should point to *newhd*, and must reset the value of $min(g)$ and set the value of $min(newhd)$. We would like to make use of Lemma 1, the characterization of bad suffix links in terms of chains of vertices. Lemma 1 applies at the end of Phase 1, except for an anomaly at *oldhd*, which may have been inserted in Stage $i-1$ into a chain of vertices with bad suffix links pointing to g, so that the arc leading to the next such vertex may not start with 0. In this case, the next vertex below *oldhd* which could have a bad suffix link to g is the vertex *oldchild* saved at the end of Stage $i-1$, because the only other child of *oldhd* is the new leaf created in Stage $i-1$. Since the label of the arc from *oldhd* to *oldchild* may not start with 0, this arc must be treated specially in *update*.

Consequently, the procedure *update* operates as follows to update suffix links and set $min(g)$ and $min(newhd)$. If $min(g)$ is undefined or the pathlength of $min(g)$ is greater than one more than the pathlength of *newhd*, $min(newhd)$ is left undefined and $min(g)$ is unchanged. Otherwise, $min(newhd)$ is set to $min(g)$, and starting at $min(g)$, as long as the pathlength of the current vertex is at most one more than the pathlength of *newhd*, *update* searches in the direction away from the root, following arcs whose labels begin with 0, except that if it reaches an internal vertex numbered $i-1$, it goes to *oldchild* instead; at each internal vertex along the way, it resets the suffix link to *newhd*. After the loop, $min(g)$ is reset to point to the last vertex 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 g, and to undefined otherwise.

While the structure of the above algorithm is a generalization of McCreight's algorithm, the existence of bad suffix links degrades the running time compared to McCreight's algorithm through both rescanning and updating. In McCreight's algorithm, once an input symbol is rescanned, the algorithm will never look before that symbol in the input again (although it may rescan that symbol again at the beginning of a later stage). By contrast, *eager* may rescan a series of symbols at vertices with bad suffix links, and then in a later stage, jump back to an earlier point in the input for rescanning. A scenario in which this could happen, as well as the updating of a series of bad suffix links, is the following.

Let $prev(S) = b_1 b_2 \cdots b_n$. Suppose that for some $i, j, r,\ i < j < r$, $psub(S, i, r)$ ends in a sequence of 0's obtained from $b_j, b_{j+1}, \ldots, b_r$ by f, and suppose that in the path of $psuffix(S, i)$ in the tree, these 0's lie along a chain of vertices with bad suffix links pointing to a vertex w. To simplify the discussion, we suppose that b_{j-1} is a symbol of Σ and there is a vertex u with pathstring $psub(S, i, j-2)$ and a good suffix link to a vertex v. If *eager* begins rescanning at u, and rescans through $b_{j-1}, b_j, b_{j+1}, \ldots, b_r$, and then creates a new vertex as the child of the last vertex with a bad suffix link, it will subsequently follow that bad suffix link to w at the beginning of Stage $i+1$. It may then create a new vertex z as the parent of w, forcing *eager* to update all the bad suffix links to w. Because the chain of bad suffix links exist, the parent of z must be the vertex v pointed to by the suffix link of u, and its pathstring is $psub(S, i+1, j-2)$. Since b_{j-1} is in Σ, v has a good suffix link to the vertex whose pathstring is $psub(S, i+2, j-2)$. At the beginning of Stage $i+2$, this good suffix link is followed, and rescanning starts at symbol b_{j-1} of the input, back where it was two stages earlier. This means that $b_{j-1}, b_j, b_{j+1}, \ldots, b_r$ may all get rescanned again. Moreover, the scenario could be repeated, causing further rescanning of these symbols.

The proof of the following theorem bounds the running time of *eager* and shows that the updating of the bad suffix links and the rescanning of input symbols transformed into 0's by f are the only places in which *eager*'s running time is degraded compared to McCreight's original algorithm.

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

Proof.

Let T_i represent the tree at the end of Stage i. We show by induction on i that after Stage i, properties P1-P3 hold for T_i, and that in addition, *oldhd* is set to the vertex with pathstring $head_i$. The properties hold vacuously at the start, corresponding to $i = 0$. Assume that they hold at the end of Stage $i-1$.

The only differences between the p-suffix tree for the first i p-suffixes and that for the first $i-1$ p-suffixes are where the pathstring for $head_i$ ends in the tree: an additional vertex with pathstring $head_i$ is required if it did not already exist, and a child of that vertex is required to represent the rest of $psuffix(S, i)$.

We claim that Phases 1-3 and 5 accomplish these changes, and consequently property P1 holds at the end of Stage i. First, we observe that at the end of Phase 1, $SL(oldhd)$ is defined correctly with respect to the tree structure of T_{i-1}. If it was already defined at the end of Stage $i-1$, this follows from property P2, and otherwise, *eager* sets it by rescanning from $SL(parent(oldhd))$, which is an ancestor of the desired vertex by the definition of tries and Nondecreasing Head Property. (Note that rescanning need only check the first symbol of each arc label because the Common Prefix Property guarantees that $psub(S, i, s)$ is the prefix of a psuffix already stored in the tree in a stage before $i-1$.) Second, we observe that in Phase 2, *eager* finds the end of pathstring $head_i$ in T_{i-1}. This is true because by the Nondecreasing Head Property, *eager* can scan in the direction away from the root starting at distance *r* from the root on the pathstring of $SL(oldhd)$ to find the desired location, which is determined by when the next symbol of $psub(S,i,n)$ is not available in the tree. In Phase 3, *eager* creates a vertex with pathstring $head_i$ if none exists, and in Phase 5, *eager* creates a new child for that vertex, with pathstring $psuffix(S,i)$.

