Detection of Maximal Repeating Patterns And Limited Length Repeating Patterns

by

David J. Martin

Project report submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTERS OF SCIENCE

IN

COMPUTER SCIENCE

APPROVED:

Roger W. Ehrich

Roger W. Ehrich, chair

[illegible] [illegible]

Lenwood S. Heath **Clifford A. Shaffer**

Department of Computer Science

Virginia Polytechnic Institute and State University

Blacksburg, Virginia 24061

June 23, 1995

c.2

LD
5655
V851
1995
M378
c.2

Detection of Maximal Repeating Patterns And Limited Length Repeating Patterns

by

David J. Martin

Roger W. Ehrich, Chairman

Computer Science

Abstract

Given a string of characters, that string may contain patterns of characters that occur more than once. These are Repeating Patterns. A Maximal Repeating Pattern (MRP) is a repeating pattern that is not a substring of a longer repeating pattern or occurs at least once where it is not a substring of another repeating pattern. This report rigorously addresses the computation of MRPs and proposes two new categories of repeating patterns whose computational bounds are more attractive for use in human-computer interaction where the computational complexity is a significant issue.

A modified trie is used to find Maximal Repeating Patterns in a given text string. The advantages in time complexity and memory usage gained by limiting the length of MRPs and the usefulness of limiting the spatial context of repeating patterns when processing large data sets are explored.

1 Introduction

1.1 Maximal Repeating Patterns

Finding repeating text patterns is of interest in several fields. In human-computer interaction studies log files are analyzed to detect repeating patterns of user commands. These repeating patterns may be indicators of a poorly designed human-computer interface, errors in the operating system, or other problems in the interface design. In the field of data compression, repeating patterns are found and replaced with shorter codes. When these patterns are not substrings of longer repeating patterns, they are called Maximal Repeating Patterns.

In 1989 Antonio Siochi was the first to define and use MRPs in human-computer interface analysis. Siochi's work (Siochi, 1989, 1991) deals with MRPs mainly in their connection with user interface analysis. His work failed to illuminate some to the subtle aspects of MRPs and their computation. This report rigorously explores the definition of MRPs, their computation, and extends the definition to several related categories of repeating patterns that are more computationally tractable.

By considering only limited length patterns, the time complexity is significantly reduced because the required data structure (a p-tree, see Section 3.1.5) may be drastically reduced in size. By requiring that patterns repeat within some spatial context, the part of the tree that must be kept in memory at any given time may be reduced. The same number of tree nodes as the non-spatially bounded tree must still be constructed, added, and deleted, but they need not exist in memory all at once. This allows a much larger data set to be processed in primary storage, avoiding the time increases associated with using secondary storage.

Consider the set of strings Σ^* constructed from a finite alphabet Σ. Let $u, w, x, y \in \Sigma^*$ denote specific strings. If $x = uyw$, we say that y is a substring of x, and we say y is a proper substring of x if $uw \neq \lambda$, where λ is the empty string.

A repeating pattern in a string x is a substring of x that occurs more than once in x.

Given a string S, of length n, constructed from the alphabet Σ, where $|\Sigma| = l$. The longest repeating pattern is of length $n-1$. A pattern of length n may not repeat in a string of length n since it can only occur once (in the first position). In the example of $S =$ "aaaa$" the pattern "aaa" occurs twice, at positions one and two.

The shortest repeating pattern for a general string is 0 if $n \leq l$ or 1 if $n > l$. If the string contains more characters than are in the alphabet, at least one of them must repeat.

A *Maximal Repeating Pattern* (MRP) is a repeating pattern in a string, where at least one of its occurrences is not a substring of another repeating pattern. The concept of an MRP and the use of a Pat-tree to find them is due to Siochi (Siochi, 1989; Siochi and Ehrich, 1991). An occurrence of a repeating pattern is considered *independent* if it is not contained in an occurrence of a larger repeating pattern.

Let S be a string of n characters, $S = s_1 s_2 s_3 \cdots s_n s_{n+1}$, where $s_{n+1} = \$$ is a special terminating character that occurs nowhere else in S. Let $\alpha = S[i, i+q]$ denote the substring $s_i \cdots s_{i+q}$ of S where α is said to occur at position i. If α occurs at distinct positions i and j, then α is a *repeating pattern* within S. If $\alpha = S[i, i+q]$, then let $|\alpha| = q+1$ be the length of α. Let α and β be repeating patterns in S. The *position set* $\mathrm{P}(\alpha)$ of α is the set of positions at which α occurs. For example, if a string S is 15 characters long, $|\beta| = 3$, and β occurs at locations 0, 5, and 10, then P(β) = {0,5,10} and the *span set* of β is SP $= \{0, 1, 2, 5, 6, 7, 10, 11, 12\}$. The complement of the span set of β (all the positions not occupied by a character of β) is $\overline{SP}(\beta)$ = {3,4,8,9,13,14,15}. A repeating pattern α is a Maximal Repeating Pattern or *MRP* of S, if there exists no repeating pattern β of S such that $\alpha < \beta$, or if a repeating pattern β exists, then P(α) $\cap$ $\overline{SP}(\beta)$ is not the empty set. In other words, at least one α occurs outside of all occurrences of β. If α is not contained in another occurrence of a repeating pattern β, then α is maximal.

Given a string S, of length n, constructed from the alphabet Σ, where $|\Sigma| = l$. The longest MRP is also of length $n - 1$. Since the longest possible repeating pattern is of length $n-1$, and since such a pattern may not be contained in a longer repeating pattern (there exist no longer repeating patterns) it is a MRP. "aaa" is a MRP of "aaaa$".

For a string of length n, there will exist at least one MRP of length $log_l(n-1)$ or greater. For $n \leq l$, there need not be any repeating patterns. Three are l^k possible patterns of length k. A string of length n contains n patterns (one starting at each position). If $n > l^k$ then at least one of those patterns must repeat. The example of "abbbbba$" where the MRPs are "a" and "bbbb" shows that there may also be MRPS shorter than length $log_l(n-1)$.

Siochi's results suggest that the number of MRPs is a linear function of the size of the input, while both the size and frequency of the MRPs is a function inversely proportional to the size of the input.

1.1.1 Examples:

Because many of them overlap, the MRPs in the examples are boxed and underlined to make them easier to read.

Siochi's example (prefix example). For the text string "$\boxed{\underline{abc}d}y\boxed{\underline{abc}d}x\underline{abc}\$$", the MRPs are "*abc*" and "*abcd*". "*abc*" is a MRP because it occurs independently of "*abcd*".

Substring example. For the text string "$\boxed{a\underline{bc}d}x\boxed{a\underline{bc}d}y\underline{bc}\$$", the MRPs are "*bc*" and "*abcd*".

Suffix example. For the text string "$\boxed{a\underline{bcd}}x\boxed{a\underline{bcd}}y\underline{bcd}\$$", the MRPs are "*bcd*" and "*abcd*". These three examples show that a pattern may be a prefix, suffix or general substring of a MRP and still be a MRP in its own right.

Overlap example. For the text string "$\boxed{xx\underline{x}}\underline{yy}abc\underline{xyy}def\boxed{xxx}\$$", the MRPs are "*xxx*" and "*xyy*".

The position of a substring in a pattern is not significant as long as the substring occurs independently. An MRP may overlap itself, as one can see in a string such as "aaa$", where the MRP is "aa".

1.1.2 k-MRPs

A k-bounded MRP, or k-MRP is a repeating pattern that is not a substring of any other pattern that is less than length k.

A repeating pattern α of S with $|\alpha| \leq k$ is a k-MRP if there is no repeating pattern β of S with $|\beta| \leq k$ such that $\alpha < \beta$, or if a repeating pattern β exists, then $\mathrm{P}(\alpha) \cap \overline{SP}(\beta)$ is not the empty set.

For the text string "*abcdxabcdybcd$*", The 4-MRPs are "*bcd*" and "*abcd*". The 3-MRPs are "*abc*" and "*bcd*". The 2-MRPs are "*ab*", "*bc*", and "*cd*". The 1-MRPs are "*a*", "*b*", "*c*", and "*d*". Note that as long as the k is as long as or longer than the longest MRP, the results are the same as for searching for the unbounded MRPs.

1.2 Scope of Work

Fully-built tree structure. Siochi used position trees to locate MRPs. This report will demonstrate and prove the validity of a simpler tree growing algorithm, especially when there is a limit on the length of the MRPs. Position trees are often implemented with compressed branches to save memory and construction time. A noncompressed position tree, or trie, (see Figure 1.1) is a position tree where each branch represents only a single character.

By using a noncompressed tree a simpler tree building algorithm may be used. Further, by placing the position data in the nodes, it is not necessary to traverse the entire tree and time is saved in the detection algorithm. On average, a tree with noncompressed branches will require more memory. However, in the worst case, a tree built from a string with only one character that repeats for the entire length is not compressible, and both trees require identical space.

k-bounded patterns. If one can guess a limit to the length of the largest MRP or better, the largest MRP useful in a given application, it will be shown that $O(n\ log(n))$ time complexity can be achieved by limiting tree depth, where n is the string length.

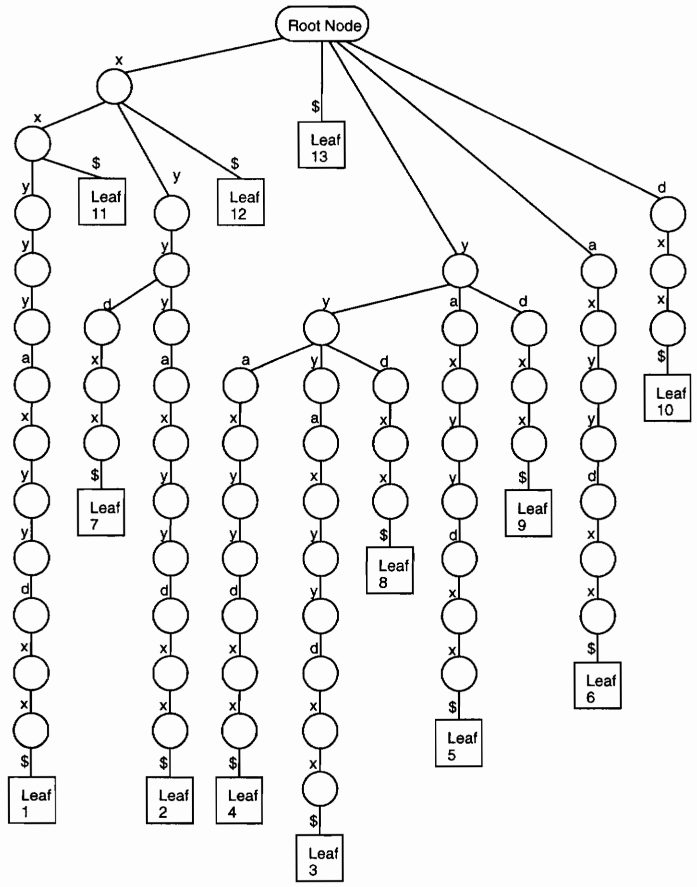


Figure 1.1. Example p-tree for the string "xxyyyaxyydxx$"

k-bounded iterative algorithm. The k-bounded algorithm has the property that if there exists an MRP, β, such that $|\beta| > k$, then all the length k substrings of β will be found. By replacing these substrings with shorter sequences and running the algorithm again all the MRPs will eventually be found. Special care must be taken, however, to ensure that the replacement sequences do not create new repeating patterns, or the algorithm will find extraneous MRPs.

