Real-Time Pattern Matching and Quasi-Real-Time Construction of Suffix Trees

Preliminary Version

S. Rao Kosaraju*

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21136

Abstract

We design simple real-time algorithms for the following problems for any text string $T = t_1t_2...t_n$ and pattern string $P = p_1p_2...p_m$: (a) given $T\#P$ as input, test whether P^R is a substring of T, and (b) given $T\#P$ as input, test whether P is a substring of T. Even though these results were claimed in a voluminous paper by Slisenko, the design of a convincing and understandable solution is a well-known open problem. Our algorithm is based on a novel *top-down* suffix tree construction algorithm. This algorithm does not construct the suffix tree in real-time; but it constructs enough of the suffix tree in real-time so that it can respond to pattern match queries in real-time.

1 Introduction

A *linear-time algorithm* runs in $O(n)$ steps on any input of length n. A *real-time algorithm* must satisfy the additional requirement that on any input symbol, the algorithm spends only $O(1)$ steps. In a *quasi-real-time algorithm* for a data structure, $O(1)$ steps can be performed on each input symbol, and all the data structure queries can be answered in real-time. There is no requirement on the internal evolution of the data structure.

We construct the suffix tree of a text in quasi-real-time in the sense that pattern match queries can be handled in real-time. In particular, we design real-time algorithms for the following problems for any text string $T = t_1t_2...t_n$ and pattern string $P = p_1p_2...p_m$: (a) given $T\#P$ as input, test whether P^R is a substring of T, and (b) given $T\#P$ as input, test whether P is a substring of T.

These problems are of some practical significance. An example is the processing of a long DNA sequence in real-time so that any intervening searches for DNA pattern strings can be done in real-time. Another application is automatic telephone message handlers. Here the text is the messages from various callers and the patterns are the identification sequences for individuals inquiring about the arrival of messages.

A new linear-time suffix tree construction algorithm is developed in section 2. The algorithm is converted into a quasi-real-time algorithm in section 3. Finally, in section 4, we show how to make use of this algorithm in answering pattern match queries in real-time. In this preliminary version we emphasize the intuition behind the algorithms at the expense of formal proofs.

2 A New Linear-time Algorithm

After reviewing the terminology for suffix trees, we develop a new linear-time suffix tree construction algorithm in this section.

2.1 Suffix Trees

For any input $X = x_1x_2...x_n$ and $1 \le i \le n$, let *suffix*$_i$, or the i^{th} suffix, be $x_i...x_n\$$ where \$ is a special symbol not in the alphabet of X, and *suffix*$_{n+1}$ be \$. For any $1 \le i \le n+1$, let $\Sigma_X(\le i)$ be the compact trie for suffixes $1, 2, \cdots, i$, and let $\Sigma_X(\ge i)$ be the compact trie for suffixes $i, i+1, \cdots, n+1$ of X. $\Sigma_X(\ge 1)$ is the *suffix tree* for X. Let $\Sigma_X(i..j, \ge k)$ be the compact trie for suffixes i, $i+1$,..., j, k, $k+1$,...,$n+1$. Thus $\Sigma_X(i..i, \ge k)$ is the compact trie for suffixes i, k, $k+1$,...,$n+1$. When X is understood, we suppress it. Thus, we write $\Sigma_X(\ge$

*Supported by NSF grant CCR-9107293

STOC 94- 5/94 Montreal, Quebec, Canada
© 1994 ACM 0-89791-663-8/94/0005..$3.50

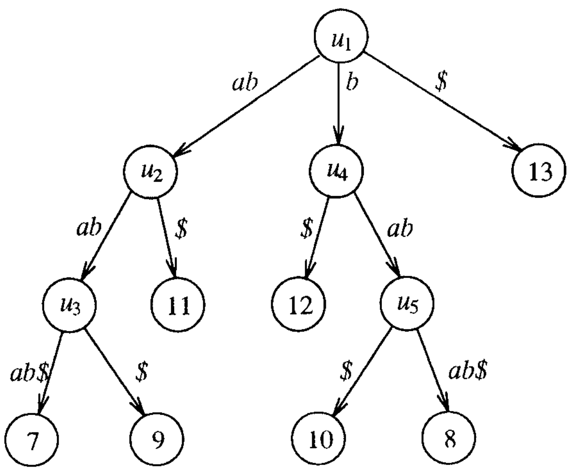

Figure 1: $\Sigma(\geq 7)$ for *babaababababab*.

j) simply as $\Sigma(\geq j)$. For X=*babaababababab*, $\Sigma(\geq 7)$ is shown in Figure 1.

