String Matching
Suffix Trees

Matthew Damigos

May 21, 2007

Problem

String Matching

- T[1,. . . ,n] $\longrightarrow$ text/document array of length n
- P[1,. . . ,m] $\longrightarrow$ pattern array of length m ($m \leq n$)
- P and T are characters drawn from a finite alphabet Σ.
- P and T are called strings

Problem

String Matching

- T[1,. . . ,n] $\rightarrow$ text/document array of length n
- P[1,. . . ,m] $\rightarrow$ pattern array of length m ($m \leq n$)
- P and T are characters drawn from a finite alphabet Σ.
- P and T are called strings

Problems:

- Is there a substring of T matching P?
- How many substrings of T match P?
- Where are first/any k occurrences of P in T?
- Where are all occurrences of P in T?

Problem

String Matching

- T[1,. . . ,n] $\rightarrow$ text/document array of length n
- P[1,. . . ,m] $\rightarrow$ pattern array of length m ($m \leq n$)
- P and T are characters drawn from a finite alphabet Σ.
- P and T are called strings

Problems:

- Is there a substring of T matching P?
- How many substrings of T match P?
- Where are first/any k occurrences of P in T?
- Where are all occurrences of P in T?

Two different approaches:

- Algorithmic approach
- Data structural approach

Tries

Definitions

A **trie** is a tree with children branches labeled with distinct letters from Σ. The branches are ordered alphabetically.(we will append a dollar sign, \$, to the end of all strings)

- Coalescing non-branching paths $\Longrightarrow$ **Compact Trie**

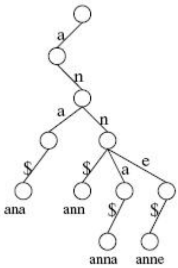
Tries

Definitions

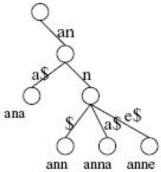
A **trie** is a tree with children branches labeled with distinct letters from Σ. The branches are ordered alphabetically.(we will append a dollar sign, \$, to the end of all strings)

- Coalescing non-branching paths $\Longrightarrow$ **Compact Trie**

Example trie for the set of strings $\{ana, ann, anna, anne\}$



Trie



Compacted Trie

Suffix of a string

Let $S = t_1 t_2 \ldots t_n$ be a string over an alphabet Σ. Each string x such that S = uxv for some (possibly empty) strings u and v is a substring of S, and each string $S_i = t_i \ldots t_n$ where $1 \leq i \leq n + 1$ is a **suffix** of S.

Suffix of a string

Let $S = t_1 t_2 \dots t_n$ be a string over an alphabet Σ. Each string x such that S = uxv for some (possibly empty) strings u and v is a substring of S, and each string $S_i = t_i \dots t_n$ where $1 \leq i \leq n+1$ is a **suffix** of S.

- $S_{n+1} = \epsilon$ is the empty suffix.
- The set of all suffixes of S is denoted $\sigma(S)$.

Suffix Tree Approach

- A tree-like data structure for solving problems involving strings.
- Compact trie-like data structure.

Suffix Tree Approach

- A tree-like data structure for solving problems involving strings.
- Compact trie-like data structure.

Definition

The suffix tree of text S is a compacted trie on all the suffixes of S.

Suffix Tree Approach

- A tree-like data structure for solving problems involving strings.
- Compact trie-like data structure.

Definition

The suffix tree of text S is a compacted trie on all the suffixes of S.

- All occurrences of P in time $O(|P| + \text{number of occurrences})$
- Find an occurrence of P in S, or determine that one does not exist, in time $O(|P|)$

Suffix Tree Approach

- A tree-like data structure for solving problems involving strings.
- Compact trie-like data structure.

Definition

The suffix tree of text S is a compacted trie on all the suffixes of S.

- All occurrences of P in time $O(|P| + \text{number of occurrences})$
- Find an occurrence of P in S, or determine that one does not exist, in time $O(|P|)$

Searching

P occurs in S $\iff$ P is a prefix of some suffix of S $\iff$ Path for P exists in SuffixTree(S)

Suffix Tree Approach

- A tree-like data structure for solving problems involving strings.
- Compact trie-like data structure.

Definition

The suffix tree of text S is a compacted trie on all the suffixes of S.

- All occurrences of P in time $O(|P| + \text{number of occurrences})$
- Find an occurrence of P in S, or determine that one does not exist, in time $O(|P|)$

Searching

P occurs in S $\iff$ P is a prefix of some suffix of S $\iff$ Path for P exists in SuffixTree(S)

- Building a suffix tree for S string of m characters, in O(m) time (On-line).
- The suffix trie of S is a trie representing $\sigma(S)$.

