A Space-Economical Suffix Tree Construction Algorithm
Edward M. McCreight (1976)

{ From Ukkonen to McCreight and Weiner: A Unifying }
View of Linear-Time Suffix Tree Construction
Overview

- Algorithm for constructing auxiliary digital search trees to help in search operations of substrings.
- Advantages over other algorithms:
  - Economical in Space.
- We describe the algorithm
- Incremental changing of the search tree corresponding to changes in the text.
Motivation

- Text editors
- Automatic command completion
Constructing a Suffix Tree Algorithm

- Given a string S, we build an index to S in the form of a search tree T, whose paths are the suffixes of S.
- Each path starting from the root of this tree represents a different suffix.
- An edge is labeled with a string.
- The concatenation of these labels on through a path gives us a suffix.
- Each leaf corresponds uniquely to positions within S.
Mapping the string \textit{ababc} into a suffix tree.
Constructing a Suffix Tree
Algorithm by McCreight:
denoted $mcc$
**Algorithm mcc**

The algorithm requires that:

**S1.** The final character of the string $S$ should not appear elsewhere in $S$.

**S1 yields:**

1. No suffix of $S$ is a prefix of a different suffix of $S$.
2. There is a leaf for each suffix of $S$. 
Algorithm *mcc*

**Constraints on the Tree**

T1. An edge of T may represent any nonempty substring of S.

T2. Each internal node of T, except the root, must have at least two outgoing edges.

T3. Siblings edges represent substrings with different starting characters.
Algorithm $mcc$

Constraints on the Tree

- Since every leaf maps uniquely to a suffix of $S$, then $T_2$ yields that the number on internal nodes in $T \leq n = |S|$ (since every branching yields another leaf).

- Proposition: The mapping of $S$ into $T$, unique, up to order among siblings.
Algorithm \textit{mcc} \\
Example

Mapping the string \textit{ababc} into a suffix tree.
Definitions

- $\Sigma$ – the alphabet
- We use $a, b, c, d$ to denote characters in $\Sigma$.
- $p, q, s, t, u, v, w, y, z$ to denote strings.
- If $t = uvw$ for some strings (possibly empty) $u, v, w$ then $u$ is a prefix of $t$, $v$ is a $t$-word, and $w$ is a suffix of $t$. 

Algorithm $mcc$
Algorithm $mcc$

Definitions

- A prefix or suffix of $t$ is *proper*, if it is different from $t$.
- By $\text{path}(k)$ we denote the concatenation of the edge labels on the path from the root of $T$ to the node $k$.
- By $T_3$ path labels are unique and we can denote $k$ by $w$, if and only if $\text{path}(k) = w$. 
Definitions

Another terminology (McCreight):

- **Definition:**
  Node $k$ is called the *locus* of the string $uv$, if the path from the root to $k$ denotes $uv$.

- hence, the locus of $uv$ is $uv$. 
Algorithm \textit{mcc}

Definitions

Example:

Path(k) = uv $\iff$ k=uv

The locus of uv
Algorithm \textit{mcc}

Definitions

- The \textbf{Extended Locus} of a string \textit{u} is the locus of the shortest extension of \textit{u}, \textit{uw} (\textit{w} is possibly empty), s.t. \textit{uw} is a node in \textit{T}.

\textit{Example}: \textit{u=hello, w=p}

![Diagram of the extended locus of \textit{u}]

The extended locus of \textit{u}
Definitions

- The **Contracted Locus** of a string $u$ is the locus of the longest prefix of $u$, $x$ ($x$ is possibly empty), s.t. $x$ is a node in $T$.