Note that each edge is labeled by a substring of X. In the actual implementation, each label is represented by two pointers into X. The parent and the grandparent of any node u are denoted by *parent*(u) and *gparent*(u), respectively. The *head* of any edge (u, v) is the node v. In addition to the nodes of the tree, it is convenient to be able to refer to each position (*locus*) within a label. The *string for a locus*, u, is the concatenation of the labels on the path from the root to that locus, and is denoted by σ_u. If u is a node then we say that σ_u *occurs explicitly*; otherwise it *occurs implicitly*. The *locus* for σ_u is u. Thus in Figure 1, *abab* occurs explicitly and its locus is u_3; *ba* occurs implicitly and its locus is in the edge between u_4 and u_5. The node of any locus u is u itself if u is a node, otherwise it is the head of the edge on which u lies. A locus is specified by its node together with the proper offset. Thus, in Figure 1, the locus of *ba* is specified by u_5 and an offset of 1. The *depth* of any locus u is the length of σ_u.

2.2 Additional Links

There are two well-known linear-time algorithms for the construction of a suffix tree: McCreight's and Weiner's algorithms [McC76, Sei77, CS85, Wei73]. As in these algorithms, we maintain additional links between nodes of the tree. Our algorithm is based on McCreight's algorithm; but **our links are the union of the links maintained by the two algorithms.** We refer to the links corresponding to McCreight's and Weiner's algorithms as M- and W- links, respectively. We first specify these links in detail.

At each node, u, for each input symbol c, there is

u:	u_1	u_2	u_3	u_4	u_5	7	8	9	10	11	12	13
$W_a(u)$:	u_2^*	–	–	u_2	u_3	–	7	–	9	–	11	–
$W_b(u)$:	u_4	u_5	8^*	–	–	–	–	8	–	10	–	12

Figure 2: W-links of Figure 1.

u:	u_1	u_2	u_3	u_4	u_5	7	8	9	10	11	12	13
$M(u)$:	–	u_4	u_5	u_1	u_2	8	9	10	11	12	13	u_1

Figure 3: M-links of Figure 1.

a W-link denoted by $W_c(u)$. It specifies the locus of $c\sigma_u$. If $c\sigma_u$ occurs explicitly the link is said to be *explicit*, otherwise the link is *implicit*. If $c\sigma_u$ doesn't occur in the tree, then the link has the special value "–". For the example of Figure 1, the W-links are listed in Figure 2. (The implicit links are superscripted with $*$).

At each node, u, there is an M-link denoted by $M(u)$. If σ_u=$c\alpha$, where c is a single symbol, then $M(u)$ specifies the locus of α. If α occurs explicitly the link is said to be *explicit*, otherwise the link is *implicit*. If α doesn't occur in the tree, then the link has the special value "–". For the example of Figure 1, the M-links are listed in Figure 3. (There are no implicit links).

Every suffix tree construction algorithm includes a new suffix into the tree by finding the maximum length prefix of this suffix that is a path in the tree and installing the new suffix at the corresponding locus. This locus is the *insertion locus* for the new suffix. The length of this prefix is the *insertion depth* of the new suffix. In Figure 1, $suffix_3$ = *baababababab$* has an insertion depth of 2. Note that this suffix can be included as the child of a new node, x, obtained by splitting the (u_4, u_5) edge into 2 edges (u_4, x) and (x, u_5) with labels a and b, respectively. The label of the $(x, leaf_4)$ edge is *abababab$*.