For P2 and P3, we begin by considering the situation after Phase 1 of Stage *i*. From above for *oldhd* and by the induction hypotheses for other vertices, $SL(v)$ is defined correctly with respect to T_{i-1} for all internal vertices existing in T_{i-1} at this point. However, $min(SL(oldhd))$ is out of date. We need to pick apart the proof of Lemma 1 to see the structure of bad suffix links at this point. From the first paragraph of the proof, we see that if the *j*th and *k*th p-suffixes do not diverge after the *p*th symbol, then the $(j-1)$st and $(k-1)$st p-suffixes diverge after the $(p-1)$st symbol only if the *p*th symbol of one is a 0 and the *p*th symbol of the other is a parameter pointer to the first symbol. In the case of *oldhd*, however, we can apply this statement only for $j<i$ and $k<i$ because the *i*th p-suffix is not yet in the tree. If *oldhd* was not created in Stage $i-1$, but has a bad suffix link, then it has three children, one of which is the new leaf created in Stage $i-1$, and the others of which have arcs whose labels start with 0 and a parameter pointer to the first symbol in the pathstring, respectively. If *oldhd* was created in Stage $i-1$, then it has exactly two children, *oldchild* and the new leaf, and the arc labels can start with anything. The second and third paragraphs still apply at all vertices, however, including *oldhd*. These observations imply that (1) Lemma 1 applies to all vertices other than $SL(oldhd)$, (2) if *oldhd* was not created in Stage $i-1$, Lemma 1 applies to $SL(oldhd)$, and (3) if *oldhd* was created in Stage $i-1$ and has a bad suffix link, the vertices with bad suffix links pointing to $SL(oldhd)$ occur in a chain, and for every vertex except *oldhd*, the arc leading to the next vertex begins with a 0, while at *oldhd*, the next vertex if *oldchild*.

To satisfy P2, the only bad suffix links that need to be changed between the end of Phase 1 and the end of Stage *i* are those that currently point to *g* but will need to point to *newhd*. The only values of *min* that may need to be changed are those for *newhd*, *g*, and $SL(oldhd)$. From above, the *update* procedure operates correctly starting at $min(g)$ to reset bad suffix links to *newhd* as needed and to update $min(newhd)$ and $min(g)$, except that it will miss *oldhd* if *oldhd* has a bad suffix link to *g* and has smaller pathlength than $min(g)$. However, *oldhd* is processed separately afterward to reset its suffix link to *newhd* if necessary and to reset $min(SL(oldhd))$ if necessary. We conclude that P2 and P3 hold at the end of Stage *i*.

Finally, we analyze the running time of *eager* phase by phase, beginning with the easy cases.

Constructing $prev(S)$ by means of a balanced tree takes time $O(n\log|\Pi|)$ and space $O(n)$. Phases 4 and 6 take constant time in each stage, or $O(n)$ time overall. Since arcs are assumed to be kept in a balanced tree with $O(\log\sigma)$ time for insertion, inserting at most two new vertices per stage in Phases 3 and 5 takes time $O(n\log\sigma)$ over all the stages.

We show that over all stages, Phase 2 scans a number of symbols that is linear in the size of the input. The proof is similar to McCreight's original proof. By the construction, if $head_{i-1}=psub(S,i-1,s)$, Phase 2 stops after reading the $(s+1)$th symbol of the input. In Stage *i*, Phase 2 starts at the $(s+1)$th symbol of the input. Thus, aside from scanning one symbol again at the beginning of Phase 2 in each stage, each symbol is scanned just once. Each access to a new symbol takes time $O(\log\sigma)$. Thus, the total time used for *scan* over all stages is $O(n\log\sigma)$.

For Phase 1 (rescanning), first consider parameter pointers that are treated as 0's when they are rescanned in Stage *i*. Since any pathstring in the tree can have at most $|\Pi|$0's, at most $|\Pi|$ such 0's can be rescanned in Stage *i*, and there are $O(n|\Pi|)$ such steps over all stages.

Now, consider rescanning other symbols other than the parameter symbols treated as 0's. In stage i, if $SL(oldhd)$ is undefined, rescanning starts at $SL(parent(oldhd))$, which means backing up in the input the equivalent of the last arc scanned or rescanned in the previous stage. Since we showed above that scanning in Phase 2 never backs up in the input, the last access to each symbol before rescanning can be by scanning rather than rescanning only once, except for the first symbol rescanned in each stage. Therefore, over all stages, there are only $O(n)$ rescanning steps that are due to rescanning symbols whose last access was by scanning.

