Paradigm 5. Data Structure

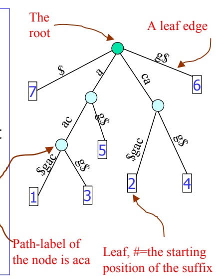
- Known examples: link table, heap,…
- Our lecture: suffix tree
  - Will involve amortize method that will be stressed shortly in this course

Suffix trees

- What is a suffix tree?
- Simple applications
- History
- Algorithms
- More applications

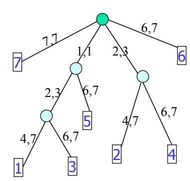
What is a suffix tree?

- A tree to represent all possible suffixes.
- E.g. S=acacag.
  - each edge is labeled with string.
  - every suffix is spelled out by a path from the root to a leaf,
  - and vice versa.
  - Why $?
    - To avoid some suffix that is a prefix of some other suffix



Suffix tree

- A suffix tree has exactly n leaves and at most $2n$-1 nodes.
- The label of each edge can be represented using 2 indices
- Thus, suffix tree can be represented using O(n log n) bits

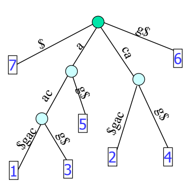


S=

a	c	a	c	a	g	$
1	2	3	4	5	6	7

Simple applications

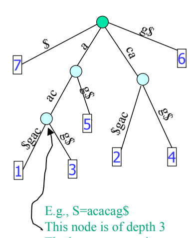
- Find all occurrences of a give Q in S
  - Start from the root, go down along the unique path Q to reach a point x.
  - All the leaves below x are the occurrences of Q.
- Time: O(|Q|+#occurrences)



E.g., S=acacag$
Q=ac
Occurrences:1, 3

Simple applications

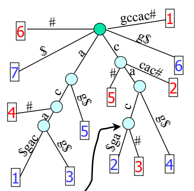
- Find the longest repeated substring in S
  - Find the deepest internal node.
    - Word "deep" is in terms of the length the string the node represents
- Time: O(n)



E.g., S=acacag$
This node is of depth 3
The longest repeat is aca

Simple applications

- Find the longest common substrings of two or more strings
  - About 30 yrs ago, Knuth conjecture that a linear time algorithm For this problem is impossible.
  - Suffix tree allows a linear time solution
  - E.g. consider two strings S_1 and S_2
    - Build a generalized suffix tree for both S_1# and S_2#
    - Mark every internal node that embraces leaves representing suffixes of S1 and S2
    - Report the deepest marked node



E.g., S_1=acacag$, S_1=gccac#
Report this node--cac

Straightforward construction of suffix trees

- Consider S[1..n], where S[n]=$
- Algorithm:
  - Initialize the tree with only root
  - For i=n to 1 do
    - Includes S[i..n] into the tree
- Time: O(n^2), quadratic time

Less than quadratic time?

- Yes, we can do it in O(n) time
- Short history of suffix trees.
  - 1973, Weiner's algorithm, called position tree
    - Linear time for constant size alphabet, but much space
    - Knuth has called it "the algorithm of 1973"
  - 1976(JACM), McGreight's
    - Linear time for constant size alphabet, quadratic space
  - 1995(Algorithmica), Ukkonen's algorithm
    - On line, linear time for constant size alphabet, linear space
  - 1997(FOCS), Farach's algorithm
    - Linear time for general alphabet

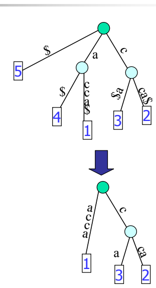
Today, we discuss Ukkonen's algorithm, which is the conceptually easiest linear-time construction alg.

The idea of Ukkonen's alg.

- Construct a sequence of implicit suffix trees;
- The last of which is converted to a true suffix tree.

Implicit suffix trees

- A implicit suffix tree is obtained by
  - Removing $ from edge labels
  - Removing edges with no label
  - Removing nodes with only one child
- E.g. S=acca$



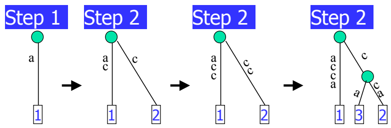
Ukkonen's alg. at high level

Denote T_i be the implicit suffix tree for S[1..i]

1. Construct T_1
2. For i=1 to n-1
   - /*phase i */
   - Construct T_{i+1} from T_i
3. Convert T_n to true suffix tree T.