*Example:* $u=$hello, $x=$hell

![Diagram of a tree with nodes labeled root, hell, op, uw. The contracted locus of $u$ is indicated by a highlighted path from the root to the node marked hell.](image-url)
Algorithm \textit{mcc}

Definitions

- Let $S$ be our main string
- $\text{Suf}_i$ is the suffix of $S$ beginning at the $i^{th}$ position (position are counted from 1 $\rightarrow$ $\text{suf}_1 = S$).
- $\text{head}_i$ is the longest prefix of $\text{suf}_i$, which is also a prefix of $\text{suf}_j$ for some $j<i$.
- $\text{tail}_i$ is defined s.t. $\text{suf}_i = \text{head}_i \text{tail}_i$
Algorithm \textit{mcc}

\textbf{Definitions}

Example:
\[ S = ababc, \ \text{suf}_3 = abc, \ \text{head}_3 = ab, \ \text{tail}_3 = c \]
\[ \text{suf}_4 = bc, \ \text{head}_3 = b, \ \text{tail}_3 = c \]

\textbullet \textbf{Constraint S1 assures that tail}_i \textbf{ is never empty.}
Algorithm *mcc*

### Overview of *mcc*

To build the suffix tree for `ababc` *mcc* inserts every step $i$ the `suf_i` into tree $T_{i-1}$:

<table>
<thead>
<tr>
<th>Step</th>
<th>Suffix Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td><code>ababc</code></td>
</tr>
<tr>
<td>Step 2</td>
<td><code>babc</code></td>
</tr>
<tr>
<td>Step 3</td>
<td><code>abc</code></td>
</tr>
<tr>
<td>Step 4</td>
<td><code>bc</code></td>
</tr>
<tr>
<td>Step 5</td>
<td><code>c</code></td>
</tr>
</tbody>
</table>
To do this we have to insert every step $\text{suf}_i$ without duplicating its prefix in the tree, so we need to find its longest prefix in the tree.

Its longest prefix in the tree is by definition $\text{head}_i$.

Example:

$\text{Suf}_3=\text{abc}$. Since we already have the word $\text{ab}$ in the tree thus we need to start from there building our new suffix. Note that indeed $\text{ab}=\text{head}_3$, $\text{tail}_i=\text{c}$.

So what we do is finding the extended locus of $\text{head}_i$ in $T_{i-1}$ and its incoming edge is split by a new node which spawns a new edge labeled $\text{tail}_i$. 
Overview of $mcc$’s operations via example of $ababc$:

Step $i = 5$

Diagram:

- $T_5$
- Root
- $ab$
- $b$
- $c$
- $abc$
- $c$
- $c$
- $abc$
Notice that $\text{head}_i$ is the longest prefix of $\text{suf}_i$ that its extended locus exists within $T_{i-1}$.

We have entered 2 suffixes by now.
For efficiency we would represent each label of an edge by 2 numbers denoting its starting and ending position in the main string.

Algorithm *mcc*

The Data Structure

- For efficiency we would represent each label of an edge by 2 numbers denoting its starting and ending position in the main string.
Thus, the actual insertion of an edge to the tree takes $O(1)$.

The introduction of a new internal node and $tail_i$ takes $O(1)$, hence,

if $mcc$ could find the extended locus of $head_i$ in $T_{i-1}$ in constant time, in average over all steps, then $mcc$ is linear in $n$.

This is done by exploiting the following lemma:
Lemma 1: If $\text{head}_{i-1} = xu$ for some character $x$ and some string $u$ (possibly empty), then $u$ is a prefix of $\text{head}_i$.

Proof. $\text{head}_{i-1} = xu$, hence, there is a $j < i$ s.t. $xu$ is a prefix of both $\text{suf}_{j-1}$ and $\text{suf}_{i-1}$.

1. $xu$ is a prefix of $\text{suf}_{j-1} \Rightarrow u$ is a prefix of $\text{suf}_j$.
2. $xu$ is a prefix of $\text{suf}_{i-1} \Rightarrow u$ is a prefix of $\text{suf}_i$.

By (1), (2): there is some $j < i$ such that $u$ is a prefix of both $\text{suf}_j$ and $\text{suf}_i$.

Hence, by definition of head: $u$ is a prefix of $\text{head}_i$.