We would like to know how many symbols rescanned in Stage i were last accessed by *rescanning*. Suppose $P[k]$ is such a symbol that is rescanned in Stage i. We will show that $P[k]$ is the first symbol rescanned in Stage i. For some $q<s$, the pathstring of $parent(oldhd)$ is $psub(S,i-1,q)$, $head_{i-1}=psub(S,i-1,s)$, and $q<k\le s$. Suppose $P[k]$ was last rescanned in Stage $j<i$. Then for some $r>k$, $head_j=psub(S,j,r)$, and there is a vertex v with pathstring $psub(S,j,k-1)$. Since v already had two children in Stage j, for some $p<j$, $psub(S,p,p+k-j-1)=psub(S,j,k-1)$, and $psub(S,p,p+k-j)\ne psub(S,j,k)$. Applying the Common Prefix Property inductively shows that $psub(S,i-1,k-1)$ = $psub(S,p+i-1-j,p+k-j-1)$. If $P[k]\in\Sigma$ or if $P[k]$ is a parameter pointer but $f(P[k],k-i+1)>0$, the Restricted Distinct Right Context Property applied at each application of the Common Prefix Property implies that $psub(S,i-1,k)\ne psub(S,p+i-1-j,p+k-j)$, and at the end of Stage $i-1$ a vertex z exists with pathstring $psub(S,i-1,k-1)$. Since $head_{i-1}=psub(S,i-1,s)$, the pathstring of $parent(oldhd)$ is $psub(S,i-1,q)$, and $q\le k-1<s$, we conclude that $z=parent(oldhd)$. Therefore, z has a good suffix link and $P[k]$ is the first symbol rescanned in Stage i. We conclude there are $O(n)$ such rescanning steps over all stages.

Since the rescanning in Phase 1 examines only the first symbol of each arc, each access to a child arc whose label begins with 0 takes time $O(1)$, and access to other children takes time $O(\log\sigma)$, we conclude that over all n stages rescanning takes time at most $O(n(|\Pi|+\log\sigma))$.

The length of a chain updated in Phase 3 by *update* is never greater than $|\Pi|$, plus one to account for *oldhd*. Since successive accesses in the chain are only to arcs beginning with 0, for which access time is a constant, the time for *update* over all stages is $O(|\Pi|n)$. □

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 [ST83] and concatenable queues [AHU74].

First, we describe how concatenable queues can be used to speed up updating of bad suffix links. Suffix links pointing directly to a vertex in the original construction will be replaced by pointers pointing indirectly through a concatenable queue. That is, for each vertex v, there will be a concatenable queue whose root has an associated pointer to v. Wherever the original construction would set $SL(w)=v$ for a vertex w, the new construction inserts a new leaf into the concatenable queue for v in time $O(\log n)$, and w 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 vertex *newhd* is made the parent of v is implemented in time $O(\log n)$ by splitting the concatenable queue for v based on the pathlength of *newhd*. Finally, in addition to maintaining these indirect suffix links for internal vertices, 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 $O(n\log n)$, and the space is linear in n.

Next, we review the properties of dynamic trees before describing how they 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 vertices in the trees. *Insert* merely inserts a new vertex into the dynamic tree. $Link(w,c)$ makes c a child of w. $Cut(a,b)$ cuts edge (a,b) to split off another dynamic tree. The *expose* operation applied to a vertex produces a solid path from that vertex to the root, represented as a locally biased search tree, and changes solid edges to dashed as needed so that solid edges occur only in paths. Each edge in a dynamic tree is assigned a cost, and the costs are maintained at vertices of the biased binary tree relative to the minimum value in subtrees, so that an integer can be added to all the costs in a subtree by changing the minimum for that subtree. Consequently, the costs can be maintained in $O(m\log k)$ time during m dynamic tree operations. Moreover, it is possible to search a biased binary tree with k leaves for the vertex of a particular cost in $O(\log k)$ time. Finally, we note that dynamic trees with k vertices use $O(k)$ space.

Except for briefly when a new vertex is being inserted into the p-suffix tree, our forest of dynamic trees will consist of a single tree. For our purposes, the cost of a solid edge will be the distance (in terms of path lengths in the p-suffix tree) from it to the vertex in its solid path that is closest to the root, while the cost of a dashed edge is 0; thus, after an *expose*, the cost of an edge is its pathlength in the psuffix tree.

The key property of dynamic trees for our purposes is the *expose* operation; $expose(v)$ produces a locally biased binary tree that represents the path from v to the root. Where the original construction follows a suffix link of the parent of *oldhd* and rescans to find the vertex 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 vertex v, calls $expose(v)$, and then searches the resulting locally biased binary tree to find the vertex that should become $SL(oldhd)$ based on its pathlength.

When the construction algorithm adds vertices to the p-suffix tree, the new algorithm uses dynamic tree operations as follows. To split an arc (a,b) of T_{i-1} to add a new vertex, it creates a new dynamic tree r consisting of a root, and then calls $cut(a,b)$, $link(a,r)$, and $link(r,b)$ to get back a single dynamic tree. When a new leaf is created as the child of a vertex v, it creates a new dynamic tree r consisting of one vertex and then calls $link(v,r)$ to connect it to the main dynamic tree.

Over all stages, there are $O(n)$ dynamic tree operations and $O(n)$ searches for a total of $O(n\log n)$ time. Since the number of children of any vertex in the p-suffix tree is bounded by the size of the input, each arc access in the p-suffix tree takes $O(\log n)$ time. Therefore, the operations besides the dynamic tree operations take $O(n\log n)$ time. Hence, the above 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 $O(n\log n)$.

