

## 15-499: Algorithms and Applications

- Computational Biology - VI
- Exact string matching
  - Suffix Trees
  - Other apps

15-499

Page 1

## Exact String Matching

- Given a text T and pattern P
- "Quickly" find an occurrence (or all occurrences) of P in T
- A Naïve solution:  
Compare P with  $T[i..i+n]$  for all  $i$  ---  $O(nm)$  time
- How about  $O(n+m)$  time?

15-499

Page 2

## Knuth-Morris-Pratt algorithm

- An  $O(n)$  preprocessing and  $O(m)$  time solution
- Want to search sequentially through T spending  $O(1)$  time on every character

```
T  x y a b c x a b c x a b c d e f e
P  a b c x a b c d e
   1
```

15-499

Page 3

## Knuth-Morris-Pratt algorithm

- An  $O(n)$  preprocessing and  $O(m)$  time solution
- Want to search sequentially through T spending  $O(1)$  time on every character

```
T  x y a b c x a b c x a b c d e f e
P  a b c x a b c d e
   1 2
```

15-499

Page 4

## Knuth-Morris-Pratt algorithm

- An  $O(n)$  preprocessing and  $O(m)$  time solution
- Want to search sequentially through  $T$  spending  $O(1)$  time on every character

```
T   x y a b c x a b c x a b c d e f e
P       a b c x a b c d e
      1 2 3 4 5 6 7 8 9 10
```

The match fails. Where should we restart?

Ideally restart from the 4<sup>th</sup> position of the pattern

15-499

Page 5

## Preprocessing the pattern

- At each position store the index of the position where we can resume the matching

```
i   1 2 3 4 5 6 7 8 9
P(i) a b c x a b c d e
l(i) 0 1 1 1 1 2 3 4 1
```

- $l(i)$ : 1 plus the length of longest suffix of  $P[2...i-1]$  that matches a prefix of  $P$ .

$l(1)$  is defined to be 0

Thus  $l(6) = 2$  ( $a = a$ ) and  $l(7) = 3$  ( $ab = ab$ )

- Can compute these values in linear time by going through the characters sequentially.

15-499

Page 6

## Knuth-Morris-Pratt algorithm

- $O(n+m)$  algorithm
  - $O(n)$  preprocessing +  $O(m)$  matching
- returns all occurrences
- What if we want to search for many patterns in the same text?
- $O(n+m)$  time per pattern  $\Rightarrow O(km)$  time
- Can we do it faster? In  $O(m+kn)$  time?

15-499

Page 7

## Suffix Trees

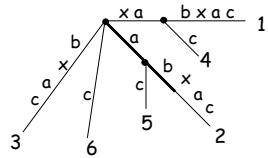
- Preprocess the text in  $O(m)$  time and search in  $O(n)$  time
- Idea:
  - Construct a tree containing all suffixes of text along the paths from the root to the leaves
  - For search, just follow the appropriate path

15-499

Page 8

## Suffix Trees

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



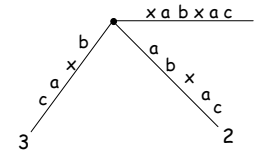
Search for the string  $a b x$

15-499

Page 9

## Constructing Suffix trees

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

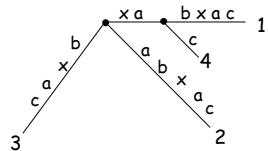


15-499

Page 10

## Constructing Suffix trees

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

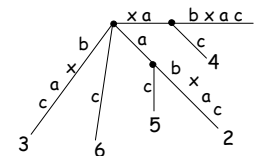


15-499

Page 11

## Constructing Suffix trees

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



15-499

Page 12

## 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  $S[1..i]$
- Incrementing  $T_i$  to  $T_{i+1}$  naively takes  $O(i^2)$  time because we insert each of the  $i$  suffixes
- Thus a total of  $O(m^3)$  time

