Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance*

Brenda S. Baker

AT&T Bell Laboratories, Room 2C-457
600 Mountain Avenue
Murray Hill, NJ 07974
email: bsb@research.att.com

ABSTRACT

As an aid in software maintenance, it would be useful to be able to track down duplication in large software systems efficiently. Duplication in code is often in the form of sections of code that are the same except for a systematic change of parameters such as identifiers and constants. To model such parameterized duplication in code, this paper introduces the notions of parameterized strings and parameterized matches of parameterized strings. A data structure called a parameterized suffix tree is defined to aid in searching for parameterized matches. For fixed alphabets, algorithms are given to construct a parameterized suffix tree in linear time and to find all maximal parameterized matches over a threshold length in a parameterized p-string in time linear in the size of the input plus the number of matches reported. The algorithms have been implemented and experimental results show that they perform well on C code.

keywords: string matching, pattern matching, duplication
AMS subject classifications: 68Q25, 68Q20, 68R15

March 23, 1993; Revised Nov 17, 1995

---

*Some of the results in this paper were presented at the 25th Annual ACM Symposium on Theory of Computing, May, 1993.

Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance*

Brenda S. Baker

AT&T Bell Laboratories, Room 2C-457
600 Mountain Avenue
Murray Hill, NJ 07974
email: bsb@research.att.com

1. Introduction

In a large ongoing systems project, introduction of new features and code maintenance by large staffs of programmers may result in code that includes many duplicated sections. Such duplication still occurs even though it has long been known that copying code may make the code larger, more complex, and more difficult to maintain, For example, when a new feature is introduced, rather than risk breaking a working feature by making a major revision, a programmer might choose to leave the old section of code untouched and to add another slightly modified copy of it for the new feature. A bug fix might also be handled by copying and modifying the code, for example if the original programmer omitted handling of special cases. The copies might be further copied and modified as time goes on. With time, the amount of duplication in a system can become substantial and can complicate maintenance.

While some of the duplication in a software system may involve sections of code that are identical, much of the duplication involves sections of code that are not identical, but the same except for a systematic change of parameters such as identifiers and constants, *e.g.* each occurrence of *first*, *last*, 0, and *fun* in one section may be replaced by *init*, *final*, 1, and *g*, respectively, in the other section; this kind of correspondence between sections of code is called a *parameterized match* [Bak1]. [Bak1] describes a program **dup** that finds all maximal parameterized matches over a threshold length in C code; application of **dup** to a million-line subsystem of a production system revealed that 21% of the lines were involved in parameterized matches of at least 30 lines (excluding comments and white space).

This paper formalizes the notion of parameterized matches for code in terms of *parameterized strings*, or *p-strings*, which are strings over two alphabets: an alphabet of constant symbols and an alphabet of parameter symbols. Two parameterized strings are a *parameterized match*, or *p-match*, if they are the same except for a one-to-one correspondence between the parameter symbols occurring in them, *e.g.* *axbxyazyx* and *aubuvaxvu* are a p-match where the one-to-one correspondence maps the x, y, and z of the first p-string into the u, v, and x, respectively, of the second p-string.

For use in searching p-strings, we define a new data structure called a *parameterized suffix tree*, or *p-suffix tree*. We show that a p-suffix tree can be built in time and space $O(n)$, where n is the length of the input, if the alphabets are fixed. Given a pattern p-string P of length m and a text p-string T of length n over fixed alphabets, a p-suffix tree for T can be used to determine in $O(n+m)$ time and $O(n)$ space whether P has a p-match in T. An algorithm is given that finds all maximal p-matches over a threshold length in a p-string S in time $O(n+r)$, where n is the length of the input and r is the number of matches reported, by searching a p-suffix tree constructed for S. The algorithms for constructing p-suffix trees and for reporting all maximal p-matches over a threshold length have been implemented. Experiments show that the program performs well on C code input.

The algorithms in this paper improve upon the somewhat *ad hoc* method of finding parameterized matches in **dup**. The parameterized-matching algorithm implemented in **dup** operates as follows: it transforms all parameter candidates such as identifiers and constants to the same symbol P, finds exact matches in the transformed code, and then checks the exact matches for possible parameterized matches; the worst-case running time is not a function of the size of the input and number of p-matches, or even of the size of the input and the total length of the p-matches. In some cases, as many as 99% of the exact matches found do not correspond to parameterized matches.

P-suffix trees are a generalization of suffix trees for strings [McC,Uk,We], but are more dynamic in that each access to an input symbol requires a transformation based on the depth in the tree. The algorithm described here for building p-suffix trees is based on the McCreight algorithm for building suffix trees [McC]; however, the original algorithm and the concept of suffix links used in it must be modified to allow for the dynamic way in which p-strings are handled and the failure of a key property of strings to generalize for p-strings. The algorithm for finding all maximal p-matches over a threshold length is a generalization of the suffix-tree-based algorithm implemented in **dup** for finding all exact matches over threshold length in strings [Bak1, Bak2]; again, the generalization is not straightforward because of the dynamic nature of p-suffix trees.

The generalization of suffix trees to p-strings is related to Giancarlo's generalization of suffix trees to L-strings [Gi] in that both have to deal with the failure of the same key property of strings to generalize. In other respects, L-strings and p-strings behave differently, because L-strings do not obey the restricted form of this property that can be proved for p-strings. Consequently, the linear bounds obtainable for constructing p-suffix trees with fixed alphabets do not appear to be obtainable for constructing L-suffix trees for L-strings.

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].

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 parts of terms.

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.

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 and do not correctly implement the definitions [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. These programs also do not address the problem of finding all duplication over a threshold length.

The paper is organized as follows. Section 2 defines parameterized strings, parameterized matches, and parameterized suffix trees, and shows how parameterized suffix trees can be used for parameterized pattern matching. The algorithm for constructing a parameterized suffix tree is given in Section 3. Section 4 gives the algorithm for reporting all parameterized matches over a threshold length in a p-string. An implementation of the algorithms and some experimental results from applying them to C code are described in Section 5. The last section discusses time bounds for the algorithms for variable alphabets and directions for further research

2. Parameterized strings, parameterized matches, and parameterized suffix trees.

In this section, we introduce parameterized strings, parameterized matches, and parameterized suffix trees (p-suffix trees), and show how parameterized suffix trees can be used for parameterized pattern matching. We assume a RAM model of computation with the uniform cost criterion [AHU].

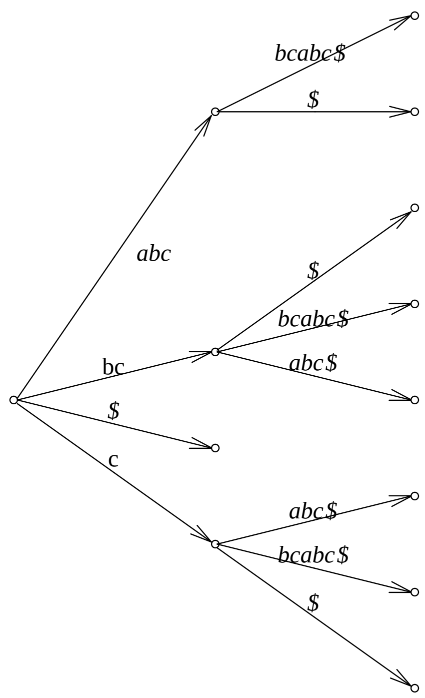

Fig. 1. A suffix tree for the string *abcbcabc$*.

