7.7. 7.8. 7.9.
Second Applications of Suffix Trees

2015. 06. 09

YunJin Choi

Second Application of Suffix Trees

- **APL7 : Building a smaller directed graph for exact matching**
- **APL8 : A reverse role for suffix trees, and major space reduction**
- **APL9 : Space-efficient longest common substring algorithm**

Second Application of Suffix Trees

- **APL7 : Building a smaller directed graph for exact matching**
- **APL8 : A reverse role for suffix trees, and major space reduction**
- **APL9 : Space-efficient longest common substring algorithm**

Introduction

- **In many applications**
  - Space is the critical constraint
  - Any significant reduction in space is of value

- **In this section**
  - We consider how to compress a suffix tree into a directed acyclic graph (DAG)
    - solve the exact matching problem (and others) in linear time
    - using less space than the tree

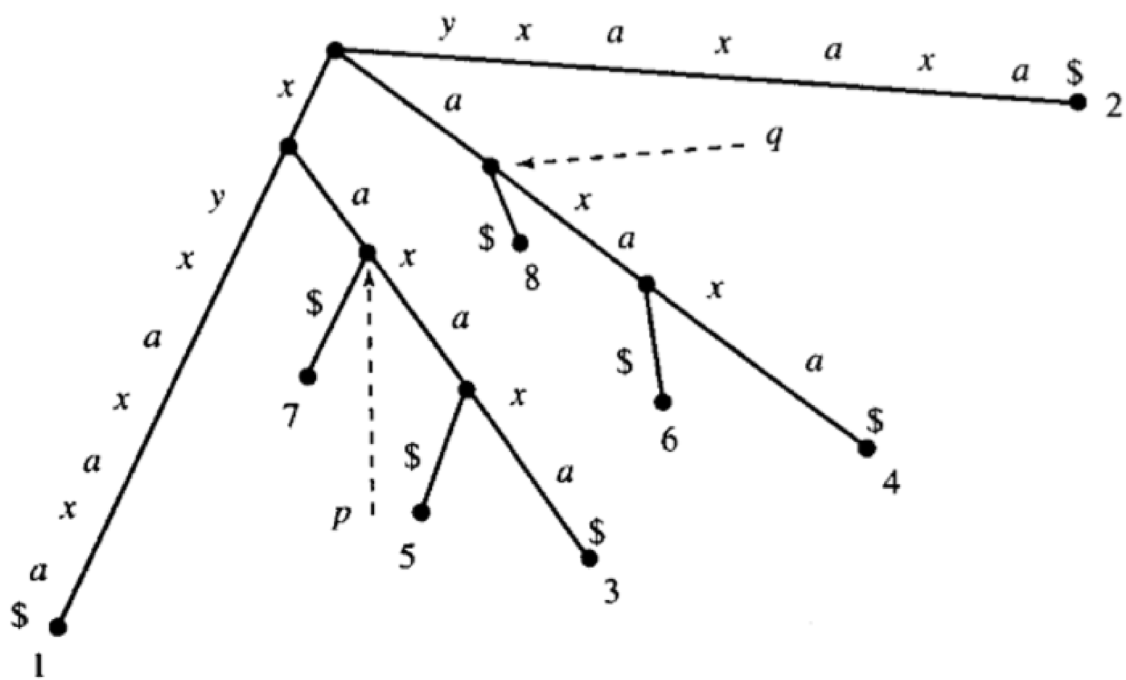
Problem

- **Problem**
  - Determine whether a pattern occurs in a larger text
    - rather than learning all the locations of the pattern occurrence(s)

  - We could merge a subtree into another subtree
    - by redirecting the labeled edge
    - by deleting the subtree

