Algorithms in the Real World

Suffix Trees

Exact String Matching

- Given a text S of length m and pattern P of length n, "quickly" find an occurrence (or all occurrences) of P in S

- A Naïve solution:

  Compare P with S[i…i+n-1] for all i --- O(nm) time

- How about O(n+m) time? (Knuth-Morris-Pratt)
  - O(n) time to build a finite-state machine recognizing P (clever algorithm)
  - O(m) time to run S through finite-state machine

- How about O(m) preprocessing time and O(n) search time?

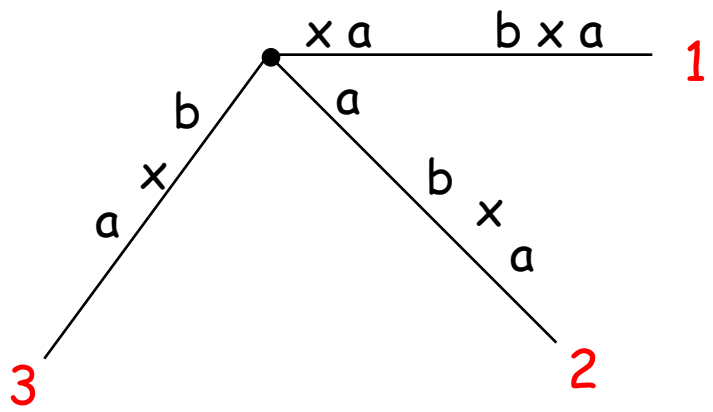
Suffix Trees

- Preprocess the text in O(m) time and search in O(n) time

- Idea:
  - Construct a tree containing all suffixes of text along the paths from the root to the leaves
  - For search, just follow the appropriate path: if the pattern occurs in the text, then it is a prefix of some suffix of the text.

Suffix Trees

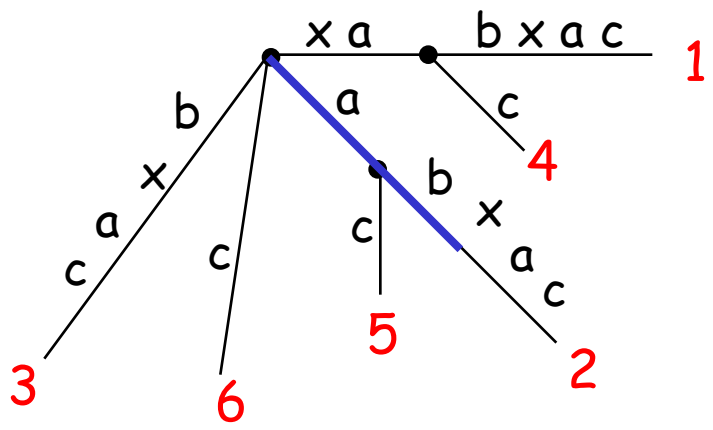
A suffix tree for the string x a b x a



Notice no leaves for suffixes xa or a

Suffix Trees

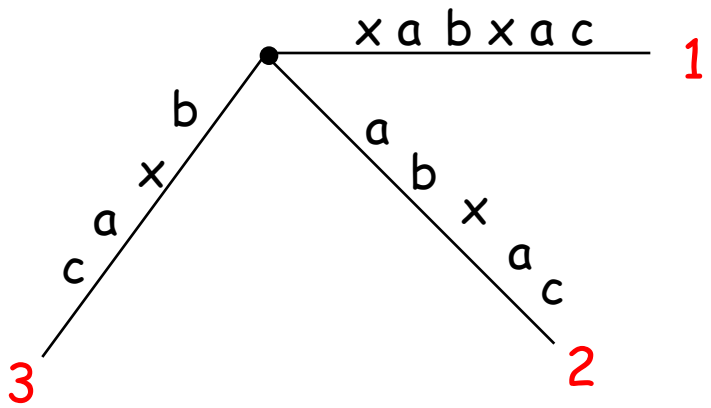
A suffix tree for the string x a b x a c