Throughout, Σ will be a fixed finite alphabet of constant symbols and Π will be a fixed finite alphabet of parameter symbols, *i.e.* the sizes of Σ and Π are $O(1)$. 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.

DEFINITION. A string of symbols in $(\Sigma \cup \Pi)^*$ 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 renaming the parameters via a one-to-one function whose domain is the set of parameter symbols occurring in one p-string and whose range is the set of parameter symbols occurring in the other p-string.

For example, if *x*, *y*, and *v* are parameter symbols and *a*, *b*, and *c* are constant symbols, 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.

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. In addition to mismatches in length, mismatches can be 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 to a different parameter. Given our definitions, checking for mismatches can be done in time linear in input length *n* and space $O(|\Pi|)$, by constructing a table for the one-to-one correspondence. Unfortunately, this approach 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 *u*, *v*, *x*, and *y* are parameter symbols and *a* and *b* are constant symbols, then $prev(abuvabuvu) = ab00ab442 = prev(abxyabxyx)$. (Each parameter pointer is a single digit here.)

Since symbols of Π can be used to index into a table, computation of *prev* can be done in time linear in input length and space linear in $|\Pi|$ by means of a table containing the position of last occurrence of each parameter symbol encountered. Proving the following proposition is straightforward from the definitions.

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

DEFINITION. Define the *i*th *p-suffix* of a p-string $S = b_1 b_2 \cdots b_n$ to be $psuffix(S,i) = prev(b_i b_{i+1} \cdots b_n)$. Define $prefix(S,i,j) = prev(b_i b_{i+1} \cdots b_j)$, for $i \le j$. Define $prefix(S,i,j)$ to be the empty string if $j < i$.

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*. For example, if $prev(S) = a0a2ab3$, then $psuffix(S,3) = a0ab3$. It is easily seen that $prefix(S,i,j)$ is the prefix of length $j-i+1$ of $psuffix(S,i)$.

The following proposition follows directly from the definitions and Proposition 1.

PROPOSITION 2. 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 $prev(P) = prefix(T,i,i+|P|-1)$.

We also note that the value of the *j*th symbol of $psuffix(S,i)$ can be computed in constant time from *j* and the corresponding $(j+i-1)$st symbol *b* of $prev(S)$. We let *f* be this function; the value of *f* is as follows.

DEFINITION. For $b \in \Sigma \cup \mathrm{N}$, if *b* is a nonnegative integer larger than $j-1$, $f(b,j) = 0$; otherwise $f(b,j) = b$.

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

Suppose that Σ is an alphabet and $S = a_1 a_2 \cdots a_n$ is a string, where each $a_i \in \Sigma$. For each i, $1 \le i \le n$, the substring $a_i \cdots a_n$ is a *suffix* of the input. Without loss of generality, we assume that the last symbol a_n is a unique endmarker; consequently, no suffix is the prefix of another suffix. A suffix tree is a compacted trie (multiway Patricia trie) over the alphabet $\Sigma \bigcup N$ representing the suffixes of the input string [McC]. A suffix tree is shown in Figure 1. Each arc of the tree is labelled with a nonempty substring of the input, each internal (non-leaf) vertex has degree at least two, and for each internal vertex, the arcs to its children have labels beginning with distinct symbols. For each leaf, the concatenation of the labels on the path from the root to the leaf is a distinct suffix of *S*. Each vertex other than a leaf has at least two children. Since the number of internal vertices is less than the number of leaves, the number of vertices is at most $2n$. Because no suffix is a prefix of another suffix, there is a one-to-one correspondence between the leaves and the suffixes.

The generalization of suffix trees to p-suffixes of a p-string gives us p-suffix trees, defined as follows.

DEFINITION. If *S* is a p-string that ends with a unique endmarker in Σ, a *parameterized suffix tree*, or *p-suffix tree*, for *S* is a compacted trie (multiway Patricia trie) that stores the p-suffixes of *S*.

Each arc in a p-suffix tree for *S* represents a nonempty substring of a p-suffix of *S*, each internal vertex has degree at least two, and the arcs from an internal vertex to its children have labels beginning with distinct first symbols. For each leaf, the concatenation of the labels on the path from the root to the leaf is a p-suffix of *S*. Since *S* ends with a unique endmarker, no p-suffix of *S* is the prefix of another p-suffix of *S*, and consequently, each p-suffix of *S* is the concatenation of the labels on a path from the root to a leaf. Thus, there is a one-to-one correspondence between leaves and p-suffixes of *S*, and the number of vertices in *S* is linear in $|S|$. Arcs are oriented from the root toward the leaves, so that arcs leaving a vertex point toward its children.

Example. Let $S = xbyyxbx\$$, where *x* and *y* are parameter symbols and *b* and *$* are constant symbols, so that $prev(S) = 0b014b2\$$. (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 2.

DEFINITION. For each vertex *v*, the *pathstring* of *v* is the concatenation of the labels on the path from the root to *v*, and *v* is the *locus* of its pathstring. The length of the pathstring of *v* is the *pathlength* of *v*.

In order for the p-suffix tree to be stored in space linear in input length, arc labels are calculated dynamically as follows. We assume that $prev(S)$ has been computed and stored in an array. For each vertex *v* other than the root, we store the pathlength $plen(v)$, an index $firstpos(v)$ into the input to specify the starting position of the label of the arc from its parent, and the length $arclen(v)$ of this arc. If a label symbol is at pathlength *j* from the root and corresponds to index *k* into the input, its value in the label is $f(b, j)$, where *b* is the *k*th symbol of $prev(S)$, and *f* is as defined above; extracting this value takes constant time.

Because up to *n* symbols of *N* can be used even when Π is fixed, it might seem that the number of children of each internal vertex is $O(n)$. However, this number is bounded by $|\Sigma| + |\Pi|$, for the following reason. The pathstring of any vertex *v* contains parameter pointers representing at most $|\Pi|$ distinct chains of parameters, and 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 of *v*), or a nonnegative integer pointing to the last element of one of these chains.

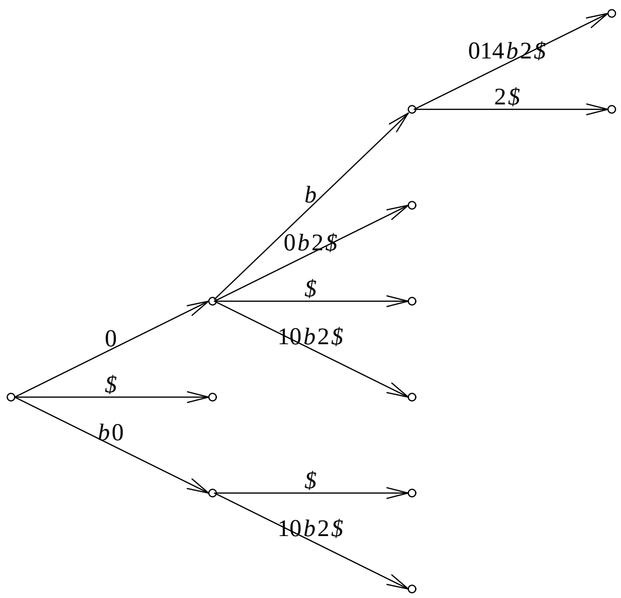

Fig. 2. A p-suffix tree for the p-string $S = xbyyxbx\$$, where $\Sigma = \{b, \$\}$ and $\Pi = \{x, y\}$.