An alternate and simpler approach that achieves the same time and space bounds is to replace concatenable queues and suffix links for non-leaf vertices by suffix links for leaves and operations in the dynamic trees. Suffix links for leaves are easily incorporated into the algorithm and remain up-to-date after creation. On creation of each non-leaf vertex, the new algorithm stores a pointer (obtained from information stored for a child) to a leaf descended from it. Where the above algorithm follows an indirect suffix link of a child of *oldhd*, the new algorithm follows a suffix link for the stored leaf descendant of *oldhd*; as before, the algorithm then does an *expose* and a search based on pathlength. This approach does not create the up-to-date suffix links for non-leaf vertices used for pattern-matching in the next section, however.

5. Searching for occurrences of a pattern

In this section,we consider how to search for p-matches of a pattern p-string P in a text p-string T using p-suffix trees. Given a p-suffix tree for T, which uses $O(T)$ space, it is possible to search it for p-matches to P in $O(|P|\log\sigma + tocc)$ time, where *tocc* is the number of matching positions [Baker93a]; this is done by starting at the root and tracing the path of P along arcs in the tree to the end of P (if possible), and reporting the p-suffix positions corresponding to the descendant leaves. By contrast, here we show how to search using a p-suffix tree for P and $O(|P|)$ space, an important difference in practice. This approach will be also be more efficient if the pattern is fixed but the text varies.

For strings, using ideas from [CL90], it is possible to search a suffix tree of a pattern P (technically, $P\$$) for occurrences of P in text T as follows. The basic idea for strings is to follow the path determined by the symbols of T through the tree, starting at the root and the first symbol of T. When the next symbol of T is not available in the tree, follow the suffix link from the last vertex reached. If this requires backing up in the input, rescan (by comparing the first symbols of arc labels to previously read input symbols) to catch up in the tree to where scanning stopped in the input, and resume scanning as before. Whenever the pathstring of the current position in the tree is P, report a match. This algorithm takes time $O(|T|\log|\Sigma|)$, if the children at each vertex are stored in a balanced tree scheme [AHU74] with $O(\log|\Sigma|)$ access time.

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 be the text p-string and P the pattern p-string.

To simplify the discussion below, we refer to $prev(T)$ as if it has been previously computed. However, computing $prev(T)$ as described in Section 2 would use $O(T)$ space or $O(|\Pi|)$ space, rather than $O(|P|)$ space. Now, symbols of $prev(T)$ are used only as the first argument to the function f described in Section 2, and the second argument is the pathlength in the tree, which is at most $|P|$in the algorithm below. Since $f(b,j) = 0$ for $b \ge j$, correctness of the algorithm is preserved by substituting a 0 for any $b \ge |P|$in $prev(T)$. Thus, we compute the values of $prev(T)$ (with 0 substituted for values $\ge |P|$) on the fly by using one balanced tree to keep track of the last occurrence of each symbol seen within the last P symbols and another balanced tree to keep track of when to delete symbols from the balanced trees because they haven't occurred within the last P symbols. These computations require $O(|P|)$ space and $O(|T|\min(\log|P|, \log|\Pi|))$ time.

Assume that a p-suffix tree has been built for the p-string $P\$$, where \$ is an endmarker that does not occur in P or T, and suffix links are available as defined in Section 3. We note that if $psub(T,i,q) = psub(P,j,j+q-i)$, then $psub(T,i+1,q)$=$psub(P,j+1,j+q-i)$ by the Common Prefix Property. Thus, if $psub(T,i,q)$ is a prefix of a pathstring in the p-suffix tree for P, so is $psub(T,i+1,q)$, and we can plan on using the suffix links created for P when we process T.

For $1 \le i \le |T|$, the algorithm compares the ith p-suffix of T to the p-suffixes of S in the ith stage. For $i = 1$, initialize the algorithm with $v = root$, and $len = 1$; in general, at the start of Stage i, v represents the vertex to begin searching from, and the next text symbol that needs to be matched is the $(i-1+len)$th symbol. The algorithm guarantees that the $psub(T,i,i+len-2)$ terminates in the tree at v, on the arc from $parent(v)$ to v, or below v. In the last case, rescanning by comparing the first symbols of arcs to the transformed input symbols is used to get to a point in the tree corresponding to pathstring $psub(T,i,i+len-2))$ and pathlength $len-1$. Starting with $j = len$, the algorithm scans in the direction away from the root in the tree and for successive values of j tries 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. If the scanning passes through the arc 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, let w be the parent of the vertex reached by the arc currently being examined for the jth symbol of the ith p-suffix. Set $v = SL(w)$ and increment i; if $j = 1$, set $len = j$, and otherwise, set $len = j - 1$ to complete Stage i.

With this algorithm, we obtain the following result.

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

The proof of Theorem 3 is omitted as it is a straightforward application of the same ideas used in the proof of Theorem 1.

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. One possibility is to use dynamic trees again. In this case, when a suffix link is to be followed, we follow the suffix link of the child vertex, do an *expose*, and then a binary search on the auxiliary biased tree to find the point at which searching should resume. The running time would be $O(|T|\log|P|)$.