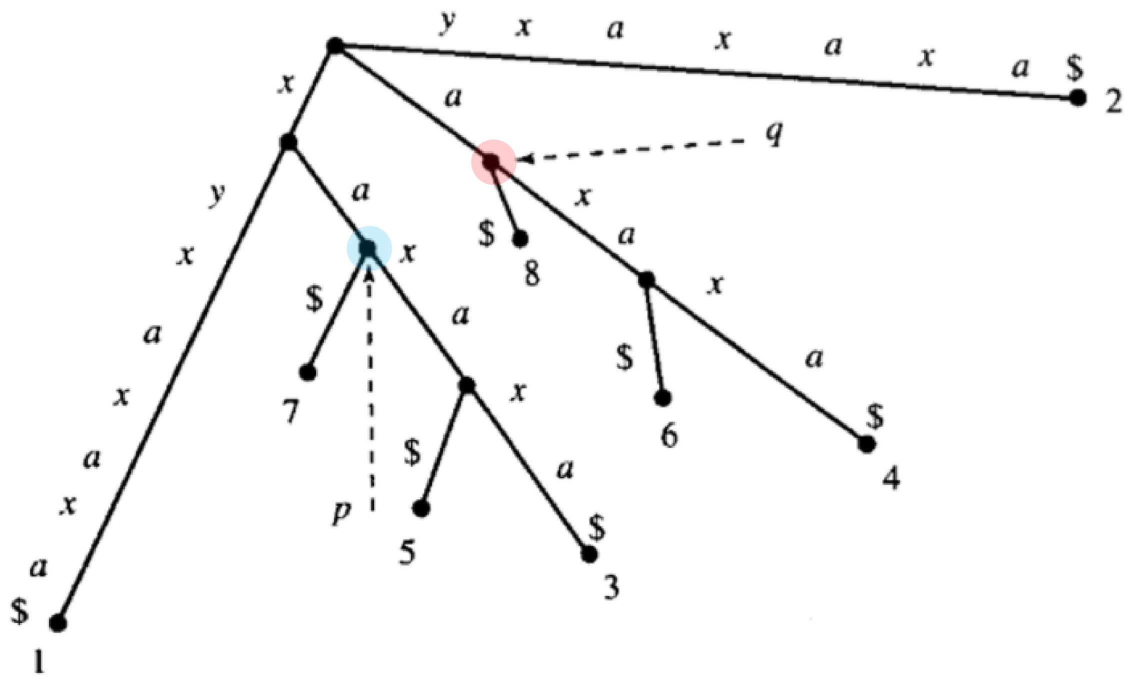
Example

- **Example**
  - *S = xyxaxaxa*



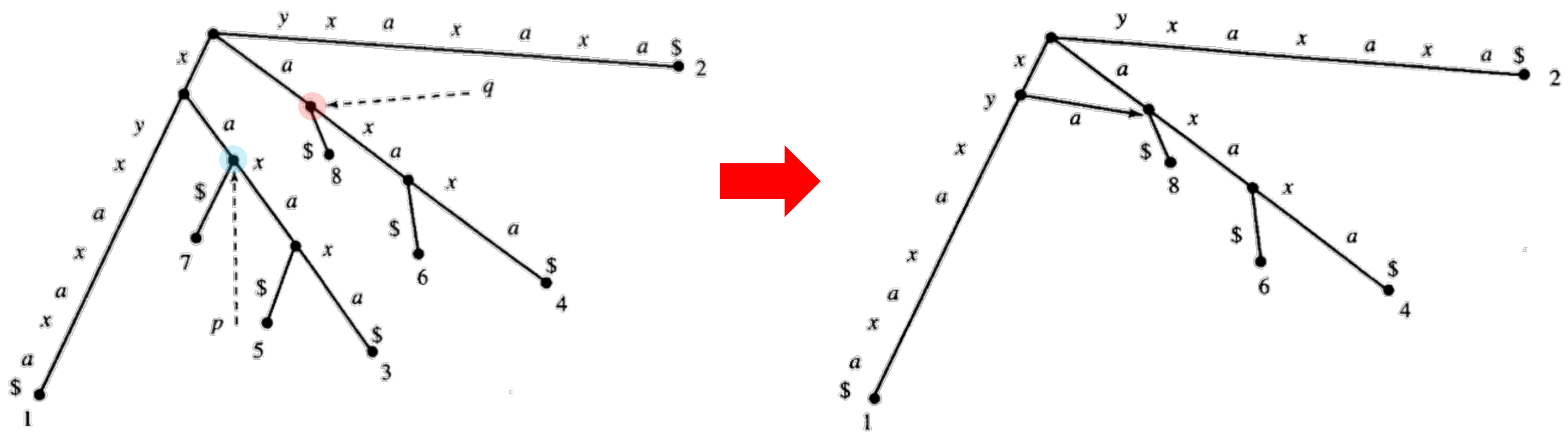
Example

- **Example**
  - The edge-labeled subtree below node *p* is *isomorphic* to the subtree below node *q*, except for the leaf numbers
    - That is, for every path from *p* there is a path from *q* with the same path-labels.



Example

- **Example**
  - We could merge *p* into *q*
    - by redirecting the labeled edge from *p*'s parent to go into *q*, deleting the subtree of *p*



- However, the leaf numbers
  - may no longer give the exact starting positions of the occurrences

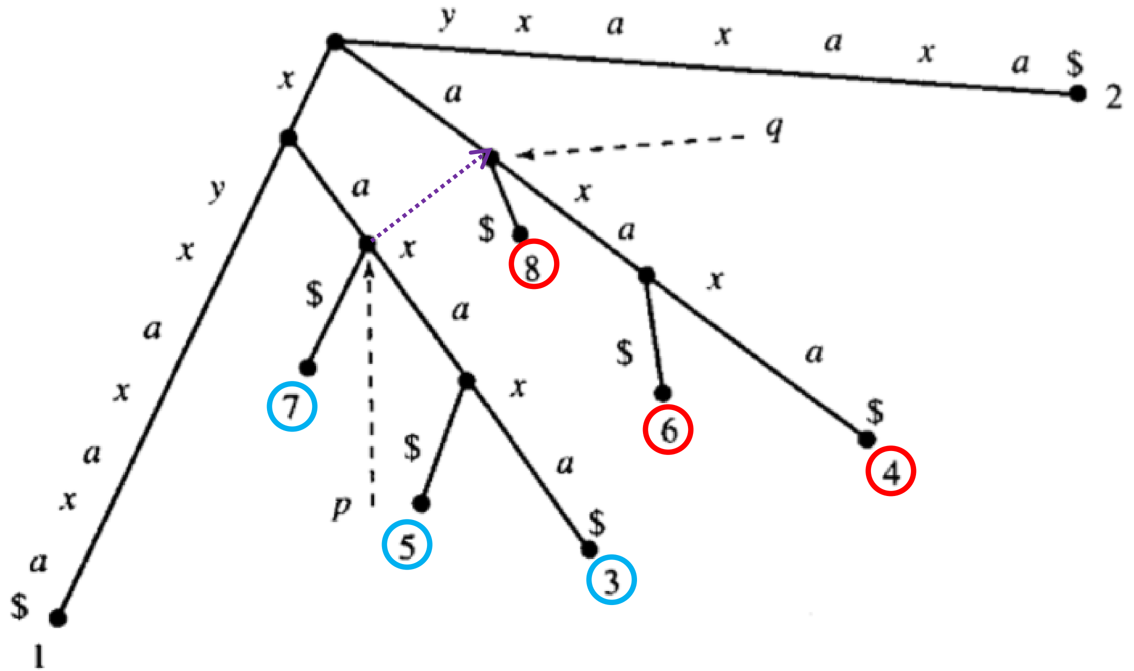
The key algorithmic issue

- **The key algorithmic issue**
  - How to find isomorphic subtrees in the suffix tree

  - **Theorem 7.7.1**
    - In a suffix tree $\mathcal{T}$ the edge-labled subtree below a node p is ***isomorphic*** to the subtree below a node q if and only if
      1. there is a directed path of suffix links from one node to the other node
      2. the number of leaves in the two subtrees is equal