For fixed alphabets, linked lists can be used to store the arcs, as far as the theoretical time bounds are concerned; in practice, for large alphabets, hashing would be used as suggested by McCreight [McC].

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 $O(|P|)$. The actual matching positions can be calculated from the descendant leaves. Thus, we obtain the following result.

THEOREM 1. Given p-strings P and T over fixed alphabets, $prev(T)$, and a p-suffix tree for $T\$$, where $\$$ is a unique endmarker, whether P has a p-match with a substring of T can be determined in time $O(|P|)$ and space $O(T)$. All positions of T at which P has a p-match can be found in time $O(|P| + k)$ and space $O(T)$, if there are k such positions.

In the next section, we will show that for fixed alphabets, a p-suffix tree can be built in time and space linear in the input length. Consequently, given p-strings P and T, whether P has a p-match with a substring of T can be determined in time $O(|P| + |T|)$ and space $O(|T|)$, and all positions of T at which P has a p-match can be found in time $O(|P| + |T| + k) = O(|P| + |T|)$ and space $O(|T|)$.

3. Building a p-suffix tree

In this section, an algorithm is given for constructing a p-suffix tree. We would like to imitate McCreight's algorithm for building suffix trees as much as possible; however, some basic changes must be made because of the difference between strings and p-strings.

Some useful properties

Strings have the following two trivial properties:

(1) (Common Prefix Property) For $a, b \in \Sigma$ and $S, T \in \Sigma^*$, if $aS = bT$, then $S = T$.

(2) (Distinct Right Context Property) Suppose $aS = bT$ and $aSc \neq bTd$, where $a, b, c, d \in \Sigma$ and $S, T \in \Sigma^*$. Then $Sc \neq Td$.

These two properties make it is possible to augment a suffix tree with pointers called *suffix links*. If an internal vertex has pathstring $a\alpha$, where a is a symbol and α is a string, its suffix link points to an internal vertex with pathstring α; in addition, the suffix link for the root points to the root. The definition of suffix links depends on the two properties in the following way. The existence of an internal vertex with pathstring $a\alpha$ implies there are two distinct strings sharing prefix $a\alpha$, the Common Prefix Property guarantees that stripping off the initial a from the two strings results in strings sharing an initial prefix α, and the Distinct Right Context Property implies that no longer prefix is shared, which in turn guarantees the existence of an internal vertex with pathstring α. 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.

L**EMMA** 1. (Common Prefix Property for p-strings) If $a, b \in \Sigma \cup \Pi$ and S and T are p-strings such that $prev(aS) = prev(bT)$, then $prev(S) = prev(T)$.

Proof. Observe that $prev(S)$ is different from $prev(aS)$ only in the deletion of the first symbol and in the changing of a parameter pointer pointing to the first position in $prev(aS)$ to a 0, if such a parameter pointer exists, and similarly for $prev(T)$ and $prev(bT)$. By equality of $prev(aS)$ and $prev(bT)$, such parameter pointers, if they exist, must be in the same position in these two p-strings. □

We would like to be able to generalize the Distinct Right Context Property to p-strings as follows: If $prev(aS) = prev(bT)$ and $prev(aSc) \neq prev(bTd)$, then $prev(Sc) \neq prev(Td)$, where $a, b, c, d \in \Sigma \cup \Pi$ and S and T are p-strings. Unfortunately, this is false, 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 form of the Distinct Right Context Property.

L**EMMA** 2. (Restricted Distinct Right Context Property for p-strings) Suppose $prev(aS) = prev(bT)$ and $prev(aSc) \neq prev(bTd)$, where S and T are p-strings of length k and $a, b, c, d \in \Sigma \cup \Pi$. If $prev(Sc) = prev(Td)$, then the last symbol of one of $prev(aSc)$, $prev(bTd)$ is $k+1$ while the last symbol of the other is 0.

Proof. Obviously, $prev(aSc)$ and $prev(bTd)$ differ only at their last symbol. Suppose $prev(Sc) = prev(Td)$. If the common last symbol is a nonzero parameter pointer or is in Σ, then the corresponding symbols in $prev(aSc)$ and $prev(bTd)$ also have a common value v, implying that $prev(aSc) = prev(aS)\,v = prev(aT)\,v = prev(aTd)$, a contradiction. The only other possibility is that the last symbols of $prev(Sc)$ and $prev(Td)$ are zero, and the last symbols of $prev(aSc)$ and $prev(bTd)$ are parameter pointers in $\{0, k+1\}$. They can't both be zero or both be $k+1$, since $prev(aSc) \neq prev(bTd)$. □

Overview of McCreight's algorithm

McCreight's algorithm for building suffix trees inserts suffixes in stages, where Stage i inserts suffix i (the suffix starting at position i of the input), for $i = 1, 2, \ldots, n$ (from left to right). 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 $|head_i|$ symbols; in Stage i, the algorithm inserts a new vertex hd_i at that point, if no vertex 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. A suffix link for a vertex with pathstring $a\alpha$, where a is a symbol and α is a string, points to the vertex with pathstring α (which must exist because of the Common Prefix property and Distinct Right Context Property); the suffix link for the root points to the root. Suppose Stage $i-1$ found that $head_{i-1}$ was $a\alpha$, where a is a symbol and α a p-string. Then $head_i$ will be at least as long as $|\alpha|$ because of the Common Prefix Property. In Stage i, if the suffix link is already defined for the vertex with pathstring $a\alpha$, there is no need to trace the common prefix α from the root; the processing can jump directly from the vertex with pathstring $a\alpha$ to the vertex with pathstring α via a suffix link. From that point, scanning can continue downward to find the desired location of 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 desired vertex, and follows a path downward in the tree while *rescanning* part of the input, up to a pathlength of $|\alpha|$. Fortunately, we know what symbols of the input were already scanned, just not where to go in the tree; thus, only the first symbol of each arc needs to be rescanned. If no vertex exists at this point (with pathlength $|\alpha|$), a new vertex hd_i is inserted, the missing suffix link is set to point to hd_i, and a leaf is created to represent suffix i. If a vertex does exist already at this point, the missing suffix link is set to it, and the *scanning* phase continues along the path corresponding to suffix i until the next symbol is not available in the tree; at this point, the algorithm creates an internal vertex hd_i if necessary, and a leaf to represent suffix i. (This explanation skips over some of the details of the algorithm.)

Our algorithm

We would like to generalize McCreight's algorithm to p-suffix trees. Unfortunately, we cannot in general define a suffix link for a vertex with pathstring $prev(aS)$ to point to a vertex with pathstring $prev(S)$, because that vertex may not exist due to the failure of the Distinct Right Context Property for p-strings. We can, however, define a modified suffix link, called a *contracted suffix link*. For a vertex with pathstring $prev(aS)$, the contracted suffix link points to the best available vertex, namely the one whose pathstring is the *contracted locus* of $prev(S)$, by the following definition.

DEFINITION. For $\alpha \in (\Sigma \cup N)^*$, the *contracted locus* of α is the vertex whose pathstring is the longest prefix of α of all vertices in the tree.

The contracted locus of α must exist because the empty string is a prefix of every string, and the root is the locus of the empty string.

The contracted locus of a pathstring may change as vertices are added to the p-suffix tree. Thus many contracted suffix links may need to be reset. But the following algorithm uses lazy evaluation, *i.e.* a contracted suffix link is reset only when it needs to be evaluated.

DEFINITION. For a p-string S and $i \geq 1$, define $head_i(S)$ as the longest prefix of $psuffix(S,i)$ that is also a prefix of $psuffix(S,j)$ for some $j < i$. Define $head_0$ to be the empty string.