However, since the p-suffix tree is static for this searching problem, we can take advantage of lowest common ancestor algorithms on static trees to do better. Recall that the factor of $|\Pi|$ in Theorem 3 is caused by the rescanning of vertices whose labels start with zero. From [HT84] or [SV88], it is possible to construct auxiliary data structures in $O(|P|)$ preprocessing time, so that the lowest common ancestor (LCA) of two vertices can be computed in $O(1)$ time. We augment the p-suffix tree with their auxiliary data structures, which we will call the LCA data structures. In addition, for each chain of 0's in the tree, we create an array that contains pointers to the vertices and the pathlengths of the vertices, so that a query to find a vertex of a given pathlength in the chain can be accomplished by binary search in time $O(\log c)$, where there are c vertices in the chain. In addition, for each vertex corresponding to an array element, we also create a pointer to the start of its array. (In this case, by a chain of 0's, we mean merely successive arcs whose labels begin with zero, and the vertices need not have bad suffix links or satisfy the other requirements of Lemma 1.) We also assume that suffix links have been created for leaves as well as for internal vertices, which is easily done since the suffix link for the leaf created in Stage i points 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 vertex we are seeking. In particular, when we follow the suffix link for an internal vertex w preparatory to rescanning, we also remember a vertex q: if the pathlength of w is exactly the length of the substring of T that has been matched, then $q = w$, but otherwise, pick q to be the child along the arc that was partially matched by the substring of T. If $q = w$, no rescanning need be done, since the suffix link for w is defined. Otherwise, the original rescanning phase would rescan down to a vertex that is an ancestor of $u = SL(q)$. This time, rescanning works as follows. When the next transformed input symbol to be matched is not a 0, rescanning finds the appropriate arc as before. However, when the next transformed input symbol to be rescanned is a 0, the corresponding arc is the start of a chain of 0's, and we find the LCA of the bottom of the chain and u; 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 vertex.

THEOREM 4. Let Σ and Π be finite disjoint alphabets. Given a text p-string T and a p-suffix tree for a pattern p-string P, all p-matches of P with T can be reported in time $O(|T| \log(\min(|P|,\sigma))$.

Proof. As for the previous algorithm, the Common Prefix Property holds for substrings of T and S, so that when a suffix link is followed, the parts of the input (up to the $(j-1)$th symbol) already matched do not need to be recompared during rescanning, except for the first symbol at certain arcs to determine what path to follow in the tree. If an arc beginning with 0 is in the path, the new rescanning algorithm finds the last 0 of this chain that is in the path by using the LCA of the bottom of the chain and a vertex u known to be below the desired vertex in the path. Thus, the arcs in the path whose first symbols are 0 do not need to be checked one by one. Only the remaining arcs in the path need to be checked. Therefore, the new rescanning method works correctly.

The preprocessing time to construct the LCA data structures is $O(|P|)$ [HT84,SV88]. For each chain, creating its array takes time and space linear in the number of vertices in the chain, even for a variable alphabet, because arcs beginning with 0 can be accessed in $O(1)$ time, and searching the tree to create all the arrays takes time $O(|P| \log(\min(|P|,\sigma)))$. As before, scanning takes $O(n \log(\min(|P|,\sigma)))$ time. In rescanning, for each chain of 0's that is encountered, $O(1)$ time is used for the LCA operation, and $O(\log(\min(|P|,|\Pi|)))$ time is used for the binary search of the array; for each arc intervening between chains, $O(\log(\min(|P|,\sigma)))$ time is used. As in the proof of Theorem 3, over all stages, the total number of arcs rescanned whose labels do not begin with 0 is linear in $|T|$. Therefore, the number of chains of 0's that are searched is also linear in $|T|$. We conclude that the total amount of time spent by the algorithm on rescanning is $O(|T| \log(\min(|P|,\sigma)))$. The amount of space required for the p-suffix tree and arrays is $O(|P|)$, and the amount of space required for the LCA data structures is also $O(|P|)$ [HT84,SV88]. □

6. An application

The p-suffix tree construction algorithm *eager* (with no auxiliary data structures) of Section 4 has been implemented and tested in combination with the implementation of the algorithm *pdup* described in [Baker93a] for finding all duplication within a p-string. 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. The implementation uses a semi-line-based approach as follows. Comments and white space are ignored. Tokens such as identifiers and constants are hashed to obtain integers representing parameter symbols. Each line is transformed by replacing each occurrence of a parameter token by the same symbol P, and the resulting string is hashed to obtain an integer representing a symbol of Σ; the P's are place-markers to record the positions of the parameters within the line. Thus, each line corresponds to a p-string consisting of one symbol of Σ followed by a (possibly empty) sequence of parameter symbols representing the parameter tokens in the order in which they occur in the original line. The parameterized string for the entire program has n symbols of Σ and p symbols of Π if the entire program has n lines other than comments and white space and p parameter tokens. Arcs are accessed by hashing (as suggested by McCreight [McC76] 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 large software system. 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 40 MHz R3000 processor (primary I and D cache 64KB, secondary cache 1 MB, and main memory 256 MB, SGI IRIX).