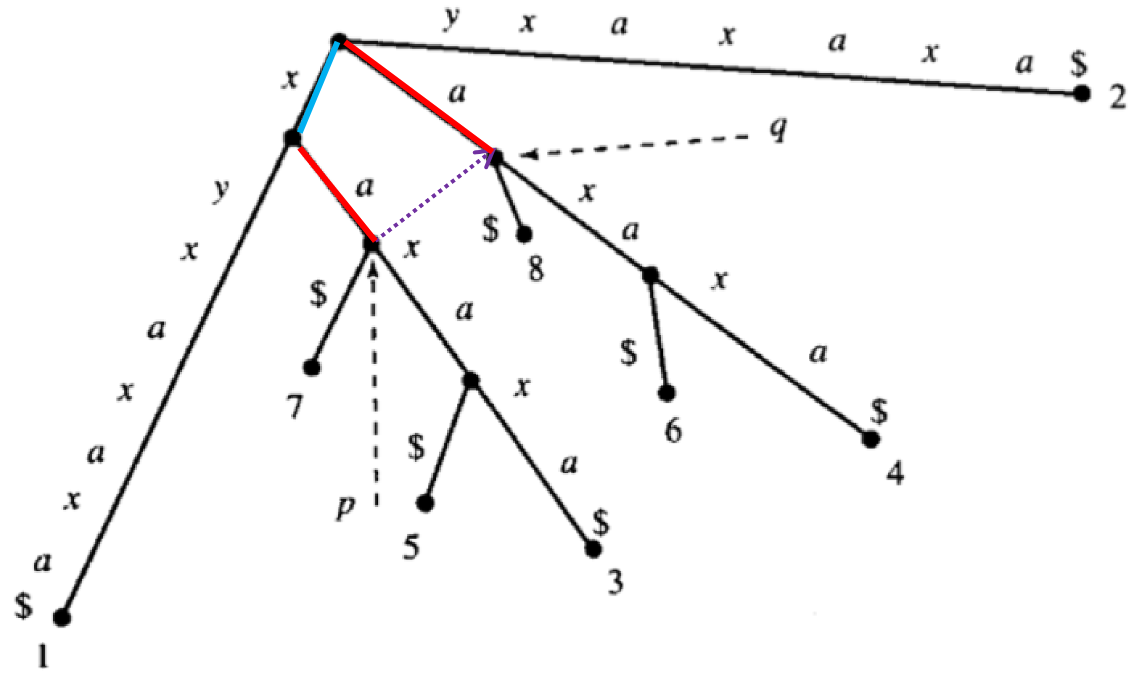
The key algorithmic issue

- **Proof (if statement)**
  - **Suppose *p* has a direct suffix link to *q*,** and those two nodes have the **same number of leaves** in their subtrees.



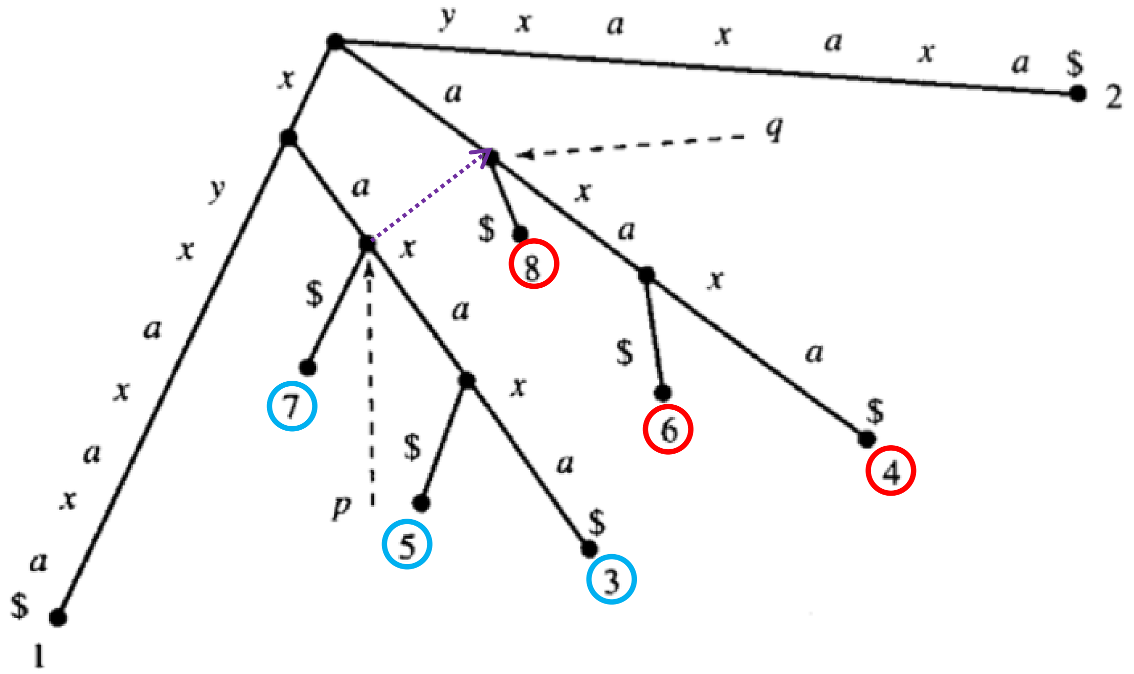
The key algorithmic issue

- **Proof (if statement)**
  - Node *p* has path-label *x*α while *q* has path-label α



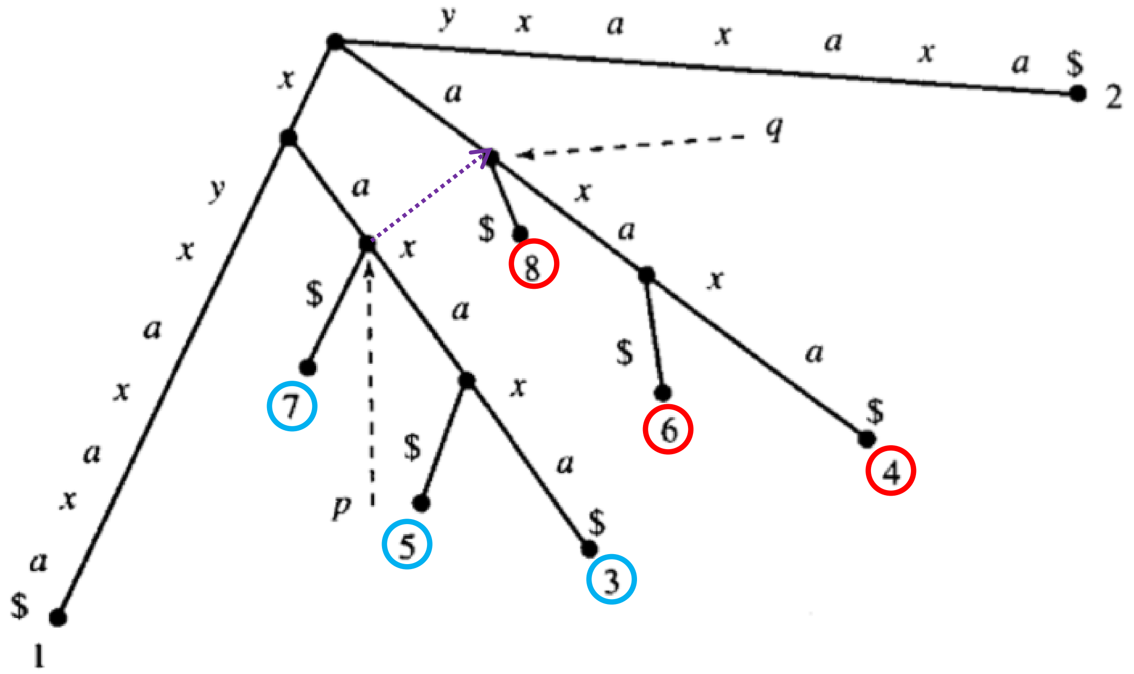
The key algorithmic issue

- **Proof (if statement)**
  - For every leaf numbered i in the subtree of p
    - there is a leaf numbered $i + 1$ in the subtree of q



