On-Line Construction

of Suffix Trees

E. Ukkonen

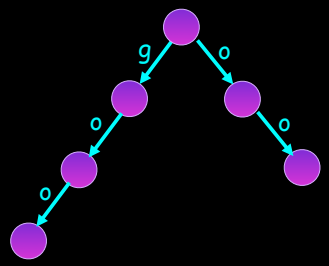
Overview

- Suffix tries
- On-line construction of suffix tries in quadratic time
- Suffix trees
- On-line construction of suffix trees in linear time
- Applications

Suffix Trees

A suffix tree is a trie-like data structure representing all suffixes of a string.

goo



Notations

- Let $T = t_1 \ldots t_n$ be a string.
- For $0 \le i \le n$, let $T^i = t_1 \ldots t_i$ denote the i-length prefix of T.
- For $1 \le i \le n + 1$, let $T_i = t_i \ldots t_n$ denote the suffix of T that starts at the i^{th} position.
- Let $\sigma(T) = \{T_i \mid 1 \le i \le n + 1\}$.

Suffix Tries

The suffix trie of T, denoted by STrie(T), is a trie representing $\sigma(T)$.

Suffix Tries (cont.)

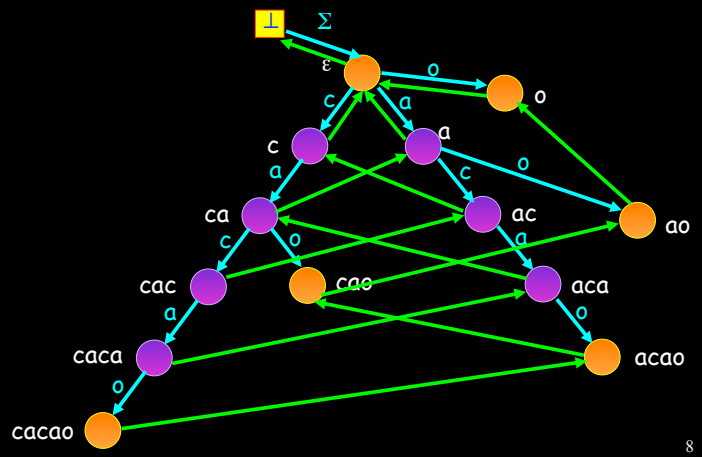
Definition: STrie(T) is an augmented DFA, STrie(T) = (Q∪{⊥}, root, F, g, f) where:

- Q = {x | x is a substring of T} is the set of the states of the DFA.
- ⊥ is an auxiliary state.
- root is the initial state, corresponding to the empty string ε.
- F = $\sigma(T)$ is the set of finite states.

Suffix Tries (cont.)

- $g : Q \cup \{\perp\} \times \Sigma \rightarrow Q$ (a partial function) is the transition function, defined as follows:
  - $g(x,a) = y$ for all $x,y \in Q$ and $a \in \Sigma$, s.t. $y = xa$.
  - $g(\perp,a) = root$ for all $a \in \Sigma$.
- $f : Q \rightarrow Q \cup \{\perp\}$ is the suffix function defined as follows:
  - $f(x) = y$ for all $x,y \in Q$, $x \neq root$, s.t $\exists a \in \Sigma$, s.t. $x = ay$.
  - $f(root) = \perp$.

An Example - STrie(cacao)



The Size of Suffix Tries

Theorem: The size of STrie(T), where $|T| = n$, is $O(n^2)$.

Proof: The size of STrie(T) is linear in the number of substrings of T.
T has at most $O(n^2)$ substrings. Thus the size of STrie(T) is $O(n^2)$.

On-Line Construction of Suffix Tries

- Let $T = t_1 \ldots t_n$.
- $\forall\ 1 \le i \le n$, the algorithm constructs $STrie(T^i)$.
- First we construct $STrie(T^0) = STrie(\varepsilon)$.
- Then, $\forall\ 1 \le i \le n$, we obtain $STrie(T^i)$ from $STrie(T^{i-1})$.