LEMMA 3. If $head_{i-1}(S) = prefix(S, i-1, s)$, where $i-1 \le s$, then $head_i(S) = prefix(S, i, t)$ for some $t \ge s$.

Proof. By definition of $head_{i-1}$, there is some $j < i-1$ such that $head_{i-1}(S)$ is a prefix of $psuffix(S, j)$ as well as of $psuffix(S, i-1)$. Since $|head_{i-1}(S)| = s-i+2$, the first $s-i+1$ symbols of $psuffix(S, j+1)$ and $psuffix(S, i)$ must be the same by the Common Prefix Property. The result follows by definition of *head*. □

The construction of p-suffix trees follows the organization of McCreight's original algorithm [McC] as much as possible; modifications are needed to allow for updating out-of-date contracted suffix links and the extra searching resulting from out-of-date contracted suffix links. The values of *plen*, *firstpos*, and *arclen* are stored for the vertices as described earlier. In addition, for each vertex *v*, the contracted suffix link $CSL(v)$ is stored, and if *v* is not the root, a pointer to its parent $parent(v)$ is stored.

Let *S* be the p-string for which the p-suffix tree is to be constructed; we assume that *S* ends in a unique endmarker in Σ and $P = prev(S)$ has already been constructed in linear time and space as described in the previous section. The main procedure of the algorithm, called *lazy*, is given in Figure 3; additional procedures *prescan*, *rescan*, and *scan* called by *lazy* are described in the text. The *i*th iteration of the main loop of *lazy* will be referred to as Stage *i* and inserts the *i*th p-suffix into the tree.

The tree is initialized to a root, with $CSL(root) = root$, and $oldhd = root$. We will prove the algorithm correct inductively by means of the following properties, which we will show must hold for Stage *i*, $i \ge 1$.

P1: At the beginning of stage *i*, $CSL(v)$ has been set for the root and for every internal vertex *v* except possibly for *oldhd*, which is the locus of $head_{i-1}$; $CSL(root)$ points to *root*, and for a vertex *v* other than the root, if $CSL(v)$ is defined and the pathstring of *v* is $prefix(S, j, k)$, where $j \le k$, $CSL(v)$ points to a vertex whose pathstring is $prefix(S, j+1, t)$ for some *t*, $j \le t \le k$.

P2: At the end of Stage *i*, the tree is a compacted trie for the first *i* p-suffixes.

The goals in Stage *i* are to set $CSL(oldhd)$, to find or create *hd*, the locus of $head_i$, and to create a new leaf as the locus for $psuffix(S, i)$. Along the way, the contracted suffix link for $parent(oldhd)$ may be updated. In the following discussion, we assume that P1 and P2 held up to the start of Stage *i*.

If *oldhd* is the root, $CSL(oldhd)$ is already set to the root, and is up-to-date; the algorithm proceeds to call *scan*, described below.

So suppose *oldhd* is not the root. For $s = plen(oldhd) + i - 2$, $head_{i-1} = prefix(S, i-1, s)$. $CSL(oldhd)$ must be set to the contracted locus of $prefix(S, i, s)$. Fortunately, $prefix(S, i, s)$ is guaranteed to be a prefix of some pathstring already existing in the tree. This follows from the definition of $head_{i-1}$ and the Common Prefix Property.

Initially, *lazy* looks for a vertex *start* that is an ancestor of the contracted locus of $prefix(S, i, s)$. If $CSL(oldhd)$ is already defined (although possibly out of date), by property P1 $CSL(oldhd)$ can be used as *start*. Otherwise *lazy* begins by updating $CSL(parent(oldhd))$ and sets *start* to its updated value. This updated value is found by calling $prescan(parent(oldhd), i)$ and will be the contracted locus of $prefix(S, i, r)$ for some $r < s$.

If $parent(oldhd)$ is the root, $prescan(parent(oldhd), i)$ returns the root. Otherwise, it follows the path of $prefix(S, i, s)$ downward in the tree from the vertex pointed to by $CSL(parent(oldhd))$. It needs to check only the first symbol of each arc label since $prefix(S, i, r)$ is a prefix of $prefix(S, i, s)$, which we showed to be a prefix of some pathstring in the tree. It finds

```
/*index(i,len) translates a tree pathlength len into an index
into prev(S) for Stage i */
#define index(i,len) i-1+len
#define FP(i,j)         f(P[i],j)

lazy() {
   int i,short;
   VERTEX c,d,hd,oldhd,start;
   create a tree consisting of a root;
   oldhd=root;
   CSL(oldhd)=root;
   for (i=1;i<=n;++i) { /* Stage i */
      if (oldhd == root) hd=scan(root,child(root,S(0,i)),0,i,i);
      else {
         if (CSL(oldhd) is defined)
           start = CSL(oldhd);
         else
           start = CSL(parent(oldhd))= prescan(parent(oldhd),i);
         pgoal=plen(oldhd)-1;   /* pgoal = |prefix(S,i,s)| */
         c=rescan(start,pgoal,i); /*contracted locus of prefix(S,i,s) */
         short = pgoal-plen(c);
         /*compare next transformed symbol of prev(S) to corresponding
                symbol on the appropriate arc out of c */
         d=child(c,f(plen(c),index(i,plen(c)+1)));
         if ((d is defined) and (short>0) and
           (FP(index(i,pgoal+1),pgoal)!= FP(firstpos(d)+short,pgoal) {
           create a new vertex hd between c and d
             with arclen(hd) = short and firstpos(hd)=i-1+pgoal;
           CSL(oldhd) = hd;
           }
         else {
             CSL(oldhd) =  c;
             hd = scan(c,d,short,index(i,pgoal+1),i);
           }
         }
      }
      add a new leaf lf as a child of hd, with firstpos(lf)=i+plen(hd),
         arclen(hd)=n-i-plen(hd)-1;  /*  locus of psuffix(i) */
      oldhd=hd;
   }
}
```

Fig. 3. The main procedure for the algorithm for constructing p-suffix trees.

the contracted locus by not exceeding the desired pathlength, $plen(parent(oldhd)) - 1$.

Once *start* has been set, *lazy* finds the current contracted locus c of $prefix(S,i,s)$ by calling a function $rescan(start,pgoal,i)$, where $pgoal = plen(oldhd) - 1$. Now, *rescan* scans downward from *start* following the path of $prefix(S,i,s)$. Like *prescan*, *rescan* checks only the first symbol of each arc label because $prefix(S,i,s)$ is known to be a prefix of some pathstring in the tree. The contracted locus of $prefix(S,i,s)$ is found by not exceeding the desired pathlength, *pgoal*; the contracted locus is *short* symbols above where the locus of $prefix(S,i,s)$ would be (if it existed), for some $short \geq 0$.

While c is currently the contracted locus of $prefix(S,i,s)$, it may no longer be so at the end of the stage if a new vertex is created as the locus of $head_i$. By Lemma 3, for some $t \geq s$, $head_i = prefix(S,i,t)$, but the value of t is not yet known. The only case in which c will not be the contracted locus of $prefix(S,i,s)$ at the end of the stage is when $short > 0$ and $t = s$. If $short > 0$, whether $t = s$ is determined by checking the $(short + 1)$st transformed symbol on the arc from c to the appropriate child d.

Thus, the algorithm proceeds as follows. If $short > 0$ and $t = s$, a new vertex is created as the locus hd of $head_i$ and $CSL(oldhd)$ is made to point to it. Otherwise, the algorithm sets $CSL(oldhd)$ to c and sets hd to the vertex returned by $scan(c,d,short,i + pgoal,i)$.

