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
R. Giegerich and S. Kurtz (1997)

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

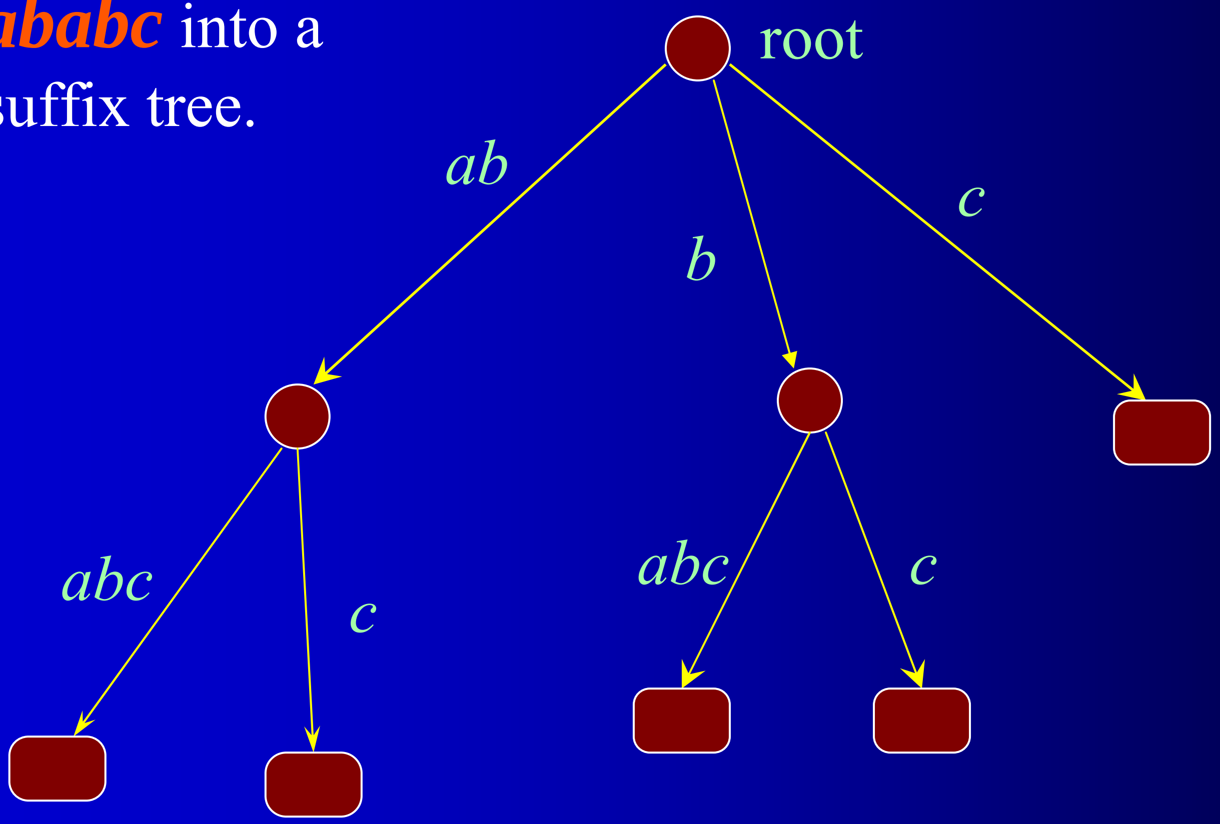
- Text editors
- Automatic command completion

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 correspond uniquely to positions within S.

Example

Mapping the string ***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.

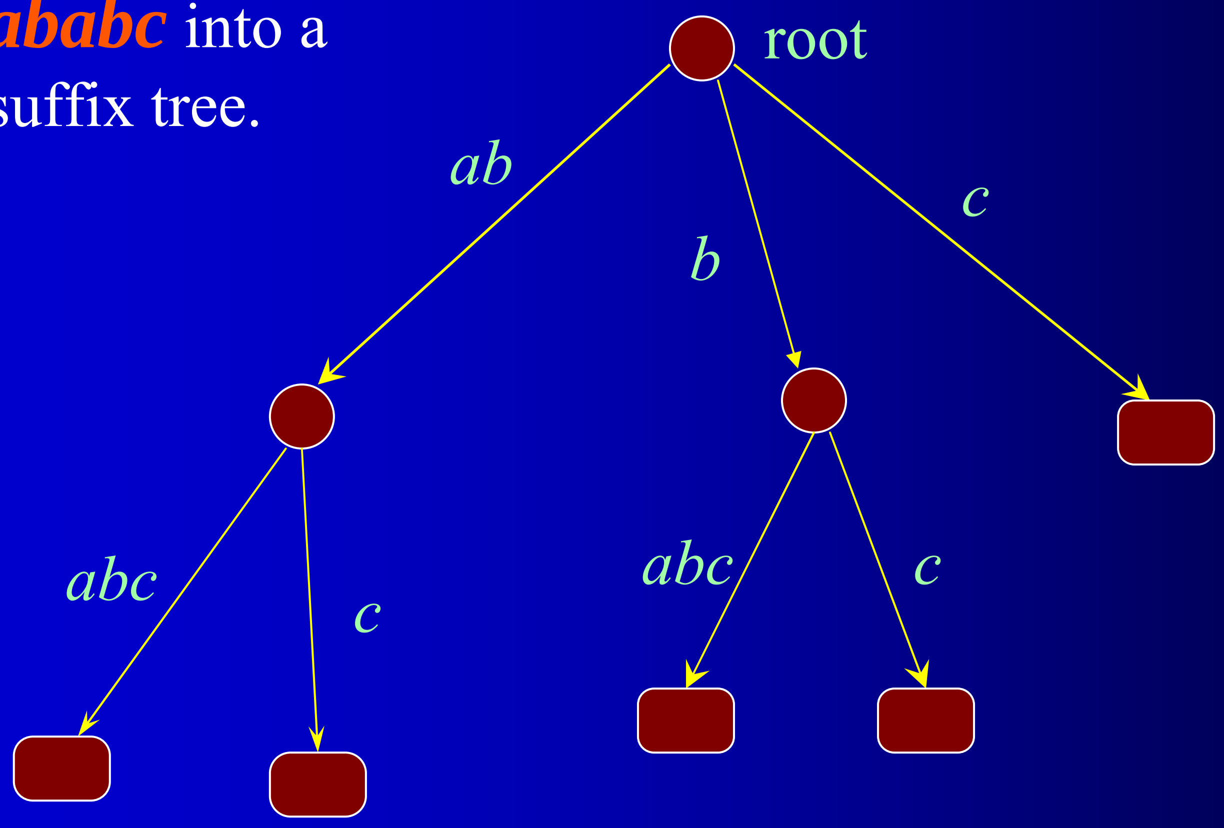
T3. Siblings edges represent substrings with different starting characters.

Algorithm *mcc*
Constraints on the Tree

- Since every leaf maps uniquely to a suffix of S, then T2 yields that the number on internal nodes in $T \leq n=|S|$ (since every branching yields another leaf).
- *Proposition: T*he mapping of S into T, unique, up to order among siblings.

Algorithm *mcc*
Example

Mapping the string ***ababc*** into a suffix tree.



Algorithm *mcc*

Definitions

- Σ – the alphabet
- We use a,b,c,d to denote characters in Σ.
- $p, q, s, t, u, v, w, y, z$ 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*

Definitions

- A prefix or suffix of t is *proper*, if it is different from t.
- By *path(k)* we denote the concatenation of the edge labels on the path from the root of T to the node k.
- By T3 path labels are unique and we can denote k by $\underline{w}$, if and only if $path(k) = w$.

Algorithm *mcc*

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.