Illustration

- S=acca



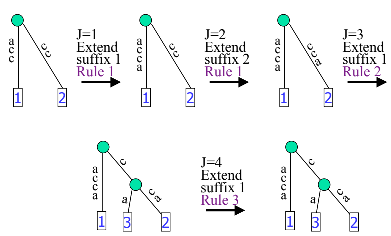
T_1 Phase 1 T_2 Phase 2 T_3 Phase 3 T_4

Phase i: Constructing T_{i+1} from T_i

- For j=1 to i+1
  - /*extension of the suffix S[j..i] */
  - Starting from the root, find the endpoint of the path labeled β= S[j..i]
  - Extend the path with character S[i+1]
    - Rule 1: if β ends at a leaf, S[i+1] is appended to the label of the last (leaf) edge
    - Rule 2: if every path from β starts with a character≠S[i+1], create a new leaf (and may also a new internal node) and a leaf edge labeled with S[i+1], number the created leaf i.
    - Rule 3: if some path from β starts with character S[i+1], do nothing

Example: From T_3 to T_4

S=acca, I=3



Summary

Algorithm:

1. Construct T_1
2. For i=1 to n-1

   /*phase I: $T_i \rightarrow T_{i+1}$ */

   - For j=1 to i+1

     /* extension of the suffix S[j..i] */

     - Locate the endpoint of the path β= S[j..i] O(|β|)
     - Extend the path with S[i+1] by the rules O(1)

Time:

- Each extension O(n) time,
- $O(n^2)$ extensions,
- Total time=$O(n^3)$.
- The algorithm may seem foolish since we already know a straightforward algorithm in $O(n^2)$ time.
- We will reduce $O(n^3)$ to $O(n)$ with
  - two observations, and
  - an implement trick.

Observation 1

- Consider phase i, once we apply rule 3 to extend S[j..i], then
  - rule 3 will be applied for extending S[k..i] for k=j+1,…,i
  - Thus, nothing to do for k=j+1,…,i
- Proof:
  - Since rule 3 is applied to extend S[j..i], a path labeled S[j..i] in the tree T_i must continue with character S[i+1].
  - Thus, there is also a path for S[k..i], followed by S[i+1].

Remark

- Based on the previous observation, in phase i, once we have applied rule 3, we can stop.
- This saves a lot of work.

Observation 2

- Once we add a leaf for a suffix in T_i, that leaf remains in T_{i+1}, T_{i+2},…
- Proof:
  - We never remove a leaf.

From the above we can infer

Fact 1: If in Phase i we have used rule 1 or 2 to extend A[j..i], then path A[j..i+1] will be in T_{i+1} and end at a leaf, and consequently, in Phase i+1, the extension for A[j..i+1] will use rule 1.

Remark

- In phase i, let j_i be the last extension involving a leaf.
- In other words, for extension due to k≤ j_i, we do not perform any rule 3 (I.e., all by rule 1 or 2).
- In phase i+1, when we perform an extension due to k≤ j_i,we always encounter a leaf at the end of S[k..i+1], thus, only rule 1 is applied (according to Fact 1 in Slide 19).

Algorithm for phase i

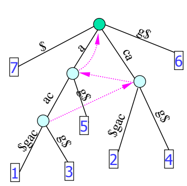
- /* for j=1…j_i, extension of j is based on rule 1, so we do nothing */
- For j=j_i +1…i+1,
  - Find the endpoint of the path from the root labeled with S[j..i]
  - Extend the path with character S[i+1] based on rule 1, 2, or 3
  - If we extend the path with rule 3
    - /* extension j' for j'=j+1…i+1 are based on rule 3. So we need to do nothing */
    - Set j_{i+1}=j-1; Break

Whole process

- Summary
  - Phase 1: compute extension 1..j_2+1.
  - Phase 2: compute extension j_2+1..j_3+1.
  - …
  - Phase i: compute extension j_i+1..j_{i+1}+1.
  - …
  - Phase n-1: compute extension j_{n-1}+1..j_n+1.
- In total we will do at most 2n extensions.
- For an extension due to j, it takes O(n) time because we need to find the endpoint of S[j..i].
- The total time is O(n^2).
- The process can be accelerated using suffix link.