$Scan(c,d,short,j,i)$ begins just after the $(short)$th symbol on the arc into d (or at c if $short = 0$) and the jth input symbol and scans downward in the tree along the path determined by $psuffix(S,i)$ until the next transformed input symbol is not available in the tree. At this point, *scan* creates a new vertex as the locus of $head_i$, if none exists already, and returns it.

Finally, a new arc is added from the locus hd of $head_i = prefix(S,i,t)$ to a new leaf, which is the locus of $psuffix(S,i)$.

Property P1 holds initially when the tree is initialized to just the root. In Stage i, the algorithm sets $CSL(oldhd)$ to the contracted locus for $prefix(S,i,s)$, implying P1 holds for *oldhd*, and care is taken to ensure that P1 still holds for $parent(oldhd)$ if its contracted suffix link was reset. No other contracted suffix links were changed. Hence, if P1 held at the beginning of this stage, it still holds at the end of the stage.

Property P2 holds at the beginning when the tree is initialized to just the root, and for $i > 1$, by the induction hypothesis, property P2 holds at the end of Stage $i - 1$. Either the locus of $head_i$ is created in Stage i by *lazy* because $short > 0$ and $t = s$, or it is created by *scan*. In either case, it is made the child of the old contracted locus of $head_i$ in the tree. Therefore, property P2 holds at the end of Stage i.

By induction, properties P1 and P2 hold for all stages. For $i = n$, P2 implies that the tree is a compacted trie for all the p-suffixes of S. Thus, the above algorithm constructs a p-suffix tree for S.

Analysis

THEOREM 2. Let Σ and Π be fixed finite disjoint alphabets. Given a p-string $S \in (\Sigma \cup \Pi)^*$ ending in a unique endmarker in Σ, a p-suffix tree can be constructed for S in time $O(n)$ and space $O(n)$, where n is the length of S.

Proof. Since correctness of the algorithm was shown above and linearity of space was shown when p-suffix trees were defined in Section 2, it only remains to analyze the running time. The proof is more complicated than that for McCreight's algorithm because of the parameter pointers and the use of contracted loci. In McCreight's algorithm, of the input symbols rescanned in a single stage, only the last can be rescanned again later, implying the time for rescanning is linear in the length of the string. In our case, the failure of the Distinct Right Context Property and the resulting use of contracted loci mean that from one stage to the next, rescanning can back up in the input and rescan again a sequence of symbols already rescanned, but at most a number proportional to $|\Pi|$ in any stage, for a total of $O(|\Pi| n)$ rescanning steps. Prescanning, not needed in McCreight's algorithm, also can recheck symbols already checked previously, but again at most a number proportional to $|\Pi| n$. The result will follow from our assumption that $|\Pi|$ is $O(1)$.

First, we observe that scan uses $O(n)$ time over all stages, because in each stage, *scan* is called at most once and scans at most one symbol scanned in an earlier stage and $|\Sigma|$ and $|\Pi|$ are

$O(1)$.

Next, we analyze the work required for prescanning.

Call a contracted suffix link *good* if it is for the root or if it is from v to $\bar{v}$, where the pathlength of $\bar{v}$ is one less than the pathlength of v. Otherwise, it is *bad*. By the Restricted Distinct Right Context Property, when a vertex is first given a bad suffix link, it has exactly two arcs, one whose label begins with 0 and one whose label begins with the pathlength of the vertex.

Let $BAD(y)$ be the set of vertices v that have bad contracted suffix links pointing to proper ancestors of y at the start of Stage i and whose contracted suffix links are reset to point to y or a descendant of y after y is created. Every prescanning step that checks the first symbol on an outarc of y is due to a distinct member of $BAD(y)$.

At the start of Stage i, every vertex in $BAD(y)$ still has exactly two outarcs, one whose label begins with 0 and one whose label begins with the pathlength of the vertex, since otherwise the vertex would have been given a good contracted suffix link already. We claim that all vertices of $BAD(y)$ lie in a single path in the tree at the start of Stage i and that the path corresponding to $BAD(y)$ goes along the arcs whose labels begin with a parameter pointer 0.

For suppose that vertices of $BAD(y)$ include w_b and w_c, where neither is an ancestor of the other. Let z be their lowest common ancestor. Since any pathstring can have at most one parameter pointer to the first symbol, by definition of *prev*, z has two arcs, whose labels begin with symbols we will call b and c, respectively, where b and c are not parameter pointers to the first symbols in the pathstrings and $b \neq c$. Without loss of generality, suppose the arc whose label begins with c was created second. By the Restricted Distinct Right Context Property, in the stage after the second arc was created, z received a good contracted suffix link to a vertex u, with arcs whose labels begin with b and c, respectively. Since all internal vertices descended from the arc of z whose label begins with c get their contracted suffix links created after u is created (by construction), w_c has a bad contracted suffix link pointing to u or to a descendant of u through the arc whose label begins with c. By definition of $BAD(y)$, the contracted suffix link of w_c points strictly above y at the start of Stage i but to y or below y after y is created, and consequently y is a descendant of u through the arc whose label begins with c. But then the $(|u|+1)$st symbol of the pathstring of y is c, whereas by definition of w_b, it must be b, contradicting the membership of w_b in $BAD(y)$. Therefore, one of w_b and w_c must be an ancestor of the other.

Moreover, at each vertex v (other than the one farthest from the root) in $BAD(y)$, the path follows the outarc whose label begins with 0. For the path can't follow the arc whose label begins with a parameter pointer to the first symbol in the pathstring, because each vertex of $BAD(y)$ has an outarc whose label begins with a parameter pointer to the first symbol in the pathstring, and a pathstring can have at most one such parameter pointer by definition of *prev*.

Since the number of 0's in a pathstring is at most $|\Pi|$, $BAD(y)$ contains at most $|\Pi|$ vertices. Therefore, vertices created before y account for at most $|\Pi|$ symbols prescanned because of y. The only other prescanning steps involving y are those in which prescanning begins at y for vertices whose bad contracted suffix links point directly at y; there are at most n such steps.

There are at most n prescanning steps that can be allocated to the first symbol prescanned in each stage. Over all vertices y, there are at most $|\Pi| n$ additional prescanning steps. Since $|\Sigma|$ and $|\Pi|$ are $O(1)$, each step takes $O(1)$ time, and the total time for prescanning is $O(n)$.

Finally, we analyze the time required for rescanning.

An argument similar to the prescanning argument shows that the number of symbols rescanned in resetting an out-of-date $CSL(oldhd)$ is $O(|\Pi| n) = O(n)$ over all stages.

Next, we consider rescanning for stages in which $CSL(oldhd)$ is initially undefined. We will show that in successive stages, rescanning can back up in the input and rescan sections of input already rescanned, but it can't back up past symbols already rescanned whose transformed value is not 0. More precisely, suppose that the *k*th symbol of $prev(S)$ is rescanned in Stage i after having been previously rescanned, $k > i$ (the *k*th symbol is not the first symbol in the label of an outarc from the root), and the transformed value in this rescanning is not 0. We will show that this symbol must be the first symbol rescanned in Stage i.