It is possible to modify Weiner's Algorithm D, (Weiner, 1973) to build a bounded length position tree with compressed branches with nodes that still contain all the extra data contained in a p-tree, but this is beyond the scope of this project. Using a compressed tree would save memory (and time where virtual memory is used to hold part of the tree).

The proposed k-bounded algorithms are of complexity $O(n\,log(n))$. Most of the subsections of the algorithm are $\Theta(n)$. This is a significant improvement over previous algorithms without significant degradation of results (if an appropriate value for k is chosen). By restricting the spatial context, it is possible to limit the number of branches in the tree at any given time, limiting the size of the entire data structure, and allowing large data sets to be processed.

1.3 Previous Work

Siochi uses a standard Pat-tree, or position tree (Weiner's algorithm D, Weiner, 1973) builder. Algorithm D restructures the tree during the building phase to maintain a compressed tree. A full traversal of the tree is required to detect all the MRPs. Although in my algorithm the trees are uncompressed, the tree traversed is equivalent to Siochi's.

A trie is a recursive tree structure that contains strings. A string is said to contained in the trie if the path from the root to a leaf is labeled with that string. (Frakes, 1992) A Pat-tree (Gonnet, 1983) or position tree is a trie built from all the suffixes in the input string, or a suffix tree. Branches in the tree are labeled with characters. The actual string data is not stored in the nodes, but in the *arrangement* of the nodes in the tree. Each of the leaves has the starting position, or index, of the first character of the string that it represents stored in it. The root node will have one branch for every different character that exists in the input string.

A given interior node, n, represents the pattern found by concatenating the labels of the branches on the path from the root node to n. If n or any of its descendants branch, then that pattern repeats. The indices in the leaf nodes descended from n indicate the starting positions of these patterns in the input string.

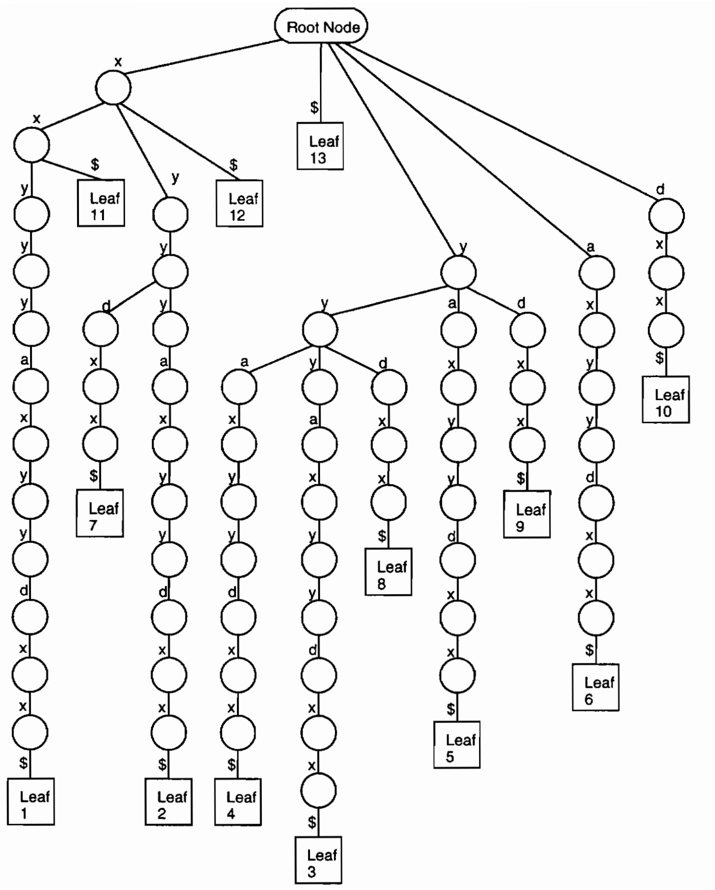


Figure 1.2. Example trie for the string “xxyyyaxyydxx$”

2 Problem Definition

2.1 Standard Problem

Given a known alphabet and an input string, find and list all the MRPs and their positions in the string.

The distance between two patterns in some string S, is the absolute difference between the positions of the first characters of the two patterns. Find and list all the MRPs in an input string, considering only patterns that are within some distance l of each other.

Present a more lucid definition of MRPs than has previously existed.

2.2 k-bounded Problem

Given a known alphabet and an input string, find and list all the k-MRPs and their positions in the string. Present and algorithm to do this in less than O(n^2) time and memory.

Patterns of length greater than k are ignored, although the possibility of their existence may be deduced from the existence of length k patterns. If no length k patterns are detected, there are no MRPs of length greater than k.

Develop an algorithm that finds all the k-MRPs and their positions in the string, considering only those that are within some distance l of each other.

3 Algorithms and Proof

3.1 Definitions

3.1.1 Suffixes

Given an arbitrary string, S, a suffix of S is a substring of S that starts at any position and continues to the end of S, for an S of any length. The number of suffixes in S is equal to the number of characters in S. Because a terminator ,\$, is added to the input, null suffixes are not allowed.

3.1.2 Occurrence

An occurrence of a pattern in the string is uniquely denoted by the index of its first character in the string. A single pattern may have several occurrences in a single string. The first character of a string has index 1.

3.1.3 Trie

Trie. Given an alphabet Σ, a trie is a tree that contains strings made up from members of Σ and have these properties:

Property 1: A trie is a rooted tree.

Property 2: The edges joining a node to its children are labeled with distinct elements of Σ.

Property 3: All leaf nodes are marked. All internal nodes are unmarked or marked. [Storer, 88] A trie is said to contain a string if the path from the root to a marked node is labeled with that string.

A trie is said to contain a string if the path from the root to a marked node is labeled with that string.

3.1.4 Pat-tree

Pat-tree. A Pat-tree of a string is a trie that contains exactly the suffixes contained in that string, except that paths for non-branching nodes are compacted together (see Figure 3.1) (Weiner, pp. 3). The path from the root node to a leaf node is labeled with the string that it represents. Pat-tree has the following properties:

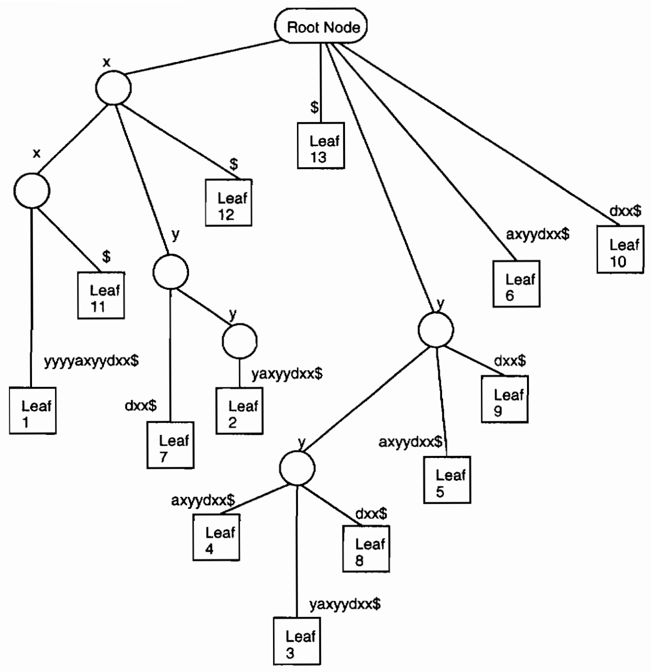


Figure 3.1. Example Pat-tree for the string “xxyyyaxyydxx$”

Property 1: A Pat-tree is a trie.

Property 2: Each leaf is labeled with the index of the string that ends at that leaf.

Property 3: There is one leaf for each index in the original string.

Property 4: A Pat-tree contains exactly the suffixes in the original input.

Note that the path from the root node to a node is labeled with is a pattern. If that node is a branching node, then that pattern is known to repeat. Since a branching node has more than one leaf descended from it, the pattern up to that point repeats.

3.1.5 p-tree

p-tree. A p-tree is defined as the particular variation of a standard Pat-tree used in this algorithm. A p-tree is the same as a Pat-tree, except that there is no node compaction (see Figure 1.2). This requires more memory, but simplifies construction and later use of the tree. p-tree nodes contain a list of pointers to

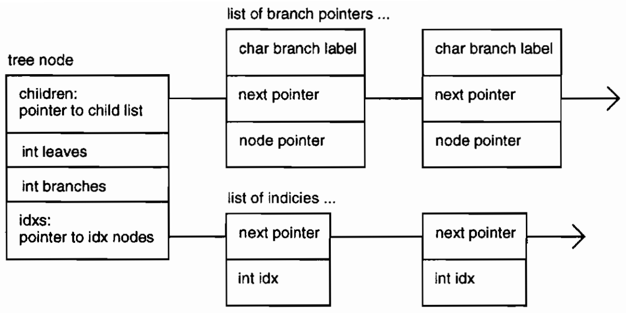


Figure 3.3. p-tree node

the node's children, the number of branches that node has, the number of leaves that are descended from that node, and a list of starting positions of occurrences that end with that node or a leaf descended from that node (see Figure 3.3). The information about the number of branches and the number of leaves descended from a node is used to limit the traversal in the detection step, later. A p-tree has the following properties:

Property 1: A p-tree is a trie.

Property 2: Each leaf is labeled with the index of the string that ends at that leaf.

Property 3: There is one leaf for each index in the original string. The leaf nodes have a one to one mapping to the characters of the input string.

Property 4: The path from the root node to a leaf node is labeled with the suffix that starts at the character to which the leaf maps.