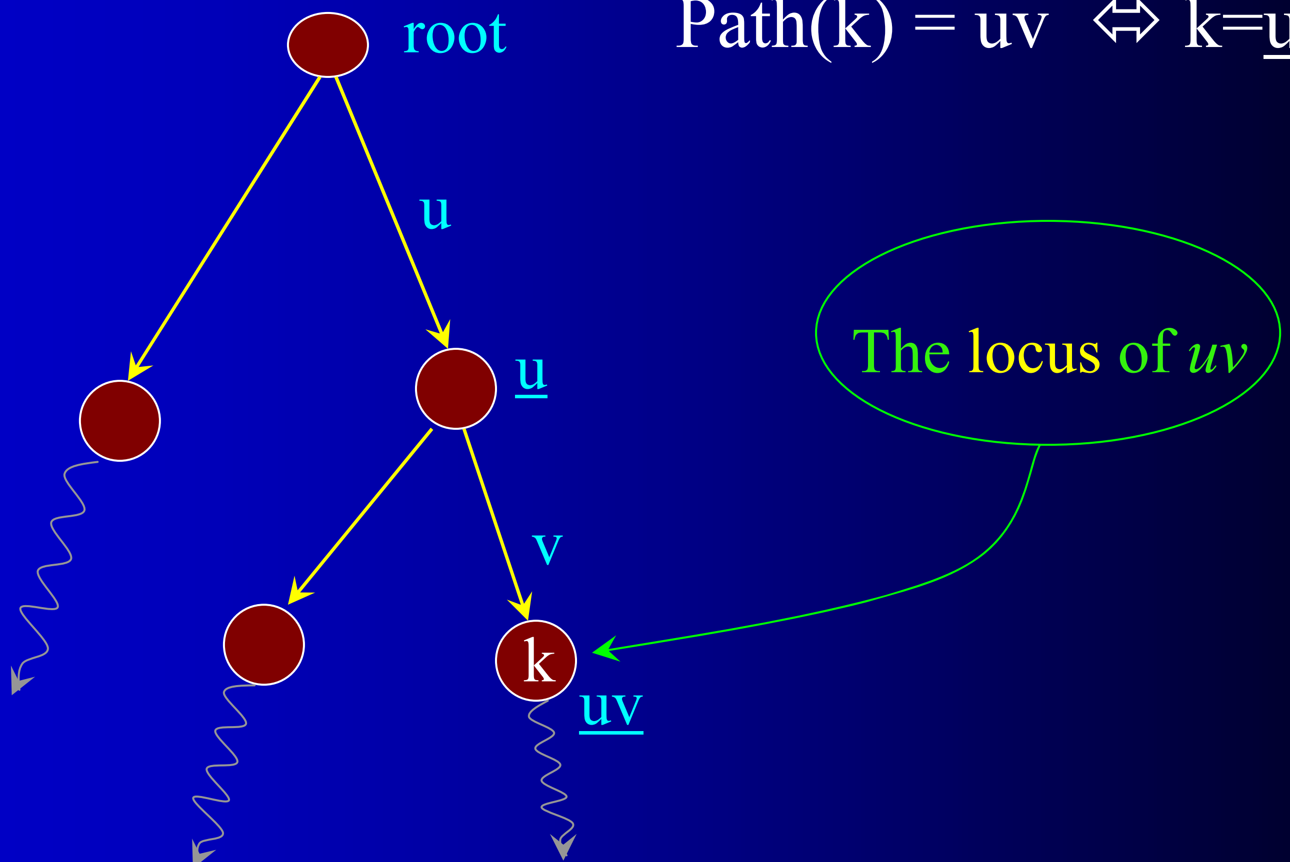
- hence, the locus of uv is $\underline{uv}$.

Algorithm *mcc*

Definitions

Example:

Path(k) = uv ⇔ k=uv

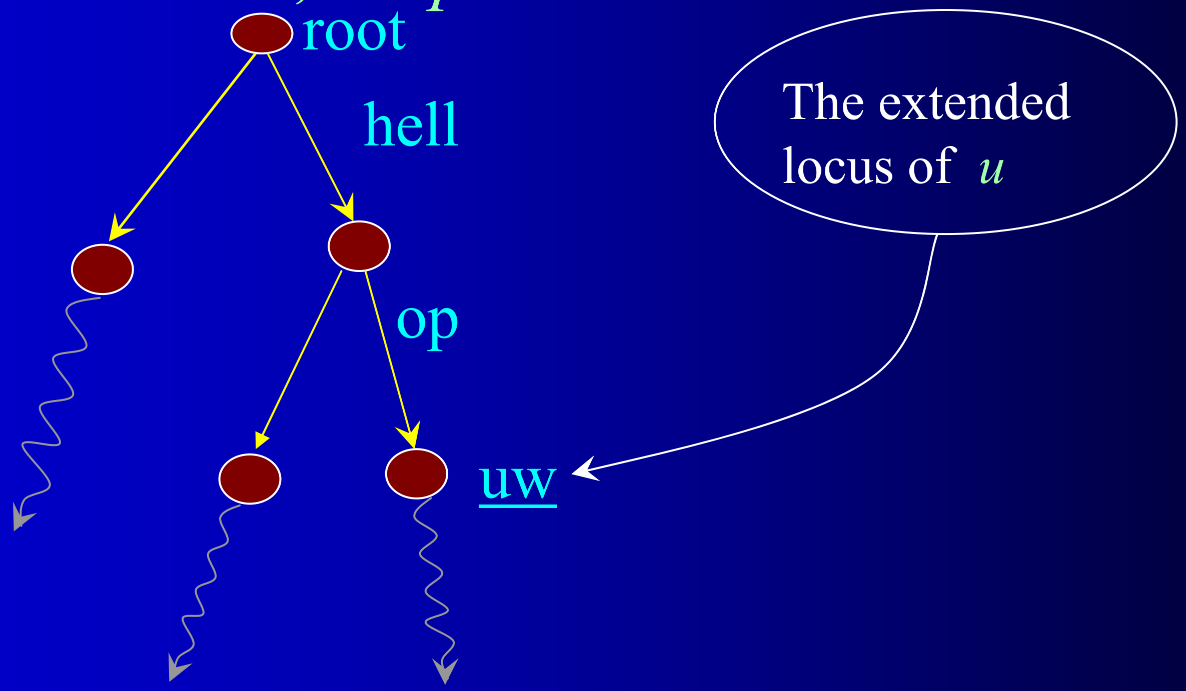


Algorithm *mcc*

Definitions

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

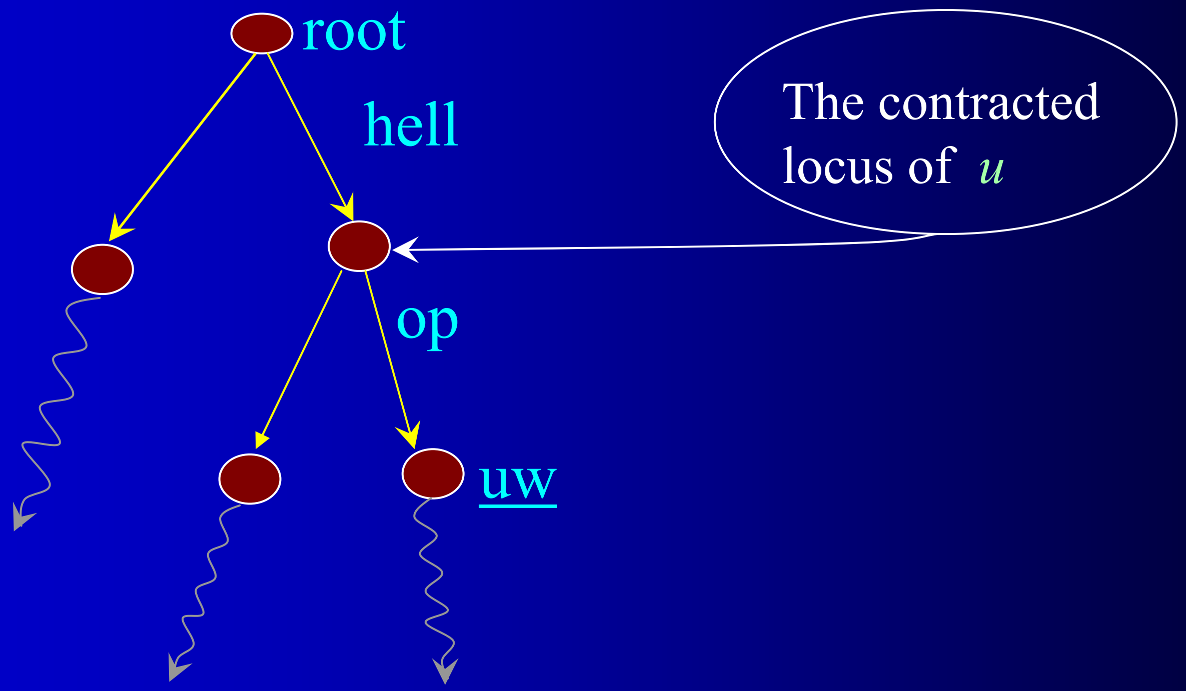
Example: u=hello, w=p



Definitions

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

Example: u=hello, x=hell



Algorithm *mcc*

Definitions

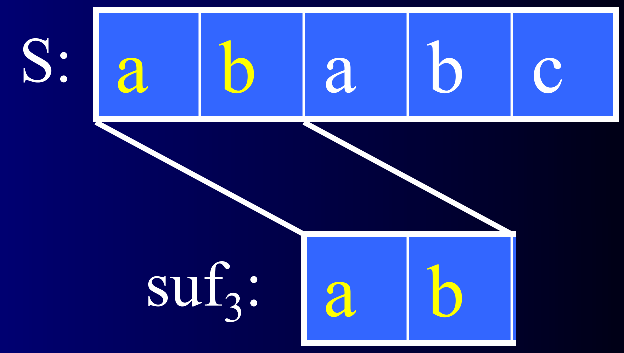
- Let S be our main string
- **Suf_i** is the suffix of S beginning at the i^{th} position (position are counted from 1 → $suf_1 = S$).
- **$head_i$** is the longest prefix of suf_i , which is also a prefix of suf_j for some $j<i$.
- **$tail_i$** is defined s.t. $suf_i = head_i\ tail_i$

Algorithm *mcc*

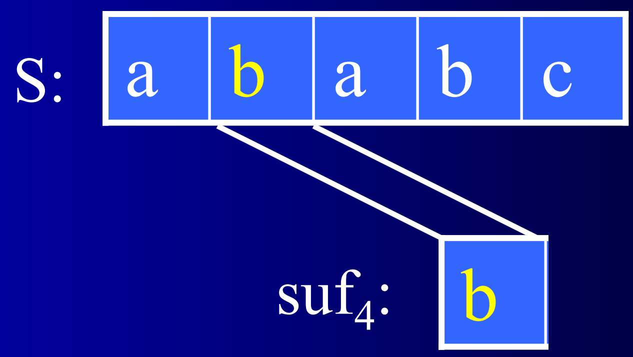
Definitions

Example:

$S=ababc$, $suf_3=abc$, $head_3=ab$, $tail_3=c$



$suf_4=bc$, $head_3=b$, $tail_3=c$



- Constraint S1 assures that $tail_i$ is never empty.