The transformed value is the first symbol in the label of an outarc of the locus of $prefix(S, i, k-1)$ and is either a symbol in Σ or an integer between 1 and $k-i$. Also, $k \le s$, where *oldhd* is the locus of $prefix(S, i-1, s)$, or the *k*th symbol of $prev(S)$ would not be rescanned. Let j be the number of the last stage in which this symbol was rescanned or scanned. In Stage j, the transformed value of the *k*th symbol was the same, because of the definition of f and the fact that this symbol was deeper in the tree in Stage j than in Stage i. In Stage j, the *k*th symbol of $prev(S)$ corresponds to the first symbol on an arc of an internal vertex. Therefore, for some q, $prefix(S, j, k-1) = prefix(S, q, q+k-j-1)$ but $prefix(S, j, k) \ne prefix(S, q, q+k-j)$, and the last symbol of $prefix(S, j, k)$ is either a symbol in Σ or an integer between 1 and $k-i < k-j$. Therefore, by the Common Prefix Property and Restricted Distinct Right Context Property, the locus of $prefix(S, j+1, k-1) = prefix(S, q+1, q+k-j-1)$ exists at the end of Stage $j+1$. Moreover, this argument can be applied inductively to show that the locus of $prefix(S, i-1, k-1)$ exists at the end of Stage $i-1$. Consequently, $k-1 \le r < s$, where $parent(oldhd)$ is the locus of $prefix(S, i-1, r)$. But the contracted locus of $prefix(S, i, r)$ cannot be any closer to the root than the locus of $prefix(S, i, k-1)$. We conclude that the *k*th symbol of $prev(S)$ must be the first symbol rescanned in Stage i.

We have shown that other than the first symbol rescanned in each stage, only symbols transformed into 0's can have been rescanned previously. Since the number of 0's in any pathstring is at most $|\Pi|$, each stage rescans at most $|\Pi|+1$ symbols already rescanned previously. Over all stages, there are at most n rescannings of symbols for the first time. Since $|\Pi|$ and $|\Sigma|$ are $O(1)$, $O(|\Pi| n) = O(n)$ symbols are rescanned and the time for rescanning each symbol is $O(1)$. Thus, the total time spent on rescanning over all stages is $O(n)$.

From above, scanning, prescanning, and rescanning use $O(n)$ time over all stages. Since the time used by *lazy* outside of these procedures is also $O(n)$, the result follows. □

4. An algorithm for finding all maximal p-matches over a threshold length.

This section defines maximal p-matches and gives algorithms for finding all maximal p-matches over threshold length in a p-string.

DEFINITION. For a p-string *S*, define S_i to be its *i*th symbol, for $1 \le i \le |S|$, and $S_{i,j} = S_i \cdots S_j$, for $1 \le i \le j \le |S|$.

Let S be a p-string of length n. If $S_{i,i+k}$ and $S_{j,j+k}$ are a p-match, where $1 \le k < n$ and $1 \le i, j \le n-k$. we denote it by $(i, j, k+1)$ or (equivalently) $(j, i, k+1)$, *i.e.* by the two starting positions and the length of the p-match. Define S_{n+1} to be an endmarker $ that does not occur in *S*, and S_0 to be a beginning marker that also does not occur in *S*, with $S_0 \ne \$$.

DEFINITION. Suppose $S_{i,i+k}$ and $S_{j,j+k}$ are a p-match, where $1 \le i \le i+k \le n$ and $1 \le j \le j+k \le n$. We say this p-match is *left-extensible* if $S_{i-1,i+k}$ and $S_{j-1,j+k}$ are a p-match, and *right-extensible* if $S_{i,i+k+1}$ and $S_{j,j+k+1}$ are a p-match. If it is neither left-extensible nor right-extensible and is not the trivial p-match $(1,1,n)$, we say it is a *maximal* p-match.

LEMMA 4. If (i, j, k) is a maximal p-match, then this p-match cannot be extended in any amount in either or both directions, *i.e.* there are no $r, s \ge 0$ with at least one of r, s nonzero such that

$(i-r, j-r, r+k+s)$ is a p-match.

Proof. If two p-strings p-match, and they are truncated on the right (left) by the same amount, the resulting p-strings will still p-match. Therefore, if there is a p-match that is more than one symbol longer in either direction, there will also be a p-match that is exactly one symbol longer, contradicting the non-extensibiliy of the p-match. □

Maximal p-matches for p-strings include as a subcase maximal matches for strings. It was shown in [Bak2] that the maximal match relation is not an equivalence relation, because it is not transitive. For example, consider the string *adbdadb*. The triple $(2,4,1)$ represents the maximal match between the first two *d*'s. Similarly, the triple $(4,6,1)$ represents the maximal match between the last two *d*'s. However, the first and last *d*'s are not a maximal match; they are part of the longer maximal match $(1,5,3)$. Thus, maximal p-matches are also not an equivalence relation, and an algorithm to report all maximal p-matches over a threshold length must report pairs of p-substrings rather than equivalence classes of p-strings.

In [Bak2], a suffix-tree based algorithm was given to find all maximal matching substrings over a threshold length in a string. We would like to generalize that algorithm to p-suffix trees and p-strings. The algorithm for strings was based on two facts: each pathstring in a suffix tree represents one or more matches that are not right-extensible, and whether the p-matches are left-extensible can be determined by checking the symbol to the left of the matching substrings.

For p-suffix trees, it is also true that each pathstring represents one or more p-matches that are not right-extensible. However, checking whether the p-matches are left-extensible is more complicated.

Suppose $S_{i,i+k}$ and $S_{j,j+k}$ are a p-match. If S_{i-1} and S_{j-1} are both in Π, the first symbols of $prev(S_{i-1,i+k})$ and $prev(S_{j-1,j+k})$ will both be 0. Nevertheless, it may happen that $S_{i-1,i+k}$ and $S_{j-1,j+k}$ are not a p-match. The reason is that for some $r \le k$, the rth symbols of $prev(S_{i,i+k})$ and $prev(S_{j,j+k})$ may both be 0, but S_{i+r-1} may be the next occurrence of S_{i-1}, while S_{j+r-1} may be the first occurrence of some parameter other than S_{j-1}, so these symbols cannot correspond under renaming in $S_{i-1,i+k}$ and $S_{j,j+k}$. For example, consider $S_{i-1,i+k} = xabcx$ and $S_{j-1,j+k} = yabcz$, where x, y, and z are parameters. Then $prev(abcx) = abc0 = prev(abcz)$, but $prev(xabcx) = 0abc3$ while $prev(yabcz) = 0abc0$. This is the failure of the Distinct Right Context Property in a different guise.

It would be convenient to have a way to determine left-extensibility just by checking the positions to the left. Our solution is to construct another string, $A = (prev(S^r))^r$, where the superscript r represents reversal; in A, the parameters are turned into forward references rather than back references as before. Note that A can be constructed in time and space linear in $|S|$by scanning $prev(S)$, replacing each parameter by a 0, and then replacing each such 0 pointed to by a parameter pointer by the value of the parameter pointer. Let $A_0 = S_0$ (the unique beginning-marker). The following proposition shows that by applying a transform function to each left context, we need check only the left context positions for equality.

LEMMA 5. Let $i,j,k > 0$ and $i \neq j$. A p-match (i,j,k) is left-extensible iff $f(A_{i-1}, k+1) = f(A_{j-1}, k+1)$.

Proof. The proof is trivial if at least one of the symbols A_{i-1} and A_{j-1} is in Σ. So consider the case where both are parameters.