15-499

Page 13

## Going from $T_i$ to $T_{i+1}$

- In the  $j^{\text{th}}$  substage of stage  $i+1$ , we insert  $S[j..i+1]$  into  $T_i$ . Let  $S[j..i] = \beta$ .
- Three cases
  - Rule 1: The path  $\beta$  ends on a leaf  $\Rightarrow$  add  $S[i+1]$  to the label of the last edge
  - Rule 2: The path  $\beta$  continues with characters other than  $S[i+1]$   $\Rightarrow$  create a new leaf node and split the path labeled  $\beta$
  - Rule 3: A path labeled  $\beta S[i+1]$  already exists  $\Rightarrow$  do nothing.

15-499

Page 14

## Idea #1 : Suffix Links

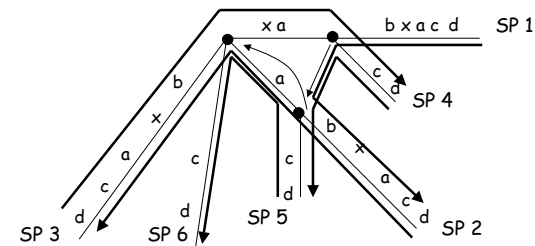
- In each substage, we first search for some string in the tree and then insert a new node/edge/label
- Can we speed up looking for strings in the tree?
- In any substage, we look for a suffix of the strings searched in previous substages
- Idea: Put a pointer from an internal node labeled  $x\alpha$  to the node labeled  $\alpha$
- Such a link is called a "Suffix Link"

15-499

Page 15

## Idea #1 : Suffix Links

Add the letter d



15-499

Page 16

## 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 the path for the remaining string
- First two steps together make up  $O(m)$  in each stage
- The third step follows only as many links as the length of the string  $S[1 .. i]$
- Thus the total time per stage is  $O(m)$

15-499

Page 17

## Maintaining Suffix Links

- Whenever a node labeled  $x\alpha$  is created, in the following substage a node labeled  $\alpha$  is created.  
Why?
- When a new node is created, add a suffix link from it to the root, and if required, add a suffix link from its predecessor to it.

15-499

Page 18

## 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  $O(m^2)$
- But notice that there are only  $2m$  edges! - Why?
- Idea: represent labels of edges as intervals
- Can easily modify the entire process to work on intervals

15-499

Page 19

## Idea #2 : Getting rid of Rule 3

- Recall Rule 3: A path labeled  $S[j .. i+1]$  already exists  $\Rightarrow$  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.

15-499

Page 20



### Extending to multiple lists

- Suppose we want to match a pattern with a dictionary of  $k$  strings
- Concatenate all the strings (interspersed with special characters) and construct a common suffix tree
- Time taken =  $O(km)$
- Unnecessarily complicated tree; needs special characters

15-499

Page 25

### Multiple lists - Better algorithm

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

15-499

Page 26

### Longest Common Substring

- Find the longest string that is a substring of both  $S_1$  and  $S_2$
- Construct a common suffix tree for both
- Any node that has leaf nodes labeled by  $S_1$  and  $S_2$  in the subtree rooted at it gives a common substring
- The "deepest" such node is the required substring
- Can be found in linear time by a tree traversal.

15-499

Page 27

### 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
- For every internal node, find the number of distinctly labeled leaves in the subtree rooted at the node
- Report  $l_k$  by a single tree traversal
- $O(Mn)$  time - not linear!

15-499

Page 28

## Lempel-Ziv compression

- Recall that at each stage, we output a pair  $(p_i, l_i)$  where  $S[p_i .. p_i+l_i] = S[i .. i+l_i]$
- Find all pairs  $(p_i, l_i)$  in linear time
- Construct a suffix tree for  $S$
- Label each internal node with the minimum of labels of all leaves below it - this is the first place in  $S$  where it occurs. Call this label  $c_v$ .
- For every  $i$ , search for the string  $S[i .. m]$  stopping just before  $c_v \geq i$ . This gives us  $l_i$  and  $p_i$ .