2.3 McCreight's Algorithm

As stated earlier, our linear-time algorithm is based on McCreight's algorithm. In the following we sketch McCreight's algorithm; the details can be obtained from [McC76]. The algorithm inserts $suffix_1$, $suffix_2$, ..., $suffix_{n+1}$ into an initially empty tree. At any instant assume that the algorithm already has constructed $\Sigma(\leq i)$ with its associated M-links with the possible exception of the M-link of *parent*($leaf_i$). The insertion of $suffix_{i+1}$ into it is shown schematically in Figure 4. The *parent*($leaf_i$) is denoted by π_i.

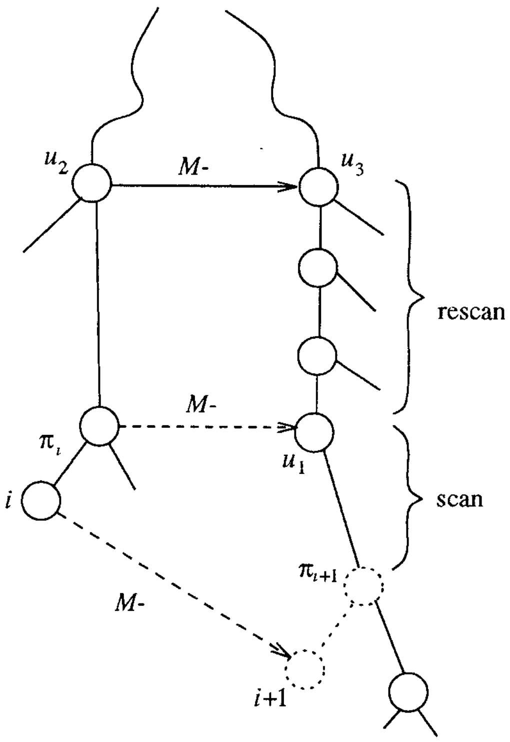

Figure 4: Insertion of $suffix_{i+1}$ by McCreight's algorithm.

In $\Sigma(\leq i)$, [McC76], all nodes other than the root, $leaf_i$, and π_i have explicit M-links. The root and $leaf_i$ cannot have M-links; π_i has an M-link but it can be implicit. McCreight's algorithm might not compute the M-link of π_i even when this link is an explicit one. The insertion of $suffix_{i+1}$ is as follows.

If π_i's M-link exists, we follow its M-link to u_1, $M(\pi_i)$, and proceed directly to *scan* described below. If $\pi_i's$ M-link is missing, we first perform the following *rescan* and then perform *scan*.

rescan: We climb up to u_2, $parent(\pi_i)$, follow its M-link to u_3, locate u_1, as described below, and then create an M-link from π_i to u_1. Initially we are at u_2 and u_3. At any instant we will be at some locus, x, on the edge (u_2, π_i) and some node, v, on the path from u_3 to u_1. We repeat the following until v becomes u_1. We choose the outgoing edge of v, say (v, v'), whose label's first symbol matches with the first next symbol on (x, π_i). If the (x, π_i) string is longer than the (v, v') string, then on (u_2, π_i) we advance by the length of the string of edge (v, v') and we move to node v'; i.e. make $v = v'$. If the (x, π_i) string is not longer than (v, v') string, we advance to node π_i, and we advance along (v, v') by the length of (x, π_i) string, reaching node u_1.
scan: Trace down a path from node u_1 by making symbol-by-symbol comparisons of the labels on the edges and the corresponding symbols of $suffix_{i+1}$ until a mismatch occurs. This locates π_{i+1}, the insertion locus for $suffix_{i+1}$, where we install $leaf_{i+1}$.
Finally we create an M-link from $leaf_i$ to $leaf_{i+1}$, completing the insertion of $suffix_{i+1}$ by McCreight's algorithm. During this insertion process, we denote the instantaneous position on the path from u_1 to π_{i+1} as the *locus of* $suffix_{i+1}$.

In the following, we modify McCreight's algorithm so that we can achieve the following additional goal: **In the suffix tree, for any locus u and any symbol c we want to compute the locus of $c\sigma_u$ in $O(1)$ steps.** Observe that this goal can be accomplished if the suffix tree has W-links at its nodes.