Algorithm *mcc*

Overview of *mcc*

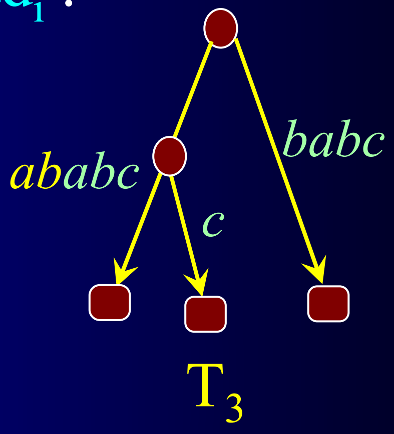
To build the suffix tree for ababc *mcc* inserts every step i the suf_i into tree T_{i-1}:

Step 1	a b a b c
Step 2	b a b c
Step 3	a b c
Step 4	b c
Step 5	c

Algorithm *mcc*

Overview of *mcc*

- To do this we have to insert every step 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 $head_i$.
- Example:

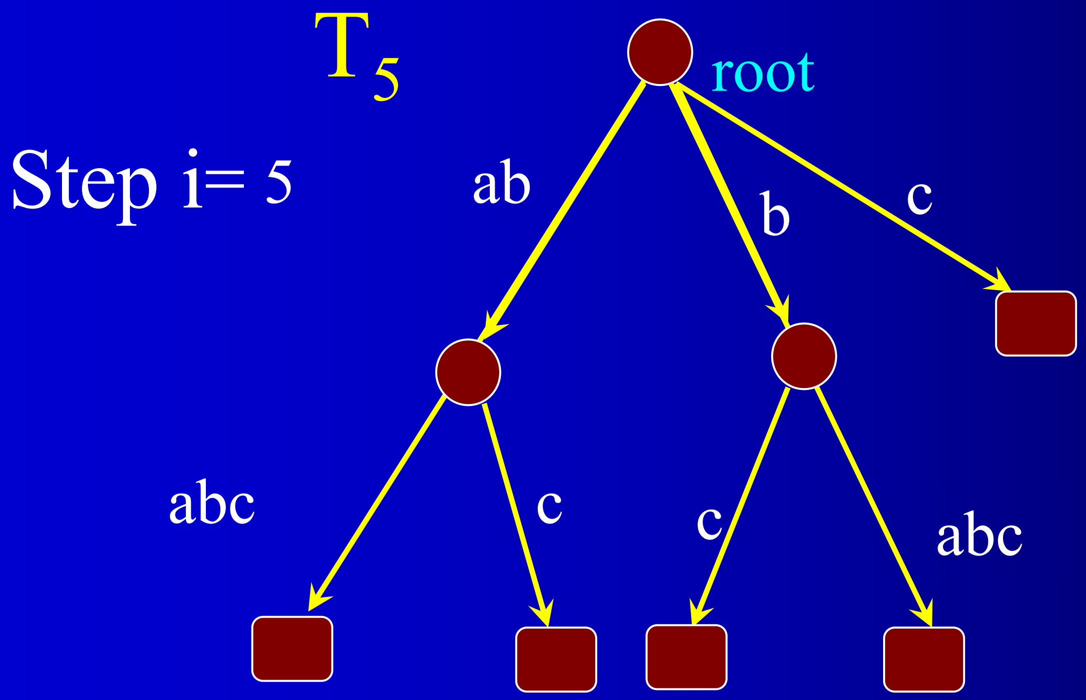


Suf_3=abc. Since we already have the word ab in the tree thus we need to start from there bulding our new suffix. Note that indeed ab=$head_3$, $tail_i$=c.

- So what we do is finding the extended locus of $head_i$ in T_{i-1} and its incoming edge is split by a new node which spawns a new edge labeled $tail_i$.

Overview of *mcc*

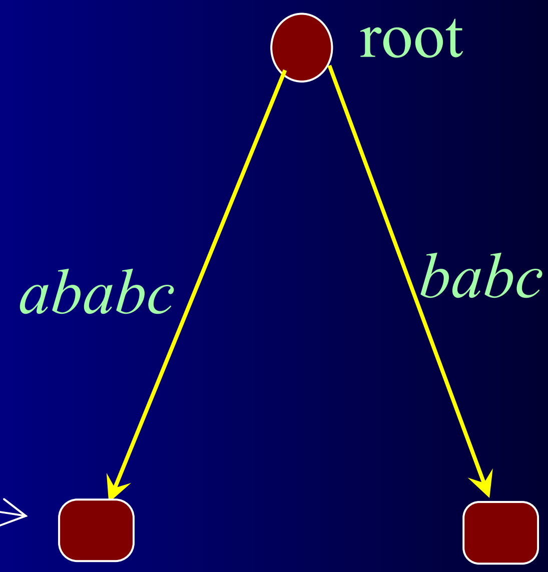
Overview of *mcc's* operations via example of *ababc*:



Algorithm *mcc*

Overview of *mcc*

Notice that head_i is the longest prefix of suf_i that its extended locus exists within T_{i-1}.



T_2

The extended locus of head_3

We have entered 2 suffixes by now.

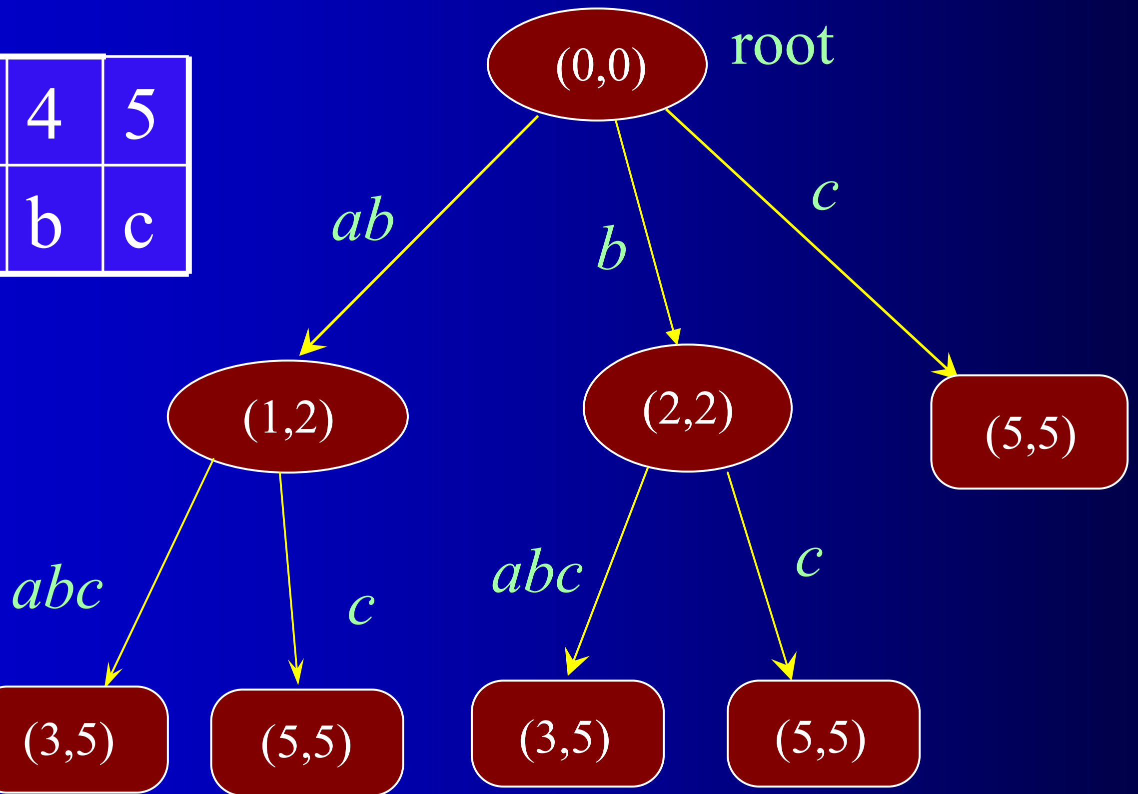
Algorithm *mcc*

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.