S: …xu…..xu….
Algorithm \textit{mcc}

The Data Structure

\begin{align*}
S &= \text{bdababdc}, \quad \text{head}_5 = \text{ab}, \quad \text{head}_6 = \text{bd} \\
S &\text{:\自在 h a l y h a l l } \\
\text{Suf}_5 &\text{:\自在 h a l l } \\
\text{Suf}_6 &\text{:\自在 h a l l } \\
\end{align*}
To exploit this we introduce **Suffix Links**:
From each internal node $xu$, where $|x| = 1$, we add a pointer to the node $u$. 

```
Algorithm $mcc$
```

```
The Data Structure
```

```
root
```

```
$u$
```

```
$xu$
```

```
Suffix link
```

```
Algorithm $mcc$
```

```
The Data Structure
```

```
root
```

```
$u$
```

```
$xu$
```

```
Suffix link
```
Algorithm \textit{mcc}

The Data Structure

Our example:
Algorithm \textit{mcc}

The Data Structure

\textbf{Note:} All suffix links are \textit{atomic} in the sense that \textit{ xu } is suffix linked to \textit{ u } where \(| x | = 1 \).
Algorithm \textit{mcc}

We shall present \textit{mcc} and prove by induction on \(i\), the step number of \textit{mcc}, that

\textbf{P1:} in \(T_i\) every internal node, except perhaps the locus of \textit{head}_{i} (\textit{head}_{i}), has a valid suffix link.

\textbf{P2:} in step \(i\) \textit{mcc} visits the contracted locus of \textit{head}_{i} in \(T_{i-1}\).

P2 yields that we can use the contracted locus of \textit{head}_{i-1} to jump with the suffix link to some prefix of \textit{head}_{i}. P1 assure us that there is such suffix link.
Algorithm $mcc$

**Base case for P1**

$i=1$: $T_1$

P1 holds since there is no internal nodes. (note that $head_1 = \varepsilon$ ($\varepsilon$=root)).
Algorithm $mcc$

**Base case for P2**

$i=1$:

$T_0 \rightarrow \text{root} \rightarrow T_1 \rightarrow \text{root} \rightarrow \text{ababc}$

$P2$ holds since $\text{head}_1=\epsilon$ and in step 1 $mcc$ visits the root which is the locus of $\epsilon$ ($\epsilon=\text{root}$) in $T_0$. 
In this substep \textit{mcc} will identify strings it had already dealt with in the previous steps, in order to make a shortcut leap to the ‘middle’ of its current \textit{head}. 
Identify 3 strings: \( xuw \) s.t.

1. \( \text{head}_{i-1} = xuw \)

2. \( xu \) is the contracted locus of \( \text{head}_{i-1} \) in \( T_{i-2} \), i.e. \( xu \) is a node in \( T_{i-2} \). If the contracted locus of \( \text{head}_{i-1} \) in \( T_{i-2} \) is the root then \( u = \varepsilon \).

3. \( |x| \leq 1. \) \( x = \varepsilon \) only if \( \text{head}_{i-1} = \varepsilon \).
Illustrating substep A in the $i^{th}$ step: the move from $T_{i-1}$ to $T_i$
Algorithm *mcc*

*mcc* – substep A

Our goal here is to go directly to the locus of *u* in the tree so that we could search for *w* (substep B) and then for *v* (substep C).
Algorithm \textit{mcc}

\textbf{mcc – substep A}

Notice that:

- In the previous step $\text{head}_{i-1}$ was found.
- Since $|x| \leq 1$ then by lemma 1: $\text{head}_i = uwv$ for some, yet to be discovered, string (possibly empty) $v$.
- By induction hyp \textbf{P2}, \textit{mcc} visited $xu$ in the previous step (i-1), hence it can identify $xu$. 

\begin{algorithm}
\begin{itemize}
\item In the previous step $\text{head}_{i-1}$ was found.
\item Since $|x| \leq 1$ then by lemma 1: $\text{head}_i = uwv$ for some, yet to be discovered, string (possibly empty) $v$.
\item By induction hyp \textbf{P2}, \textit{mcc} visited $xu$ in the previous step (i-1), hence it can identify $xu$.
\end{itemize}
\end{algorithm}
Algorithm *mcc*

**mcc – substep A**

If $u = \varepsilon$ then $c \leftarrow \text{root}$

(note that root = $u$)

else, $c \leftarrow \{\text{suffix link of } xu\}$ (note that $c = u$)

explanation:

- $u \neq \varepsilon$ thus by definition $xu$ existed (as the contracted locus of $xuw$) in $T_{i-2}$ hence by P1: the internal node $xu$ has a suffix link.