2.4 Modified McCreight's Algorithm: Algorithm MM

Algorithm MM maintains both M- and W-links at every instant. As in McCreight's algorithm, in $\Sigma(\geq i)$, node π_i might not have its M-link. We make a corresponding natural compromise for the W_{x_i}-link value of every node between u_3 and u_1 (excluding u_3, but including u_1 if u_1 is a node). Note that the correct value for each of these nodes is π_i. At this instant the W_{x_i}-link we maintain at each of these nodes is the child of π_i which was the head of the edge that was split to create π_i.

We implement each M-link by a pointer in the obvious way. However, our implementation of W-links is non-standard. It is easy to observe [Wei73] that in the tree several nodes can have their W_c-links, for some c, point to the same node. In our implementation, all such links will point to an *auxiliary node* which in turn points to the correct node. (This indirect linking mechanism will allow us to achieve certain monotonicity properties that are crucial in the development of the quasi-real-time algorithm.) Each W-link is specified by 2 fields: the subscript field, and the link field. In Figure 1, the W_b-links of both u_3 and $leaf_9$ are $leaf_8$. Thus the first field at both the nodes is b, and the second field contains the same auxiliary node. The second field of the auxiliary node contains 8, the index of the leaf pointed to, and the first field contains a special designated value to indicate that the node is an auxiliary node. Our algorithm keeps track of the depth of any node in the tree in a separate field at that node. We ignore the trivial problem of how this computation is incorporated into the algorithm. Even though each W-link chains through an auxiliary node, we describe each link as a single link. Even though an implicit link points to a node, we can find the exact locus in $O(1)$ steps. For example, in Figure 1, $W_b(u_3) = 8$, $depth(u_3) = 4$, and $depth(leaf_8) = 5$. We can infer that W_b-link of u_3 is an implicit link and it

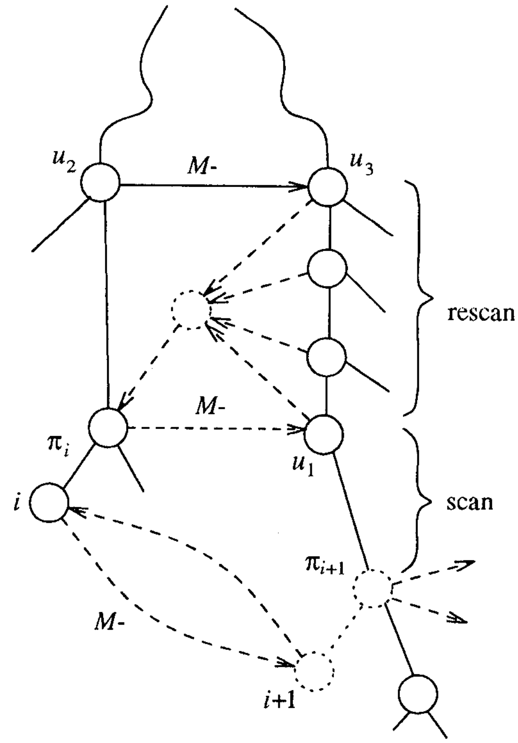

Figure 5: W-links created when $suffix_{i+1}$ is inserted.

points to $(u_5, leaf_8)$ edge between ab and \$.

Assume that we have already constructed the suffix tree $\Sigma(\leq i)$ with the associated M- and W- links. The insertion of $suffix_{i+1}$ into it and the creation of M-links are as described in the previous subsection. Figure 5 shows the W-links that get created. The links are shown unlabeled.

Note that the W_{x_i}-link of $leaf_{i+1}$ points to $leaf_i$, and all the other W-links at $leaf_{i+1}$ are undefined. If π_{i+1} is created by splitting an edge, then its W-links are the same as the W-links of the head of the edge it was created from. The W_{x_i}-link from each node between u_3 and u_1 (excluding u_3, but including u_1) must point to node π_i. During the *rescan* step when the first node on this path is encountered, we create an auxiliary node with a link to π_i, and replace the previous W_{x_i}-link of the node encountered by a W_{x_i}-link to the auxiliary node. On each subsequent node on the path between u_3 and u_1, we replace its previous W_{x_i}-link by a W_{x_i}-link to the auxiliary node.