Constructing Suffix Trees - Naive Algorithm

- Start with a root and a leaf numbered 1, connected by an edge labeled S$.
- Enter suffixes $S[2 \ldots m]\$$, $S[3 \ldots m]\$$, $\ldots$, $S[m]\$$ into the tree as follows:
  - To insert $K_i = S[i \ldots m]\$$, follow the path from the root matching characters of K_i until the first mismatch at character $K_i[j]$ (which is bound to happen)
    1. If the matching cannot continue from a node, denote that node by w
    2. Otherwise the mismatch occurs at the middle of an edge, which has to be split
  - If the mismatch occurs at the middle of an edge e = (u,v), let the label of that edge be $a_1 \ldots a_l$
  - If the mismatch occurred at character a_k, then create a new node w, and replace e by edges (u,w) and (w,v) labeled by $a_1 \ldots a_{k-1}$ and $a_k \ldots a_l$
  - Finally, in both cases (a) and (b), create a new leaf numbered i, and connect w to it by an edge labeled with $K_i[j \ldots |Ki|]$.

Example - papua

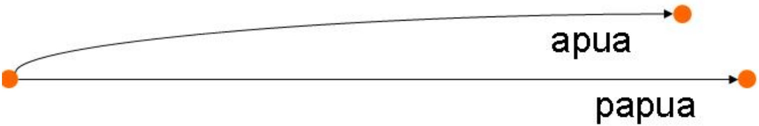
Suffix tree of S=papua

papua$
apua$
pua$
ua$
a$
$

Example - papua

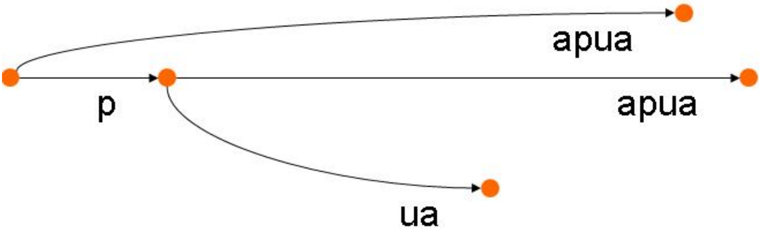
papua$

Example - papua



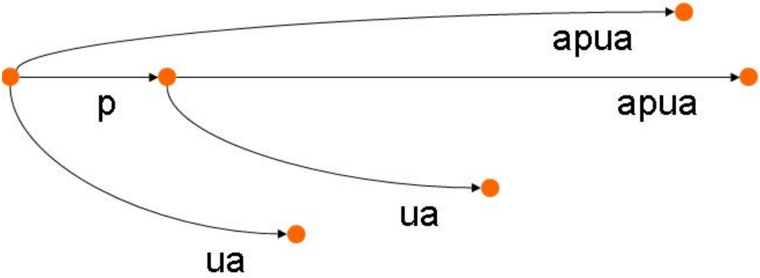
apua$

Example - papua



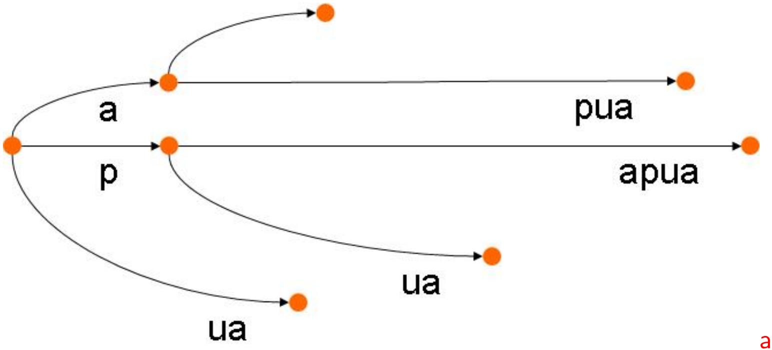
pua$

Example - papua



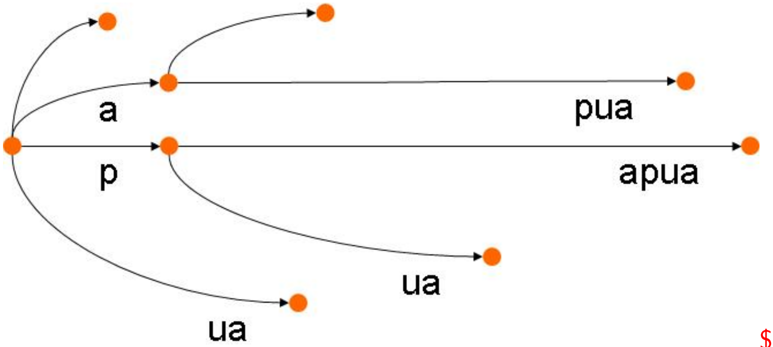
ua$

Example - papua



a$

Example - papua



$

- This is an $O(m^3)$ time and $O(m^2)$ space algorithm (Too much time and space)
- We need a few implementation speed-ups to achieve the O(m) time and O(m) space bounds

- This is an $O(m^3)$ time and $O(m^2)$ space algorithm (Too much time and space)
- We need a few implementation speed-ups to achieve the O(m) time and O(m) space bounds

Speed ups

1. Suffix Links: speed up navigation to the next extension point in the tree

- This is an $O(m^3)$ time and $O(m^2)$ space algorithm (Too much time and space)
- We need a few implementation speed-ups to achieve the O(m) time and O(m) space bounds

Speed ups

1. Suffix Links: speed up navigation to the next extension point in the tree
2. Skip/Count Trick: instead of stepping through each character, we know that we can just jump, as long as we are the right distance

- This is an $O(m^3)$ time and $O(m^2)$ space algorithm (Too much time and space)
- We need a few implementation speed-ups to achieve the O(m) time and O(m) space bounds

Speed ups

1. Suffix Links: speed up navigation to the next extension point in the tree
2. Skip/Count Trick: instead of stepping through each character, we know that we can just jump, as long as we are the right distance