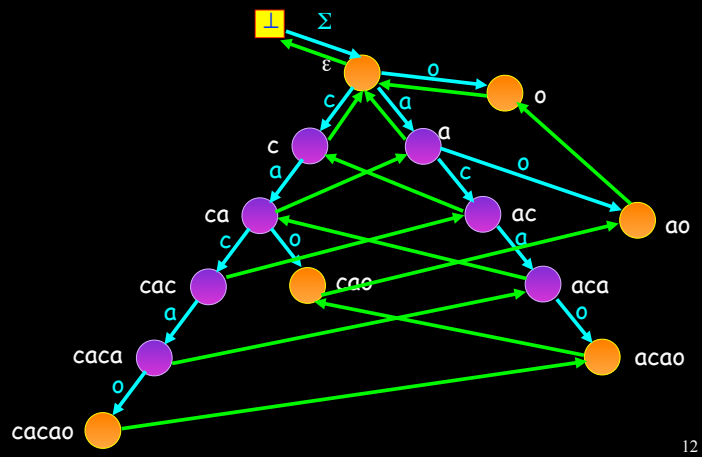
On-Line Construction of Suffix Tries (cont.)

Observation 1: $\sigma(T^i) = \{x t_i \mid x \in \sigma(T^{i-1})\} \cup \{\varepsilon\}$.

Observation 2: The suffixes of T^i can be found by starting at the state T^i and following the suffix links, until ε.
Thus, $\sigma(T^i) = \{f^j(T^i) \mid 0 \le j \le i\}$.

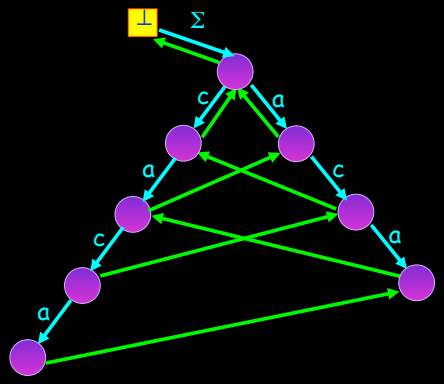
Definition: The path from T^i to $\perp$ following the suffix links is called the boundary path of $STrie(T^i)$.

On-Line Construction of Suffix Tries (cont.)



$STrie(T^{i-1}) \Rightarrow STrie(T^{i})$

cac ⇒ caca



The Algorithm

create STrie(ε)

top ← ε

for i ← 1 to n do

  r ← top

  while $g(r,t_i)$ is undefined do

    create new state r′ and $g(r,t_i) \leftarrow r'$

    if r ≠ top then f(old-r′) ← r′

    old-r′ ← r′

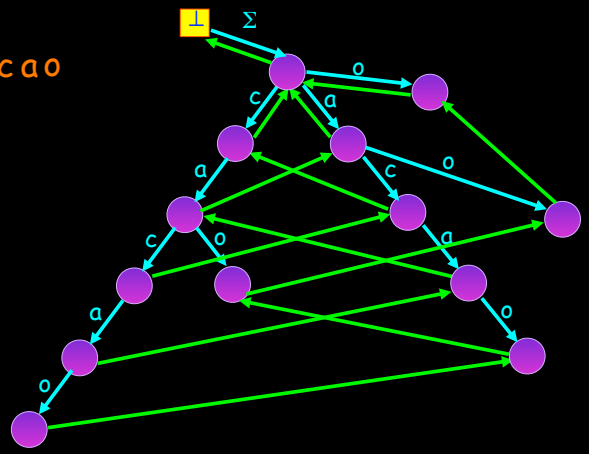
    r ← f(r)

  $f(\text{old-}r') \leftarrow g(r,t_i)$

  $top \leftarrow g(top,t_i)$

The Algorithm (cont.)

cacao



Running Time

Theorem: The running time of the algorithm is linear in the size of STrie(T), which is, in worst case, $O(|T|^2)$.