Lemma 1 *In $\Sigma(\leq i)$, given any locus u and any symbol c, we can find the locus of $c\sigma_u$, when it exists, in $O(1)$ steps.*

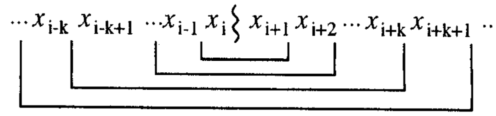

Figure 6: Path extension.

2.5 New Linear-Time Algorithm: Algorithm LT

Algorithm LT inserts suffixes in phases. Even though it is based on Algorithm MM, which inserts $suffix_1$ first and $suffix_{n+1}$ last, it inserts $suffix_{n+1}$ first.

Assume that at the beginning of a phase we have $\Sigma(\geq i+1)$ with all its M-links and W-links. (That is, we have already inserted $suffix_{i+1}$, $suffix_{i+2}$, ..., $suffix_{n+1}$.) Here we require that every node in the tree must have its M- and W- links correct as defined originally. In particular, note that every node other than the root must have an M-link. (Algorithm MM makes an exception for the parent of the last leaf installed.) During this phase we insert into $\Sigma(\geq i+1)$ an interval of suffixes $i-k+1, i-k+2, ..., i$ for a suitable k, resulting in $\Sigma(\geq i-k+1)$ with all its M-links and W-links. The insertions are done in the order $i-k+1, i-k+2, ..., i$. First we describe how the interval of suffixes is identified.

Interval Identification

On input symbol x_i we compute the locus of $x_i x_{i+1}$. This clearly can be done in $O(1)$ steps. (If this locus does not exist, then $suffix_i$ forms the interval. We insert it and create the necessary links, completing the current phase.) Then on input symbol x_{i-1} we compute the locus of $x_{i-1}x_i x_{i+1} x_{i+2}$, starting at the locus of $x_i x_{i+1}$. In general, on input x_{i-j} we compute the locus of $x_{i-j}x_{i-j+1} \dots x_{i+j}x_{i+j+1}$ starting the locus of $x_{i-j+1} \dots x_{i+j}$, as shown in Figure 6.

If the locus of $x_{i-j+1} \dots x_{i+j}$ is u, then the locus of $x_{i-j}x_{i-j+1} \dots x_{i+j}$ can be computed in $O(1)$ steps as given in lemma 1. Let this locus be u'. Then from u' we simply walk down along the symbol x_{i+j+1} in the tree $\Sigma(\geq i+1)$ to complete the step. Hence each length 2 path extension can be performed in $O(1)$ steps.

Suppose the locus for $x_{i-k} \dots x_{i+k+1}$ does not exist in $\Sigma(\geq i+1)$. This can be due to the non-existence of the $W_{x_{i-k}}$-link or the non-existence of the extension on symbol x_{i+k+1}. Then suffixes $i-k+1, i-k+2, ..., i-1, i$ form the interval of suffixes inserted in the current phase. For the example of Figure 1, let the previous phase end with $\Sigma(\geq 7)$. On $x_6 = b$, the locus is between u_4 and u_5, since $x_6x_7 = ba$. On $x_5 = a$, the locus is u_3, since $x_5x_6x_7x_8 = abab$. Since

$x_4x_5x_6x_7x_8x_9 = aababa$ is not a path in $\Sigma(\geq 7)$, suffixes 5 and 6 form the current interval of suffixes.

Before describing the details of the insertion process, let us highlight the importance of what has been achieved. Since $x_{i-k+1}...x_{i+k}$ is a path in $\Sigma(\geq i+1)$, each $x_\ell x_{\ell+1}...x_{i+k}$, $i-k+1 \leq \ell \leq i$, is a path in $\Sigma(\geq i+1)$. (A suffix of a path is a path.) Thus, we have:

Lemma 2 *Irrespective of the order in which we insert the suffixes of the current interval, every one of the suffixes $i-k+1, ..., i-1, i$ gets inserted at an insertion depth not less than $k+1$.*

In the next section when the algorithm is converted into a quasi-real-time algorithm, this property plays a critical role. (Since no insertion depth is less than $k+1$, any pattern match query for a pattern of length at most k can be performed correctly while the current insertions are in progress.)