3. Edge-Label Compression: since we have a copy of the string, we don't need to store copies of the substrings for each edge

- This is an $O(m^3)$ time and $O(m^2)$ space algorithm (Too much time and space)
- We need a few implementation speed-ups to achieve the O(m) time and O(m) space bounds

Speed ups

1. Suffix Links: speed up navigation to the next extension point in the tree
2. Skip/Count Trick: instead of stepping through each character, we know that we can just jump, as long as we are the right distance
3. Edge-Label Compression: since we have a copy of the string, we don't need to store copies of the substrings for each edge
    - $O(m^2)$ space becomes O(m) space

- This is an $O(m^3)$ time and $O(m^2)$ space algorithm (Too much time and space)
- We need a few implementation speed-ups to achieve the O(m) time and O(m) space bounds

Speed ups

1. Suffix Links: speed up navigation to the next extension point in the tree
2. Skip/Count Trick: instead of stepping through each character, we know that we can just jump, as long as we are the right distance
3. Edge-Label Compression: since we have a copy of the string, we don't need to store copies of the substrings for each edge
    - $O(m^2)$ space becomes O(m) space
4. Once a leaf, always a leaf: We don't need to update each leaf, since it will always be the end of the current string. We can get these updates for free. (Either maintain a global end-of-string index or insert the whole string for every leaf)

- This is an $O(m^3)$ time and $O(m^2)$ space algorithm (Too much time and space)
- We need a few implementation speed-ups to achieve the O(m) time and O(m) space bounds

Speed ups

1. Suffix Links: speed up navigation to the next extension point in the tree
2. Skip/Count Trick: instead of stepping through each character, we know that we can just jump, as long as we are the right distance
3. Edge-Label Compression: since we have a copy of the string, we don't need to store copies of the substrings for each edge
   - $O(m^2)$ space becomes O(m) space
4. Once a leaf, always a leaf: We don't need to update each leaf, since it will always be the end of the current string. We can get these updates for free. (Either maintain a global end-of-string index or insert the whole string for every leaf)

Observation

Suffix Trie/Tree of string S can be seen as a DFA: language accepted = the suffixes of S

Suffix Trie

The suffix trie of S is the STrie(S)=$(Q \cup \{\perp\}, root, F, g, f)$ augmented DFA which has a tree-shaped transition graph representing the trie for $\sigma(S)$ and which is augmented with the so-called suffix function f and auxiliary state $\perp$.