Running Time (cont.)

```
create STrie(ε)
top ← ε
for i ← 1 to n do
    r ← top
    while g(r, t_i) is undefined do
        create new state r' and g(r, t_i) ← r'
        if r ≠ top then f(old-r') ← r'
        old-r' ← r'
        r ← f(r)
    f(old-r') ← g(r, t_i)
```

O(1) for each node added to STrie(T)

Suffix Trees

- A suffix tree STree(T) represents STrie(T) in space linear in |T|.
- This is achieved by representing only a subset of Q'∪{⊥} of Q∪{⊥}, called the explicit states.

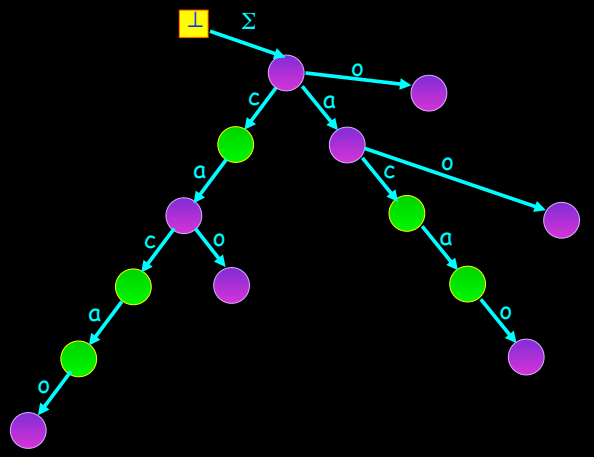
Explicit and Implicit States

Definition: A state q is called explicit in the following cases:

- q is a leaf
- q is a branching state (has at least two transitions)
- root and ⊥ are also defined to be branching states.

Otherwise (if q has exactly one transitions and is not the root or ⊥), q is called implicit.

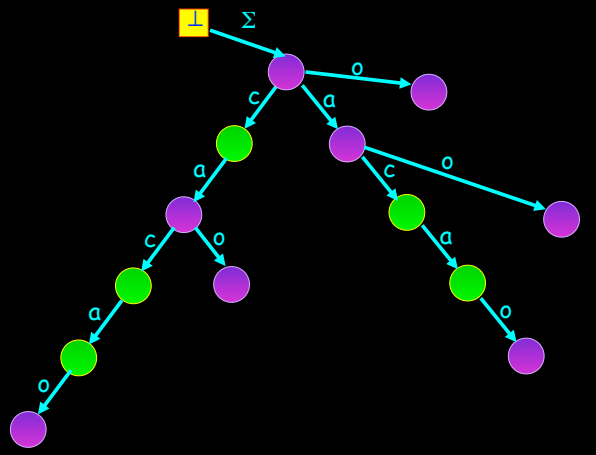
Explicit and Implicit States (cont).



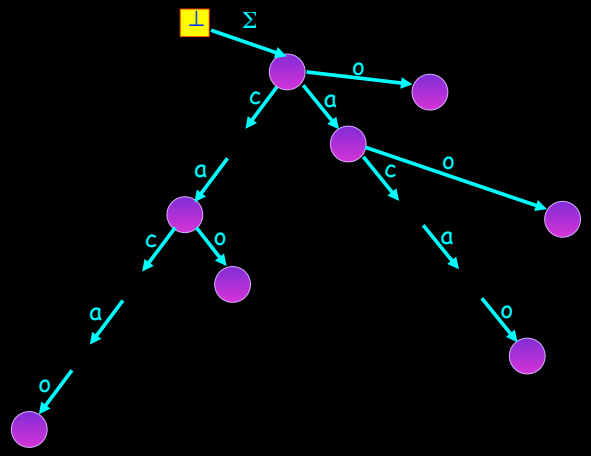
Generalized Transition Function

- The string w spelled out by the transition path in STrie(T) between two explicit states s and r is represented in STree(T) as a generalized transition $g'(s,w) = r$.
- A generalized transition $g'(s,w) = r$ is called an a-transition if $\exists a \in \Sigma$ and $v \in \Sigma^*$ s.t. $w = av$.
- Note that for each explicit state s and $a \in \Sigma$ there is at most one a-transition from s.