S:

1	2	3	4	5
a	b	a	b	c



Algorithm *mcc*

The Data Structure

- 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:

Algorithm *mcc*

The Data Structure

Lemma 1: If $head_{i-1} = xu$ for some character x and some string u (possibly empty), then u is a prefix of $head_i$.

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

1. xu is a prefix of suf_{j-1} → u is a prefix of suf_j .
2. xu is a prefix of suf_{i-1} → u is a prefix of suf_i .

By (1), (2): there is some j<i such that u is a prefix of both suf_j and suf_i .

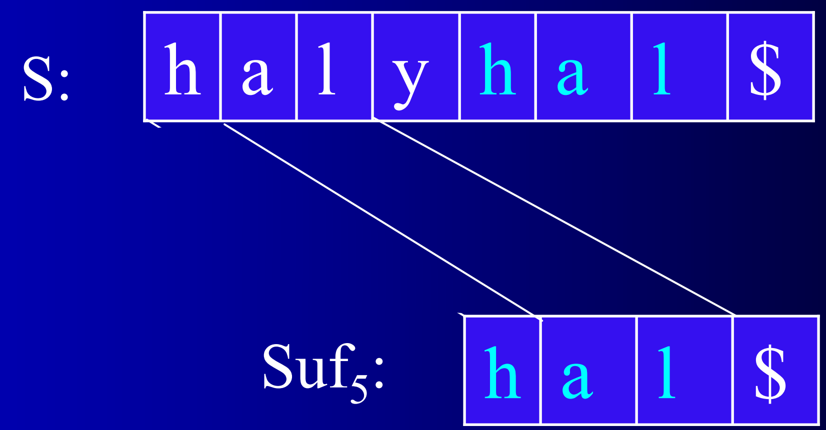
Hence, by definition of head: u is a prefix of $head_i$.

S: …xu…..xu…

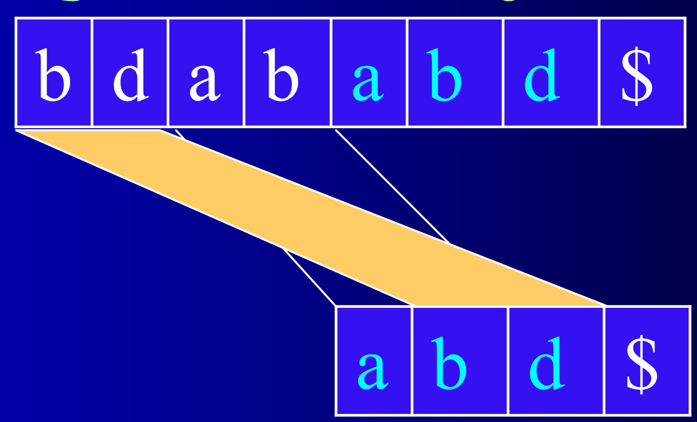
j-1 i-1

Algorithm *mcc*

The Data Structure



S=bdababdc, head$_5$=ab, head$_6$=bd

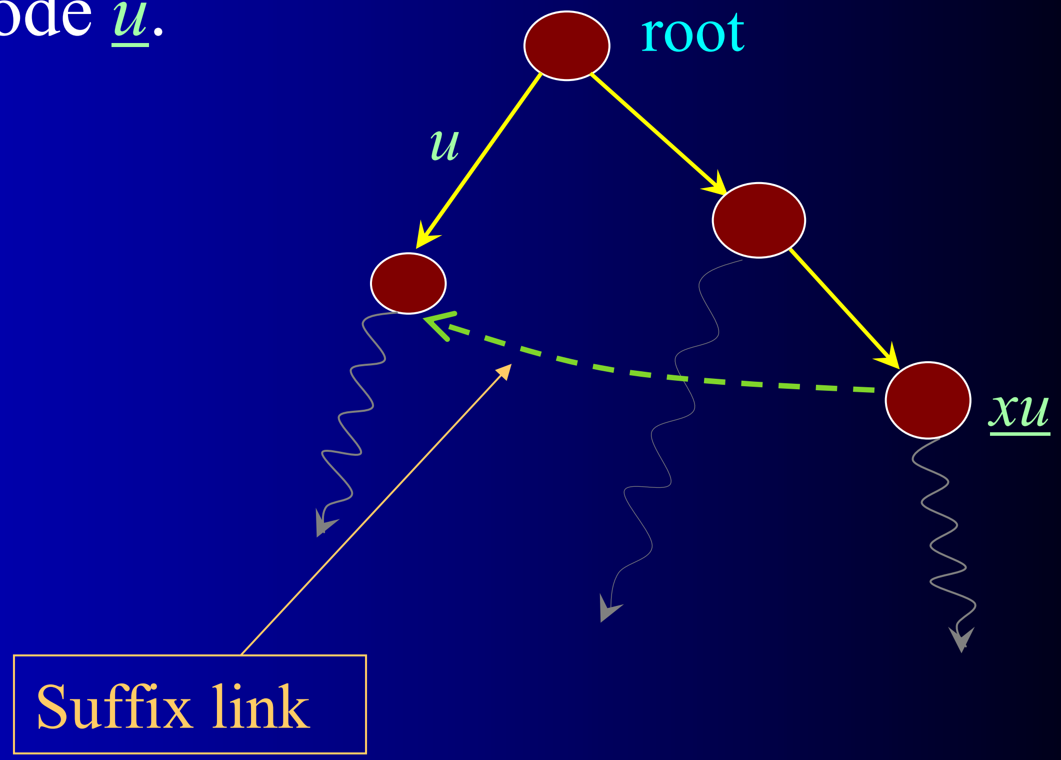


Algorithm *mcc*

The Data Structure

To exploit this we introduce Suffix Links:

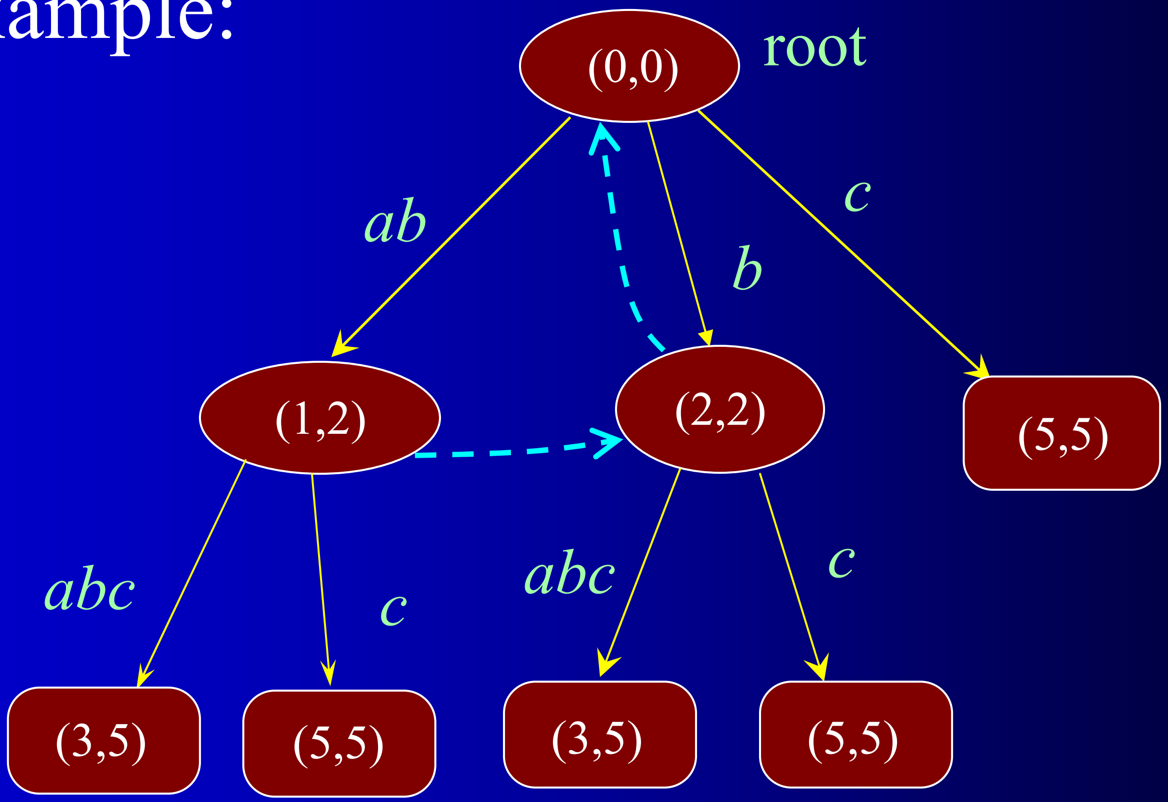
From each internal node xu , where $|x|=1$, we add a pointer to the node $\underline{u}$.



Algorithm *mcc*

The Data Structure

Our example:



Algorithm *mcc*

The Data Structure

Note: **All suffix links are *atomic* in the sense that *xu* is suffix linked to *u* where $|x| = 1$.**

Algorithm *mcc*

We shall present *mcc* and prove by induction on i, the step number of *mcc,* that

P1: in T_i every internal node, except perhaps the locus of $head_i$ ($\underline{head}_i$), has a valid suffix link.