- Q is a set of the states of STrie(S) (1-1 correspondence with the substrings of S).
- $\overline{x}$ is the state that corresponds to a substring x.
- F is the set of the final states corresponds to $\sigma(S)$.
- The transition function g is defined as g$(\overline{x}, \alpha) = \overline{y}$, $\forall \overline{x}, \overline{y} \in Q$ such that $y = x\alpha$, where $\alpha \in \Sigma$. Moreover, $g(\perp, \alpha) = root$
- The suffix function f is defined $\forall \overline{x} \in Q$ as follows: Let $\overline{x} \neq root$. Then $x = \alpha y$ for some $\alpha \in \Sigma$, and we set g$(\overline{x}) = \overline{y}$. Moreover, f(root) $=\perp$.

On-line Procedure for building $STrie(T^i)$ from $STrie(T^{i-1})$

1. $r \leftarrow top$;
2. **while** $g(r, t_i)$ is undefined **do**
   1. create new state r' and new transition $g(r, t_i) = r'$;
   2. if $r \neq top$ **then** create new suffix link $f(oldr') = r'$;
   3. $oldr' \leftarrow r'$;
   4. $r \leftarrow f(r)$;
3. create new suffix link $f(oldr') = g(r, t_i)$;
4. $top \leftarrow g(top, t_i)$.

Example: construction of Suffix Trie for S=cacao

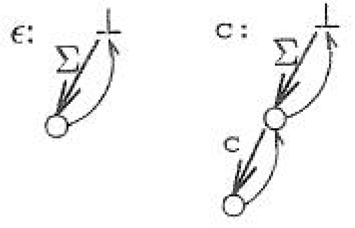
1^{st} iteration:

- Initially: $g(\bot, \Sigma) = root$ and $f(root) = \bot$
- top = root ($=\overline{\epsilon}$)
- $r \leftarrow root$ ($=\overline{\epsilon}$)
- g(root,c)=undefined :
- New state: r_1 and New transition: g(root,c)=r_1
- (Checking if r=top): r=top
- $oldr' \leftarrow r_1$
- $r \leftarrow f(root)(= \bot)$
- $top \leftarrow g(root, c)(= r_1)$

Example: construction of Suffix Trie for S=cacao

1^{st} iteration:

- Initially: $g(\perp, \Sigma) = root$ and $f(root) = \perp$
- top = root ($=\overline{\epsilon}$)
- $r \leftarrow root$ ($=\overline{\epsilon}$)
- g(root,c)=undefined :
- New state: r_1 and New transition: g(root,c)=r_1
- (Checking if r=top): r=top
- $oldr' \leftarrow r_1$
- $r \leftarrow f(root)(= \perp)$
- $top \leftarrow g(root, c)(= r_1)$

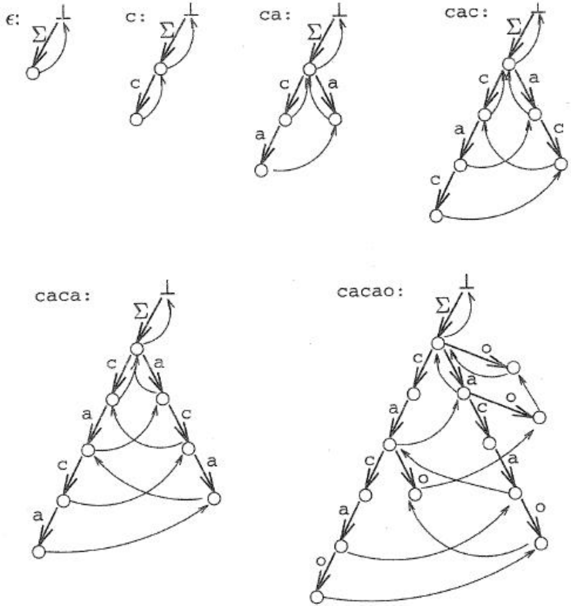


Example (cont.)

2^{nd} iteration:

- $r \leftarrow top$ $(=r_1)$
- g(r_1,a)=undefined :
- New state: r_2 and New transition: g(r_1,a)=r_2
- (Checking if r=top): r=top
- $oldr' \leftarrow r_2$
- $r \leftarrow f(r_1)(= root)$
- g(root,a)=undefined :
- New state: r_3 and New transition: g($root$,a)=r_3
- (Checking if r=top): $r \neq top \Rightarrow f(oldr') = f(r_2) = r_3$
- $oldr' \leftarrow r_3$
- $r \leftarrow f(root)(= \perp)$
- $f(r_3) \leftarrow g(\perp, a)(= root)$

Example (cont.)