3.1.6 Definition of a k bounded p-tree

k-bounded p-tree. For the k bounded algorithm, bounded p-trees are used. The depth of the bounded tree is limited to $k + 2$ and the leaf nodes contain lists of indices, instead of single values. These values have the 1 to 1 mapping to the input that the leaf nodes have in an unbounded p-tree. A leaf node contains index values for the leaves that would descend from it in the unbounded version except

that the strings passed to InsertString (see Section 3.1.2) are limited to length k. The bounded version is built in the same manner as the unbounded version. (See Example 3.11) A k bounded p-tree has the following properties:

Property 1. A k bounded p-tree is a trie.

Property 2. Each leaf is labeled with the indices of the string(s) that end at that leaf.

Property 3. There is one index entry for each character in the original string. The index entries have a one to one mapping to the characters of the input string.

Property 4. The path from the root node to a leaf node is labeled with the suffix that starts at the character(s) to which the entries in the index map.

3.2 Tree Builder

3.2.1 Tree Building Algorithm

Build the tree by inserting each of the suffixes into the tree. BuildTree takes a string S, terminated by \$, as an argument and returns the root node of a p-tree constructed from all the suffixes in S.

Algorithm: BuildTree(S)

```
(1) Let $R$ be a new, empty p-tree
(2) For every suffix $s_i s_{i+1} s_{i+2} \cdots s_n s_{n+1}$, do
(3)         InsertString($s_i s_{i+1} s_{i+2} \cdots s_n s_{n+1}, R, i$)
(4) Return $R$
```

Example 3.1. For the input string "xxyyyaxyydxx\$", BuildTree will send the following suffixes to InsertString: "xxyyyaxyydxx\$", "xyyyaxyydxx\$", "yyyaxyydxx\$", "yyaxyydxx\$", "yaxyydxx\$", "axyydxx\$", "xyydxx\$", "yydxx\$", "ydxx\$", "dxx\$", "xx\$", "x\$", and "\$".

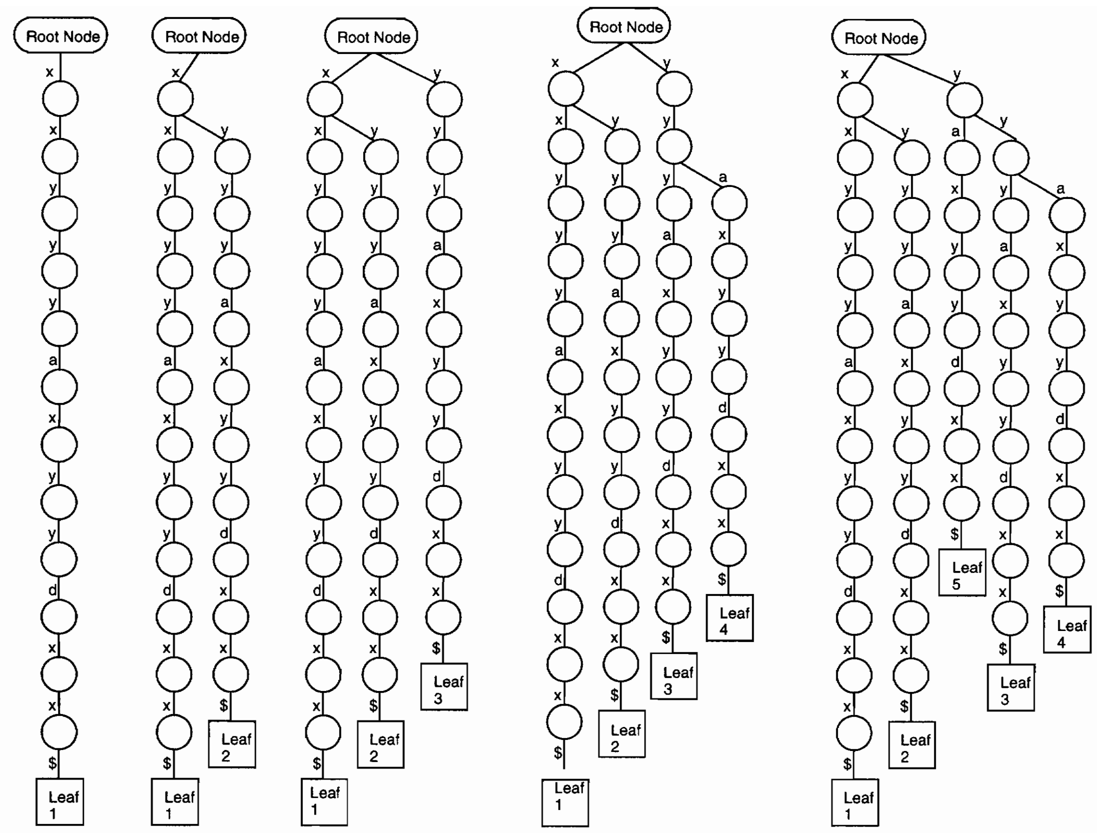
AddPtr takes the current node and a character as an argument. If the current node has a child labeled with that character it returns a pointer to that child. Otherwise it creates a new node and a pointer, labeled with that character, to it from the current node.

InsertString takes a string, x, a node and an index number, index, as arguments. The string is added to the tree. Insert string requires the terminating character \$ to be unique in the string, i.e., $\$ \notin \Sigma$.

Algorithm: InsertString($x_1 x_2 \cdots x_n, node, index$)

```
(1) Is there a child of $node$ connected by an edge labeled with $x_1$?
(2) If so:
(3)         Delete $node$'s list of indices.
(4) If not, create one by calling AddPtr($node, x_1$).
(5)         Increment $node$'s branch counter.
(6)         Add $index$ to the child's list of indices.
(7) Increment $node$'s leaf counter.
(8) Call InsertString($x_2 x_3 \cdots x_n, child, index$).
(9) Return.
```

Example 3.2. Examples 3.2.1 through 3.2.11 show the intermediate steps of building the p-tree, after each suffix is added.



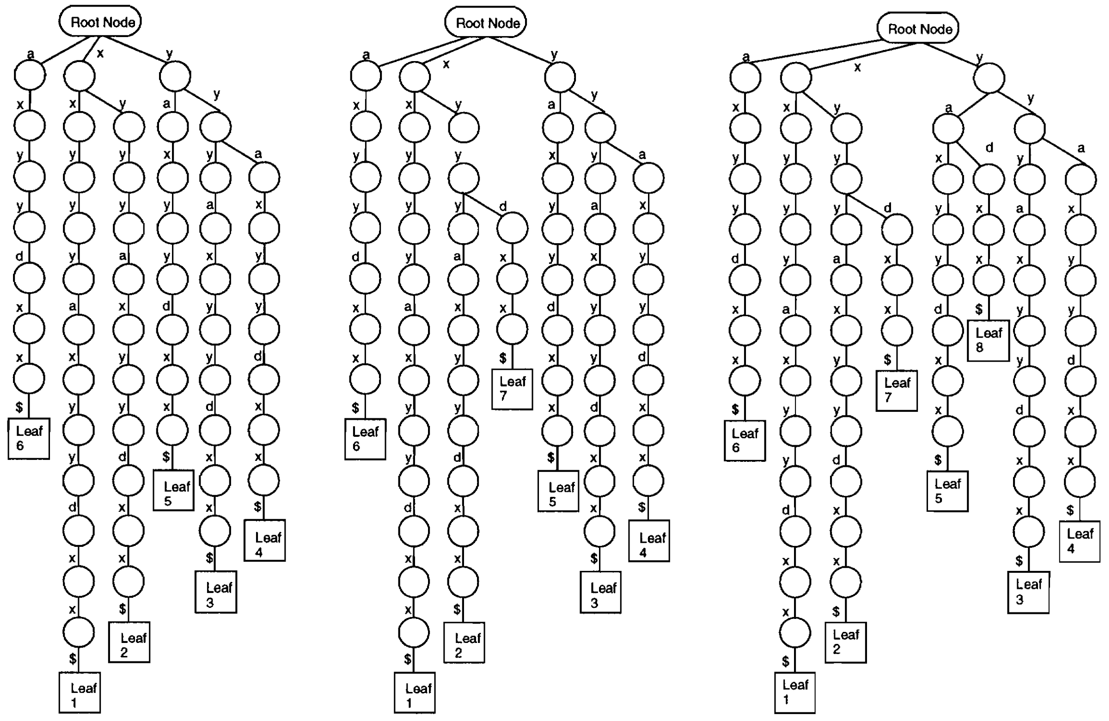
Examples 3.2.1, 3.2.2, 3.2.3, 3.2.4, and 3.2.5

3.2.2 Proof of Tree Building

The proof of correctness of the tree building algorithm is by induction on the length of the string, n. A p-tree is correct if it has the four properties listed in 3.1.5. Given a string, β, it is shown why the tree for the string $a\beta$ is correct if the string for β is correct. This may seem backwards, but the tree for $a\beta$ is the tree for β with the string $a\beta$ inserted into it.

Lemma 3.1: The algorithm InsertString is correct for $n \geq 1$.

Proof by Induction on the length of the string x.