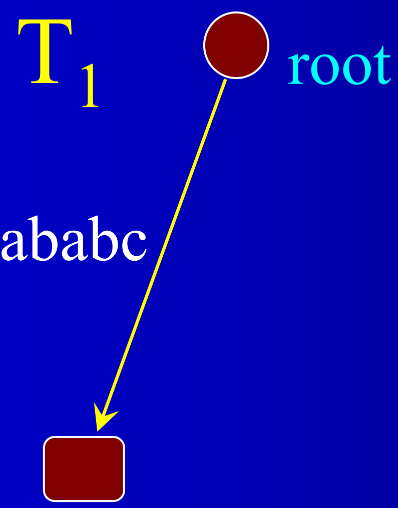
P2: in step i *mcc* visits the contracted locus of $head_i$ in T_{i-1} .

P2 yields that we can use the contracted locus of $head_{i-1}$ to jump with the suffix link to some prefix of $head_i$. P1 assure us that there is such suffix link.

Algorithm *mcc*

Base case for P1

i=1:



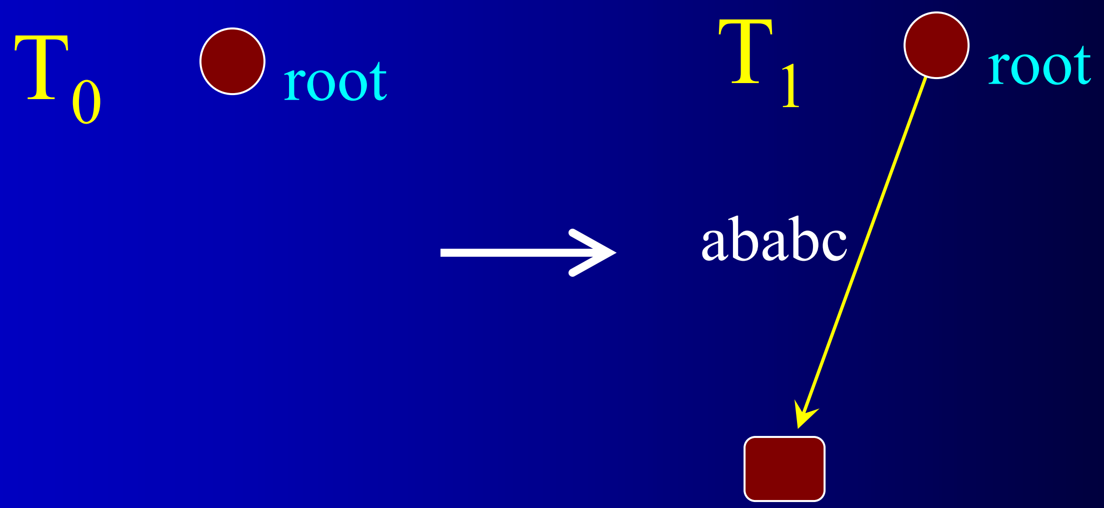
P1 holds since there is no internal nodes.

- (note that $head_1 = \varepsilon$ ($\underline{\varepsilon}$=root)).

Algorithm *mcc*

Base case for P2

i=1:



P2 holds since $head_1=\varepsilon$ and in step 1 mcc visits the root which is the locus of ε ($\underline{\varepsilon}$=root) in T_0 .

mcc – substep A

In this substep *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 *head*.

Algorithm *mcc*

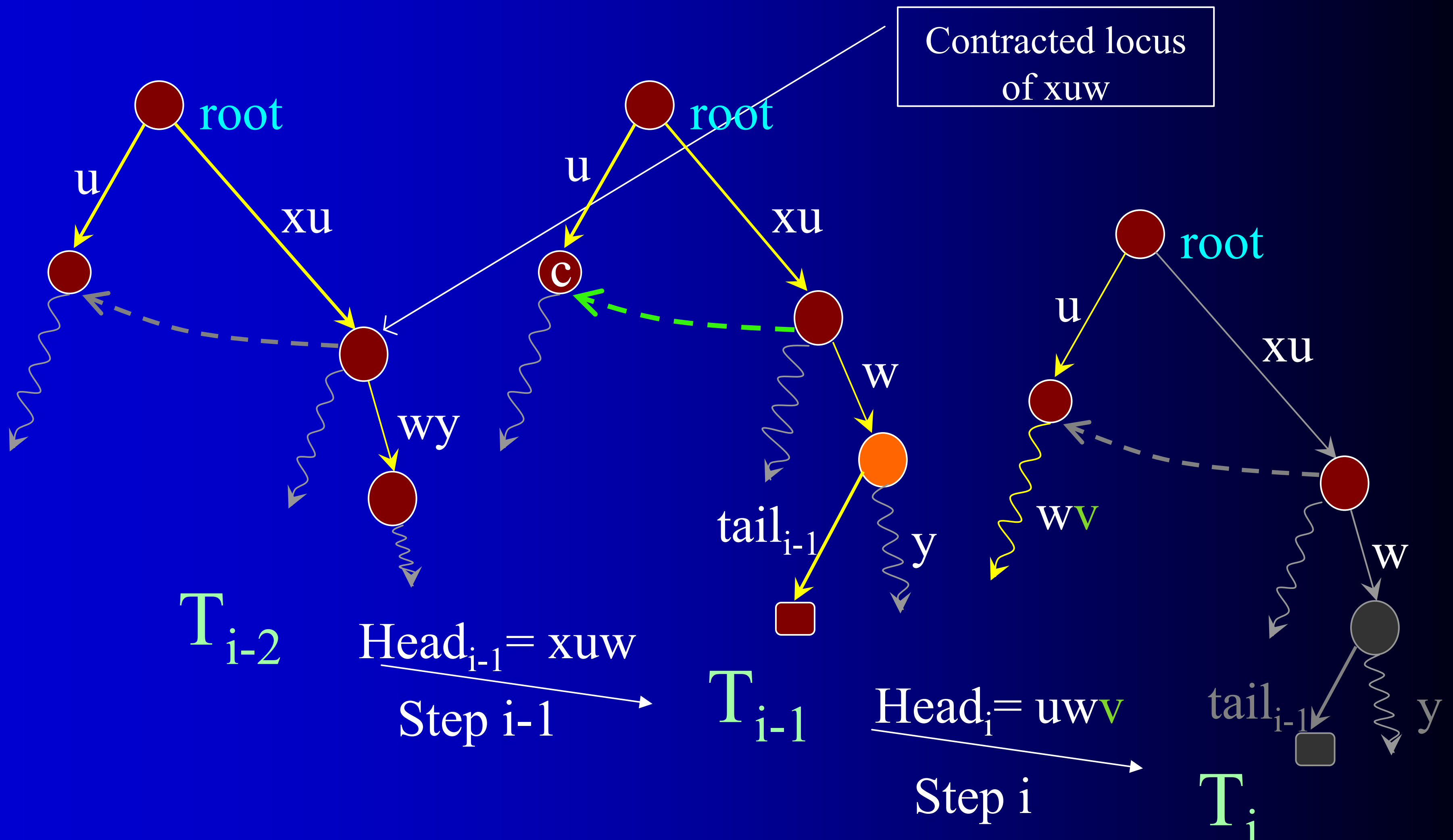
mcc – substep A

Identify 3 strings: *xuw s.t.*

1. $\text{head}_{i-1} = xuw$
2. *xu* is the contracted locus of head_{i-1} in T_{i-2} , i.e. *xu* is a node in T_{i-2} . If the contracted locus of head_{i-1} in T_{i-2} is the root then $u=\varepsilon$.
3. $|x| \leq 1$. $x=\varepsilon$ only if $\text{head}_{i-1}=\varepsilon$.

Algorithm *mcc:* substep A

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 seach for w (substep B) and then for v (substep C).

mcc – substep A

Notice that:

- In the previous step 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 P2, *mcc* visited $\underline{xu}$ in the previous step (i-1), hence it can identify xu.

Algorithm *mcc*

mcc – substep A

If u=ε then c← root

(note that root = <u>*u*</u>)

else, c←{suffix link of <u>*xu*</u>} (note that c=<u>*u*</u>)

explanation:

- u≠ε thus by definition <u>*xu*</u> existed (as the contracted locus of *xuw*) in T_{i-2} hence by P1: the internal node <u>*xu*</u> has a suffix link.
- By P2 we remember <u>*xu*</u> from step *i-1* and we can now follow its suffix link.

mcc – substep A

- uwv = $head_i$ 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>u</u> to find the extended locus of *uw*.