Search for the string a b x

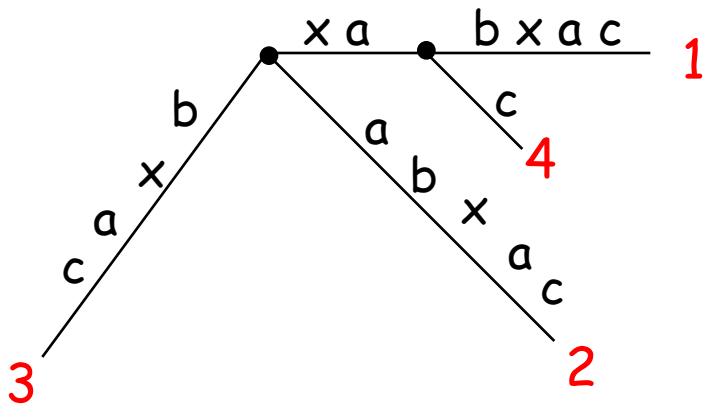
Constructing Suffix trees

- Naive $O(m^2)$ algorithm to extend string
- For every i, add the suffix S[i .. m] to the current tree



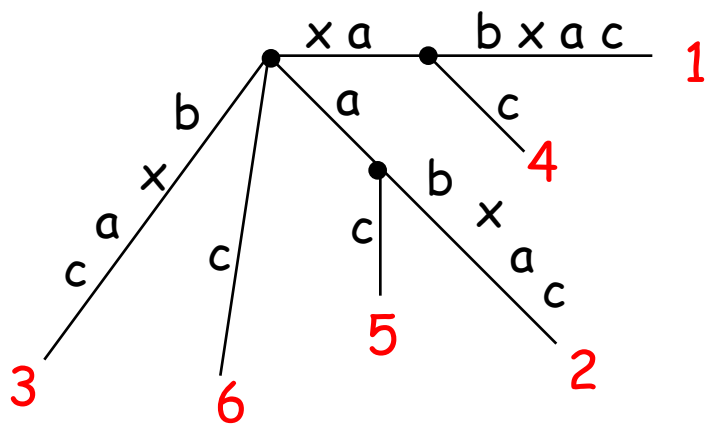
Constructing Suffix trees

- Naive $O(m^2)$ algorithm to extend string
- For every i, add the suffix S[i .. m] to the current tree



Constructing Suffix trees

- Naive $O(m^2)$ algorithm to extend string
- For every i, add the suffix S[i .. m] to the current tree



Ukkonen's linear-time algorithm

- We will start with an $O(m^3)$ algorithm and then give a series of improvements
- In stage i, we construct a suffix tree T_i for S[1..i]
- Building T_{i+1} from T_i naively takes $O(i^2)$ time because we insert each of the i+1 suffixes S[j..i+1] ($1 \leq j \leq i+1$)
- Thus a total of $O(m^3)$ time

Going from T_i to T_{i+1}

- In the jth substage of stage i+1, for j = 1 to i+1, we insert S[j..i+1] into T_i. Let S[j..i] = β.

- Three cases
  - Rule 1: The path β ends on a leaf $\Rightarrow$ add S[i+1] to the label of the last edge
  - Rule 2: The path β continues with characters other than S[i+1] $\Rightarrow$ create a new leaf node and split the path labeled β
  - Rule 3: A path labeled β S[i+1] already exists $\Rightarrow$ do nothing.