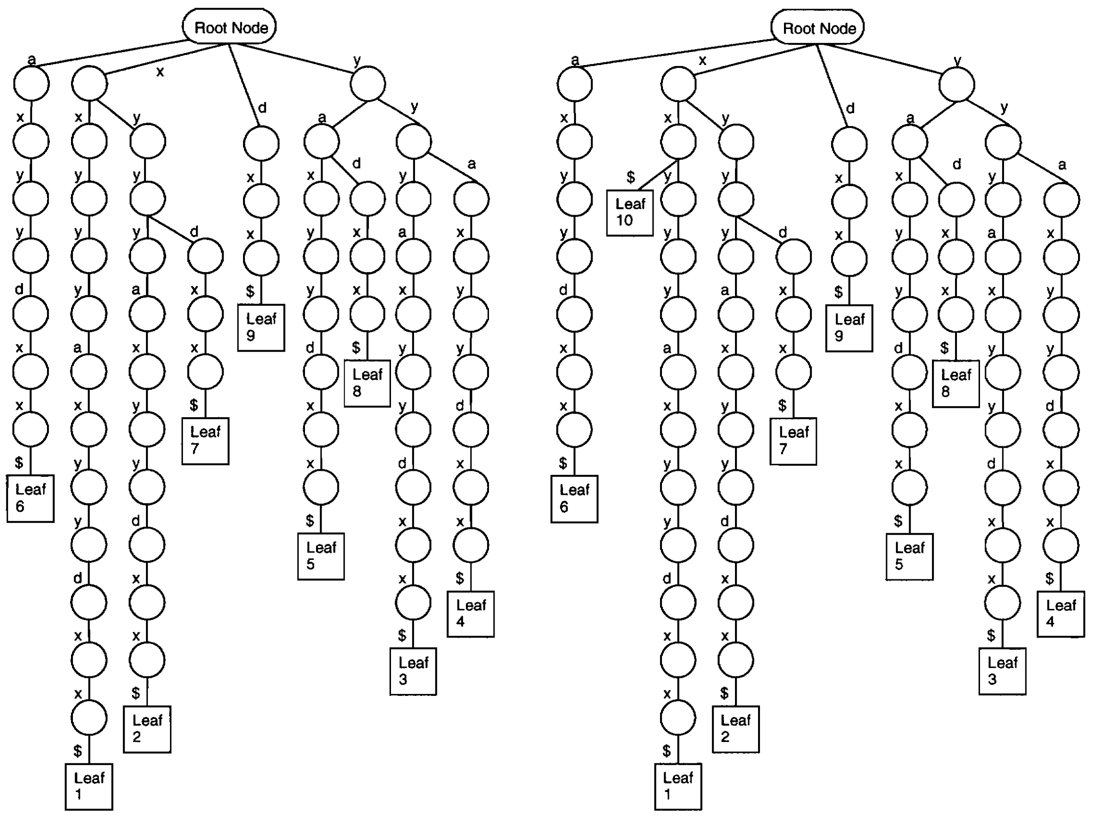


Examples 3.2.6, 3.2.7, and 3.2.8

Part I: (base cases)

n **= 0:** InsertString returns with no changes to the tree.

n **= 1:** x = "$". Because all the suffixes are unique, there will not be a branch labeled with '$', therefor a new node will be created. Since a new pointer is added, the branch counter is incremented. Since a new leaf will be added the leaf counter is incremented. The index of the new leaf is added to the idxs list of the new node. InsertString is called with x = "". All the properties are met: The tree is still a trie (Property 1). One new leaf with a properly labeled path is created (Properties 3 and 4). The new leaf contains its index (Property 2). The new leaf has zeros for the branch and leaf counters (zeroed in the node creation process).

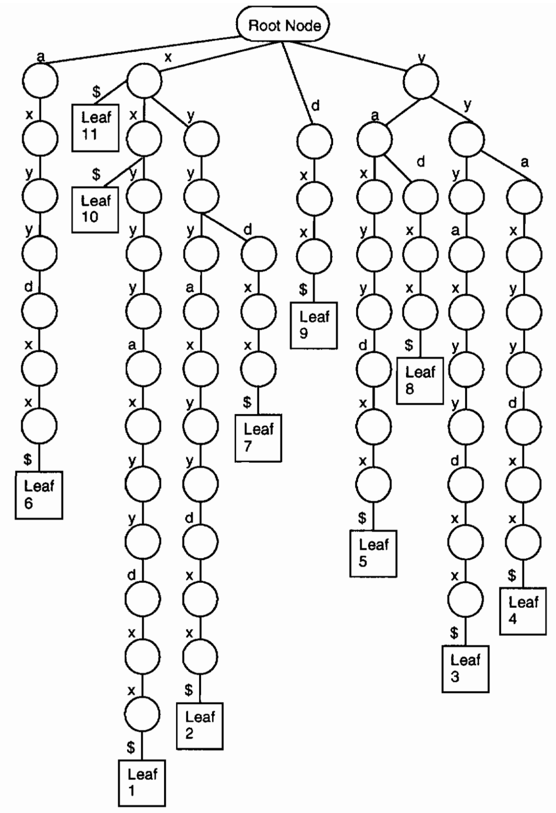


Examples 3.2.9 and 3.2.10

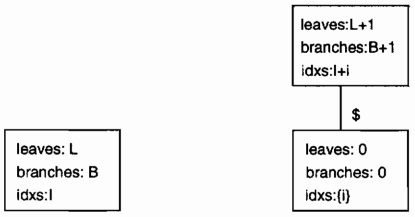
The old node has incremented leaf and branch pointers. Example 3.3.1 shows the original old node with values of L, B and I for the number of leaves, number of branches, and the index list. Example 3.3.2 shows both nodes, after the values have been changed. Note that the old node may or may not have any descendants before InsertString is called with x = "\$".

(Basis) $n = 2$**:** s = "a\$" where 'a' is a character. Either the node has an edge labeled 'a' or not.

Case 1: The node does not have a child connected by an edge labeled 'a'. A new node will be created. Since a new pointer is added, the branch counter is incremented. Since a new leaf will be added the leaf counter is incremented. The

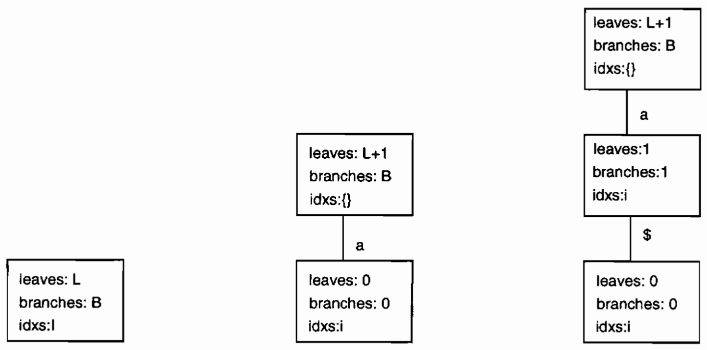


Example 3.2.11



Examples 3.3.1 and 3.3.2

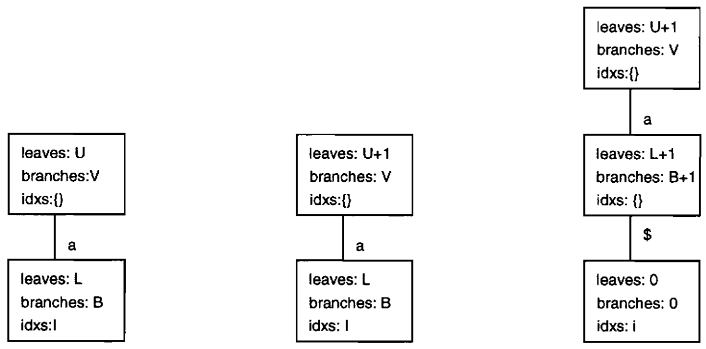
index of the new leaf is added to the idxs list of the new node. InsertString is called with x = "\$" and *node* = the new child of *node*. This is known to properly add a leaf node. All the properties are met: The tree is still a p-tree (Property 1). One new leaf with a path of "a\$" is created (Properties 3 and 4). The new



Examples 3.4.1, 3.4.2 and 3.4.3

leaf contains its index (Property 2). The new leaf has zeros for the branch and leaf counters (zeroed in the node creation process). The old node has incremented leaf and branch pointers. The new node has leaf and node counters of 1 and index list containing index. Example 3.4.1 shows the original node. Example 3.4.2 shows the tree after the 'a' has been added but before InsertString is called with x = "$". Example 3.4.3 shows the final state of the tree after InsertString returns.

Case 2: The node does have a child connected by an edge labeled 'a'. Since a new pointer is not added, the branch counter is not incremented. Since a new leaf will be added the leaf counter is incremented. The index of the new leaf is added to the idxs list of the new node. InsertString is called with x = "$" and *node* = the child of *node*. This is known to properly add a leaf node. All the properties are met: The tree is still a p-tree (Property 1). One new leaf with a path of "a$" is created (Properties 3 and 4). The new leaf contains its index (Property 2). The new leaf has zeros for the branch and leaf counters (zeroed in the node creation process). The old node has incremented leaf and branch pointers. The new node has leaf and node counters of 1 and index list containing index. Example 3.5.1 shows the original nodes. Example 3.5.2 shows the tree after the 'a' branch has been traversed



Examples 3.5.6, 3.5.7 and 3.5.8

but before InsertString is called with x = "\$". Example 3.5.3 shows the final state of the tree after InsertString returns.

Part II: (induction)

Assume: InsertString(β\$,node,index) is correct.

Prove: InsertString($a\beta$\$,node,index) is correct.

Case 1: *node* does not have a child connected by an edge labeled 'a'. Since a new pointer is added, the branch counter is incremented. Since a new leaf will be added the leaf counter is incremented. The index of the new leaf is added to the idxs list of the new node. InsertString is called with x = "β\$" and *node* = the new node. This is assumed to work, as it maintains the four properties in 3.1.5.

Case 2: *node* does have a child connected by an edge labeled 'a'. Since a new pointer is not added, the branch counter is not incremented. Since a new leaf will be added the leaf counter is incremented. The index of the new leaf is added to the idxs list of the new node. InsertString is called with x = "β\$" and node = the child of *node* connected by the edge labeled 'a'. This is assumed to work, as it maintains the four properties in 3.1.5.

All nodes have correct branch and leaf counters and index lists. The path to each leaf is labeled with the suffix used to create it. Each leaf has a unique index corresponding to the index of the suffix.

By parts I and II, Lemma I is correct.

Proof of BuildTree.

Theorem 3.1: The algorithm BuildTree is correct

Proof by Induction on the length of S.

Part I: (base case)

n **= 1:** BuildTree calls InsertString("\$",root,i), creating one leaf with an index of i and a path labeled "\$"(Lemma 3.1). This correctly maintains the four properties in 3.1.5.

(Basis) n **= 2:** BuildTree calls InsertString("a\$", root, i) and InsertString("\$", root, i+1) creating two leaves with indices of i and i+1 respectively, and paths labeled "a\$" and "\$". This also correctly maintains the four properties in 3.1.5.

Part II: (induction)

Assume: BuildTree(β\$) holds, producing $|\beta|$+1 leaves numbered from 1 to $|\beta|$+1

Prove: BuildTree($a\beta$\$).

Build tree calls InsertString("$a\beta$\$", root, i), InsertString("β\$", root, i) $\cdots$ InsertString("\$", root, i). Calling InsertString("β\$", root, i) $\cdots$ InsertString("\$", root, i) is the same as calling BuildTree($\beta\$$), which is assumed to produce a tree that maintains all four properties. Calling InsertString("$a\beta\$$", root, i) produces one leaf (Property 3) with index i (Property 2), and a path to it labeled "$a\beta\$$" (Property 4). The tree created is still a p-Tree (Property 1).

By Parts I and II, Theorem 3.1 is correct for $n \geq 1$.