Algorithm *mcc*

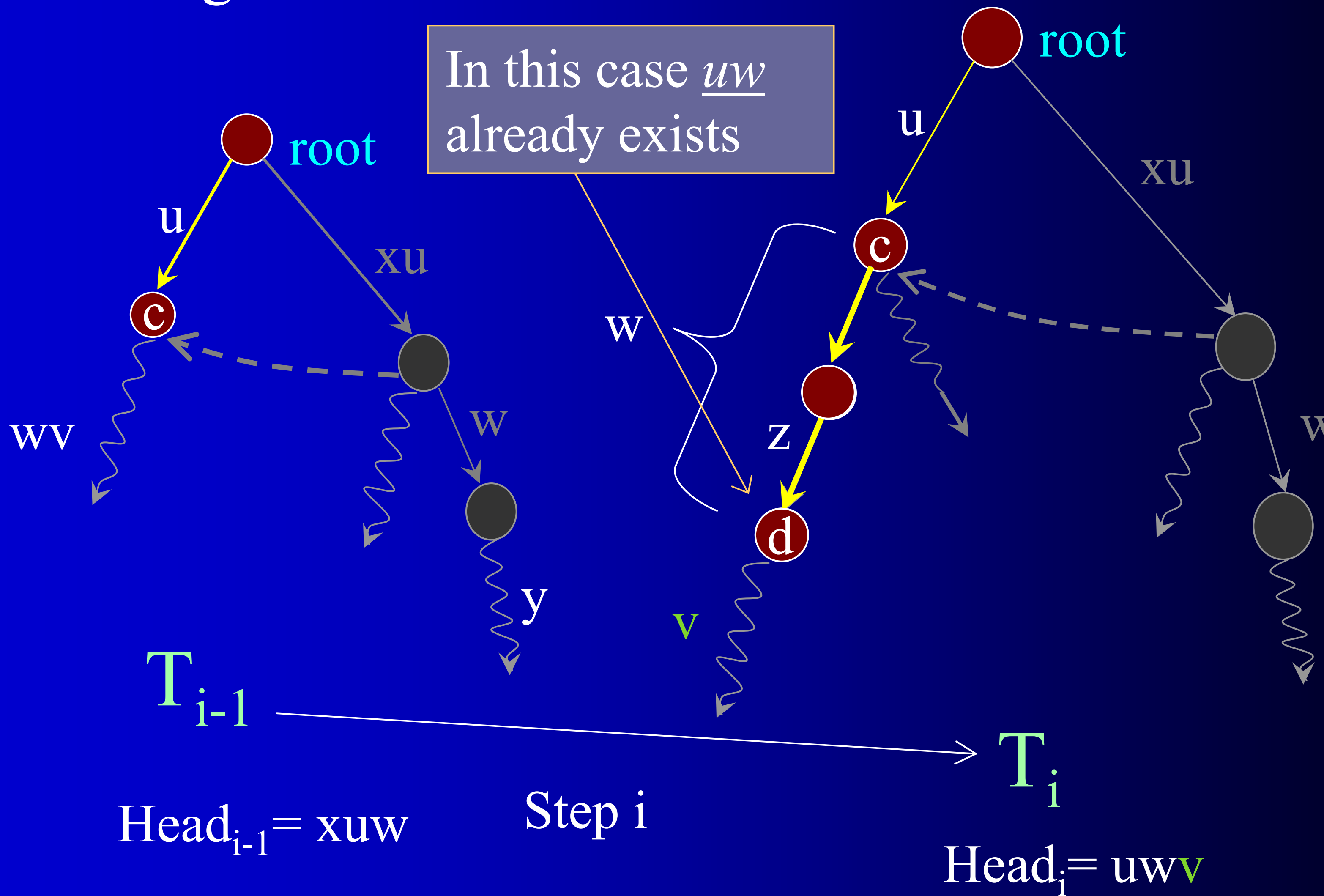
Mcc – substep B: Rescanning

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): $\underline{uw}$.
- d ← $\underline{uw}$

Algorithm *mcc:* substep B - rescanning

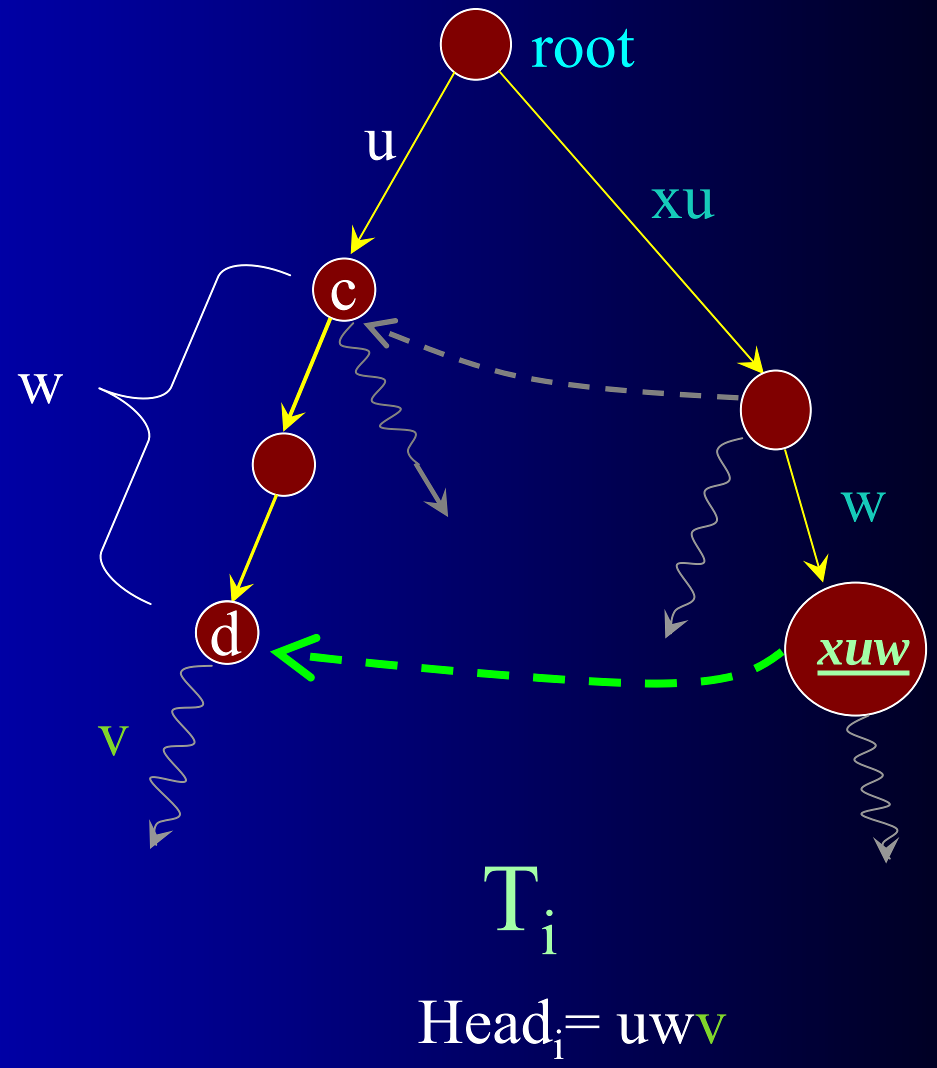
Illustrating substep B in the i^{th} step: rescanning substring w.



mcc – substep B: Scanning

Make the suffix link of *xuw* point to d .

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



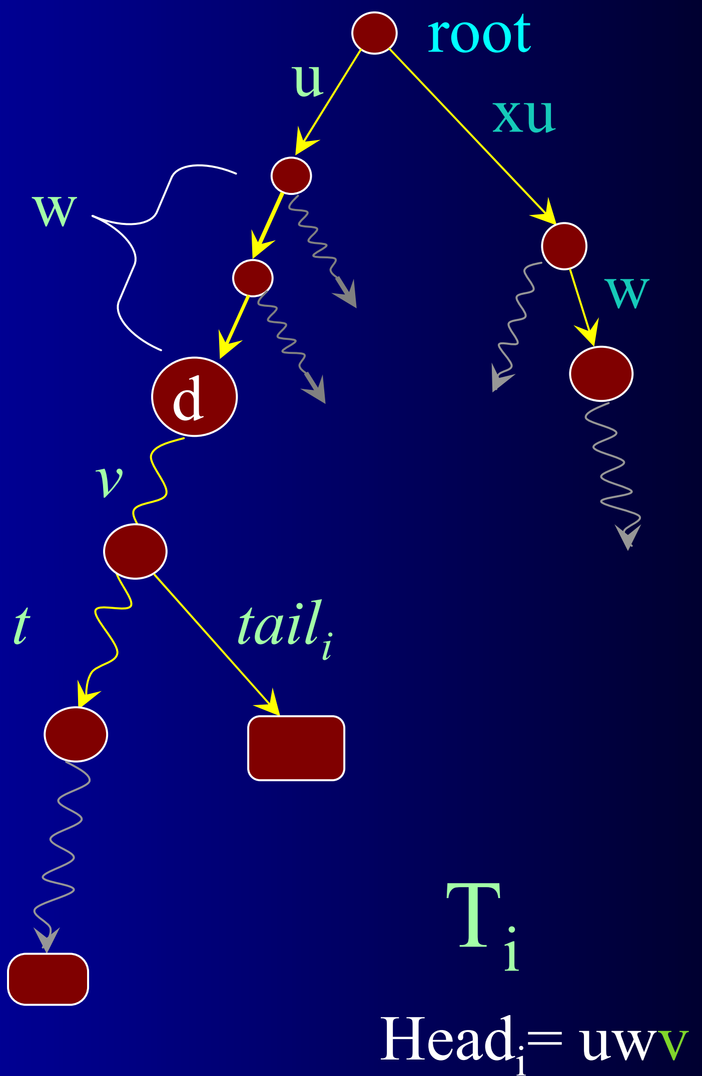
mcc – substep C - Scanning

- Scan the edges from *d* in order to find the extended locus of *uwv*.
- Since **<u>we don't know</u>** 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 <u>*uwv*</u>, if needed.
- Construct the new leaf edge $tail_i$.