Idea #1 : Suffix Links

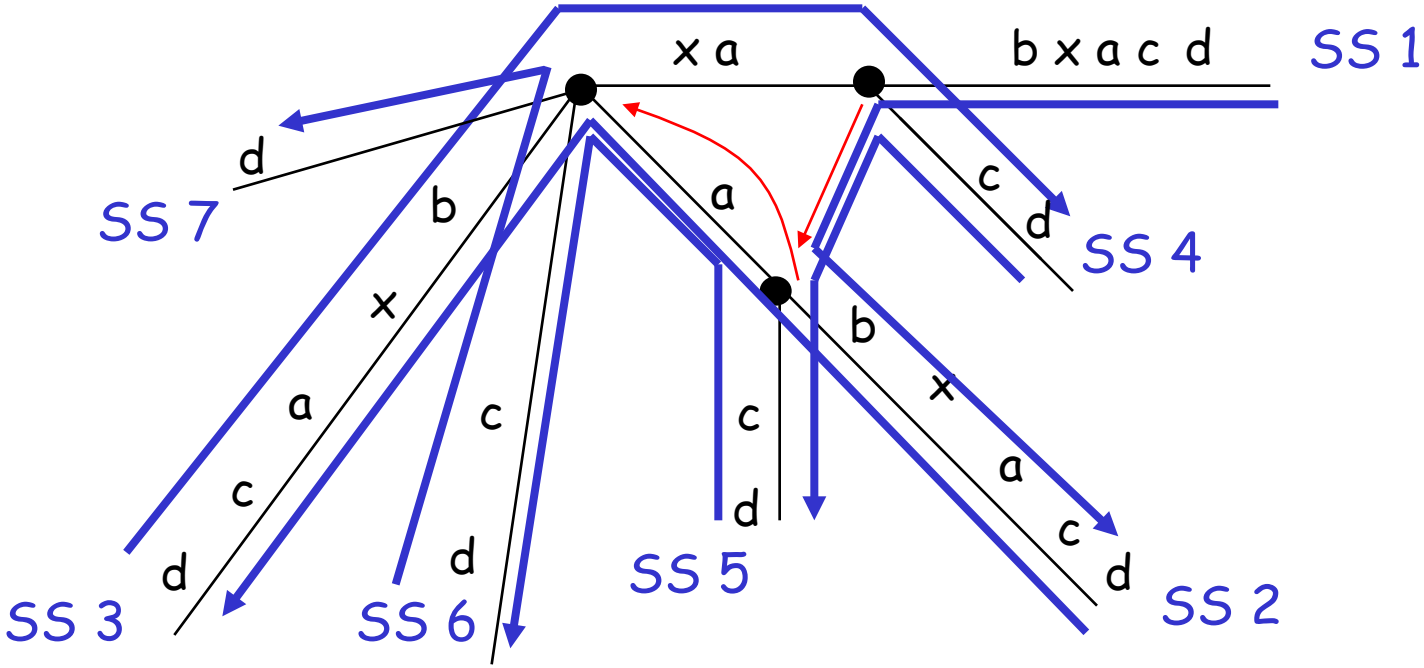
- Note that in each substage, we first search for some string in the tree and then insert a new node/edge/label
- Can we speed up looking for strings in the tree?

- Note that in any substage, we look for a suffix of the string searched in the previous substage
- Idea: Put a pointer from an internal node labeled $x\alpha$ to the node labeled α

- Such a link is called a "Suffix Link"

Idea #1 : Suffix Links

Add the letter d to the string x a b x a c
(SS stands for substage)

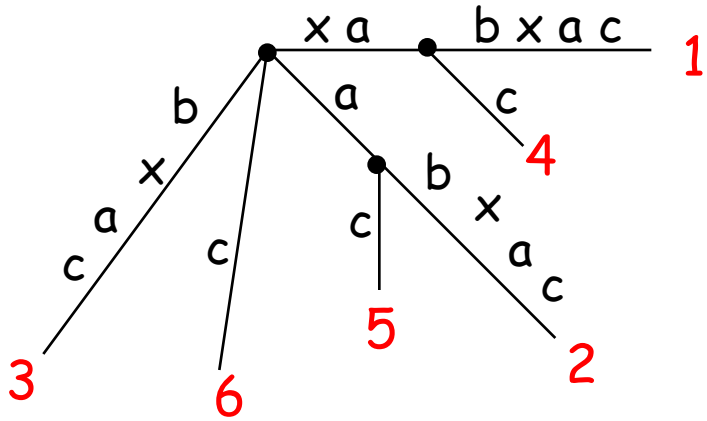


Suffix Links – Bounding the time

- Steps in each substage
  - Go up 1 link to the nearest internal node
  - Follow a suffix link to the suffix node
  - Follow path link for the remaining string
- First and second steps happen once per substage.
- Suffix links ensure that in third step, each character in S[1..i+1] is used at most once to traverse a downward tree edge to an internal node. Hence O(m) time over stage.
- Our example: (x a)SS1 b (x a) SS4 c d
- Thus the total time per stage is O(m)

Maintaining Suffix Links

- When an internal node labeled $x\alpha$ is created (where x is a single character), in the following substage an internal node labeled α is created. E.g., "x a" followed by "a" in substages 4 and 5:



Why? If $x\alpha$ is a suffix and a prefix of suffix $x\alpha\gamma$, but $x\alpha S[i+1]$ isn't a suffix, then α is also a suffix and a prefix of suffix $\alpha\gamma$, but $\alpha S[i+1]$ isn't a suffix.

- When a new internal node is created, add a suffix link from it to the root, and if required, add a suffix link from its predecessor to it.

Going from $O(m^2)$ to $O(m)$

- Can we even hope to do better than $O(m^2)$?
- Size of the tree itself can be $O(m^2)$ as shown so far
- But notice that there are at most 2m edges! Why? (at most m leaves, and all internal nodes have at least two children)

  (still O(m) even if we double count edges for all suffixes that are prefixes of other suffixes)
- Idea: represent labels of edges as intervals
- Can easily modify the entire process to work on intervals

Idea #2 : Getting rid of Rule 3

- Recall Rule 3: A path labeled S[j .. i+1] already exists) do nothing.
- If S[j .. i+1] already exists, then S[j+1 .. i+1] exists too and we will again apply Rule 3 in the next substage
- Whenever we encounter Rule 3, this stage is over – skip to the next stage.