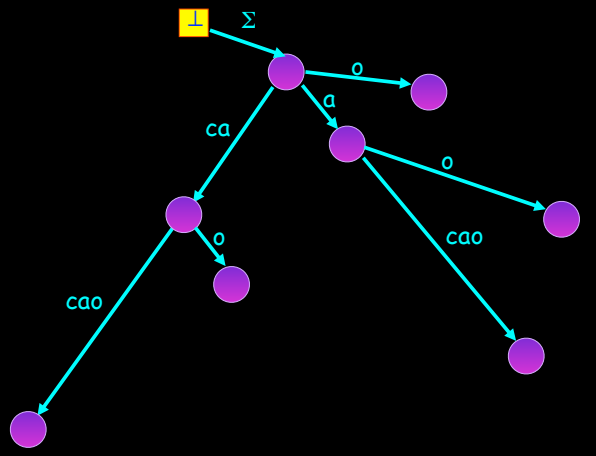
STrie(T) ⇒ STree(T)



STrie(T) ⇒ STree(T)



STrie(T) ⇒ STree(T)



Suffix Links

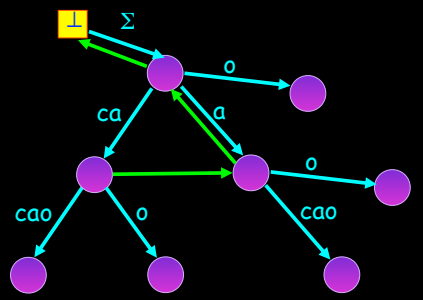
Definition: If $x \in Q'$ is a branching state and $x = ay$, where $a \in \Sigma$, then the suffix link of x is defined by $f'(x) = y$, and $f'(\varepsilon) = \perp$.

Proposition: If $x \in Q'$ is a branching state and $f'(x) = y$ then y is also a branching state.

Proof: $\exists a \neq b \in \Sigma$ s.t. xa and xb are substrings of T. y is a suffix of x. Thus ya and yb are also substrings of T.

STree(T)

STree(T) = $(Q' \cup \{\perp\}, root, g', f')$.



The Size of Suffix Trees

Theorem: The size of STree(T), where $|T| = n$, is $O(n)$.

Proof: Since we represent each substring $w = t_k \ldots t_p$ of T by a pair pointers (k,p), the size of STree(T) is linear in the number of explicit states. STree(T) has at most n leaves, and thus at most $n - 1$ branching states. Therefore, the size of STree(T) is $O(n)$.

Reference Pairs

Definition: Let r be an explicit or implicit state. (s,w) is called a reference pair for r if:

- s is an explicit state and an ancestor of r.
- w is the string spelled out by the transitions from s to r in the corresponding suffix trie.

Definition: A reference pair (s,w) for r is called canonical if s is the closest explicit ancestor of r (or r itself, if it is explicit).

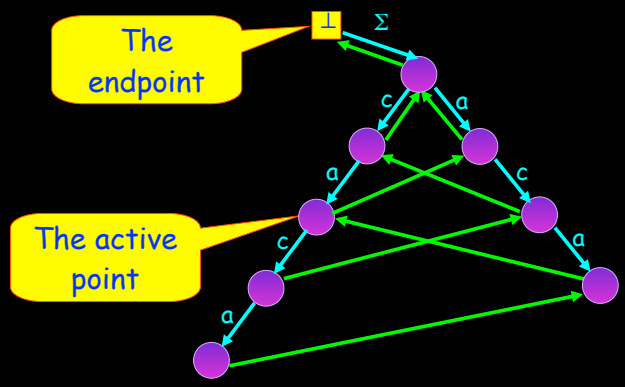
Active Point and Endpoint

Let $s_1 = T^{i-1}$, s_2, …, s_i = root, $s_{i+1} = \perp$ be the boundary path of STrie(T^{i-1}).