In addition, since $x_{i-k}...x_{i+k+1}$ is not a path in $\Sigma(\geq i+1)$, if we were to insert $suffix_{i-k}$ into $\Sigma(\geq i+1)$, its insertion depth will not be more than $2k+1$. We can easily establish the following lemma.

Lemma 3 *If the insertion depth of $leaf_{i-k}$ in $\Sigma(\geq i+1)$ is not more than δ, then the insertion depth of $leaf_{i-k}$ in $\Sigma(\geq i-k+1)$ is not more than $\delta + k$.*

The following is an immediate consequence of this lemma.

Corollary 4 *The insertion depth of $leaf_{i-k}$ in $\Sigma(\geq i-k+1)$ is not more than $3k+1$.*

Note that $\Sigma(\geq i-k+1)$ is the suffix tree at the beginning of the next phase. The above bound on the insertion depth will be important in establishing the $O(n)$ speed bound for the overall algorithm.

Insertion of Suffixes

Now we describe the details of the insertion of suffixes $i-k+1$, ..., $i-1$, i into $\Sigma(\geq i+1)$.

We first insert $suffix_{i-k+1}$. The interval identification step has already located the locus of $x_{i-k+1}x_{i-k+2}...x_{i+k}$. ¿From that locus, which is at depth $2k$, we walk down the tree one symbol at a time until we locate the insertion locus for $suffix_{i-k+1}$, where we install $suffix_{i-k+1}$ resulting in $\Sigma(i-k+1...i-k+1, \geq i+1)$. (So far suffixes $i-k+1$, $i+1$, $i+2$, ... , $n+1$ have been inserted.) Then we apply Algorithm MM and insert suffixes $i-k+2, ..., i-1, i$ in order. Then we create a W_{x_i}-link from $leaf_{i+1}$ to $leaf_i$ and an M-link from $leaf_i$ to $leaf_{i+1}$. At this stage, even though we have inserted suffixes $i-k+1, i-k+2, ..., i$ into $\Sigma(\geq i+1)$, the resulting $\Sigma(\geq i-k+1)$ need not have all its links. Since we followed Algorithm MM, the M-link of $\pi_i = parent(leaf_i)$ and the W-links that point to π_i might be absent. Note that $M(\pi_i)$ must be a node in $\Sigma(\geq i-k+1)$. Thus if the M-link of π_i is absent, we climb up to the parent of π_i, traverse its M-link and create the W-links for nodes that need to have W-links to $parent(i)$ by following the rescan step of Algorithm MM. Finally we create the M-link of π_i.

We now argue that this algorithm runs in linear-time. By making use of corollary 4 we can easily prove the following lemma.

Lemma 5 *Let the next phase insert k' suffixes. Then the insertion depth of any suffix in the next phase is not more than $3k+k'$. The number of steps needed to perform the next phase is not more than $c_1(k+k')$, for a suitably large constant c_1.*

Now let Algorithm LT insert the $n+1$ suffixes in m phases, and let it insert k_i suffixes in the i^{th} phase. Then, by lemma 5, the total number of steps performed by the algorithm is not more than $c_1(0+k_1) + c_1(k_1+k_2) + \cdots + c_1(k_{m-1}+k_m)$ which is not more than $2c_1(n+1)$. Hence Algorithm LT runs in $O(n)$ steps.

The following lemma specifies the progress that can be achieved during an intermediate instant of the insertion stage.

Lemma 6 *There exists a constant c_2 such that if we perform $c_2\alpha$ steps, for any $\alpha \geq 1$, on the current task starting with $suffix_{i-k+1}$ and at locus depth $2k$, then the locus depth of insertion of any unfinished suffix in the current interval is at least $max\{k, \alpha\}$.*

3 Quasi-Real-time Algorithm: Algorithm Q

Now we show that Algorithm LT can be adapted to run in quasi-real-time so that the suffix tree construction can proceed uninterrupted as more input is received. Throughout we assume that the input symbols get stored in an array so that any previous input symbol can be accessed in a single step. We also assume that the input is received right-to-left. In addition, whenever a leaf gets created a pointer from the corresponding position in the array to the leaf will also be created. This will permit accessing $leaf_i$ for any given i in a single step.