Algorithm *mcc*

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 *mcc*

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 T2 that an internal node has at least 2 son edges.

Algorithm *mcc*

Maintaining T2

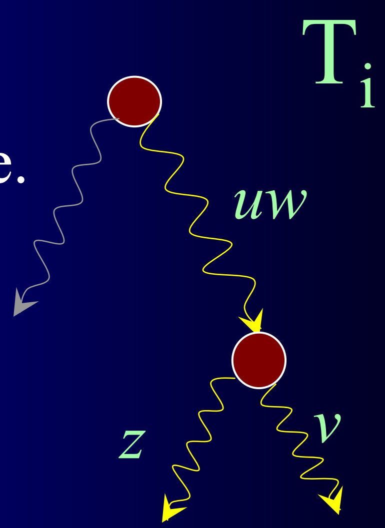
Lemma: In step i, at the end of substep B we add a new node only if *v* is empty.

Proof. In step *i*, If *v* is not empty then $head_i = uwv$ and $head_{i-1} = xuw$ hence, w.l.g. we can write S as follows:

S=……xuwz…..uwv…..xuwv…

Thus, we have 2 occurences of uw with different extensions, *uwv, uwz*, that occur already in the tree. Hence, there is a branching node <u>uw</u>.

Position i-1



Algorithm *mcc*

Maintaining T2

Corollary: In the i^{th} step if v is empty then we add an outgoing edge from the locus of $uv=head_i$. Thus the only case where we add a node we add an outgoing edge to it.

Algorithm *mcc*

Time Complexity Analysis

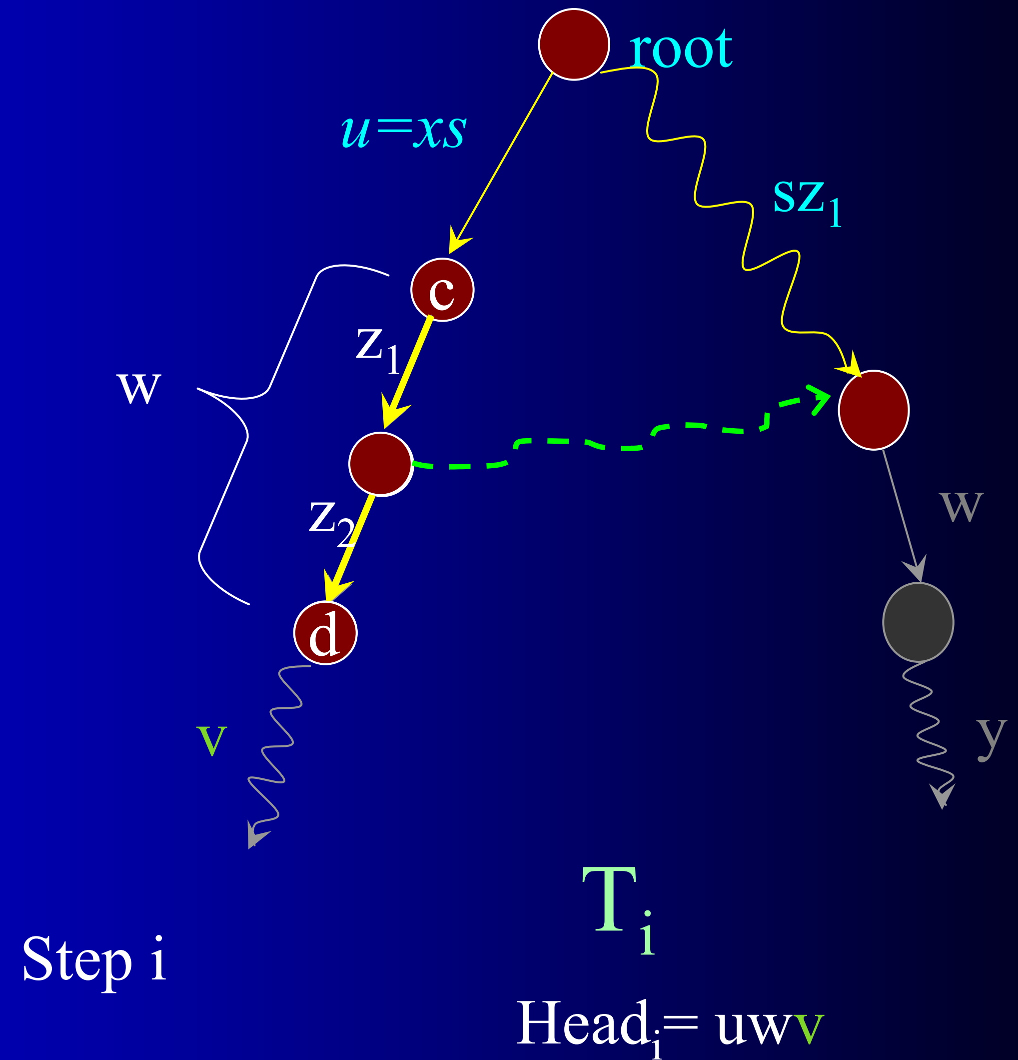
Define: $\mathrm{res}_\mathrm{i} = wv\{tail_i\}$ in step i .

Hence, res_i is the suffix of S rescanned and scanned during step i.

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 res_i but not in res_{i+1}.

Time Complexity Analysis

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



Algorithm *mcc*

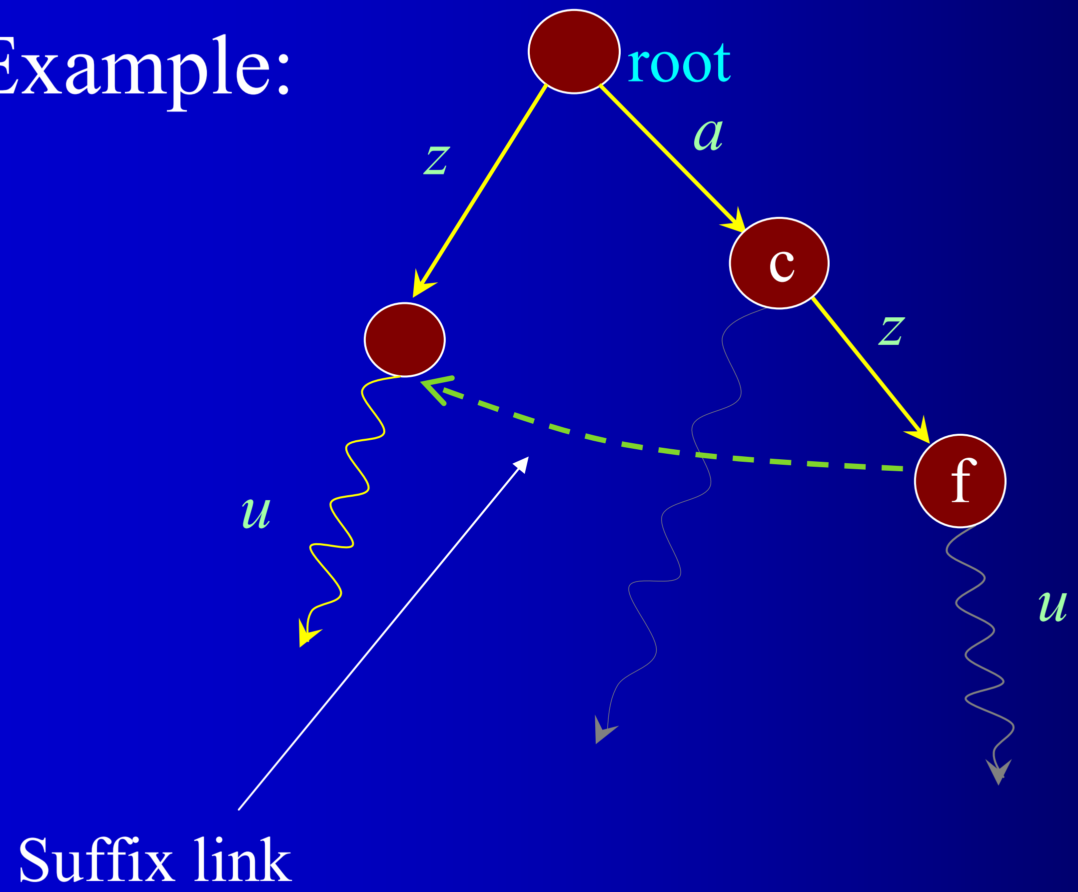
Time Complexity Analysis

- **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 = $\underline{az}$
- This suffix link serves us in substep A of step $i+1$ to reach the node $\underline{z}$, hence we do not have to rescan substring z again.