Definition: s_j is called the active point of STrie(T^{i-1}) if j is the smallest index for which s_j is not a leaf.

Definition: $s_{j'}$ is called the endpoint of STrie(T^{i-1}) if j' is the smallest index for

Active Point and Endpoint (cont.)



Active Point and Endpoint (cont.)

Proposition: s_j and $s_{j'}$ are well defined and $j \le j'$.

Proof:

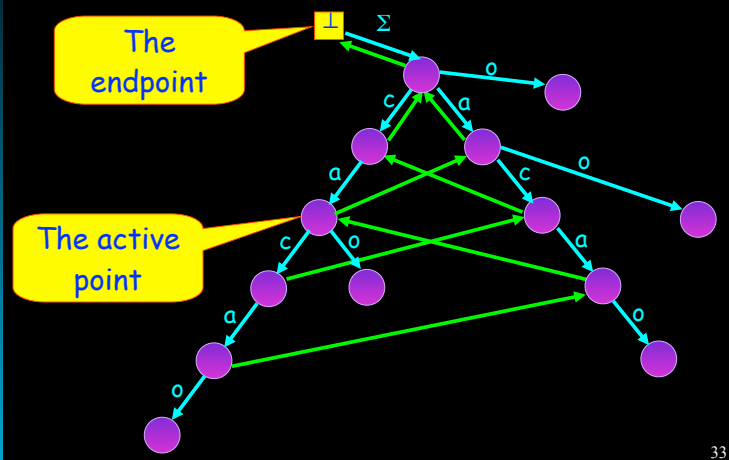
- root is not a leaf $\Rightarrow$ s_j is defined.
- $g(\perp, t_i)$ is defined $\Rightarrow$ $s_{j'}$ is defined.
- $g(s_{j'}, t_i)$ is defined $\Rightarrow$ $s_{j'}$ is not a leaf $\Rightarrow j \le j'$.

Adding t_i-Transitions to STrie(T^{i-1})

Lemma: When obtaining STrie(T^i) from STrie(T^{i-1}) the algorithm adds a t_i-transition to each state s_h s.t. $1 \le h < j'$, and only to these states, as follows:

- For $1 \le h < j$, the new transition expands an old branch of the trie that ends at s_h.
- For $j \le h < j'$, the new transition initiates a new branch from s_h.

Adding t_i-Transitions to STrie(T^{i-1}) (cont.)



On-Line Construction of Suffix Trees

- We create STree(ε), and then ∀ 1 ≤ i ≤ n we obtain STree(T^i) from STree(T^{i-1}).
- When obtaining STree(T^i) from STree(T^{i-1}), we update STree(T^{i-1}) according to the transitions we would add to STrie(T^{i-1}).
- Note that $s_1,\ldots,s_{i-1}$ are not necessarily explicit states.

On-Line Construction of Suffix Trees (cont.)

For $1 \le h < j$:

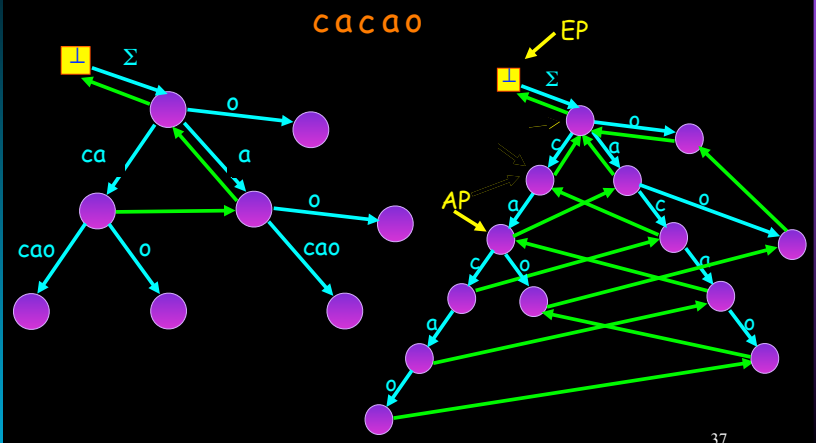
- s_h is a leaf. Thus, ∃ s, $0 \le k \le i-1$ s.t. $g'(s,(k,i-1)) = s_h$. We replace this transition by $g'(s,(k,i)) = s_h$.
- This would take too much time. Thus, we denote transitions of the type $g'(s,(k,i-1))$ in STree(T^{i-1}) by $g'(s,(k,\infty))$. Hence, no updates are needed.