If (i,j,k) is left-extensible, $(i-1, j-1, k+1)$ is a p-match and S_{i-1} occurs in $S_{i,i+k-1}$ iff S_{j-1} occurs in the corresponding positions of $S_{j,j+k-1}$. If these symbols do occur, then

$f(A_{i-1}, k+1) = f(A_{j-1}, k+1) > 0$, while if they don't occur, then $f(A_{i-1}, k+1) = f(A_{j-1}, k+1) = 0$.

Now, suppose (i, j, k) is a p-match and $f(A_{i-1}, k+1) = f(A_{j-1}, k+1) = r$. If $r = 0$, then the initial symbols are both parameter symbols that don't occur in the rest of the p-strings, and the p-match is left-extensible. If $r > 0$, then the initial symbols are the same as the parameters r symbols to the right, and the 1-1 correspondence between parameter symbols for (i, j, k) also implies that $(i-1, j-1, k+1)$ is a p-match and (i, j, k) is left-extensible. □

The algorithm for finding all maximal p-matches over a threshold length will be based on p-suffix trees augmented by lists of the following forms.

DEFINITION. A *plist* is a list of integers, and a *clist* is a list of plists.

An integer *i* in a plist will represent the *i*th p-suffix. Each plist will be constructed so that all of its member elements correspond to p-suffixes with the same transformed left context. The intent is to construct a clist for each vertex *v* to represent the descendant leaves of *v*, sorted by transformed left context in *A*.

For strings, the algorithm recurses over a suffix tree as follows [Bak2]. For each leaf *L*, it creates a clist containing a single plist with one index corresponding to the suffix represented by the leaf. At each internal vertex, after constructing the clists for the children, the algorithm sorts the information represented by the clists of the children into a new clist. The sorting is accomplished by processing the children from left to right and merging their information into a new clist. At the same time, any longest p-matches that are found are reported.

The following example illustrates the operation of the string algorithm at a vertex *v*. Suppose we represent a plist with positions p_1, p_2, ... p_k and left context σ by $\sigma : p_1, p_2, \cdots, p_k$. If the first child of *v* has a clist containing plists $a:35,72,46$, $b:43,7$, and $c:25$, the second child has a clist consisting of plists $a:66,2$ and $c:56$, and the third child has a clist consisting of plists $b:64,31$ and $c:82,13,59$, where $a, b, c \in \Sigma$, then the clist constructed for the parent, *v*, will be $a:35,72,46,66,2$, $b:43,7,64,31$, and $c:25,56,82,13,59$.

For strings, this algorithmic structure is adequate [Bak2]. For p-strings, an extra step must be performed, because the transformed left context of a plist may change from nonzero to zero when it is transformed with respect to a smaller number. For example, if the transformed left context is 35 when evaluated with respect to the pathlength 38 of a child, and the parent has pathlength 32, then the transformed left context will be 0 when transformed with respect to the pathlength of the parent.

Thus, a clist that is sorted by left context for a child may contain more than one plist with left context 0 when the left contexts are evaluated with respect to the parent. Thus, after a clist is constructed for a child *c* of a vertex *v*, the clist is scanned for any plists corresponding to left contexts of 0 when transformed with respect to the pathlength of *v*, and such plists are merged into a single plist.

Figure 4 gives the three procedures needed to perform this algorithm for a threshold *t*: *pdup*, *concatz*, and *pcombine*. The main procedure is *pdup*, which recurses over the p-suffix tree. For each internal vertex, *pdup* calls *pcombine*, which sorts and combines the plists produced for the children, and applies *concatz* to handle the special case where a nonzero value of a transformed left context changes to a zero value. The concatenations of plists, which are not described explicitly, are done via pointers rather than by copying; by maintaining pointers to the beginning and end of each plist, each concatenation is done in $O(1)$ time. For conciseness, the following definitions are assumed in the pseudocode.

```
/*f is defined in Section 2 */
clist pcombine(clist cl1, clist cl2, int len, int t) {
   plist pl1, pl2;
   clist outputlist;
   if (len < t) return (NULL);
   for each plist pl1 of cl1 and each plist pl2 of cl2
      if (f(LCA(pl1),len+1) ≠ f(LCA(pl2),len+1))
         for every p1 in pl1 and every p2 in pl2
            report a maximal match (p1,p2,len);
   /*construct outputlist */
   for each plist pl1 of cl1 and each plist pl2 of cl2
      if (f(LCA(pl1),len+1) == f(LCA(pl2),len+1)) {
         include in outputlist the plist obtained by concatenating
              pl2 to pl1;
         mark pl1 and pl2 as used;
      }
   for each plist pl1 of cl1 that is not marked used
      include pl1 in outputlist;
   for each plist pl2 of cl2 that is not marked used
      include pl2 in outputlist;
   remove marks from all plists in outputlist;
   return outputlist;
}
clist concatz(clist cl, int len) {
   scan cl and concatenate all plists pl for which f(LCA(pl),len+1)=0
      into one plist while leaving others unchanged
}

clist pdup(vertex v, int t) {
   clist cl;
   if (v is a leaf) return  a clist containing one plist
      consisting of start(v);
   cl = NULL;
   for each child s of v
      cl = pcombine(cl,concatz(pdup(s,t),plen(v)),plen(v),t);
   return cl;
}
```

Fig. 4. The algorithm *pdup* for reporting all maximal p-matches of length at least t.

DEFINITION. For a leaf L, the starting position of the p-suffix corresponding to L is $start(L) = firstpos(L) + arclen(L) - plen(L)$. For an integer i, define $LCA(i)$ to be A_{i-1}. For a plist pl, define $LCA(pl)$ to be A_{i-1}, where the first element of pl is i.

Thus, for a vertex v, $f(LCA(pl), plen(v)+1)$ is the transformed left context used for pl while processing vertex v.

THEOREM 3. Given a p-string S of length n over fixed alphabets Σ and Π and a positive integer t, all p-matches of length at least t in S can be found in time $O(n+r)$ and space $O(n)$, where r is the number of matches reported.

Proof. Construct the p-suffix tree T for $S\$$, where $\$$ is an endmarker that doesn't occur in S, using linear time by Theorem 2. Then, call $pdup(root(T), t)$, where *pdup* is given in Figure 4.

We claim that the algorithm correctly merges the clists of the children at each vertex v, transforming the left context values as required by the pathlength of v, and reporting exactly the longest p-matches over threshold length. A formal proof would be by induction on the depth of v; details are left to the interested reader.

Now, we need to analyze the time and space bounds for the algorithm. The time spent on linear searches of clists and plists in *pcombine* and *concatz* is dominated by the number of steps spent on cross-products. Two cross-products of clists are performed in *pcombine*. We partition the work into *same-work*, namely the steps required when the transformed left contexts are the same, and *different-work*, namely the steps required when the transformed left contexts are different. We bound the amount of same-work and different-work as follows.

By examining the second cross-product of clists, where plists are merged, it is easy to see that the same-work done at vertex v is linear in the number of new links created in combining plists, and consequently, the amount of same-work performed over all vertices is linear in the sum of the lengths of the plists for the root, and is consequently linear in n.