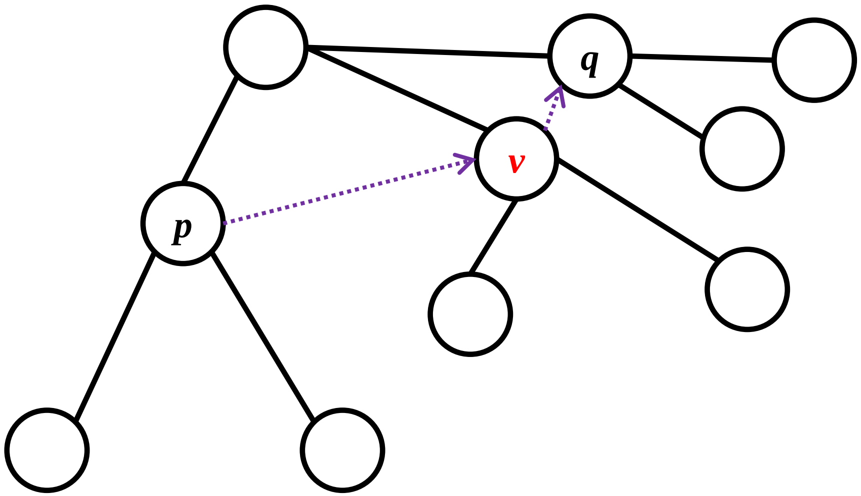
The key algorithmic issue

- **Proof (if statement)**
  - Therefore, for every path from *p* to a leaf, there is an identical path from *q* to a leaf
  - Hence the two subtrees are **isomorphic**



The key algorithmic issue

- **Proof (if statement)**
  - If there is a path of suffix links from p to q going through a node v
    - $|p| \leq |v| \leq |q|$
  - If p and q have the same number of leaves, then all the subtrees have the same number of leaves
  - All these subtrees are isomorphic to each other



The key algorithmic issue

- **Proof (only if statement)**
  - Suppose that the subtrees of *p* and *q* are isomorphic

    1. **there is a directed path of suffix links from one node to the other node**

    2. the number of leaves in the two subtrees is equal

       -> Clearly they have the same number of leaves

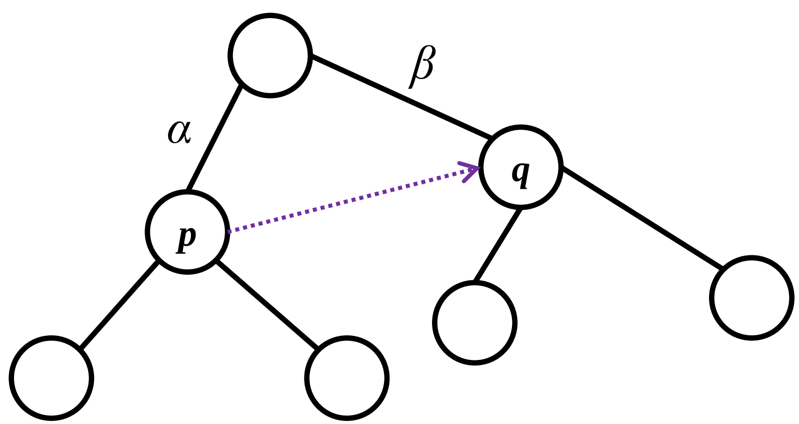
The key algorithmic issue

- **Proof (only if statement)**
  - Assume that $|\beta| \leq |\alpha|$

    α is the path-label of p

    β is the path-label of q

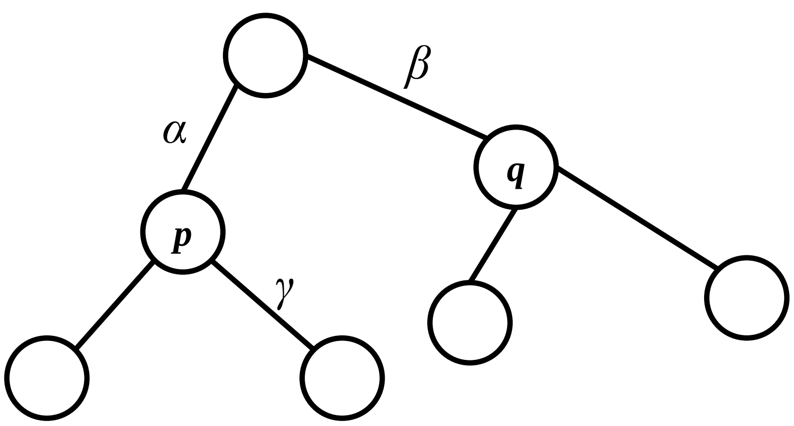
A. If β is a suffix of α
   - it must be a proper suffix (since α != β)
   - Then by properties of suffix links,
     - there is a directed path of suffix links from *p* to *q*



The key algorithmic issue

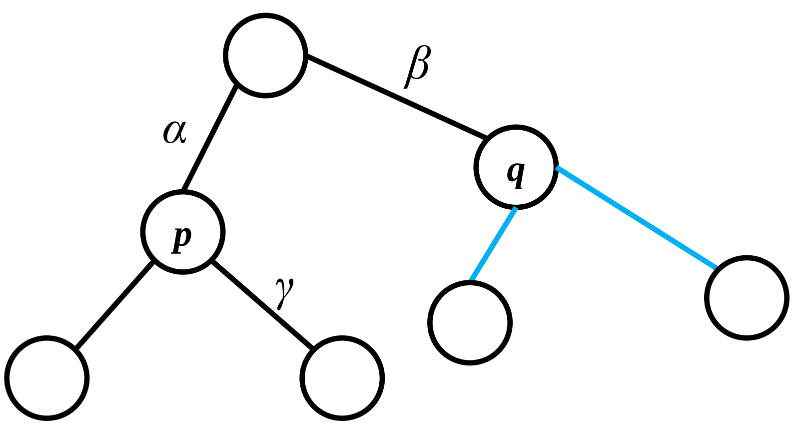
- **Proof (only if statement)**
  - Now we will prove that β must be a suffix of α
    - by contradiction