On-Line Construction of Suffix Trees (cont.)

For $j \le h < j'$:

- If s_h is an implicit state, we turn it into an explicit state by splitting the transition containing it.
- We create a new leaf $s_h t_i$ and add a new transition $g'(s_h,(i,\infty))$.

On-Line Construction of Suffix Trees (cont.)



Lemma 1

Lemma 1: Let (s,(k,p)) be some reference pair for a state r. Then ∃ s', k' s.t. (s',(k',p)) is the canonical reference pair for r.

Proof: Let s' be the closest explicit ancestor of r, or r itself if r is explicit. $t_k...t_p$ is the path from the explicit state s to r. Thus, the path from s' to r is a suffix $t_{k'}...t_p$ of $t_k...t_p$.

Lemma 2

Lemma 2: Let r be a state on the boundary path of STrie(T^i). Then ∃ s, k s.t. (s,(k,i)) is the canonical reference pair for r.

Proof: r is on the boundary path of STrie(T^i).
⇒ r refers to some suffix $t_{k'}...t_i$ of T^i.
⇒ (ε,(k',i)) is a reference pair for r.
⇒ the claim holds by lemma 1.

Lemma 3

Lemma 3: Let (s,(k,i-1)) be a reference pair for the endpoint of STrie(T^{i-1}). Then (s,(k,i)) is a reference pair for the active point of STrie(T^i).

Proof:

- s_j is the active point of STrie(T^{i-1}) iff $t_j...t_{i-1}$ is the longest suffix of T^{i-1} that occurs at least twice in T^{i-1}.

Lemma 3 (cont.)

Proof (cont.):

- $s_{j'}$ is the endpoint of STrie(T^{i-1}) iff $t_{j'}...t_{i-1}$ is the longest suffix of T^{i-1} such that $t_{j'}...t_{i-1}t_i$ is a substring of T^{i-1}.

- Thus, if $s_{j'}$ is the endpoint of STrie(T^{i-1}), then $t_{j'}...t_{i-1}t_i$ is the longest suffix of T^i that occurs at least twice in T^i. Therefore, $s_{j'}t_i$ is the active point of

The Algorithm

```
create STree(ε)
s ← root
k ← 1
for i ← 1 to n do
   (s,k) ← update(s,(k,i))
   (s,k) ← canonize(s,(k,i))
```

Transforms STree(T^{i-1}) into STree(T^i).
Input: (s,(k,i)) s.t. (s,(k,i-1) is the active point of STrie(T^{i-1}).
Output: (s',k') s.t. (s',(k',i-1) is the endpoint of STrie(T^{i-1}).

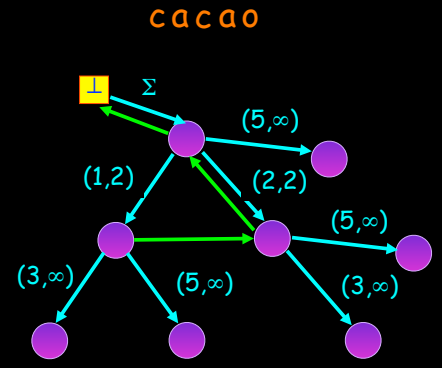
Input: a reference pair (s,(k,p)) for some state r.
Output: (s',k') s.t. (s', (k',p)) is the canonical reference pair for r.

update(s,(k,i))