Suffix trees On-line

- Ukkonen's method (On-line construction of Suffix Trees) constructs a suffix tree for $S[1 \dots m]$ in time O(m)
- The method builds, as intermediate results, for each prefix $S[1 \dots 1]$, $S[1 \dots 2]$, $\dots$, $S[1 \dots m]$ an implicit suffix tree
- The **implicit suffix tree** of a string is what results by applying suffix tree construction to the string without an added end marker $
- Denote the implicit suffix tree of the prefix $S[1 \dots i]$ by $\mathcal{I}_i$
  - $\mathcal{I}_1$: single edge labeled by S[1] leading to leaf 1
- Phase i+1 updates T from $\mathcal{I}_i$ (with all suffixes of $S[1 \dots i]$) to $\mathcal{I}_{i+1}$ (with all suffixes of $S[1 \dots i+1]$)
- Each phase i + 1 consists of extensions j = $1, \dots, i+1$

Suffix Extension Rules

- **Rule 1** If $S[j \dots i]$ leads to a leaf (j), catenate S[i+1] to its edge label
- **Rule 2** If path $S[j \dots i]$ ends before a leaf, and does not continue by S[i+1]: Connect the end of the path to a new leaf j by an edge labeled by char S[i+1]. (If the path ended at the middle of an edge, split the edge and insert a new node as the parent of leaf j)
- **Rule 3** If the path continues by S[i+1], do nothing. (Suffix $S[j \dots i+1]$ is already in the tree)

Improvement

- Total time for all phases i $= 2, \ldots, m+1$ is $\Theta(m^3)$.
  - How to improve this?
  - we need to avoid or speed up path traversals ($O(m^2)$)
- Suffix Links: For each internal node v of T "labeled" by $x\alpha$, where $x \in \Sigma$ and $\alpha \in \Sigma^*$, define s(v) to be the node "labeled" by α. Then a pointer from v to s(v) is the **suffix link** of v.
  - Extension j (of phase i + 1) finds the end of the path $S[j \ldots i]$ in the tree (and extends it with char S[i+1])
  - Extension j+1 finds the end of the path $S[j+1 \ldots i]$
  - Assume v is an internal node "labeled" by $S[j]\alpha$ on the path $S[j \ldots i]$: we can avoid traversing path α when locating the end of $S[j+1 \ldots i]$, by starting from node s(v)

Improvement (cont.)

- The path $S[j \dots i]$ followed in extension j is in the tree (it is a suffix of $S[1 \dots i]$)
  - no need to check all of its characters; it suffices to choose the correct edges
- Let S[k] be the next char to be matched on path $S[j \dots i]$:
  - An edge labeled by $S[p \dots q]$ can be traversed simply by checking that S[p] = S[k], and skipping the next $q - p$ chars of $S[j \dots i]$ (skip/count)

Improvement (cont.)

- The path $S[j \dots i]$ followed in extension j is in the tree (it is a suffix of $S[1 \dots i]$)
  - no need to check all of its characters; it suffices to choose the correct edges
- Let S[k] be the next char to be matched on path $S[j \dots i]$:
  - An edge labeled by $S[p \dots q]$ can be traversed simply by checking that S[p] = S[k], and skipping the next $q - p$ chars of $S[j \dots i]$ (skip/count)

Using suffix links and the skip/count trick, a single phase takes time **O(m)**

Improvement (cont.)

- The path $S[j \dots i]$ followed in extension j is in the tree (it is a suffix of $S[1 \dots i]$)
  - no need to check all of its characters; it suffices to choose the correct edges
- Let S[k] be the next char to be matched on path $S[j \dots i]$:
  - An edge labeled by $S[p \dots q]$ can be traversed simply by checking that S[p] = S[k], and skipping the next $q - p$ chars of $S[j \dots i]$ (skip/count)

Using suffix links and the skip/count trick, a single phase takes time **O(m)**

- If path $S[j \dots i+1]$ is already in the tree, so are paths $S[j+1 \dots i+1], \dots, S[i+1]$, too $\Rightarrow$ phase i + 1 can be finished at the first extension j that applies Rule 3; all the rest are void, too
- Use of compressed edge representation (i.e., indices p and q instead of substring $S[p \dots q]$), and represent the end position of each terminal edge by a global value e for "the current end position".