3.3 Detection

3.3.1 Detection Algorithm

The MRP detection algorithm makes use of a data structure called mlist. mlist is a list of occurrences, denoted by their starting and ending indices in S. In the worst case, n items will be inserted into mlist. If mlist is implemented as a flat list, the detection algorithm requires O(n^2) time complexity because the list has to be traversed for each insertion. If mlist is implemented as a balanced tree, sorted by starting index, the algorithm is O(($n\,log(n)$)). (n searches of a balanced tree of maximum depth $log(n)$.) FindMRP takes a node, the root of the p-tree to be searched, as an argument and sends all the repeating patterns to mlist using AddNode. FindMRP traverses the tree and sends the occurrences of repeating patterns represented by nodes with branches to mlist. Since any maximal repeating pattern that repeats must end at a branching node, only nodes that branch need be considered. This is shorter, in general, than traversing the entire tree because branches are only followed to their last branch. Further, if all of a node's direct descendants have branches in their trees, then that node can be ignored also, because its pattern is always a substring of a longer pattern and is not independent. Depth is a global variable that is initialized to zero before FindMRP is first called. mrplist is a global list of MRP occurrences, sorted by their starting position in the input. It empty when FindMRP is first called.

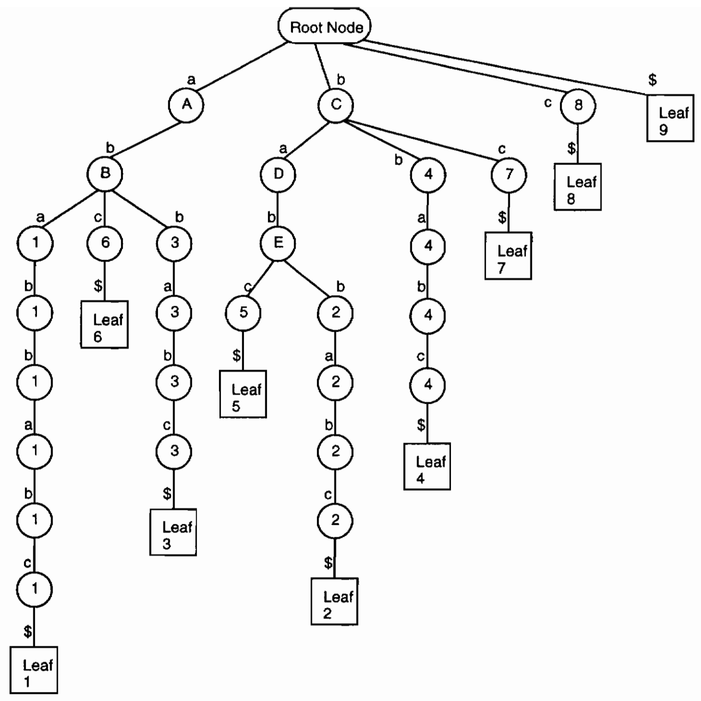
Algorithm: FindMRP(*node*)

```
(1) Does node have more than one leaf descended from it?
(2) If not, return.
(3) Increment depth.
(4) Does node have exactly one branch?
(5) If so:
(6)     Let child be the node on that branch.
(7)     Call FindMRP(child).
(8)     Decrement depth.
(9)     Return.
(10) For all child, child child of node do:
(11)     Does child have exactly one leaf descended from it?
(12)     If so:
(13)         Let nnode be a new mrpnode created with child's list of indicies and depth.
(14)         Call AddNode(mrplist, nnode).
(15)     If not:
(16)         Call FindMRP(child).
(17) Return
```

AddNode takes as an argument an occurrence (a start and stop index) of a pattern and adds it to the list of MRPs, provided it is not a substring of any previous member of the list. It also removes any occurrences from the list that are substrings of the new member.

Let L be the list of MRPs.

Let b be the occurrence to be added.



Example 3.6. p-tree for the string "ababbabc$"

Algorithm: AddNode(L,b)

```
(1) For every $p \in L$ do
(2)     If ($p$ is entirely contained in $b$) delete $p$ from $L$
(3)     If ($b$ is entirely contained in $p$)
(4)         delete $b$ from $L$
(5)         Return
(6)     Add $b$ to $L$
(6) Return
```

Example 3.6. Example 3.6 demonstrates how FindMRP detects repeating patterns and adds occurrences to mlist. FindMRP starts at the root node with the depth counter initialized to 0. First FindMRP considers the ‘a’ branch and determines that there is more than one leaf descends from it. Each node contains a count of the number of branches it has and the number of leaves that descended from it (see Figure 3.3). The depth counter is incremented, and FindMRP is called on node A. Node A does not branch, so no occurrences are sent to mlist. Node A has a single child, node B, so the depth counter is incremented again and FindMRP is called on node B. At node B, each of the three branches, ‘a’, ‘b’, and ‘c’, have only one leaf each. The pattern represented by node B is “ab”. The depth of node B is 2, the same as the length of the pattern it represents. FindMRP constructs the occurrences (1,2),(3,2), and (6,2) from the indices in node B’s children (1,3, and 6) and the depth of node B (2). FindMRP sends the occurrence (1,2) (the occurrence starting at character 1 that is 2 characters long, or the first “ab” in the input string) to mlist for the ‘a’ branch, the occurrence (3,2) for the ‘b’ branch, and the occurrence (6,2) for the ‘c’ branch. Each of these occurrences represents the pattern “ab” from the input string. mlist is now (1,2),(3,2),(6,2). The depth counter is decremented and FindMRP returns to node A. The depth counter is decremented again and FindMRP returns to the root node. Note that none of the nodes below node B are ever traversed.

Next, FindMRP considers the ‘b’ branch from the root node. This branch also has more than one leaf and must be traversed. The depth counter is incremented to 1 and FindMRP is called on node C. First the ‘a’ branch is considered. The ‘a’ branch has more than one leaf so the depth counter is incremented and FindMRP is called on node D. Node D is like node A, with only one branch, but has multiple leaves descending from it. The depth counter is incremented again and FindMRP is called on node E. Node E has two branches, each with exactly one leaf descended from them. Since the depth is 3, FindMRP will add the occurrences (5,3) and (2,3) to mlist. The occurrence (3,2) is entirely contained in the occurrence (2,3), so mlist automatically deletes (3,2) when (5,3) is added. The occurrence (6,2) is contained in (5,3) and is deleted. The depth counter is decremented and FindMRP returns to node D. When FindMRP leaves node E, mlist is {(1,2),(2,3),(5,3)}. At node D, the depth counter is decremented again and FindMRP returns to node C.

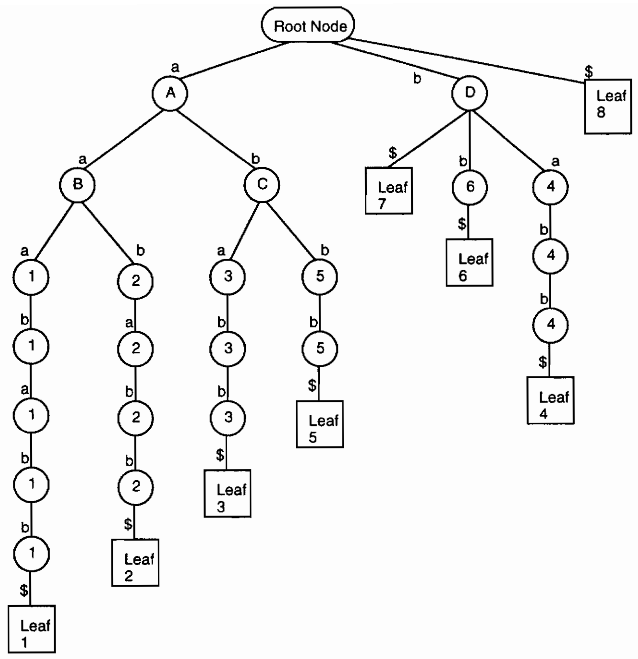
After returning to node C from the 'a' branch, FindMRP considers the 'b' branch. The 'b' branch has only one leaf descended from it so FindMRP adds the occurrence (4,1) to mlist. The occurrence (4,1) is entirely contained in the occurrence (2,3), so (4,1) is deleted immediately. Similarly, the 'c' branch has only one leaf descended from it and FindMRP adds the occurrence (7,1) to mlist. The occurrence (7,1) is entirely contained in the occurrence (5,3), so (7,1) is also deleted immediately. The depth counter is decremented and FindMRP returns to the root node. mlist remains unchanged.

FindMRP next considers the 'c' branch from the root node. The 'c' branch has only one leaf, and since the depth is currently 0, FindMRP knows that this represents a singleton character. This branch is skipped.

The last branch, the '$' branch also represents a singleton character. It is also the terminating character for the string. This branch is also skipped.

When FindMRP returns from the root node, mlist is {(1,2),(2,3),(5,3)}, representing the patterns "ab", "bab", and "bab", respectively.

Example 3.7. Example 3.7 works the same way. At node B, FindMRP adds the occurrences (1,2) and (2,2) to mlist. Note that these overlap, but neither fully contains the other, so neither is deleted by mlist. At node C, FindMRP adds the occurrences (3,2) and (5,2). At node D the occurrences (7,1), (4,1), and (6,1) are added. (4,1) and (6,1) are deleted because they are contained in (3,2) and



Example 3.7. p-tree for the string "aaababb$"

(5,2), respectively. The final mlist is {(1,2),(2,2),(3,2),(5,2),(7,1)} representing the patterns "aa", "aa", "ab","ab", and "b". Note that no occurrences are added at node A, because none of its children have only one leaf descended from it.

3.3.2 Proof of Detection Algorithm

Lemma 3.2: All members of mlist are MRPs. Proof by contradiction.

Suppose an occurrence, o, of pattern p, exists in mlist at the conclusion of the algorithm. Assume that p is not an MRP. If p is not an MRP then either p does not repeat or p is not maximal and o is not independent. If p does not repeat, then o would not exist in mlist because only occurrences of repeating patterns are added to mlist. p must repeat. If p is not maximal and o is not independent, there must exist o', occurrence of p', such that $p \leq p'$ and o is entirely contained in o'. If o is entirely contained in o' then o would have been removed from mlist. o exists in mlist. Therefore o' does not exist. p is maximal or o is independent.

Since p is maximal or o is independent and p repeats, p is an MRP.

All members of mlist are MRPs.

Lemma 3.3: All MRPs are placed in mlist. Proof by contradiction.

Suppose an occurrence, o, of pattern p, does not exist in mlist. Assume that p is an MRP. If p is an MRP, then p repeats. All occurrences of all repeating patterns are added to mlist. If o does not exist in mlist and p repeats, o must have been removed from mlist. Only occurrences that are entirely contained in other occurrences are removed from mlist. If o was removed from mlist, then there must exist o', an occurrence of p', such that o is entirely contained in o'. If o is entirely contained in o' then o is not an occurrence of an MRP. By contradiction o does not exist. Therefore, all occurrences of MRPs are placed in mlist.