Idea #3 : Fast-forwarding Rules 1 & 2

- Rule 1 applies whenever a path ends in a leaf
- Note that a leaf node always stays a leaf node – the only change is to append the new character to its edge using Rule 1
- An application of Rule 2 in substage j creates a new leaf node

  This node is then accessed using Rule 1 in substage j in all the following stages

Idea #3 : Fast-forwarding Rules 1 & 2

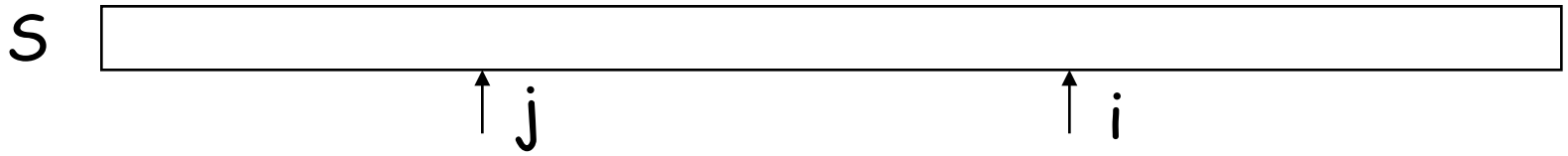
- Fast-forward Rule 1 and 2
  - Whenever Rule 2 creates a node, instead of labeling the last edge with only one character, implicitly label it with the entire remaining suffix
- Each leaf edge is labeled only once!

Loop Structure

i	j	rule 2 (follow suffix link)
	j+1	rule 2 (follow suffix link)
	j+2	rule 3
i+1	j+2	rule 3
i+2	j+2	rule 3
i+3	j+2	rule 2
	j+3	rule 3

- Rule 2 gets applied once per j (across all stages)
- Rule 3 gets applied once per i

Another Way to Think About It



insert finger

increment when S[j..i] not in tree (rule 2)

1) insert S[j..n] into tree by branching at S[j..i-1]
2) create suffix pointer to new node at S[j..i-1] if there is one
3) use parent suffix pointer to move finger to j+1