Algorithm *mcc*

Time Complexity Analysis

Example:



Algorithm *mcc*

Time Complexity Analysis

- **Define**: int_i = number of intermediate nodes (f) rescanned during step i.
- The observation yields:

  $|(res_{i+1})| \leq |(res_i)| - int_i$
- Hence,

$1 = |(res_n)| \leq |(res_{n-1})| - int_{n-1} \leq \ldots \leq |(res_1)| - \sum^n_{i=1} int_i$ (since int_n=0) $\Rightarrow$

$1 \leq |(res_1)| - \sum^n_{i=1} int_i = n - \sum^n_{i=1} int_i \Rightarrow$

$\sum^n_{i=1} int_i \leq n - 1$

i.e., the total number of intermediate nodes rescanned $\leq$ n .

Algorithm *mcc*

Time Complexity Analysis

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:

$$\sum^{n}_{i=1} [|(head_i)|-|(head_{i-1})|+1] = |(head_n)|-|(head_0)|+n = n$$

- 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 = uwv, 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 *uwv* to *uzv*.
- This requires a large pool of position numbers.

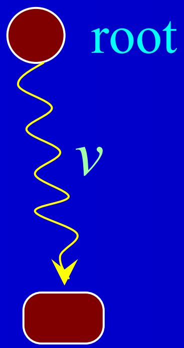
Paths in need to be changed

- 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.
- <u>Definition</u>: 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:

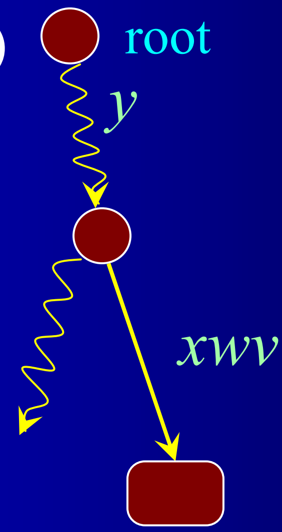
(1)



***Suffix* = *v*:**

Stays as it is.

(2)



***Suffix* =** *u'u*wv:*

➔(by definition of u*) yx=u'u*, for some y,x≥0 and |xy|>0, s.t. the branching occurs at the end of *y*. Stays as it is.

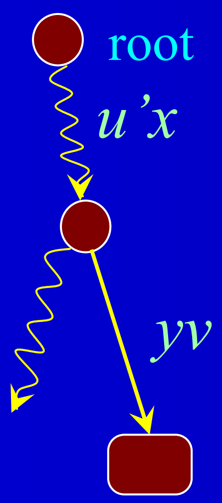
Note that due to our numbering scheme and data structure the label *xwv* 'changes' automatically to *xzv*.

Paths in need to be changed

Illustrating paths that **are** affected by the change:

Suffix = u'wv, u'=suf(u*):

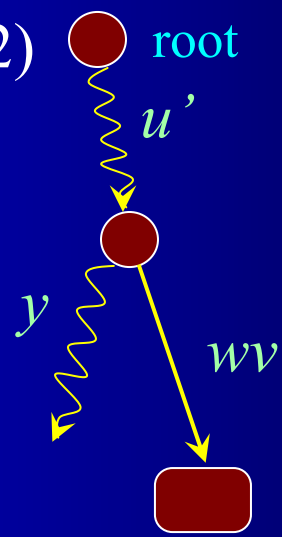
(1)



$xy=w$:

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, *umcc*:

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

Stage 1

Phase 1:

- Denote $u^{(i)}$ the suffix of u of length i.
- Examine the paths $u^{(1)}wv$, $u^{(2)}wv$, $u^{(4)}wv$, $u^{(8)}wv$, …

  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 $u^{(k)}wv$, $u^{(k-1)}wv$, ...
  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 $O(u^*)$

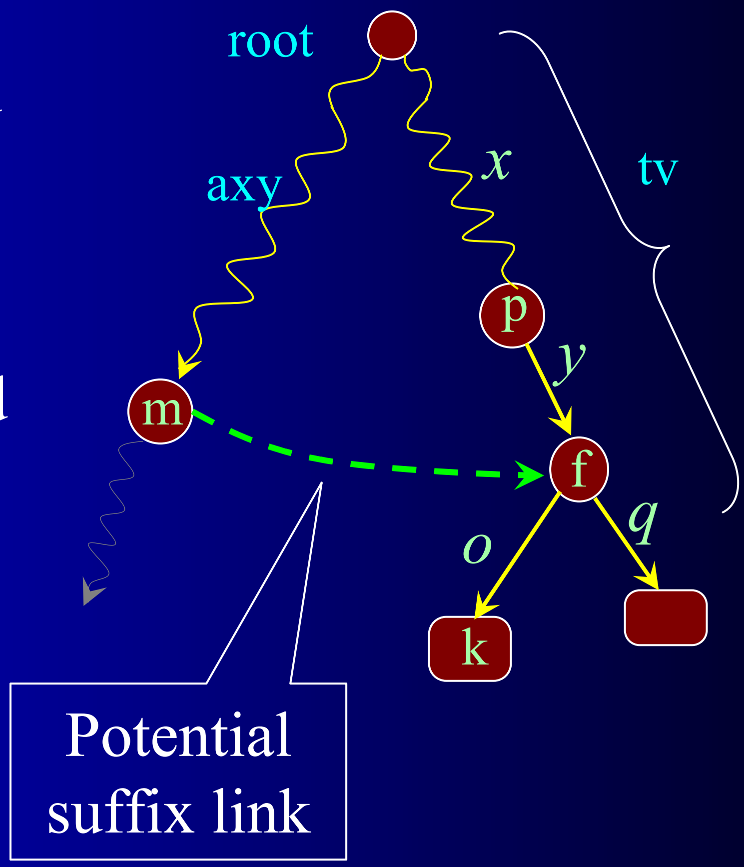
Stage 2

- 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*.



Stage 2

- 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:

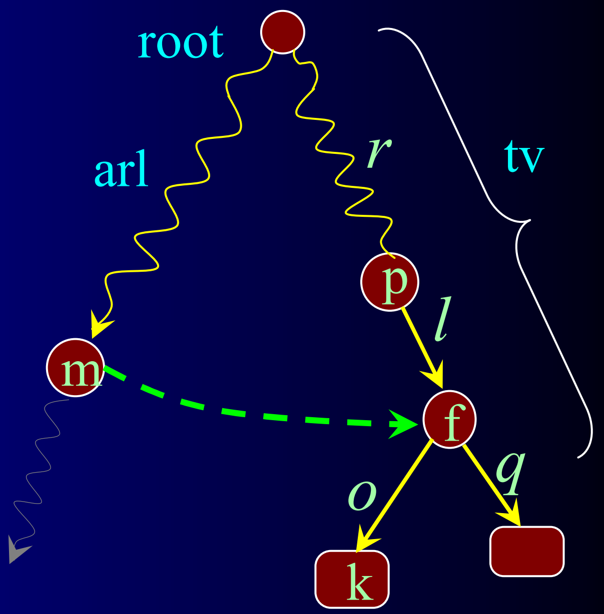
1. Whenever a node f is deleted there is no suffix link pointing to it, except perhaps that of node s^*.
2. Every path in T has a suffix path, except perhaps xu^*zv.

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 *rlq* 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=suf(u*z), into the tree.

- We do it as if we are running *mcc* with a pre-initialized suffix tree that already contains all suffixes of *v*. Denote that tree *T(v),* and this variant algorithm as *umcc(v).*
- We already have all the suffixes longer than *u*zv*, so we start running *mcc* from there:
- Denote $j=|(u)|-|(u^*)|+1$
- Denote $k=|(uz)|$
- We will insert the paths *u*zv, …, dv* (where *d* is the last character of *uz*), by running *mcc* from step *j* through *k+1's* rescanning substep (in order to connect the a suffix link to 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:

$\sum^{k+1}_{i=j} int(v)_i \leq |(res(v)_j)| - |(res(v)_{k+1})| + int(v)_{k+1}$

- $|(res(v)_j)| \leq |(suf_j)| = |(u^*zv)|$
- For all i: $int(v)_i \leq |(w)| \leq |(head_{i-1}(v))|$, where w is the substring rescanned in substep B.
- Hence, $int(v)_{k+1} \leq |(head_k(v))|$.
- For all i: $|(res(v)_i)| \geq |(suf(v)_{i-1})| - |(head_{i-1}(v))|$.
- Hence, $|(res(v)_{k+1})| \geq |(suf(v)_k)| - |(head_k(v))|$.

Time Complexity Analysis

Thus, $\sum^{k+1}_{i=j} int(v)_i \le |(u^*zv)| + |(head_k(v))| - |(suf(v)_k)| + |(head_k(v))|$

$= |(u^*zv)| + 2\,|(v^*)| - |(dv)|$

Hence, rescanning takes $O(|u^*zv|+|v^*|)$.

- 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

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