In Algorithm LT we assumed that when the current phase started, $\Sigma(\geq i+1)$ was completely computed. In Algorithm Q, $\Sigma(\geq i+1)$ will be constructed only partially, and there will be unfinished insertion tasks, running in the background, arranged in a stack. Each unfinished task corresponds to the unfinished insertion stage of one of the previous phases of Algorithm LT.

Each task on the stack carries enough information so that on a subsequent update step we can resume the unfinished insertions of the task from where we left off. This information is the following: the interval of suffixes, the index of the current suffix, if the current suffix is not the first suffix of the interval then a pointer to the last leaf inserted, and a pointer to the locus of the current suffix. The last pointer is specified by the head of its edge and the depth of the locus. This depth of the locus of the current suffix is denoted as the *current depth of the task*. Even though from the above information we can compute in $O(1)$ steps whether the task is currently performing rescan or scan, for simplicity, we assume that a separate field carries this information.

When we initiate an update of a task on the stack, if the stack was in rescan step then we make a **significant change** to the rescan step. We do not simply resume where we left off. Assume that the task was in rescan, the current suffix is $suffix_{i+1}$, and the locus of this suffix was node v, as shown in Figure 8 (ignoring the rectangular nodes). When we initiate the update, let $v' = M(gparent(leaf_i))$. If $depth(v') \leq depth(v)$, then we resume the rescan normally; otherwise we make $v = v'$ and resume the rescan from the new v. This important modification takes into consideration the possibility that from the instant the task was last updated, the $gparent(leaf_i)$ might have changed. Many additional nodes could have been inserted during this time interval. Such inserted nodes are shown as rectangles. (This modification permits skipping over such inserted nodes. All the links will also be set properly because of the way the W-links are implemented.)

When updates are performed on a task, we make sure that no M- or W-link is left "dangling". That is, if we were in the process of changing a link, then we finish the change as part of the update step.

The following algorithm makes use of a suitably large constant c.

Outline of Algorithm Q:

Initially the stack is empty.

At any stage, let the top two tasks on the stack be D and D', with D at the top. (If D' is absent, then ignore the operations on it.) Let the number of suffixes in the interval of tasks D and D' be d and d', respectively.

On each new input symbol received, perform path extension by length 2 in $O(1)$ steps. In addition, on each input symbol, perform c update steps on each of D and D' until $depth(D')$ equals $2d' + d$ or until D' finishes or until D finishes.

If one of the first two conditions holds, then perform c update steps on D_k only. If the third condition holds, then pop D and terminate the updates.

If the path cannot be extended by length 2, the task of inserting the current interval of suffixes is pushed onto the stack as a new task.

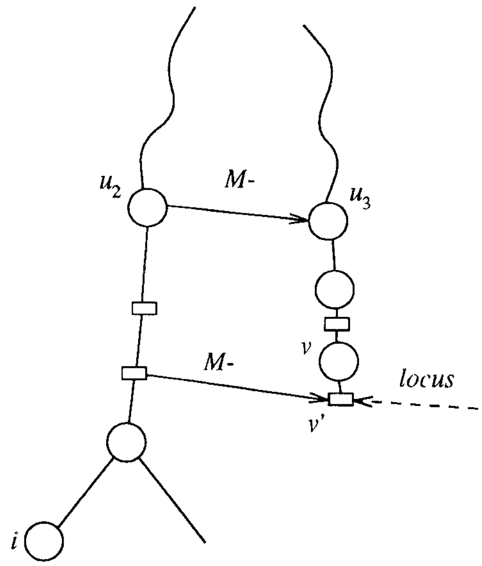

Figure 7: Locus at update time.

In the above algorithm when we update D' special care needs to be exercised. If the current loci of D and D' are on the same edge and if D' splits that edge, then the corresponding W-link of D has to be properly adjusted. We claim that a suitably large constant c can be chosen for the algorithm such that the invariants below can be maintained.

At any stage, let the stack contain the tasks D_k, D_{k-1},..., D_0, with D_k at the top, and let the number of suffixes in the interval of each task D_i be d_i. Assume that we have already performed c updates s_i times on task D_i. Let the number of input symbols received during the current phase be d_{k+1}. (This can be less than the number of input symbols received from the instant D_k was pushed on the stack.)