Theorem 3.2: By Lemmas 3.2 and 3.3, FindMRP finds all the MRPs and only the MRPs.

3.4 k-bounded Problem

Often it is enough to limit the search to patterns of some fixed length k, or less. This allows us to limit the depth of the p-tree, reducing build time and memory used. Let $l = |\Sigma|$, the number of characters in the alphabet Σ. The maximum size of the p-tree is limited by the number of branches possible at each node, l, not the number patterns in the p-tree. Construction time for the p-tree is $O(nk)$ and memory used is $O(\sum_{i=0}^{k+1} l^i) = O(1)$, and the size of a full tree of depth $k+2$. Note that as k approaches n, the algorithm returns to $O(n^2)$. Extracting the k-MRPs from the tree takes $O(n)$ time. The complexity for simply finding all the repeating patterns of length k or less, after the tree is built, is $O(nk)$.

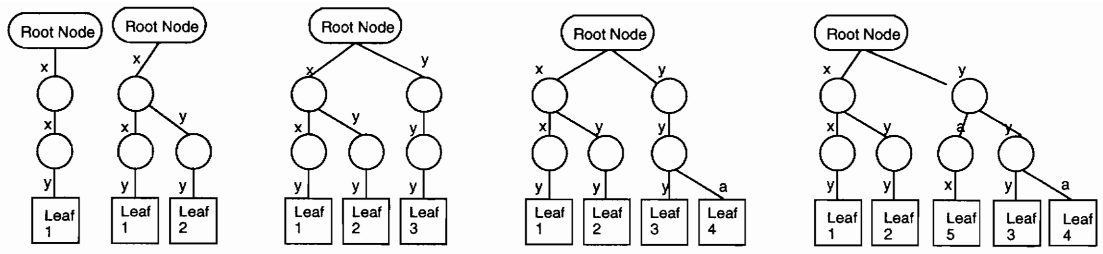
3.4.1 k-bounded Tree Builder

Build the tree by inserting each of the suffixes into the tree. BuildBoundedTree takes a \$-terminated string S and k, the maximum pattern length, as arguments and returns the root node of a p-tree constructed from all the substrings of length k or less.

```
Algorithm: BuildBoundedTree($s_{1\cdots n}, k$)
(1) Let $R$ be a new, empty p-tree
(2) For every suffix $s_i s_{i+1} s_{i+2} \cdots s_n s_{n+1}$, do
(3)         InsertString($s_i s_{i+1} s_{i+2} \cdots s_n s_{min(i+k,n)}, R, i$)
(4) Return $R$
```

Example 3.8. For the input "xxyyyaxyydxx\$" and $k = 3$, BuildBoundedTree will send the following suffixes to InsertBoundedString: "xxy", "xyy", "yyy", "yya", "yax", "axy", "xyy", "yyd", "ydx", "dxx", "xx\$", "x\$", and "\$".

InsertBoundedString takes a string x, a node, n, an index number, index, and the maximum length pattern to be found, k, as arguments. The string is added to the tree. InsertBoundedString requires the terminating character '\$' to



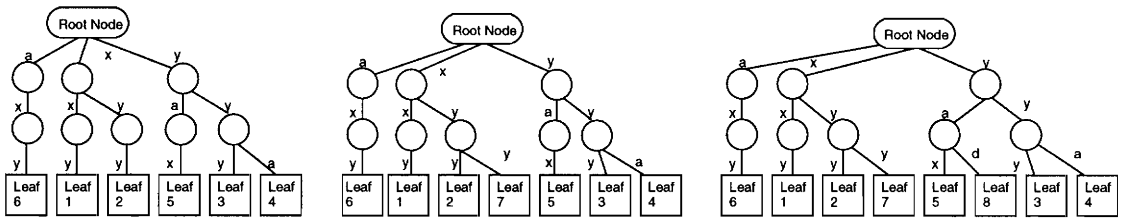
Examples 3.9.1, 3.9.2, 3.9.3, 3.9.4, and 3.9.5

be unique in the string. Depth is a global variable that is initialized to zero before InsertBoundedString is called the first time.

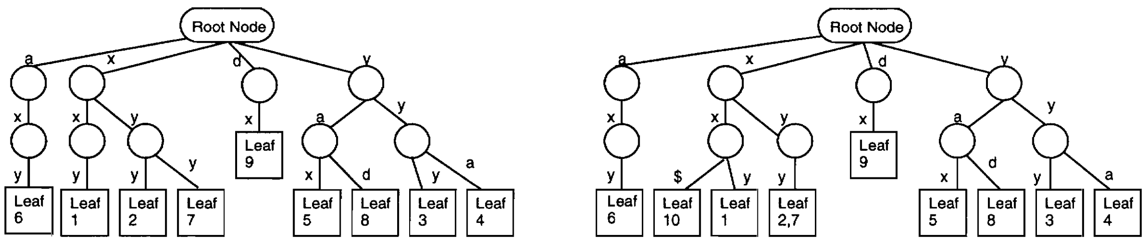
Algorithm: InsertBoundedString($x_1 x_2 \cdots x_n$,node,index,k)

```
(1)  Is $depth = k + 1$?
(2)  If so:
(3)         Add index to $node$'s list of indicies.
(4)         Return.
(5)  Is there a child of $node$ connected by an edge labeled with $x_1$?
(6)  If so:
(7)         Delete $node$'s list of indicies.
(8)  If not, create one by calling AddPtr($node, x_1$).
(9)         Increment $node$'s branch counter.
(10)        Add $index$ to the child's list of indicies.
(11) Increment $node$'s leaf counter.
(12) Call InsertString($x_2 x_3 \cdots x_n, child, index$).
(13) Return.
```

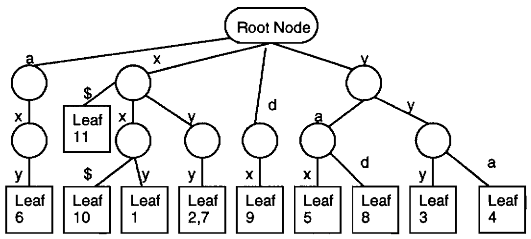
Example 3.9. Examples 3.9.1 through 3.9.11 show the intermediate steps of building the p-tree, after each suffix is added.



Examples 3.9.6, 3.9.7 and 3.9.8



Examples 3.9.9 and 3.9.10



Example 3.9.11

3.4.2 Proof of k-bounded Tree Building

The first k characters of each suffix are placed into the p-tree, and there are exactly n entries in the index lists in the leaf nodes in the tree. These entries have the same one to one mapping to the characters of the input string as the leaves in the unbounded tree.

The proof of correctness of the tree building algorithm is by induction on the length of the string. Again, a k-bounded tree is correct if Properties 3.1.6. hold. Given a string, β, it is shown why the tree for the string $a\beta$ is correct if the string for β is correct.

Lemma 3.4: The algorithm InsertBoundedString is correct.

Proof by Induction on the length of the string x.

Part I: (base case)

n **= 0:** If depth $= k$, InsertBoundedString adds index to the current leaf's list of indices. Otherwise, InsertBoundedString returns with no change to the tree. This allows the depth k leaves to have multiple index values.

InsertBoundedString returns with no changes to the tree.

(basis) n **= 1:** $s =$ "a" where 'a' is a character. Either node has a pointer labeled 'a' or not. In either case the resulting tree is a p-tree (Property 1).

Case 1: *node* does not have a child connected by an edge labeled with 'a'. Since a new pointer is added, the branch counter is incremented. Since a new leaf will be added the leaf counter is incremented. The index of the new leaf is added to the idxs list of the new leaf node. All the properties are met: One new index value with a path of "a" is created (Properties 3 and 4). The new leaf contains its index (Property 2). The new leaf has zeros for the branch and leaf counters (zeroed in the node creation process). The old node has incremented leaf and branch pointers.

Case 2: *node* does have a child labeled 'a'. Since a new pointer is not added, the branch counter is not incremented. Since a new leaf will be added the leaf counter is incremented. InsertString is called with x = "" and node =

GetPtr(node, 'a'). This is assumed to properly add a leaf node. All the properties are met: One new leaf with a path of "a" is created (Properties 3 and 4). The new leaf contains its index (Property 2). The new leaf has zeros for the branch and leaf counters (zeroed in the node creation process). The old node has incremented leaf and branch pointers. The new node has leaf and node counters of 1 and index list containing index.

Part II: (induction)

Assume: InsertString($\beta, node, index$) is correct.

Prove: InsertString($a\beta, node, index$) is correct.

Case 1: *node* does not have a child connected by an edge labeled with 'a'. Since a new pointer is added, the branch counter is incremented. Since a new leaf will be added the leaf counter is incremented. The index of the new leaf is added to the idxs list of the new node. InsertString is called with x = "β" and node = GetPtr(node, 'a'). This is assumed to be correct for "β" (Properties 2 and 3). Since the new branch is appended to a node with a path labeled "a", the new an entire path of $a\beta$ is created (Property 4). The tree is still a p-tree (Property 1).

Case 2: *node* does have a child connected by an edge labeled with 'a'. Since a new pointer is not added, the branch counter is not incremented. Since a new leaf will be added the leaf counter is incremented. The index of the new leaf is added to the idxs list of the new node. InsertString is called with x = "β" and node = GetPtr(node, 'a'). This is assumed to be correct for "β" (Properties 2 and 3). Since the new branch is appended to a node with a path labeled "a", the new an entire path of $a\beta$ is created (Property 4). The tree is still a p-tree (Property 1).

All nodes have correct branch and leaf counters and index lists. The path to each leaf is labeled with the suffix used to create it. Each leaf has a unique index corresponding to the index of the suffix.

By parts I and II, Lemma 3.4 is correct for $n \geq 1$.

Proof of BuildBoundedTree.

Theorem 3.3: The algorithm BuildTree is correct

Proof by Induction on the length of S.

Part I: (base case)

n **= 1:** BuildTree calls InsertString("a",$root, i$), creating one leaf with an index of i and a path labeled "a". This upholds the four properties in 3.1.6.

(Basis) n **= 2:** BuildTree calls InsertString("a",$root$, i) and InsertString("", $root$, $i+1$) creating two leaves with indices of i and $i+1$ respectively, and paths labeled "a" and "". This upholds the four properties in 3.1.6.