- By P2 we remember $xu$ from step $i-1$ and we can now follow its suffix link.
Algorithm \textit{mcc}

\textbf{mcc – substep A}

- \(uwv = \text{head}_i\) hence from the definition of head, the extended locus of \(uw\) exists in \(T_{i-1}\).
- Now we can start going down the edges, from \(u\) to find the extended locus of \(uw\).
Algorithm $mcc$

**Mcc – substep B: Rescanning**

To rescan $w$:
- Find the edge that starts with the first character of $w$. Denote the edge’s label $z$ and the node it leads to $f$.
- If $|w| > |z|$ then start a recursive rescan of $w-z$ (or $w_{|z|}$) from $f$.
- If $|w| \leq |z|$ , then $w$ is a prefix of $z$ , and we found the extended locus of $uw$.
- Construct a new node (if needed): $uw$.
- $d \leftarrow uw$
Algorithm $mcc$: substep B - rescanning

Illustrating substep B in the $i^{th}$ step: rescanning substring $w$.

In this case $uw$ already exists.

$T_{i-1}$

Head$_{i-1}$ = $xuw$

Step i

$T_i$

Head$_i$ = $uwv$
Algorithm \textit{mcc}

\textbf{mcc – substep B: Scanning}

Make the suffix link of \textit{xuw} point to \textit{d}.

Hence, we have defined a suffix link to the node constructed in step \(i-1\). By this and induction hyp \(\Rightarrow P1\) holds in \(T_i\).

\[\text{Head}_{i} = \text{uwv}\]
Scan the edges from $d$ in order to find the extended locus of $uwv$.

Since we don’t know yet what is $v$ we must scan each character in the path from $d$ downward, comparing it to $tail_{i-1}$.

When we ‘fall out of the tree’ we have found $v$.

The last node in this trek is the contracted locus of $head_i$ in $T_{i-1}$, which proves P2.

When we reach the extended locus of $uwv$ we construct the new node $uwv$, if needed.

Construct the new leaf edge $tail_i$. 

**Algorithm mcc**

**mcc – substep C - Scanning**
Algorithm \textit{mcc}

\textbf{mcc – substep C: Scanning}

Scanning for the requested \(v\).

Comparing each character of the downward path beginning at \(d\) to \(tail_i\). When the comparison fails we have reached \(head_i\).
Algorithm \textit{mcc}

\textbf{Maintaining T2}

We shall prove that when we add a new node in the end of substep B as the locus of uw then we obey constraint $T_2$ that an internal node has at least 2 son edges.
Lemma: In step i, at the end of substep B we add a new node only if \( v \) is empty.

Proof. In step \( i \), If \( v \) is not empty then \( \text{head}_i = uwv \) and \( \text{head}_{i-1} = xuw \) hence, w.l.g. we can write \( S \) as follows:

\[ S = \ldots\ldots xuwz\ldots\ldots uwv\ldots\ldots xuwv\ldots \]

Thus, we have 2 occurrences of \( uw \) with different extensions, \( uwv, uwz \), that occur already in the tree. Hence, there is a branching node \( uw \).
Algorithm \textit{mcc}

Maintaining T2

\textbf{Corollary}: In the \(i\text{th}\) step if \(v\) is empty then we add an outgoing edge from the locus of \(uv=\text{head}_i\). Thus the only case where we add a node we add an outgoing edge to it.
**Algorithm \textit{mcc}**

**Time Complexity Analysis**

\textbf{Define}: \( \text{res}_i = \text{wv}\{\text{tail}_i\} \) in step \( i \).