Input: the canonical reference pair for some state r, and t_i.
Output: true/false if r is the endpoint or not, and the explicit state r (creating it if needed).

```
old-r ← root
(endpoint,r) ← test-and-split(s,(k,i-1),t_i)
while not endpoint do
   create new state r'; g'(r,(i,∞)) ← r'
   if old-r ≠ root then f'(old-r) ← r
   old-r ← r
   (s,k) ← canonize(f'(s),(k,i-1))
   (endpoint,r) ← test-and-split(s,(k,i-1),t_i)
if old-r ≠ root then f'(old-r) ← s
return (s,k)
```


update



s = ⊥

k = 5

i = 5

test-and-split(s,(k,p),t)

if $k \le p$ then

find the t_k-transition $g'(s,(k',p')) = s'$ from s

if $t = t_{k'+p-k+1}$ then return (true,s)

else

create a new state r

replace $g'(s,(k',p')) = s'$ by $g'(s,(k',k'+p-k)) = r$

and $g'(r,(k'+p-k+1,p')) = s'$

return (false,r)

else if $\neg\exists$ t-transition from s then return (false,s)

canonize(s,(k,p))

if $p < k$ then return (s,k)

else

find the t_k-transition $g'(s,(k',p')) = s'$ from s

while $p' - k' \le p - k$ do

$k \leftarrow k + p' - k' + 1$

$s \leftarrow s'$

if $k \le p$ then

find the t_k-transition $g'(s,(k',p')) = s'$ from s

return (s,k)

Running Time

Theorem: The running time of the algorithm is $O(n)$.

Proof: We divide the running time into two components:

1. The total time of the procedure canonize.
2. The rest.

update

Called n times

old-r ← root

(endpoint,r) ← test-and-split(s,(k,i-1),t_i)

while not endpoint do

In each execution of the loop, a new state is created.

create new state r'; $g'(r,(i,\infty)) \leftarrow r'$

if old-r ≠ root then f'(old-r) ← r

old-r ← r

(s,k) ← canonize(f'(s),(k,i-1))

(endpoint,r) ← test-and-split(s,(k,i-1),t_i)

O(1)

if old-r ≠ root then f'(old-r) ← s

canonize

Called O(n) times

```
if p < k then return (s,k)
else
    find the t_k-transition g'(s,(k',p')) = s' from s
    while p' - k' ≤ p - k do
            k ← k + p' - k' + 1
            s ← s'
            if k ≤ p then
                find the t_k-transition g'(s,(k',p')) = s'
    from s
```

In each execution of the loop, the value of k increases.

Applications - Exact String Matching

Input: two strings: a text T and a pattern P.

Output: all the occurrences of P in T.

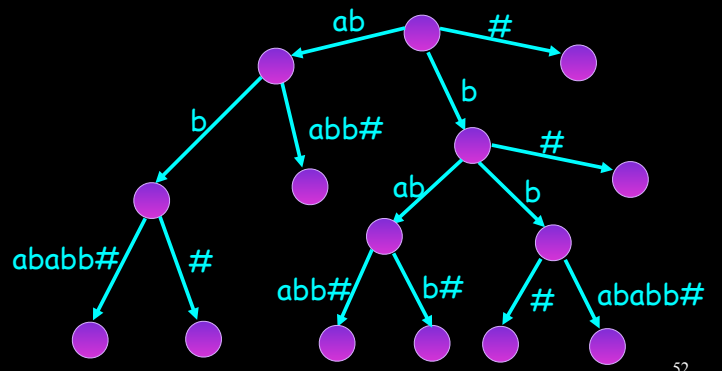
This problem can be solved in O(|T|+|P|) time (Boyer-Moore, Knuth-Morris-Pratt).

Applications - Exact String Matching (cont.)

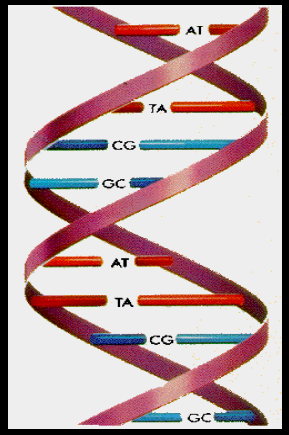
- We look at the case where we have a text T first, and then a sequence of patterns $P_1,\ldots,P_r$.
- This problem can be solved using suffix trees.
  - Preprocessing time: O(|T|).
  - Finding a pattern P: O(|P|+k), where k is the number of occurrences of P in T.