Part II: (induction)

Assume: BuildTree(β) holds, producing $|\beta|$+1 leaves numbered from 1 to $|\beta|$+1. In other words, assume that the four properties in 3.1.6 are maintained.

Prove: BuildTree($a\beta$).

Build tree calls InsertString("$a\beta$",$root, i$), InsertString("β",$root, i$) $\cdots$ InsertString("",$root, i$). Calling InsertString("β", $root, i$) $\cdots$ InsertString("",$root, i$) is the same as calling BuildTree(β), which is assumed to produce $|\beta|$+1 leaves numbered from 1 to $|\beta|+1$. Calling InsertString("$a\beta$", $root, i$) produces one leaf with index i, and a path to it labeled "$a\beta$". InsertString is assumed to maintain the properties if a k bounded p-tree, with respect the suffix that it is given. If all the substrings are added to the tree by InsertString, the tree will be a proper p-tree.

By Parts I and II, Theorem 3.3 is correct for $n \geq 1$.

3.4.3 k-bounded Detection

Since the definition of k-bounded MRPs basically ignores patterns longer than k, and builds a truncated tree, the same detection algorithm may be used on the truncated tree to find the k-MRPs. The only modification is that the leaf nodes in the tree need to be able to hold more than one index, and the detection algorithm needs to add all indices to MRPlist.

3.4.4 Proof of k-bounded Detection

The same list adder is used in k-bounded detection. The difference is that since the tree is limited to patterns of length k, only patterns of length k or less are added. No changes to MRPlist or the adder are needed. Since there are no differences between the unbounded detection algorithm and the k-bounded detection algorithm, the proof for the unbounded case also applies to the k-bounded case.

3.5 Local MRPs

Sometimes it is interesting to determine the MRPs that occur close to each other and disregard any that occur more than a certain distance from each other.

This can be useful in Human Computer Interaction studies where a command log is being analyzed. Frequently used patterns (or rather, patterns used close to each other) will be detected, but commands used only once a day might not.

3.5.1 Local MRP

A pattern p is a local MRP of S if there exists $s' \leq S$, such that p is an MRP of s'. Note that all repeating patterns of S are local MRPs to some substring of S. In the extreme case, any two repeating patterns are local to the substring that begins at the beginning of the first pattern and ends at the end of the second pattern.

Local MRPs Within a Radius of l. A pattern p, occurring at location r is a local MRP with radius l if p is an MRP of the pattern $S[r,\max(|S|,r+l)]$. A pattern of S is considered to repeat if two or more occurrences start within l characters. It is possible for the MRPs detected to have lengths longer then l. Consider the example where $S =$ "ababababab\$" and $l = 4$. When $r = 4$, the p-tree will contain the strings: "ababababab\$", "babababab\$", "abababab\$", and "bababab\$". The MRPs detected will be "abababab", and "bababab".

3.5.2 Global MRP

An MRP is a global MRP if it is an MRP of S. Section 1.1.1 defines global MRPs.

3.5.3 Detection of Local MRPs

If it is only important or useful to detect patterns that are MRPs in some locality, it is possible to conserve considerable memory, although time complexity is not improved because the pattern length, k, is potentially unbounded. If both k and R are bounded the algorithm becomes linear. (n insertions of length k strings and a traversal of n branches of length R.)

Detection of MRPs Within a Radius of l. If it is decided that only patterns that repeat within l characters of each other are interesting, it is only necessary to maintain $l + 1$ branches in the p-tree. When the $r + l^{th}$ branch is added; the r^{th} branch is traversed, adding any repeating patterns to MRPlist; then the r^{th} branch is removed, decrementing the appropriate counters in the nodes. Since each branch of the tree is fully traversed, this will add slightly to the cost of the detection algorithm, but the complexity will not change. Instead of using $O(n^2)$ memory, the tree will take $O(ln)$ memory.

At the end of the algorithm, MRPlist will contain all the patterns of S that are local MRPs with a radius of l. To get only the patterns that are MRPs within radius l of character r, take only those entries that have a starting index between l and $l + r$. This is a quick calculation since MRPlist is sorted by starting index.

3.6 Local k-MRPs

If it is further possible to limit interesting MRPs to a length of k, then it is possible to save even more memory by using a tree with radius r, as described in Section 3.5.3 but limit the depth of the tree to k. A radius r, k-bounded p-tree takes O(lk) memory (a maximum of l branches, each with a maximum depth of $k + 2$) no matter what size the input is. The mlist, however, still requires O(n) memory in the worst case.

3.7 Local MRPs and Large Input Sets

If it is desired to detect local k-MRPs in a very large data set, it is possible to set up a circular buffer of length k for the input. This requires that only k characters of the input to be stored in memory at any one time. However, if the actual patterns of the MRPs are desired, and not just the locations, MRPlist must be modified to retain them. This would cost at most, O(nk) memory (a possible n patterns of length k). The total size of the input must still be indexable (smaller than the largest integer type on the machine it is run on).

Example 3.7.1. Suppose that S = "abxabya$" and $k = 4$. The first k characters of the input are read into the buffer. The buffer will then contain "abxa". The string "abxa" is added to the p-tree. The next character of the input is read in and the first character is dropped from the buffer. The buffer is then "bxab". This string is added to the p-tree. The next character is read in and the second character is dropped from the buffer. The buffer contains "xaby". The third character is dropped and the next character is read into the buffer. The buffer is then "abya". This string is added to the tree.

The p-tree now contains the three strings: "abxa", "bxab", "xaby", and "abya". The branch "abxa" is traversed, and removed. As is it traversed, the occurrence (1,1) is added to MRPlist because it repeats. Since there are no other entries in MRPlist at this time, it is not deleted. "a" is a radius 4 local MRP of "abxabya$". After this step the p-tree contains only "bxab", "xaby", and "abya".

The next string, "bya\$" added to the p-tree. The "bxab" branch is traversed and removed, adding (2,1) to MRPlist. The string "ya\$" does not add any occurrence to MRPlist. The string "a\$" adds the occurrences (4,1). This represents the pattern "a", but is not deleted because it is an independent occurrence. The string "\$" also does not add any occurrences.

The final MRPlist is {(1,1),(2,1),(4,1)}, representing the patterns {"a", "b", "a"}.

Alternatively, for unlimited size input, using a circular buffer for the input, modify MRPlist to hold only patterns and not the indices. The limiting factor would then be fitting MRPlist into memory. Since at most, only the last l entries in MRPlist are necessary to determine if a new entry is an MRP, only O(lk) space is required, if the rest of MRPlist is spooled off to disk. The entire file could still be quite large but not require much memory.

4 Performance Analysis

4.1 Performance of Standard Algorithm

4.1.1 Time Complexity of Different Parts of the Unbounded Algorithm

Let l = the number of legal characters in the alphabet.

Let k = the limit on the length of the MRPs.

Let q = the number of nodes that have to be added for a given string.

ReadInput. The input is read in character by character. There are n characters so this takes O(n) time.

InsertString. Each node may have up to l branches. The list of branches must be traversed to determine if a given node already has a branch with a given label. This takes at most O(l) time. For a string of length m to be added to the tree, there are $m - q$ nodes that have children along the path of the pattern to be inserted. There are q nodes that need to be created. New nodes do not have branches to be traversed. Therefore, InsertString takes O(l)O($m - q$) + O(q) or O(m) time.

BuildTree. BuildTree causes InsertString to add $n + (n-1) + (n-2)...2 + 1$ nodes to the tree. This is $\sum_{i=1}^{n} i = \frac{n^2+n}{2}$ or O(n^2).

FindMRP. At most, the traversal will cover every node once, and there are O(n^2) nodes in the tree. This is O(n^2).

AddNode. There are n additions to the list of MRPs which may contain up to n patterns. Because [at least a partial] list traversal is required for each addition, the algorithm requires $\frac{n^2+n}{2}$ steps and is O(n^2). If the list is organized in a balanced tree, sorted by starting index of the occurrence, insert time can be reduced to O($log(n)$) for a total time of O($n\, log(n)$). Since items in the MRPlist have unique starting positions, they are easy to sort.

Theorem 4.1: The total time complexity of the unbounded algorithm is O(n^2). The proof is in the preceding arguments.

4.1.2 Space Complexity and Large Input

The p-tree may require up to O(n^2) nodes. There can be as many as n entries in MRPlist. Total space required is O(n^2). Since the input string is indexed, the input size is limited to the largest index type on the implementation platform. Using Borland C++, this is **longint**.

4.2 *Performance of k-bounded Detector*

4.2.1 *Complexity of Different Parts of the Algorithm*

Let l = the number of legal characters in the alphabet.

Let k = the limit on the length of the MRPs.

Let q = the number of nodes that have to be added for a given string.

ReadInput. The input is read in character by character. There are n characters so this takes O(n) time.

InsertBoundedString. Each node may have up to l branches. The list of branches must be traversed to determine if a given node already has a branch with a given label. This takes at most O(l) time. For a string of length k, there are $k - q$ nodes that have children along the path of the pattern to be inserted. There are q nodes that need to be created. New nodes do not have branches to be traversed. Therefore, InsertString takes O(l)O($k - q$) + O(q) or O(k) time.

BuildBoundedTree. While reading the input it is passed through a filter to remove illegal characters. This adds nothing to the O(n) time for reading a string character by character. Then, for each character in the input, InsertString is called. Therefore BuildTree takes O(n) + O(n)O(k) or O(nk).

FindMRP. There can be at most $n(k-1)$ +1 nodes in a tree of depth k that is has exactly n leaves. If $l < n$, then the depth 1 layer will have at most l nodes, because the root node can have at most l branches. FindMRP is a traversal of the tree so it takes O(nk) time.

AddNode. There are n additions to a list of up to length n. Because (at least a partial) list traversal is required for each addition, the algorithm requires $\frac{n^2+n}{2}$

steps and is O(n^2). If the list is organized in a balanced tree, sorted by starting position of the occurrence, insert time can be reduced to $log(n)$ for a total time of O($n\,log(n)$).

4.2.2 Complexity When Both k and R are bounded

ReadInput. The input is read in character by character. There are n characters so this takes O(n) time.

InsertBoundedString. Each string inserted into the tree is of length k or less. This takes O(k) time.

BuildBoundedTree. There are n strings inserted into the tree. This takes O(nk) time.

FindMRP. There are n branches of up to length k to traverse. This takes O(nk) time.

AddNode. Since only the last R patterns need be considered, there will be as many as n inserts into a sorted list of length R. This is O($n\,log(R)$).

The total complexity is O($n + k + nk + nk + n\,log(R)$) or O($n$).