Suffix-tree on-line: main procedure

Input: String $S = t_1 \ldots t_n$, $t_{n+1} = \$$

- $T \longleftarrow T_1$
- while $i \leq n$ do:
  1. Set current end-position: e := i + 1; (to implement extensions $1, \ldots, j_i$ implicitly)
  2. Compute extensions $j+1, \ldots, j^*$ until $j^* > i+1$ or Rule 3 was applied in extension j^*;
  3. Set $j_{i+1} := j^* - 1$; (for the next phase)

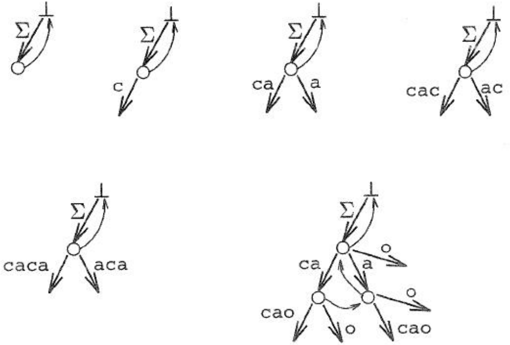
Example (On-line construction of suffix tree for S=cacao)

- Initial state: root
- SLink(root)=root
- $T \longleftarrow T_1$

- e=2 (S[2]=a)
- Rule 1: catenate s[2] to edge S[1]
- Rule 1: create a arc from root with label=S[2]=a
- inplicitly: slink(“ca“)=“a“
- inplicitly: slink(“a“)=root

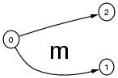
⋮

Example (On-line construction of suffix tree for S=cacao)



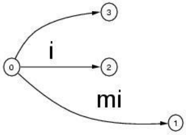
Example - mississippi

m



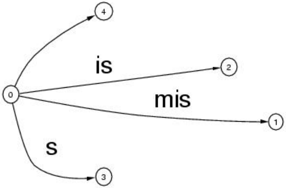
Example - mississippi

mi



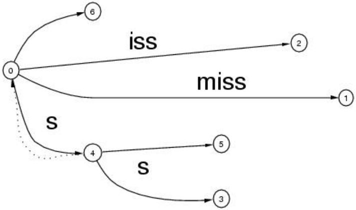
Example - mississippi

mis



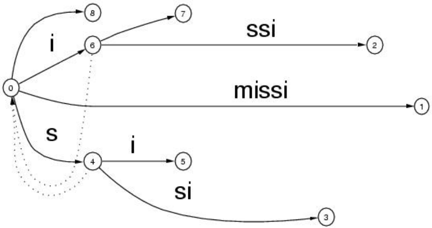
Example - mississippi

miss

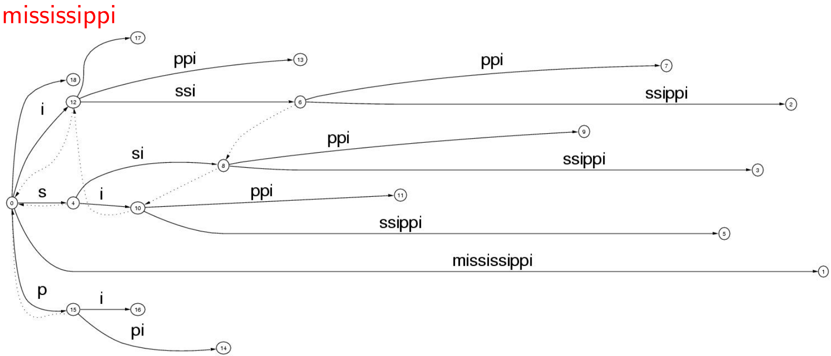


Example - mississippi

missi



Example - mississippi



Main Construction Method

- Create Tree(t_1); slink(root) := root
- (v, α) := (root, ϵ) /* (v, α) is the start node */
- for i := 2 to n+1 do
- v' := 0
- while there is no arc from v with label prefix αt_i do
- if $\alpha \neq \epsilon$ then /* divide the arc w = son(v, $\alpha\eta$) into two */
- son(v, α) := v''; son(v'', t_i) := v'''; son(v',η) := w
- else
- son(v,t_i) := v'''; v'' := v
- if $v' \neq 0$ then slink(v') := v''
- v' := v''; v := slink(v); (v, α) := Canonize(v, α)
- if $v' \neg 0$ then slink(v') := v
- (v, α) := Canonize(v, αt_i) /* (v, α) = start node of the next round */

On-line procedure for suffix tree

Input: string $S = t_1 t_2 \ldots t_n \$$ Output: Tree(S) Notation:

- son(v,α) $=$ w iff there is an arc from v to w with label α
- son(v,ϵ) $=$ v

Function Canonize(v, ϵ):

- while son(v, α') $\neq 0$ where $\alpha = \alpha'\alpha''$, $|\alpha'| > 0$ do
- v := son(v, α'); $\alpha := \alpha''$
- return (v, α)