B. Suppose β is not a suffix of α
  - Let γ be the suffix of T just to the right of α
    - That means that $\alpha\gamma$ is a suffix of T



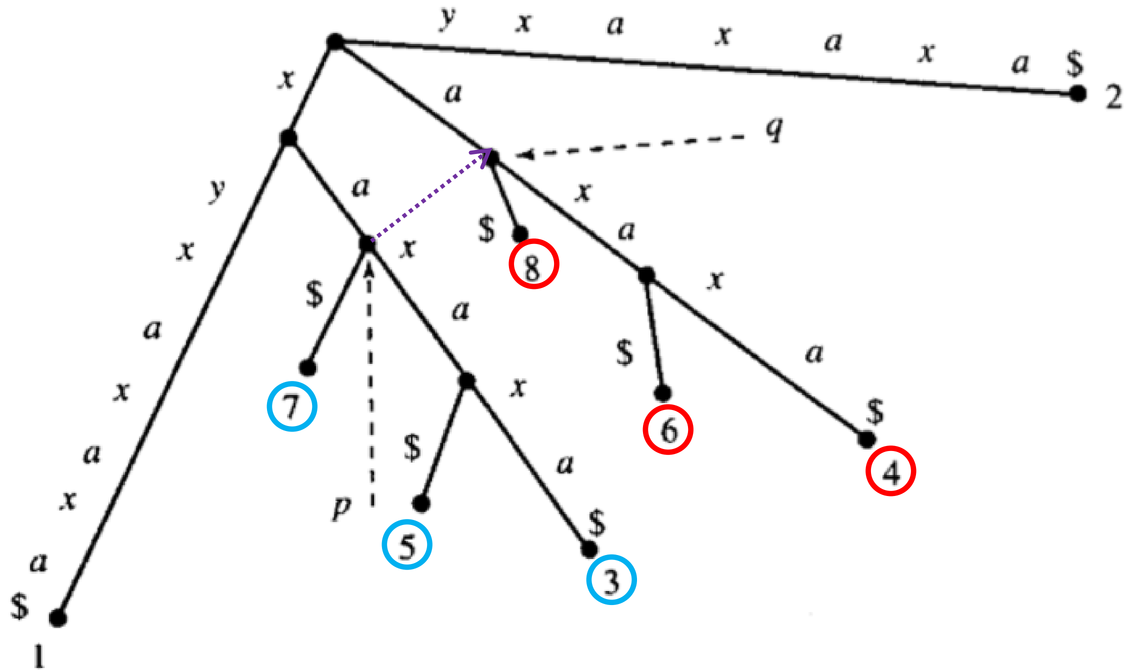
The key algorithmic issue

- **Proof (only if statement)**
  - Since β is not a suffix of α
    - there is no path of length $|\gamma|$ from q to a leaf
  - Therefore, the subtrees rooted at p and at q are not isomorphic
    - which is a contradiction



Suffix tree compaction

- **Definition**
  - let ***Q*** be the set of all **pairs (*p*, *q*)** such that
    - A. there exists **a suffix link from *p* to *q* in *T***
    - **B. *p* and *q* have the same number of leave**s in their respective subtrees



Suffix tree compaction

- **The entire procedure to compact a suffix tree**

Suffix tree compaction

begin

Identify the set Q of pairs (p, q) such that there is a suffix link from p to q and the number of leaves in their respective subtrees is equal.

While there is a pair (p, q) in Q and both p and q are in the current DAG,
Merge node p into q.

end.

Suffix tree compaction

- **Correctness**
  - **Theorem 7.7.2**
    - Let $\mathcal{T}$ be the suffix tree for an input string S
    - Let D be the DAG resulting from running the compaction algorithm on $\mathcal{T}$
    - Any directed path from the root in D
      - enumerates a substring of S
    - and every substring of S is
      - enumerated by some such path
    - Therefore, the problem of determining whether a string is a substring of S
      - can be solved in linear time using D instead of $\mathcal{T}$.

Suffix tree compaction

- **DAG *D* can be used**
  - to determine whether a pattern occurs in a text
  - but the graph seems to lose the location(s) where the pattern begins
- **It is possible**
  - to add simple (linear-space) information to the graph
  - so that the locations of all the occurrences can also be recovered
- **In the algorithm**
  - Pairs are merged in arbitrary order

DAGs versus DAWGs

- **DAWG**
  - represents a finite-state machine
  - and each edge label is allowed to have only one character

  - Moreover, the main theoretical feature of the DAWG for a string *S*
    - is that it is the finite-state machine with the fewest number of states (nodes)
    - that recognizes suffixes of *S*

  - Still, DAG *D* for string *S* has as few (or fewer) nodes and edges than DAWG for S
    - so is as compact as the DAWG
  - Therefore, construction of the DAWG for *S* is mostly of theoretical interest

Second Application of Suffix Trees

- APL7 : Building a smaller directed graph for exact matching
- **APL8 : A reverse role for suffix trees, and major space reduction**
- APL9 : Space-efficient longest common substring algorithm

Introduction

- **Exact matching problem**
  - Suffix tree
    - Preprocessing time and space: $O(n)$
      - n: length of the text
    - Search time: $O(m+k)$
      - m: length of the pattern
      - k: the number of occurrences
  - KMP (or Boyer-Moore)
    - Preprocessing time and space: $O(m)$
      - m: length of the pattern
    - Search time: $O(n)$
      - n: length of the text

Introduction

- **Exact set matching problem**
  - Suffix tree
    - Preprocessing time and space: $O(n)$
      - n: length of the text
    - Search time: $O(m+k)$
      - m: total length of all the patterns
      - k: the number of occurrences
  - Aho-Corasick
    - Preprocessing time and space: $O(m)$
      - m: total length of all the patterns
    - Search time: $O(n+k)$
      - n: length of the text
      - k: the number of occurrences

Introduction

- **Suffix tree methods that preprocess the text**
  - as efficient as the methods that preprocess the pattern
  - $O(n+m)$ time and $\Theta(n+m)$ space

  - However, the practical constants for suffix trees
    - unattractive compare to the other methods

  - Moreover, the situation that the pattern(s) will be given first and held fixed while the text varies

  - Solve those problems by building a suffix tree for the pattern(s)