Suffix link

- For an internal node v with path-label xα, if there is another node s(v) with path-label α, then we create a suffix link from v to s(v)

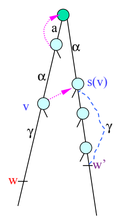


Is suffix link well defined?

- For a (implicit) suffix tree, every internal node (except the root) has a suffix link.
- Proof:
  - Consider any internal node v with path-label xα.
  - xα is the common prefix of S[i..n] and S[j..n]
    - The two leaves labeled i and j under v
  - α is the common prefix of S[i+1..n] and S[j+1..n]
  - Thus, there is an internal node u with path-label α.
  - Suffix link of v=u.

How to use suffix link?

- Assume before extension due to j, we maintain suffix links for all internal nodes.
- In the extension for j, we have located w which is the end of S[j..i].
- To start extension for j+1, we need to go to the end of S[j+1..i].
  1. From w, go up on edge to v
  2. Through suffix link, go to s(v)
  3. Go down until we find the end of S[j+1..i], say, w'
  4. If w has led to a newly created internal node, create a suffix link from w to w'



Time complexity

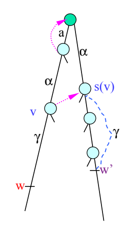
- Find the end of S[j+1..i]:
  - Step 1, 2, and 4 take O(1) time.
  - Step 3 takes amortized O(1) time. (?)
- So, each extension can be done in amortized O(1) time.
- As there are 2n extensions, the total time is O(n).

Why Step 3 takes amortized O(1) time?

- Step 3 is to walk down from node s(v) along a path labeled γ.
- There surely must be such a γ path from s(v).
- Direct implemented, this walk takes O(|γ|) time.
- A simple trick, called skip/count trick, will reduce the traversal time to O(# of edges on the path).
- So, define node-depth of u to be the # of edges on the path from the root to u. Our task is then to justify the above claim about skip/count and that
  - By amortization, each step 2 goes down O(1) edge.

The skip/count trick

- Let $g = |\gamma|$, $u = s(v)$
- Repeat
  - Find the edge $e=(u, u')$ whose first character=$\gamma[1]$.
  - Let $l=|label(e)|$
  - If $l<g$ then
    - $\gamma = \gamma[l+1,g]$; $g=g-l$; $u=u'$
  - Else
    - Skip to $label(e)[g]$; exit

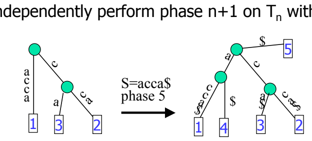


Step 3 go down amortized O(1) edges

- Note that for each extension,
  - Step 1 reduces the node-depth by 1
  - Step 2 reduces the node-depth by at most 1
  - Step 3 increases the node-depth
- Since there are 2n extensions,
  - All steps 1 and 2 can reduce the node-depth by at most 4n
- Since the maximum node-depth is n-1,
  - All steps 3 can at most increase the node-depth by 5n-1
  - By amortization, each step 3 goes down O(1) nodes.

Creating the true suffix tree

- Convert the implicit suffix tree T_n to true suffix tree in O(n) time
  - Append S with the terminal character $
  - Independently perform phase n+1 on T_n with S$.



Implicit T_4 Explicit T

More applications

- Maximum unique match. O(n)
  - Given two strings S_1 and S_2
  - Find all substrings *w* such that
    - *w* appear exactly once in both strings, and
    - *w* is maximal (I.e., any substring *x* including *w* cannot appear exactly once in both strings)
- Longest common prefix. O(n)
  - Given a string S[1..n], for i,j, the problem is to find the length of the longest common prefix of S[i,..n] and S[j..n]
- Maximum palindrome (最大回文)
- Palindrome is a string X s.t. $X=X^R$. e.g., level
  - The problem is to find the longest substring of S that is a palindrome.

10/5/2007 Algorithms @ CS, OUC 31

Additional applications

- Ziv-Lempel data compression
- Minimum length encoding of DNA
- All-pairs suffix-prefix matching
  - For
    - Recover DNA
    - Data compression
    - …
- …

10/5/2007 Algorithms @ CS, OUC 32