4.2.3 Space Complexity and Large Input

The p-tree may require up to O(nk) nodes. There can be as many as n entries in MRPlist. Total space required is O(nk). Since the input string is indexed, the input size is limited to the largest index type on the implementation platform. Using Borland C++, this is longint().

Theorem 4.2: The total time complexity of the unbounded algorithm is O($n\,log(n)$). The proof is in the preceding arguments.

5 Implementation and Empirical Results

A working implementation was developed and compiled using Borland C++ 4.0 on an IBM clone 486, running DOS 5.0, 4Dos 4.0, and MS-Windows 3.1. The program was tested with input files ranging from 100 bytes to 5 megabytes and run with a variety of parameter combinations.

Figure 5.2 shows the marked improvement in runtimes gained by limiting the pattern length over a variety of input sizes. Figure 5.1 shows that enforcing locality does not reduce the complexity of the problem. Figures 5.3 and 5.4 show the runtimes for a variety of k and R values.

Limitations. Since I use the index numbers of the string, I am limited to input sizes smaller than (long unsigned) integers on whatever platform the algorithm is running on.

6 Summary

This work extends Siochi's work and sets forth straightforward algorithms that are useful in a variety of practical situations. By reducing the problem complexity by either limiting the length of patterns considered or by limiting the context in which the patterns must repeat, the algorithms are feasible for a wider variety of applications.

6.1 Multiple Character Tokens

In certain applications like Human Computer Interaction, the input data will be a stream of tokens. These tokens could represent the commands used by a user over the course of a day. The algorithms do not differentiate between single character tokens and multiple character tokens. Multiple character tokens would only be longer repeating patterns. Care must be taken to remove the strings containing

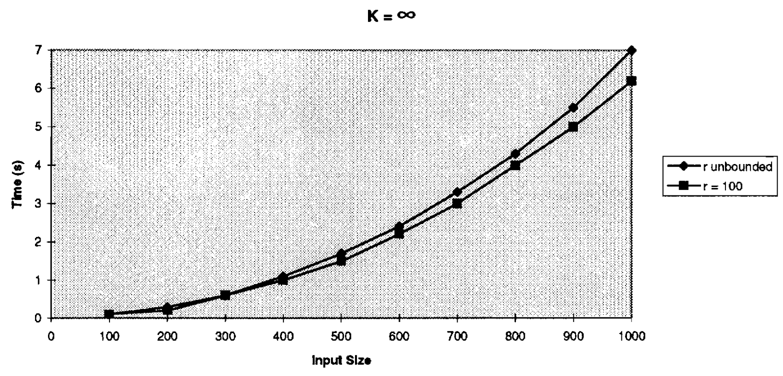


Figure 5.1 Performance times for unbounded MRP's

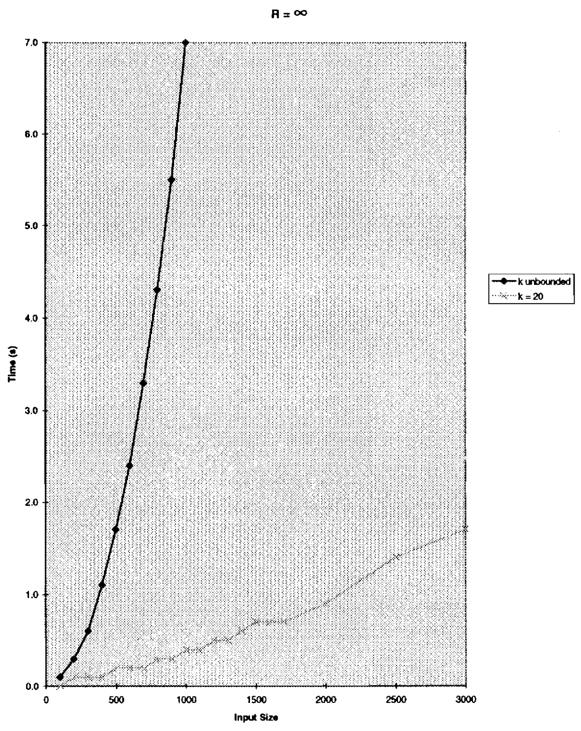
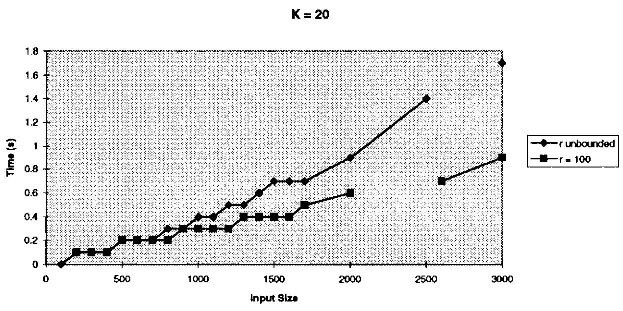
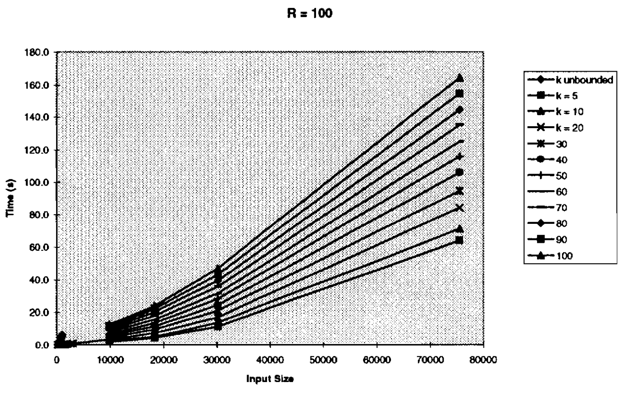


Figure 5.2. Performance times for global neighborhoods

fragmented tokens from the final output. If these tokens were to start or are delimited with a single (or one of a few) character not used elsewhere in the input an exclusion filter (see Appendix I) can be used to eliminate repeating patterns that



Figures 5.3 Performance times for $k = 20$



Figures 5.4. Performance times for $R = 100$

start with fragmented tokens.

References

Amir, A., Farach, M. (1991). Adaptive Dictionary Matching. *32nd FOCS* p. 760-766 IEEE Press

Amir, A., Farach, M., Gali, Z., Giancarlo, R., Park, K. (to appear). Dynamic dictionary matching. *Journal of Computer and Systems Sciences*

Frakes, W. B., Baeza-Yates, R. (1992). Information Retrieval Data Structures & Algorithms. p 21, 68 Prentice Hall, Englewood Cliffs, NJ

Gonnet, G. (1983). Unstructured data bases or very efficient text searching. *ACM PODS* vol. 2, p. 117-24 Atlanta, Ga.

Siochi, A. C. (1989). Computer based user interface evaluation by analysis of repeating usage patterns in transcripts of user sessions. Virginia Polytechnic Institute and State University, Blacksburg, Virginia

Siochi, A. C., Ehrich, Roger W. (1991). Computer analysis of user interfaces based on repetition in transcripts of user sessions. *ACM Transactions on Information Systems* vol. 9, No 4, October 1991, Pages 309-335.

Storer, J. A. (1988). *Data compression: methods and theory.* p. 15. Computer Science Press

Weiner, P. (1973). Linear pattern matching algorithms. *Proceedings of IEEE 14th Annual Symposium on Switching and Automata Theory* Rand Corp. Santa Monica, California

Appendix I

mrp

Usage: **mrp [-hlt] [-rR] [-kK] [-mM] [-f<filter file>] [-e<exclusion file>]**
The mrp command reads from standard input, finds the maximal repeating patterns and writes them to standard output.

FLAGS

-h Displays this help message.

-l Displays the locations of the patterns in the output. By default, locations are NOT printed. Locations are of the form [startpos, length].

-t Disables the printing of the text of the patterns found. If R and K are both nonzero, only pattern location may be displayed. Otherwise patterns are printed by default. Note that the text patterns are terminated with a $ (dollar sign). The text patterns cannot be printed when both R and K are nonzero because the input string is discarded as it is processed in order to conserve memory.

OPTIONS

-rR Only patterns that start within R characters of each other are considered. If R is 0 then all patterns are considered. (R is the radius or locality in which patterns will be checked for in.) Set R to a non zero value for very large data sets. Only R strings are kept in memory at any one time. The i^{th} suffix is removed from the tree when the $i + R^{th}$ suffix is added. Memory use for the tree is then O(Rn) rather than O(n^2). For most applications a value for R of between 200 and 1000 is recommended.

-kK Ignore patterns of length greater than K. The length K prefixes of longer patterns will be detected. If K is set to 0, then unlimited length patterns are considered. WARNING: Setting K to 0 will use enormous amounts of memory. K defaults to 100. This limits the depth of the tree to K+2 (K characters, plus the root node and a leaf node). Memory use for the tree is then O(nK) rather then O(n^2).

If both R and K are set to nonzero values then memory use for the tree is O(RK). Memory used for mlist is still O(n), however.

-f<filter file> Use <filter file> to redefine the input filter. Non valid characters are removed from the input as they are read in and Ignored. The default filter allows all printable characters, except space. This is primarily an error prevention feature to disallow unexpected characters into the data.

-e<exclusion file> Use <filter file> to redefine the exclusion filter. The exclusion filter determines valid starting characters for patterns. Patterns that start with excluded characters are ignored. For example, it is possible to set an exclusion filter to that only patterns that start with capital letters are considered. Only patterns starting with valid characters are added to tree. By only allowing a few characters to start valid patterns, the number of strings to added to the tree can be drastically reduced. This does not affect the order of the algorithm because the number of strings to be inserted into the tree remains O(n), although with a lower constant. In some applications the data will consist of multi-character tokens.

If these tokens either begin or end with a few common characters or are delimited with a specific character, creating an exclusion list of these characters would filter out a great deal of extraneous information.

Caution: When a small list of valid starting characters is combined with a small radius, it is possible to detect few or no MRPs because the tree will be very sparse.

-mM Where M is the minimum length of patterns to be considered.

CAUTIONS

This program can use enormous amounts of memory(n^2), especially if poor parameters are chosen. Since most data sets are much larger than the repeating patterns in them setting K to some small number (<20) will often not cause any loss of data. For very large input, setting R to a nonzero value may allow the entire data structure to stay in real memory.

FILTER FILES

The filter files are composed of 128 integers separated by whitespace. These integers correspond to the first 128 characters of the ASCII set. Any character for which the integer is 1 (nonzero) is considered valid. Any character for which the integer is 0 is considered invalid.

USEFUL RELATED COMMANDS

Commands: more(1), uniq(1), sort(1), grep(1)/egrep(1)/fgrep(1).