    $\Rightarrow$ **the reverse of the normal use of suffix trees**

Matching statistics

- **Definition**
  - $ms(i)$
    - the length of the longest substring of T starting at position i
    - that matches a substring somewhere (but we don't know where) in P
    - These values are called the matching statistics
  - Ex)
    - $T = abcxabcdex$
    - $P = wyabcwzqabcdw$
    - $ms(1)$

Matching statistics

- **Definition**
  - $ms(i)$
    - the length of the longest substring of T starting at position i
    - that matches a substring somewhere (but we don't know where) in P
    - These values are called the matching statistics
  - Ex)
    - T = *abcxabcdex*
    - P = *wyabcwzqabcdw*
    - $ms(1) = 3$

Matching statistics

- **Definition**
  - $ms(i)$
    - the length of the longest substring of T starting at position i
    - that matches a substring somewhere (but we don't know where) in P
    - These values are called the matching statistics

  - Ex)
    - $T = abcxabcdex$
    - $P = wyabcwzqabcdw$

    - $ms(5)$

Matching statistics

- **Definition**
  - $ms(i)$
    - the length of the longest substring of T starting at position i
    - that matches a substring somewhere (but we don't know where) in P
    - These values are called the matching statistics
  - Ex)
    - $T = abcxabcdex$
    - $P = wyabcwzqabcdw$
    - $ms(5) = 4$

Matching statistics

- **There is an occurrence of *P* starting at position *i* of *T***
  - if and only if $ms(i) = |P|$

  - Thus, the problem of finding the matching statistics
    - is a generalization of the exact matching problem

Matching statistics

- **Matching statistics**
  - can be used to reduce the size of the suffix tree
  - are central to a fast approximate matching method
    - designed for rapid database searching
  - provide one bridge
    - between exact matching and approximate string matching

How to compute matching statistics

- **Compute *ms*(*i*) for each position *i* in *T***
  - in $O(m)$ time
  - using only a suffix tree for P

  - Build a suffix tree $\mathcal{T}$ for P
    - but do not remove the suffix links

How to compute matching statistics

- **The naïve way**
  - Match the initial characters of $T[i \ldots n]$ against $\mathcal{T}$
  - by following the unique path of matches
  - until no further matches are possible

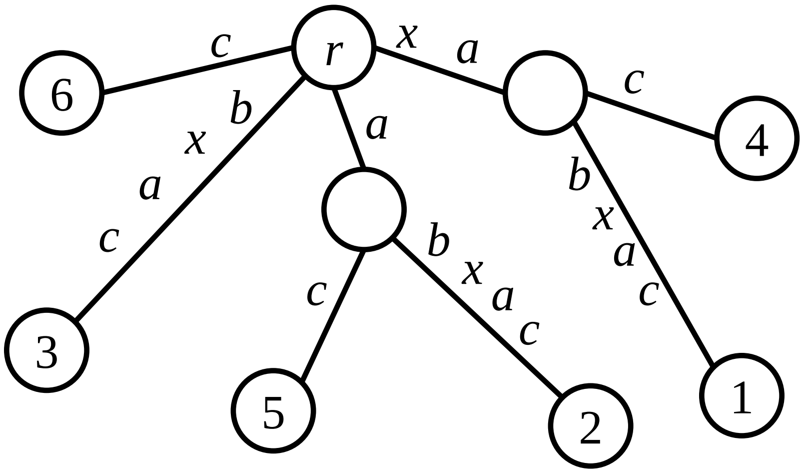
  - Repeating this for each i
    - not achieve the linear time bound

How to compute matching statistics

- **To accelerate the entire computation**
  - **The suffix links** are used
  - similar to the way they accelerate the construction of $\mathcal{T}$ in Ukkonen's algorithm

How to compute matching statistics

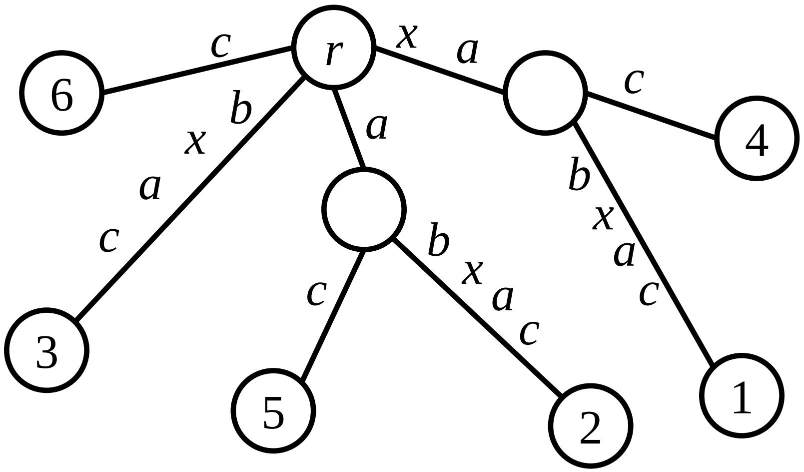
- **Example**
  - $T = cxabxat$
  - $P = xabxac$



How to compute matching statistics

- **Example**
  - T = *cxabxat*
  - P = *xabxac*

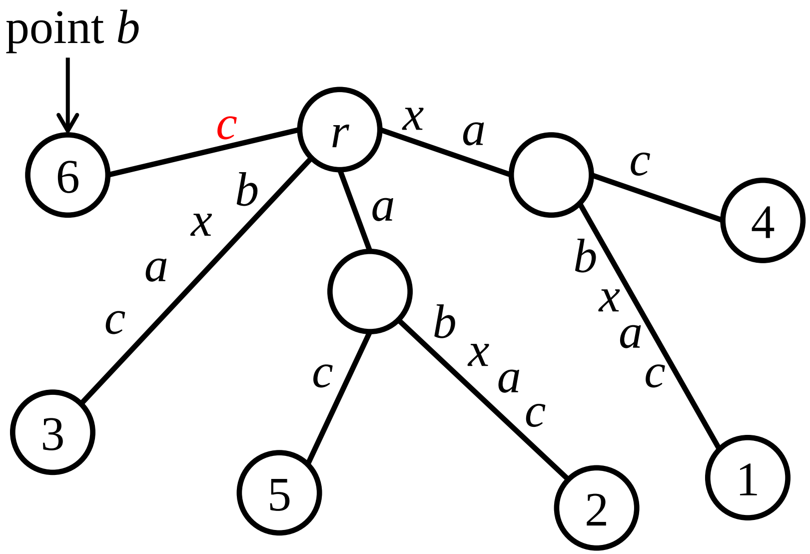
  - $ms(1)$



How to compute matching statistics

- **Example**
  - $T = cxabxat$
  - $P = xabxac$

  - $ms(1) = 1$



How to compute matching statistics

- **Compute *ms*(*i*+1)**
  A. If point *b* is an internal node *v*

  B. If point *b* is not an internal node

How to compute matching statistics

- **Compute *ms*(*i*+1)**

  A. If point b is an internal node v

  - can follow its suffix link to a node $s(v)$

  B. If point b is not an internal node

  - Back up to the node v just above b

    a. If v is the root

    b. If v is not the root

How to compute matching statistics

- **Compute *ms*(*i*+1)**
  A. If point *b* is an internal node *v*
     - can follow its suffix link to a node *s*(*v*)