At each step in the first cross-product with distinct transformed left contexts, a maximal p-match is reported for each pair of positions in the cross-product of the plists. Therefore, the amount of time spent over all vertices on different-work in the two cross-products of clists is linear in m. □

5. Implementation and Experiments.

This section describes an application of the p-suffix tree to finding parameterized duplication in C code. The algorithms *lazy* and *pdup* have been implemented in a C program running under UNIX. The sets Σ and Π are disjoint sets of integers, varying with the input. For each line of C code, the lexical analyzer turns tokens such as identifiers and constants (but not keywords or operators) into integer parameter symbols, generates a new version of the line in which each such token is replaced by a "P", and obtains an integer in Σ corresponding to the transformed line. Thus, each line of input corresponds to one symbol of Σ and zero or more symbols of Π (in the order in which the corresponding tokens occur in the line). Since Σ and Π can be very large, accessing the children of each vertex is accomplished by hashing rather than by linked lists in *lazy*. How to do this was suggested by McCreight[M]. However, since *pdup* requires being able to scan successive children of each vertex, which cannot easily be done with the hashing method, the outarcs of each vertex are converted into linked lists for running *pdup*; the transformation is done in time linear in input length.

Number of Lines	Number of Symbols	$\lvert\Sigma\rvert$	$\lvert\Pi\rvert$	Number of Symbols Prescanned	Number of Symbols Rescanned	Time for *lazy* in Seconds
35233	112146	2406	6761	60730	27371	4.97
101874	320947	3992	11718	176690	74149	15.06
158579	517415	8808	24168	271069	121202	25.91

Table 1. Data for *lazy* on three subsystems

The program was applied to code from three different subsystems of a production system. The experiments were run on one processor of an SGI machine with eight 33 MHz R3000 processors, data and instruction caches of 64 Kbytes, a secondary data cache size of 256 Kbytes, and main memory size of 256 Mbytes.

Table 1 gives the data related to *lazy*. The first four columns give the data for the input subsystems: the number of lines (including comments and white space), the length of the resulting p-string, the size of Σ, and the size of Π. The last three columns give the running time for *lazy*, (not including the lexical analysis), the number of symbols prescanned, and the number of symbols rescanned.

The proof of Theorem 2 suggests that in the worst case, the number of steps executed for prescanning and rescanning might be proportional to $|\Pi|$times the length of the input. Since $|\Pi|$ is not constant in our application, and could be in principle proportional to n, the running time could be quadratic rather than linear in n. Table 1 shows that this blowup was not observed in the experiments: in each case, the number of symbols prescanned and the number of symbols rescanned were less than the length of the input. The question of why this blowup was not observed is partially answered by looking at the maximum number of prescannings and rescannings in any stage. For the largest subsystem, the maximum number of prescannings in any stage was 6, and for the other two, it was 3; for the largest subsystem, the maximum number of rescannings in any stage was 99, for the second largest, it was 33, and for the smallest, it was 16. Either the bound of $|\Pi|n$ steps is not tight, or the input from code does not generate worst case behavior. In any case, we conclude that *lazy* runs fast even with large alphabets and large input.

Table 2 gives data regarding running times of *pdup* and running times of the whole program (including lexical analysis and *lazy*) on the three subsystems, with a threshold length of 75, which is a value which would be reasonable for the application. The times given for *pdup* include the time for transforming the arc representation from hashing into linked lists and the time for running *pdup*. Note that with a threshold of 75, the number of p-matches reported is much less than input length; hence the running time of *pdup* is dominated by the part of the algorithm linear in input length. With a threshold of 10 for the largest subsystem, with p-string length of 517415, *pdup* reports 5017926 maximal p-matches and takes 467 seconds, and the whole program takes 555 seconds.

Threshold Length	Number of Lines	Number of Symbols	Number of Matches	*pdup* Time in Seconds	Total Time in Seconds
75	35233	112146	512	4.28	27.48
75	101874	320947	3837	15.14	69.64
75	158579	517415	16088	26.06	111.33

Table 2. Running times for *pdup* and for the whole program

6. Discussion

The theorems in this paper were stated in terms of fixed alphabets, but in the application, as described in the last section, the alphabets are not fixed. It is interesting to examine further the worst-case time bounds in the case of variable alphabets.

The construction of A from $prev(S)$ takes time linear in $|S|$even for variable alphabets, and in *pdup*, arcs are only accessed sequentially. Thus, for variable alphabets, once the p-suffix tree is constructed for a p-string S, the time to report all maximal p-matches over a threshold length in S is $O(n+r)$, where n is the input length and r is the number of p-matches reported.

On the other hand, as mentioned in the last section, the bounds proved for constructing a p-suffix tree do depend on alphabet size: the proof of Theorem 2 shows that the number of symbols prescanned and rescanned when running *lazy* on input of length n is bounded by $O(n|\Pi|)$. If arcs are stored in a balanced tree scheme with $O(\log(|\Sigma|+|\Pi|))$ access time, then *lazy* constructs a p-suffix tree in worst-case time $O(n(|\Pi|\log(|\Sigma|+|\Pi|))$. As discussed in the last section, these bounds may not be tight.

The author is continuing work on p-suffix trees, some of which will be described in [Bak3]. As a step in improving the worst-case bounds for variable alphabets, [Bak3] will show that a p-suffix tree for a p-string S of length n can be constructed in $O(n\log n)$ time and $O(n)$ space if Σ and Π can vary. However, that algorithm is not practical, since it relies on complicated auxiliary data structures.

An issue regarding the choice of problem statement is that reporting all maximal p-matches over a threshold length does not necessarily result in a natural analysis of certain kinds of duplication. For example, a structure of $(abc)^4$ will be reported as one p-match of length 3, one of length 6, and one of length 9 (with the matching p-strings overlapping). It could be that some other set of definitions would enable a more natural analysis of the structure of duplication.

A related issue regarding the choice of definitions is the issue of how the amount of output relates to the amount of information it conveys. Tufte has stressed in his beautiful book [Tu] on the visual display of data that it is important to maximize the data-ink ratio. The same principle should apply to how much output is reported by algorithms. In the case of duplication, if there are c copies of the same p-string, then a natural way to report them would be by listing c starting positions and the p-string; however, because the p-match relationship is not an equivalence relation, they could in general be part of $c(c-1)/2$ distinct maximal p-matches, and under the maximal p-match definition, these must be reported separately. It is not obvious what definitions would enable reporting a minimal amount of information in each case, in a way easily understandable to the user of the program.

Acknowledgements.

The author would like to thank Raffaele Giancarlo for helpful discussions relating to this work, and William Chang for providing an implementation of McCreight's algorithm which was modified to implement `lazy`.

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.

[Bak1] Brenda S. Baker, A Program for Identifying Duplicated Code, *Computing Science and Statistics 24* (1992), Interface Foundation of North America, pp. 49-57.

[Bak2] Brenda S. Baker, On finding duplication in strings and software, technical report, AT&T Bell Laboratories, February, 1993.

[Bak3] Brenda S. Baker, Parameterized pattern matching: algorithms and applications, *Proc. of 25th Annual ACM Symposium on Theory of Computing*, (May, 1993),pp. 71-80.

[CL] W. I. Chang and E. L. Lawler, Sublinear Approximate String Matching and Biological Applications, *Algorithmica 12* (1995), pp. 327-344.

[CH] Kenneth W. Church and Jonathan I. Helfman, Dotplot: A program for exploring self-similarity in millions of lines of text and code, *Journal of Computational and*

Graphical Statistics 2,2 (June, 1993), pp. 153-174.

[GG] Zvi Galil and Raffaele 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.

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

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

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

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

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

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

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

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

[Tu] Edward R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT, 1983.

[Ukk] E. Ukkonen, On-line construction of suffix trees, *Algorithmica 14* (1995), pp. 249-260.

[We] Peter Weiner, Linear pattern matching algorithms, *Proc. 14th Annual IEEE Symp. on Switching and Automata Theory* (1973), pp. 1-11.

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