To relate this performance with the theoretical worst case bound of Theorem 1, $|\Pi|$ can be taken to be *n* since it would be expected to grow linearly with *n* in this application. Since Theorem 1 gives a bound of $O(n(|\Pi| + \log\sigma))$ time for *eager*, where *n* is input length, this worst case bound is quadratic in *n*. 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. 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 in terms of maximal p-matches, defined as follows. For a p-string $S = b_1 b_2 \cdots b_n$, we say that $b_i b_{i+1} \cdots b_{i+k}$ and $b_j b_{j+1} \cdots b_{j+k}$ are a *maximal p-match* if $b_i b_{i+1} \cdots b_{i+k}$ and $b_j b_{j+1} \cdots b_{j+k}$ are a p-match and the p-match cannot be extended by a symbol in either direction, *i.e.* $psub(S, i-1, i+k) \neq psub(S, j-1, j+k)$ and $psub(S, i, i+k+1) \neq psub(S, j, j+k+1)$; this p-match is represented in terms of the starting positions and length by $(i, j, k+1)$ or (by symmetry) $(j, i, k+1)$. In fact, a maximal p-match cannot be extended in either or both directions by any amount [Baker93a].

For a p-string *S* and any positive integer *t*, recall that $m(t, S)$ is the number of maximal parameterized matches of length at least *t* within *S*. Thus, $m(t, S)$ measures all the parameterized duplication in *S* over the threshold length *t*; for our intended application, experiments with **dup** have found $m(t, S)$ to be less than $|S|$ unless *t* is small. The time for *eager* to build a p-suffix tree for a p-string *S* can be analyzed in terms of *t* and $m(t, S)$ as follows.

THEOREM 5. Given disjoint alphabets Σ and Π and a p-string *S* of length *n*, *eager* constructs a p-suffix tree for *S* in time $O(\min(tn + m(t, S) \mid t > 0) \log\sigma)$, and σ is the sum of the alphabet sizes.

Proof. Let *t* be a positive integer. From the proof of Theorem 1, the time used for scanning and inserting new vertices is bounded by $O(n \log\sigma)$. We will show that over all stages, rescanning takes time $O((tn + m(t, S)) \log\sigma)$, and updating suffix links takes time $O(tn + m(t, S))$.

To get the bound on rescanning, we show that rescanning checks the first symbol of at most $O(tn + m(t, S))$ arcs, with $O(\log\sigma)$ access time for each arc. We claim that except for the first and last vertex rescanned in each stage, each vertex visited in rescanning corresponds to a distinct maximal p-match in the input string. For consider Stage *i*. For some $s \geq i-1$, the pathstring of *oldhd* is $psub(S, i-1, s)$. For some $r \leq s$, the pathstring of *parent*(*oldhd*) is $psub(S, i-1, r)$. In Stage *i*, if rescanning occurs, it begins at *SL*(*parent*(*oldhd*)), whose pathstring is $psub(S, i, p)$ for some p such that $r \le p \le s$, and ends at the vertex whose pathstring is the shortest extension of $psub(S,i,s)$. Consider a vertex v visited during rescanning, other than the first or last; thus $p-i+1 < pathlen(v) < s-i+1$. Since v is a branching vertex, it has an arc that is not traversed in rescanning. Let L be any leaf reached via this arc. For some $k<i$, the pathstring of L is $psuffix(S,k)$, and the pathstring of v is $psub(S,k,k-1+pathlen(v))$. We claim that $(k,i,pathlen(v))$ is a maximal p-match. For it can't be extended to the right, because $psub(S,k,k+pathlen(v)) \ne psub(S,i,i+pathlen(v))$, by choice of L. Moreover, if it could be extended one position to the both the left and right, *i.e.* $psub(S,k-1,k+pathlen(v)) = psub(S,i-1,i+pathlen(v))$, the Common Prefix Property would imply $psub(S,k,k+pathlen(v)) = psub(S,i,i+pathlen(v))$, contradicting the choice of L. So suppose the p-match can be extended one position to the left, *i.e.* $psub(S,k-1,k-1+pathlen(v)) = psub(S,i-1,i-1+pathlen(v))$. Since this match can't be extended to the right by one position, these two prefixes branch at pathlength $pathlen(v)+1$ in the tree. This means that they branch at a vertex between *oldhd* and its parent, because $pathlen(parent(oldhd)) = r-i+2 \le p-i+2$ (from above), $p-i+2 < pathlen(v)+1 < s-i+2$ (by choice of v), and $s-i+2 = pathlen(oldhd)$. But this is a contradiction, implying that the p-match could not really be extended one position to the left. We conclude that $(k,i,pathlen(v))$ is a maximal p-match. This maximal p-match cannot be reported in any stage besides i, since it includes i and $k<i$. Also, at most t of the maximal p-matches corresponding to this stage have length less than t. We conclude that the number of arcs searched in rescanning during all stages is $O(tn+m(t,S))$, for a total time used of $O((tn+m(t,S))\log\sigma)$.