Invariant 1: If we complete D_0, D_1,..., D_k in order on the current suffix tree and then insert the suffixes of the current phase, the resulting suffix tree is the suffix tree for the input received so far.

Invariant 2: For each $i = 0, 1, \cdots, k$, $max\{d_i, s_i\} \geq 3d_{i+1} + d_{i+2}$, where $d_{k+2} = 0$.

Invariant 3: For each $i = 0, 1, \cdots, k-1$, $max\{d_i, s_i\} >$ current depth of D_{i+1}.

As a consequence of the second invariant and lemmas 5 and 6, we can infer that the maximum depth of insertion of any unfinished suffix in D_{i+2} is less than the minimum depth of insertion of any unfinished suffix in D_i, and the maximum depth of insertion of any suffix in the current phase is less than the minimum depth of insertion of any unfinished suffix in D_{k-1}. The third invariant implies that the current depth of D_{i+1} is less than the minumum depth of insertion of any unfinished suffix in D_i.

We prove that the above two properties remain invariant in the final version. We now present an informal justification of these invariants.

If the other invariants hold, the first invariant might fail because W-links can be implicit. Suppose current loci of several tasks are on the same edge, and suppose that one of these tasks splits that edge into two edges. We can show, based on invariants 2 and 3, that the split must happen at the shallowest locus. In such a case our indirect linking scheme maintains all the links properly.

Between the instants D_i and D_{i+1} got pushed on the stack at least d_{i+1} c updates must have been performed on D_i. During the time when D_{i+2} was the current phase an additional d_{i+2} c updates must have been performed on D_i. Hence invariant 2 holds. An analogous argument establishes invariant 3.

4 Pattern Matching Problems

We make use of the above quasi-real-time algorithm to develop real-time algorithms for some pattern matching problems.

Problem 1: Let the input be $T\#P$ where $T = t_1t_2...t_n$ is the text and $P = p_1p_2...p_m$ is the pattern. We want to test whether P^R is a substring of T.

We apply the quasi-real-time algorithm and construct the suffix tree for $X = T^R\$$. Since our quasi-real-time algorithm processes X right-to-left, the input requested is in the proper order. After receiving $\#$, we walk down the path $p_1p_2...p_m$ starting at the root. On each input symbol p_i, we update the tasks on the stack assuming that the path extension step has succeeded. (In reality, we don't perform the path extension test at all.) We can infer from invariants 2 and 3 of Algorithm Q that at every instant the minimum depth of insertion of any suffix of any task on the stack is greater than the length of the pattern received. Consequently, the cleaning up of the suffix tree can be maintained ahead of the pattern. The path for P exists in the suffix tree if and only if P is a substring of T^R, *i.e.*, if and only if P^R is a substring of T.

Problem 2: Let the input be $T\#P$, as before. We want to test whether P is a substring of T.

As before we apply the quasi-real-time algorithm for $X = T^R\$$. After receiving $\#$ we process P by applying lemma 1. After processing $p_1p_2...p_i$ we will be at the locus of $p_i...p_1$. On input symbol p_{i+1} we apply lemma 1 and reach the locus of $p_{i+1}p_i...p_1$ in $O(1)$ steps. As before, the tasks on the stack get updated while each input symbol is processed. The path $p_mp_{m-1}...p_1$ exists if and only if P^R is a substring of T^R, *i.e.*, if and only if P is a substring of T.

Acknowledgements: I acknowledge my deep gratitude to Professor Art Delcher for enthusiastically listening through my ideas and for suggesting several improvements to the presentation.

References

[CL90] W. L. Chang and E. L. Lawler. Approximate string matching in sublinear expected time. In *Proc. of 31st Annual IEEE Symp. on Foundations of Computer Science*, pages 116–124, 1990.

[CS85] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, pages 97–107. Springer-Verlag, 1985.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of ACM*, pages 262–272, 1976.

[Sei77] J. Seiferas. Subword trees, February 1977. Class Notes.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proc. of 14th Annual IEEE Symp. on Switching & Automata Theory*, pages 1–11, 1973.