Applications - Exact String Matching (cont.)

abbabab



Applications in Biology



Finding Repeats in DNA

- The DNA contains many repetitive sequences with different biological functions.
- We want to find all maximal repeats in a DNA sequence.

ACCAGTTCGCGCATGAACGTTCGACCGGTTCGAT

Finding Repeats in DNA (cont.)

Theorem: All maximal repeats in a sequence T can be found in O(|T|) time using suffix trees.

Finding Repeats in DNA (cont.)

Lemma: If w is a maximal repeat in T, then the state w in STree(T) is explicit.

Proof: If w is a maximal repeat then there are at least two occurrences of w in T s.t. the character following w is different. Thus w is a branching state, and therefore it is explicit.

Finding Repeats in DNA (cont.)

Corollary: There are at most $O(|T|)$ maximal repeats in T.

Proof: By the above lemma, each maximal repeat corresponds to an explicit state. Since $STree(T)$ has $O(|T|)$ explicit states, T has $O(|T|)$ maximal repeats.

Finding Repeats in DNA (cont.)

Definition: The left character of a leaf $t_i \ldots t_n$ of $STree(T)$ is t_{i-1}.

Definition: A node w of $STree(T)$ is called left diverse if there are at least two leaves in w's subtree with different left characters.

Note that, by definition, a left diverse node is not a leaf.

Finding Repeats in DNA (cont.)

Lemma: A substring w of T is a maximal repeat iff w is a left diverse explicit state in $STree(T)$.

Finding Repeats in DNA (cont.)

Proof:

1. Suppose w is a maximal repeat.
   i. By the previous lemma w is explicit.
   ii. $\exists a \neq b \in \Sigma$ s.t aw and bw are substrings of T. Let awu and bwv be the corresponding suffixes.
   $\Rightarrow$ wu and wv are two leaves in the subtree of w with different left characters.

Finding Repeats in DNA (cont.)

2. Suppose that w is explicit and left diverse.

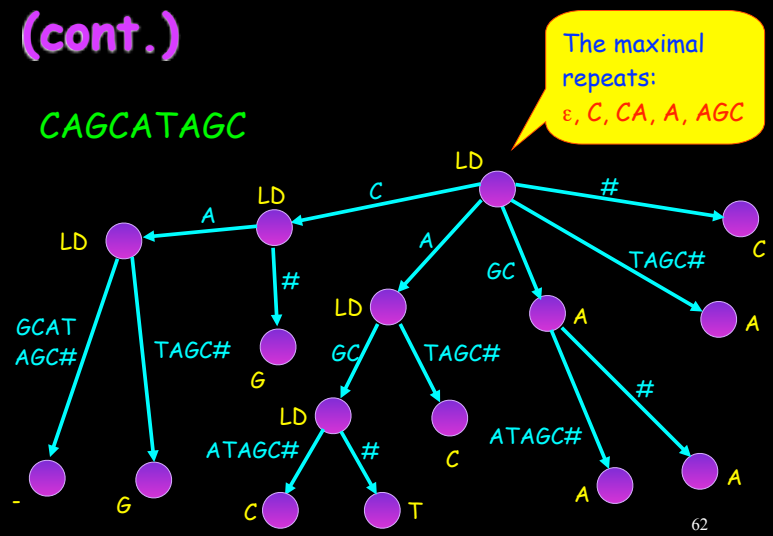
aw bw

(i) awc bwd

(ii) awc bwc

wd

Finding Repeats in DNA (cont.)



Bibliography

- On-Line Construction of Suffix Trees
  E. Ukkonen

- Algorithms on String, Trees, and Sequences
  Dan Gusfield