To get the bound on *update*, we show that at most $O(tn+m(t,S))$ suffix links get updated at $O(1)$ cost for each update. Recall that the suffix links updated are those that are changed to point to *newhd*. Of the vertices on the path from the root through the chain whose suffix links gets changed in Stage i, let $\bar{v}$ be the one of greatest pathlength that is either *oldhd* or an ancestor of *oldhd*. Since the root is on this path, $\bar{v}$ is defined. For any vertex v' whose suffix link gets modified, pick a leaf L reached through the child arc of v' whose first symbol is not 0. For some $j<i-1$, the pathstring of L is $psuffix(S,j)$. We claim that $(i,j+1,|head_i|)$ is a maximal p-match if v' is not $\bar{v}$ or *oldhd*.

Since $SL(v')$ is set to *newhd* in Stage i, the leaf whose pathstring is $psuffix(S,j+1)$ is a descendant of *newhd*. Since the leaf for $psuffix(S,i)$ is a descendant of *newhd* through a different arc, $(i,j+1,|head_i|)$ is a p-match that cannot be extended by one position to the right.

Next, we show that this p-match cannot be extended by one position to the left. Both $\bar{v}$ and v' are at pathlengths of at most $|head_i|+1$ in the tree. The length of the common prefix of $psuffix(S,i-1)$ and $psuffix(S,j)$ is the length of the common prefix of the pathstrings of $\bar{v}$ and v', by choice of $\bar{v}$ and v'. Since v' and $\bar{v}$ were chosen to be distinct, the length of the common prefix of their pathstrings is at most $|head_i|$, which means the p-match of length $|head_i|$ starting at i and $j+1$ cannot be extended one position to the left.

We conclude that $(i,j+1,|head_i|)$ is a maximal p-match. Since the j's that would be picked for different leaves defined as above would all be different, and at most *oldhd* and one chain vertex in each stage are excluded as v', the number of vertices updated in all stages is at most the number of distinct maximal p-matches plus two per stage. At most t of these in each stage have length less than t. Therefore, the total number of suffix links updated is at most $O(tn+m(t,S))$. Since the time to access each arc in the chain and change it is $O(1)$, the total time used for updating suffix links is $O(tn+m(t,S))$. □

Now, the time bound of Theorem 5 is quadratic in input length in the worst case, even for fixed alphabets, since $m(t,S)$ can grow quadratically in $|S|$ in the worst case. However, experiments suggest that in practice, $m(t,S) < |S|$, unless t is small. In particular, experiments with

dup on amounts of code ranging from 1000 to a million lines, with thresholds of 10-50 matching lines, found $m(t,S)$ always to be less than n. **Dup**'s definitions of maximal match and threshold length are slightly different, namely line-based, but that should not affect these order-of-magnitude conclusions; preliminary experiments with *eager* and *pdup* confirm these observations. (To be fair, the code did not include large tables such as font tables, where quadratic growth in the number of matches might be more likely. For ordinary C code, when $t=1$, $m(t,S)$ grows quadratically in $|S|$ due to lines consisting of only a closing brace.) 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 $m(t,S)$ exceeds n for a sample of 158579 lines of code, and the experiments revealed that $m(t,S)<|S|$ for $t \geq 20$. Thus, this alternative analysis suggests a reason that the observed running time of *eager* (using hashing to access arcs) should appear linear.

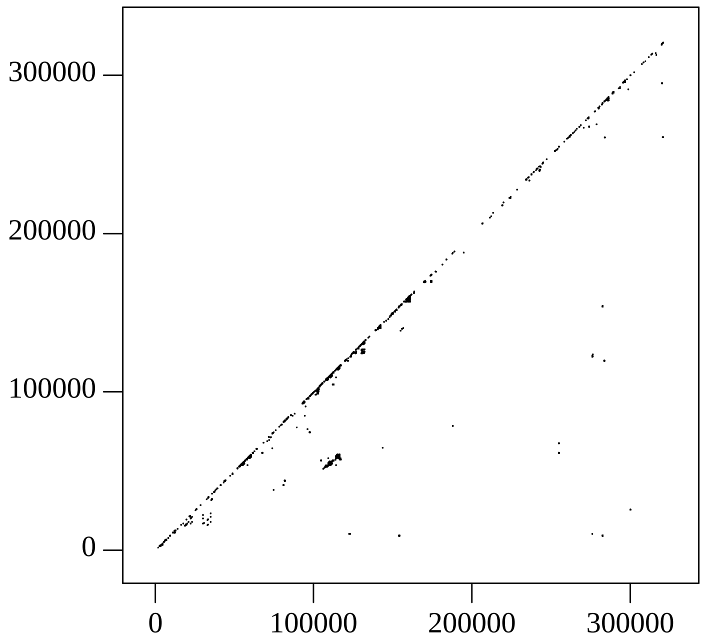

Figure 4. A scatter plot of maximal p-matches of length at least 75 symbols in about 101,000 lines of C code (excluding comments), corresponding to a p-string of about 321,000 symbols.

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 $m(t,S)$ does not grow faster with respect to $|S|$: the density near the main diagonal and the sparseness away from the main diagonal show that much of the duplication is fairly local. For small thresholds, under about 10 lines, there are many pairs of short matching code segments dominated by closing braces, and these are less likely to be local; however, these matches are not likely to be useful, either. 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.

While the construction algorithm for p-suffix trees described in [Baker93a] also appears to run in linear time in practice, based on the number of rescanning steps and updating steps observed, the proof of Theorem 5 does not apply to that algorithm. In particular, the pathlength arguments used in the proof break down because the definition of suffix links is different.