Hence, \( \text{res}_i \) is the suffix of \( S \) rescanned and scanned during step \( i \).

\textbf{Observation}: For every intermediate node \( f \) encountered in the rescan phase of step \( i \), the substring \( z \), labeling the edge to \( f \), is contained in \( \text{res}_i \) but not in \( \text{res}_{i+1} \).
Time Complexity Analysis

Illustrating substep B in the \(i^{th}\) step: rescanning substring \(w\).

\[\text{Head}_i = uwv\]
Algorithm $mcc$

**Time Complexity Analysis**

- **Explanation**: if we encounter node $f$ in step $i$ during the rescan phase of substep B then $f$ must be an internal node in $T_{i-1}$ hence P1 yields that in $T_{i+1}$, $f$ has a suffix link.

- Assume w.l.g that $f = az$

- This suffix link serves us in substep A of step $i+1$ to reach the node $z$, hence we do not have to rescan substring $z$ again.
Algorithm $mcc$

Time Complexity Analysis

Example:

Suffix link
**Algorithm mcc**

**Time Complexity Analysis**

- **Define**: $\text{int}_i = \text{number of intermediate nodes (f) rescanned during step } i$.  
- The observation yields:  
  $$|\text{res}_{i+1}| \leq |\text{res}_i| - \text{int}_i$$  
- Hence,  
  $$1 = |\text{res}_n| \leq |\text{res}_{n-1}| - \text{int}_{n-1} \leq \ldots \leq |\text{res}_1| - \sum_{i=1}^{n-1} \text{int}_i \quad (\text{since } \text{int}_n=0) \Rightarrow$$  
  $$1 \leq |\text{res}_1| - \sum_{i=1}^{n} \text{int}_i = n - \sum_{i=1}^{n} \text{int}_i \Rightarrow$$  
  $$\sum_{i=1}^{n} \text{int}_i \leq n - 1$$  
- i.e., the total number of intermediate nodes rescanned $\leq n$.  

The total number of characters scanned in substep C to locate $head_i$ (the length of $v$):

- In step $i$ the number of characters scanned during step C is $|(head_i)| - |(head_{i-1})| + 1$ since we already rescanned $w$ (the suffix of $head_{i-1}$) in substep B. +1 comes from the first character of $head_{i-1}$.

- The number of characters scanned is:
  $$\sum_{i=1}^{n} [|{(head_{i})}|-|{(head_{i-1})}|+1] = |{(head_{n})}|-|{(head_{0})}|+n = n$$

- Therefore, the total time complexity is $O(n+n)=O(n)$
Updating the suffix tree

We shall see how to update the suffix tree (not online), when a substring of the main string is being replaced by another.
Updating the suffix tree

Goal:

- Given a string $S = uvw$, and its corresponding suffix tree, we change $S$, so that: $S = uzv$.
- We wish to update the suffix tree to represent the change in $S$. 

Updating the suffix tree

In order to make it possible to update the tree effectively, i.e., not change the whole tree, we would adopt a numbering scheme representing the positions of $S$, in which a position number need never change after it has been assigned.

Also, the position numbers are strictly monotonic.

Hence, the suffixes of $v$, for instance, need not to be changed, when we change $uvw$ to $uzv$.

This requires a large pool of position numbers.
We consider what kind of paths might need to change, by the change: $uwv \rightarrow uzv$.

Denote $u^*$ as the longest suffix of $u$ that appears elsewhere in $uwv$.

Definition: a $w$-splitters (w.r.t $uwv \rightarrow uzv$) are the strings of the form $tv$, where $t$ is a nonempty suffix of $u^*w$.

Equivalently, splitters are the paths which properly contain $v$ and whose last edge do not contain $wv$. 
Paths in need to be changed

Illustrating the paths not affected by the change:

(1) \[ \text{root} \]

\textbf{Suffix} = \( v \):
\begin{itemize}
  \item \( v \)
\end{itemize}

Stays as it is.