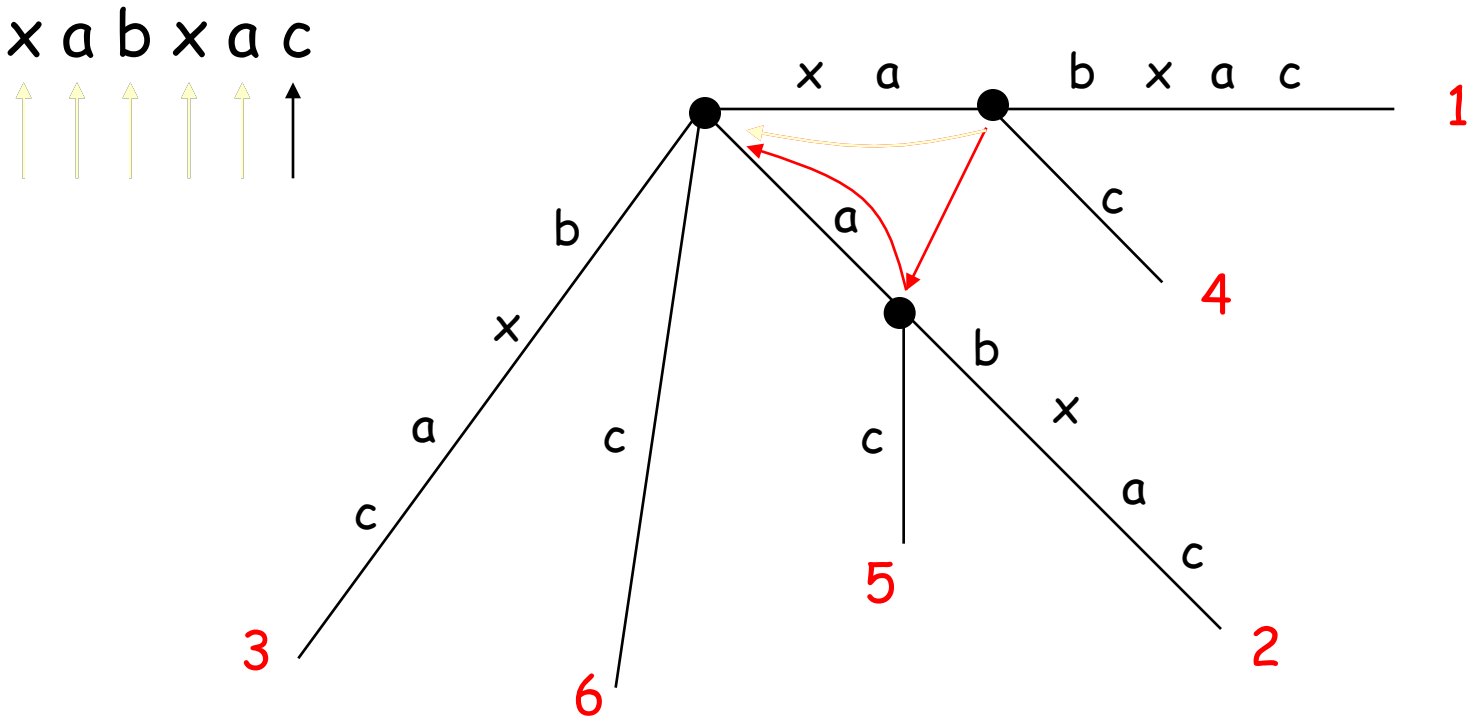
search finger

increment when S[j..i] in tree (rule 3)

Invariants:

1. j is never after i
2. S[j..i-1] is always in the tree

An example



Leaf edge labels are updated by using a variable to denote the start of the interval

Complexity Analysis

- Rule 3 is used only once in every stage
- For every j, Rule 1 & 2 are applied only once in the jth substage of all the stages.
- Each application of a rule takes O(1) steps
- Other overheads are O(1) per stage

- Total time is O(m)

Extending to multiple texts

- Suppose we want to match a pattern with a dictionary of k texts of length $m_1, \ldots, m_k$
- Concatenate all the texts (separated by special characters) and construct a common suffix tree
- Time taken = $O(m_1+\cdots+m_k)$
- Unnecessarily complicated tree; needs special characters

Multiple texts – Better algorithm

- First construct a suffix tree on the first text, then insert suffixes of the second text and so on
- Each leaf node should store values corresponding to each text
- $O(m_1+\cdots+m_k)$ as before

Longest Common Substring

- Find the longest string that is a substring of both text S_1 and text S_2
- Construct a common suffix tree for both texts
- Any node that has in its subtree at least one leaf node labeled by S_1 and at least one leaf node labeled by S_2 yields a common substring
- The "deepest" such node is the required substring
- Can be found in linear time by a tree traversal

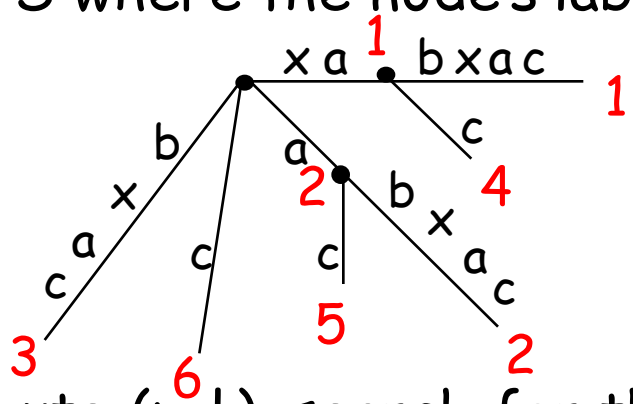
Common substrings of M strings

- Given M strings of total length n, find for every k, the length l_k of the longest string that is a substring of at least k of the strings

- Construct a common suffix tree – O(n) time
- For every internal node, find the number of distinctly labeled leaves in the subtree rooted at the node – might take O(Mn) time – not linear!
- Report l_k by a single tree traversal

Lempel-Ziv compression

- Recall that at each stage, we output a pair (p_i, l_i) where $S[p_i .. p_i+l_i] = S[i .. i+l_i]$
- Here's how to find all pairs (p_i,l_i) in linear time
- Construct a suffix tree for S
- Let the position of each internal node be the minimum of the positions of all leaves below it – this is the first place in S where the node's label occurs.



Example: for i=4,
S[i..m] = x a c
Stop at node labeled x a
with position 1
$(p_i,l_i)=(1,2)$

- To compute (p_i,l_i), search for the string S[i .. m], but stop before reaching a node with position i or more. This gives us l_i and p_i.