7. Related work and areas for further research

The algorithm *eager* from this paper and the algorithm *pdup* from [Baker93a] have been incorporated into a new version of **dup** [Baker92], using threshold lengths measured in lines. On the software subsystem of 1085432 lines mentioned in the last section, corresponding to a p-string of length 1931508, the new algorithms produced a factor of two speedup for a threshold of 30 lines, and a factor of 8 speedup for a threshold of 15 lines. In the latter case, the improvement from about an hour to seven minutes is quite noticeable to the user.

The simplicity of the *prev* encoding of p-strings suggests that other string pattern matching results may generalize to p-strings. Some work along these lines has begun. The author has developed generalizations of the Knuth-Morris-Pratt and Boyer-Moore algorithms for parameterized pattern-matching [Baker93c]. Amir *et al* have also generalized the Knuth-Morris-Pratt algorithm to find p-matches in $O(n \log(\min(m, |\Pi|)))$ time, where n is text length and m is pattern length, and have shown that this is optimal under a comparison model [AFM93]. For p-matching a set of p-string patterns, Idury and Schaffer [ID93] have developed generalizations of the Aho-Corasick algorithm [AC75] for matching multiple patterns and of their dynamic dictionary algorithm for strings [ID92]. Let *tocc* be the number of occurrences found, and d be the sum of the lengths of the patterns. Their parameterized multiple pattern matching algorithm finds p-matches to a set of p-string patterns in time $O(|T| \log \sigma + tocc)$, with preprocessing time $O(d \log \sigma)$. Their parameterized dynamic dictionary algorithm finds p-matches to a set of p-string patterns in time $O((|T| + tocc)(\log \sigma + \log d)$, inserts a pattern of length p in time $O(p(\log \sigma + \log^2 d))$, and deletes a pattern of length p in time $O(p(\log \sigma + \log d))$. The work on p-suffix trees in this paper was motivated by the usefulness of p-suffix trees for finding all maximal p-matches within a p-string in the software application. These other approaches may be more efficient in practice for parameterized pattern matching.

From a practical point of view, finding all p-matches to a pattern or to an arbitrary set of patterns is not always exactly the right model. If a programmer is fixing a bug and would like to know if the same bug occurs elsewhere, searching for just a line or two would retrieve an impossibly large set of p-matches, while searching for a long section of text might retrieve no p-matches because it misses copies that had a change in the middle. Therefore, while the programmer would presumably want to include some code surrounding the bug being fixed, he/she would not know how much to include in the pattern. Looking for an approximate p-match of restricted edit distance is a possibility, except that it is not clear how to mesh the definition of edit distance with that of parameterized patterns: if two lines in the middle violate the one-to-one pairing of parameters for all the other lines, is that acceptable? Another possibility would be to look for p-matches between pattern and text of minimum threshold length; some combination of the pattern-matching techniques of this paper with the techniques for finding parameterized duplication over a threshold length in [Baker93a] might yield an efficient algorithm.

Acknowledgements.

The author would like to thank Raffaele Giancarlo for many discussions relating to this work.

References

[Aho90] 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.

[AC75] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, *Comm. ACM 18* (1975), pp. 333-340.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (1974), Addison-Wesley, Reading, Massachusetts.

[AFM93] Amihood Amir, Martin Farach, and S. Muthukrishnan, Alphabet dependence in parameterized matching, submitted for publication, 1993.

[Baker92] Brenda S. Baker, A program for identifying duplicated code, *Computing Science and Statistics 24* (1992), Interface Foundation of North America, pp. 49-57.

[Baker93a] Brenda S. Baker, Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance, submitted for publication, March, 1993.

[Baker93b] Brenda S. Baker, A theory of parameterized pattern matching: algorithms and applications (extended abstract), *Proc. 25th ACM Symposium on Theory of Computing*, May, 1993, pp. 71-80.

[Baker93c] Brenda S. Baker, Parameterized pattern-matching using a Boyer-Moore approach, in preparation.

[CL90] 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.

[Croch87] 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.

[Gian93] Raffaele Giancarlo, The suffix tree of a square matrix, with applications, *Proc. of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (1993), pp. 402-411.

[HT84] 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).

[IS92] Ramana M. Idury and Alejandro A. Schaffer, Dynamic dictionary matching with failure functions, *Proc. Third Symp. on Combinatorial Pattern Matching*, *Lecture Notes Comp. Sci. 644* (1992), pp. 276-287.

[IS93] Ramana M. Idury and Alejandro A. Schaffer, Multiple Matching of Parameterized Patterns, submitted for publication, 1993.

[McC76] E.M. McCreight, A space-economical suffix-tree construction algorithm, *J. ACM* 23,2 (1976), pp. 262-272.

[SG86] R.W. Scheifler and J. Gettys, The X window system, *ACM Transactions on Graphics 5,2* (1986), pp. 79-109.

[SV88] Baruch Schieber and Uzi Vishkin, On finding lowest common ancestors: simplification and parallelization, *SIAM J. on Computing 17* (1988), pp. 1253-1262.

[ST83] Daniel D. Sleator and Robert E. Tarjan, A data structure for dynamic trees *J. Comput. Sys. Sci. 26* (1983), pp. 362-391.