(2) \[ \text{root} \]

\textbf{Suffix} = \( u'u*vw \):
\begin{itemize}
  \item \( yx = u'u* \), for some \( y, x \geq 0 \) and \( |xy| > 0 \), s.t. the branching occurs at the end of \( y \). Stays as it is.
\end{itemize}

\textbf{Note} that due to our numbering scheme and data structure the label \( xwv \) ‘changes’ automatically to \( xzv \).
Paths in need to be changed

Illustrating paths that are affected by the change:

$\text{Suffix} = u'wv, u' = \text{suf}(u^*)$:

(1) $x\ y = w$:

need a change, since the positions indices of the leaf and its father change.

(2) Need to update the last edge label, since the position index in the leaf changes.
Overview of the algorithm

- The updating algorithm removes all w-splitters paths and inserts all z-splitters paths,
- while preserving properties T1, T2, T3.
Overview of the algorithm

3 stages of the algorithm, \textit{umcc}:

1. Discover $u^*wv$, the longest $w$-splitter.
2. Delete all paths $tv$, $t=\text{suf}(u^*w)$, from the tree.
3. Insert all paths $sv$, $s=\text{suf}(u^*z)$, into the tree.
Stage 1

Phase 1:

- Denote $u^{(i)}$ the suffix of $u$ of length $i$.
- Examine the paths $u^{(1)}wv, u^{(2)}wv, u^{(4)}wv, u^{(8)}wv, \ldots$ until a non $w$-splitter is discovered, say, $u^{(k)}wv$.
- Every path $u^{(i)}wv$ examined takes $O(i)$ time.
Stage 1

Phase 2:
- Examine the paths $u^{(k)}wv, u^{(k-1)}wv, \ldots$ until the longest $w$-$splitter$ is discovered, $u^*wv$.

Time complexity:
- This search can take full advantage of the suffix links, as in $mcc$, since $k$ is incremented by 1, each step, hence it takes $O(k)$ time.
- $|u^*| > k/2$
- Phase 1 takes $O(1+2+4+\ldots+k)$
- Phase 2 takes $O(k)$
- Hence, stage 1 takes $O(u^*)$
Stage 2

- Delete all paths tv, t=suf(u*w), |t|≠0, from the tree.
- The deletion is done in order, from the longest to the shortest.
- Suppose that for all suffixes s of u*w longer than t, the deletion of sv has been already done.
- We now consider how to delete tv.
Stage 2

The general case is illustrated
- delete the edge labeled \( q \) and its leaf.
- If node \( f \) has more than 2 sons than this is enough.
- Otherwise, delete node \( f \), and make \( k \) the son of \( p \); label turn to \( yo \).
- Potential problem: an existing suffix link to \( f \).
Denote the last internal node in the path \( xu^*zv \) by \( s^* \) where \( |x| = 1 \).

We show that this problem could arise only for a unique node, \( s^* \).

**Lemma 2:**

1. Whenever a node \( f \) is deleted there is no suffix link pointing to it, except perhaps that of node \( s^* \).

2. Every path in \( T \) has a suffix path, except perhaps \( xu^*zv \).
Stage 2

proof:

Base: (1) is trivially true. (2) is true, since we haven’t change the tree, so the only path without a suffix path is the path whose suffix path is the longest \( w \)-splitter, \( xu^{*}zv \).

Induction:

(1)

assume \( m \) is a node having its suffix link point to \( f \), than \( m \) could not have an outgoing edge labeled \( q \), since \( arlq \) would be a longer splitter than \( rlp \) so it would have been already deleted.
Stage 2

- Thus, node $f$ has only one son edge that has a prefix path in $T$. Hence, node $m$ has exactly 2 son edges (otherwise there would be more than 1 paths in $T$ without a suffix path, in contrast to induction hyp), and the path having no suffix path must pass through $m$, so node $m$ is actually $s^*$.  

- (2)  
  - If we delete $u^*wv$ then since we have already changed $xu^*wv$ to $xu^*zv$, hence its prefix path doesn’t exist anyway.  
  - If we delete a proper suffix of $u^*wv$ then, we have already deleted its prefix path in $T$.  