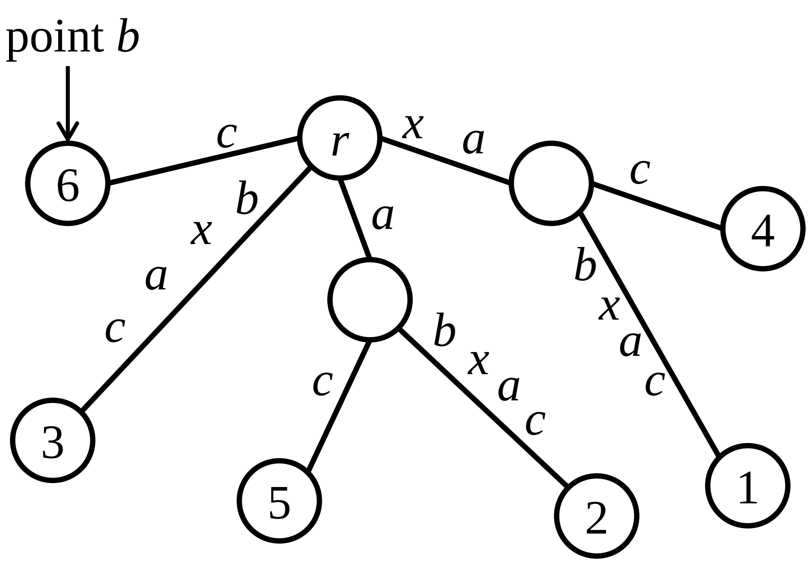
  B. If point *b* is not an internal node
     - Back up to the node *v* just above *b*

     a. If *v* is the root
        - begins at the root

     b. If *v* is not the root
        - follows the suffix link from *v* to *s*(*v*)