- In both cases we haven’t prevented any path in $T$ of a suffix path.
Stage 3

Insert all paths \( sv, s = \text{suf}(u^*z) \), into the tree.

- We do it as if we are running \( \text{mcc} \) with a pre-initialized suffix tree that already contains all suffixes of \( v \). Denote that tree \( T(v) \), and this variant algorithm as \( \text{umcc}(v) \).

- We already have all the suffixes longer than \( u^*zv \), so we start running \( \text{mcc} \) from there:

  - Denote \( j = |(u)| - |(u^*)| + 1 \)
  - Denote \( k = |(uz)| \)
  - We will insert the paths \( u^*zv, \ldots, dv \) (where \( d \) is the last character of \( uz \)), by running \( \text{mcc} \) from step \( j \) through \( k + 1 \) ’s rescanning substep (in order to connect the a suffix link to \( \text{head}_k \)).
Stage 3

- We remember node $s^*$ and its father (this settles the problem of the suffix link of $s^*$).
- The following 2 observations, corresponding to $P1$, $P2$, enable us to start running $mcc$ from the $j^{th}$ step, with $T(v)$:
  1. $s^*$ or its father are the contracted locuses of $head(v)_{j-1}$ in $T(v)_{j-1}$.
  2. $s^*$ is the only internal node that might not have a suffix link in $T(v)_{j-1}$.
Time Complexity Analysis

- We saw that finding $u^*$ takes $O(|u^*|)$.
- Deleting all the paths of the form $tv$, where $t$ is a nonempty suffix of $u^*w$, requires finding the leaf edge of each path and deleting its leaf. Deleting the leaf is constant.
- Finding the leaf edges of all these paths can be done in a similar manner of $mcc(v)$:
  - Find the path $u^*wv$; remember its last internal node; follow its suffix link to find the last internal node of $suf_i(u^*wv)$.
- Hence, deleting the paths takes $O(|u^*wv|)$.
Time Complexity Analysis

Running *mcc* from step $j$ through step $k+1$:

- Everything but scanning and rescanning takes constant time.
- Denote the last character of $u^*w$ by $d$.
- Define $v^*$ as the longest prefix of $dv$ that occurs elsewhere in $uzv$. 
Time Complexity Analysis

During rescanning (substep B) we encounter:

\[ \sum_{i=j}^{k+1} \int(v)_i \leq |(\text{res}(v)_j)| - |(\text{res}(v)_{k+1})| + \int(v)_{k+1} \]

- \[ |(\text{res}(v)_j)| \leq |(\text{suf}_j)| = |(u^*zv)| \]
- For all \( i \): \( \int(v)_i \leq |(w)| \leq |(\text{head}_{i-1}(v))| \), where \( w \) is the substring rescanned in substep B.
- Hence, \( \int(v)_{k+1} \leq |(\text{head}_k(v))| \).
- For all \( i \): \( |(\text{res}(v)_i)| \geq |(\text{suf}(v)_{i-1})| - |(\text{head}_{i-1}(v))| \).
- Hence, \( |(\text{res}(v)_{k+1})| \geq |(\text{suf}(v)_k)| - |(\text{head}_k(v))| \).
Time Complexity Analysis

Thus, \( \sum_{i=j}^{k+1} v_i \leq |u*zhv| + |(head_k(v))| - |(suf(v)_k)| + |(head_k(v))| \)

= \( |u*zhv| + 2 |v*| - |dv| \)

Hence, rescanning takes \( O(|u*zhv| + |v*|) \).

Scanning (substep C):

As in mcc analysis:

- The number of character scanned in steps \( j \) through \( k \) is exactly \( (k-j+1) + |head(v)_k| - |head(v)_{j-1}| \)
- \( |head(v)_k| = |v*| , |head(v)_{j-1}| = 0 \), hence
- Scanning takes \( O(|u*z| + |v*|) \)
Algorithm $mcc$

Time Complexity Analysis

In total, updating the suffix tree takes

$O(|u^*| + |w| + |z| + |v^*|)$