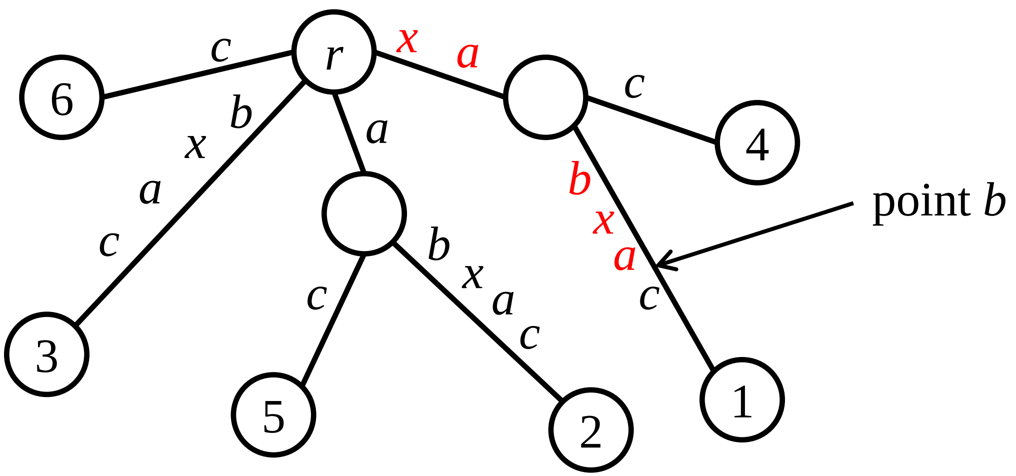
How to compute matching statistics

- **Example**
  - $T = cxabxat$
  - $P = xabxac$
  - $ms(2) =$



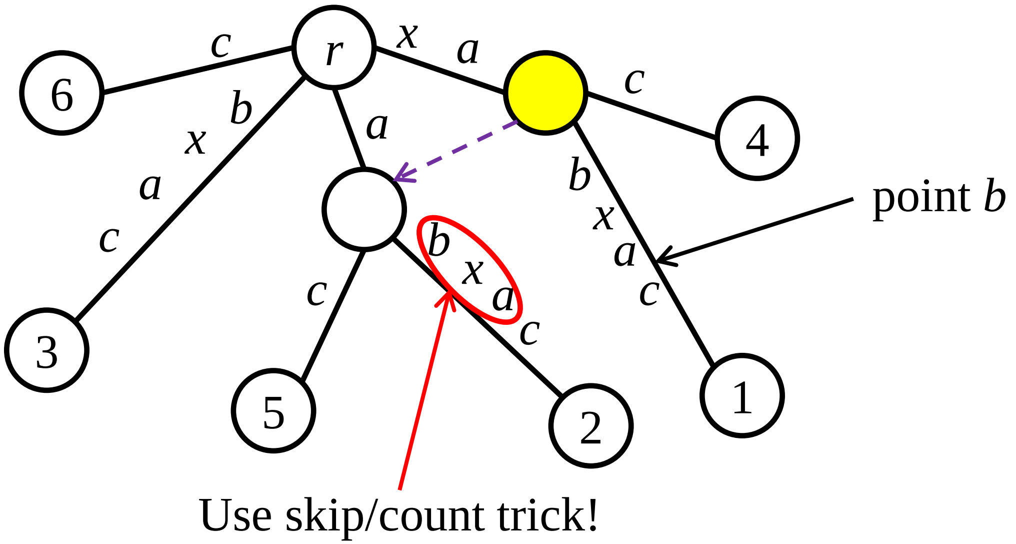
How to compute matching statistics

- **Example**
  - $T = cxabxat$
  - $P = xabxac$
  - $ms(2) = 5$



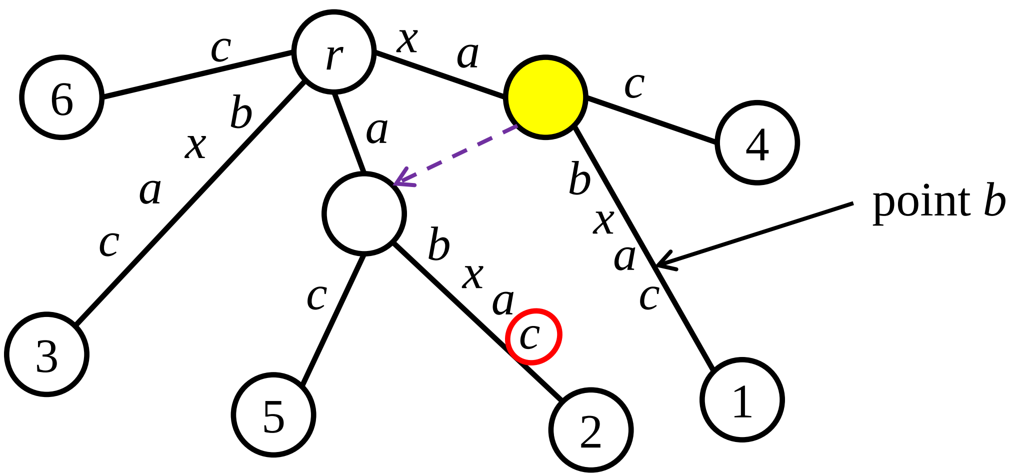
How to compute matching statistics

- **Example**
  - $T = cx$abxat
  - $P = xabxac$
  - $ms(3) =$



How to compute matching statistics

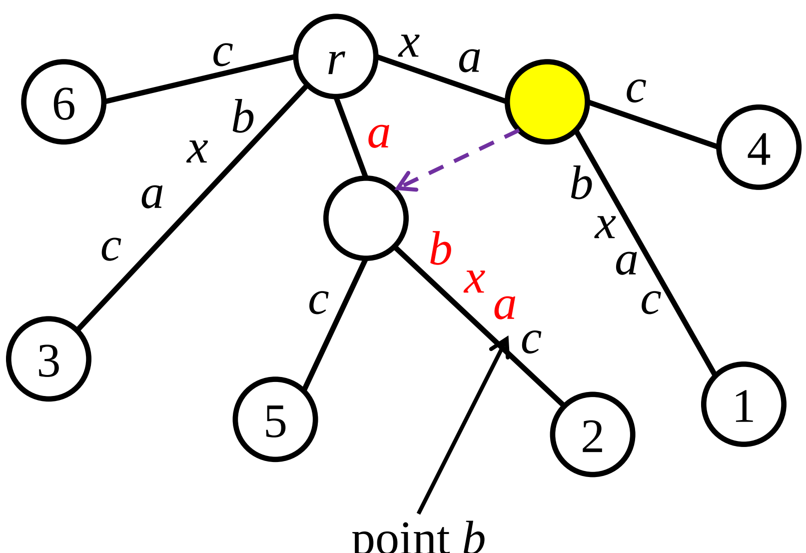
- **Example**
  - $T = cxabxac$
  - $P = xabxac$
  - $ms(3) =$



How to compute matching statistics

- **Example**
  - $T = cxabxat$
  - $P = xabxac$

  - $ms(3) = 4$



How to compute matching statistics

- **One special case that can arise in computing *ms*(*i*+1)**
  - If $ms(i) = 1$ or $ms(i) = 0$
    - (so that the algorithm is at the root)
  - and $T(i+1)$ is not in P

  - then $ms(i+1) = 0$

Correctness and time analysis for matching statistics

- **The proof of correctness of the method is immediate**
  - since it merely simulates the naïve method for finding each *ms*(*i*)

- **The proof of time**
  - very similar to that done for Ukkonen's algorithm

  - **Theorem 7.8.1**
    - Using only a suffix tree for *P* and a copy of *T*
    - All the *n* matching statistics can be found in *O*(*n*) time

Correctness and time analysis for matching statistics

- **Proof**
  - **Backing up**
    - Constant time per i
  - **Suffix link traverse**
    - Constant time per i

$O(n)$

Correctness and time analysis for matching statistics

- **Proof**
  - **The total time to traverse the various β path**

  - Each backup reduces the current depth by one
  - A link traversal reduces the current node depth by at most one
    - Ukkonen's algorithm

$\Rightarrow$ total decrement cannot exceed $2n$

  - But current depth cannot exceed n or become negative
  - The total increments to current depth are bounded by $3n$

$\Rightarrow$ total time for all β traversal is at most $3n$

Correctness and time analysis for matching statistics

- **Proof**
  - Total time used in all the character comparisons
    - done in the ‘after- β’ traversals
  - The ‘after- β’ character comparisons needed to compute *ms*(i+1)
    1. begin with the character in T that ended the computation for *ms*(i)
    2. or with the next character in T
  - Hence the after-β comparisons performed
    - when computing *ms*(i) and *ms*(i+1) share at most one character in common
  - At most 2n comparisons in total are performed during all the after- β comparisons

A small but important extension

- ***ms*(*i*)**
  - does not indicate the location of match in *P*
  - For some applications
    - We must also know the location of at least one such matching substring
- ***p*(*i*)**
  - For each position i in T,
    - the number $p(i)$ specifies a starting location in P
    - such that the substring starting at $p(i)$ matches a substring starting at position i of T for exactly $ms(i)$ places

A small but important extension

- **To accumulate the $p(i)$ values**
  - First do a depth-first traversal of $\mathcal{T}$
    - marking each node v with the leaf number of one of the leaves in its subtree
    - Takes time linear in the size of $\mathcal{T}$
  - Then, when using $\mathcal{T}$ to find each $ms(i)$
    - If the search stops at a node u
      - $p(i)$ is the suffix number written at u
    - If the search stops on an edge (u, v)
      - $p(i)$ is the suffix number written at v

Second Application of Suffix Trees

- **APL7 : Building a smaller directed graph for exact matching**
- **APL8 : A reverse role for suffix trees, and major space reduction**
- **APL9 : Space-efficient longest common substring algorithm**

Longest common substring

- **In section 7.4**
  - Solve the problem of finding the longest common substring of S_1 and S_2
  - by building a generalized suffix tree
  - That solution used $O(|S_1|+|S_2|)$ time and space

  - Problem
    - The practical space overhead required to construct and use suffix tree

Longest common substring

- **The longest common substring**
  - has length equal to the longest matching statistics $ms(i)$
  - The actual substring occurs
    - in the longer string starting at position i
    - in the shorter string starting at position $p(i)$
  - So, using only a suffix tree for the smaller of the two strings
  - The use of matching statistics reduces the space needed to solve the longest common substring problem.