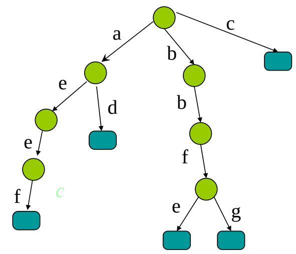
Trie, Suffix Trees and Suffix Arrays

Trie

- A tree representing a set of strings.

{
aeef
ad
bbfe
bbfg
c
}

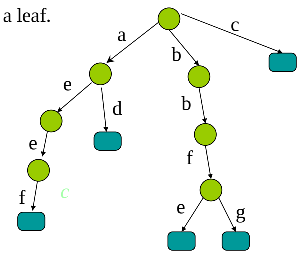


Trie (Cont)

Assume no string is a prefix of another

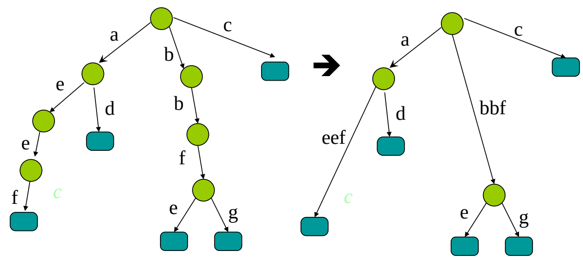
Each edge is labeled by a letter,
no two edges outgoing from the same node are labeled the same.

Each string corresponds to a leaf.



Compressed Trie

- Compress unary nodes, label edges by strings



Problems

- Given a pattern P = P[1..m], find all occurrences of P in a text S = S[1..n]
- Another problem:
  - Given two strings $S_1[1..n_1]$ and $S_2[1..n_2]$ find their longest common substring.
    - find i, j, k such that $S_1[i .. i+k-1] = S_2[j .. j+k-1]$ and k is as large as possible.
- Solve these problems efficiently?

Exact string matching

- Finding the pattern P[1..m] in S[1..n] can be solved simply with a scan of the string S in O(m+n) time. However, when S is very long and we want to perform many queries, it would be desirable to have a search algorithm that could take O(m) time.
- To do that we have to preprocess *S*. The preprocessing step is especially useful in scenarios where the text is relatively constant over time (e.g., a genome), and when search is needed for many different patterns.

Applications in Bioinformatics

- Multiple genome alignment
  - Longest common substring problem
  - Common substrings of more than two strings
- Selection of signature oligonucleotides for microarrays
- Identification of sequence repeats

Suffix trees

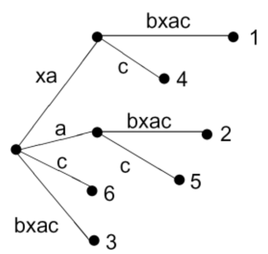
- Any string of length m can be degenerated into m suffixes.
  - abcdefgh (length: 8)
  - 8 suffixes:
    - h, gh, fgh, efgh, defgh, cdefgh, bcefgh, abcdefgh
- The suffixes can be stored in a suffix-tree and this tree can be generated in O(*n*) time
- A string pattern of length *m* can be searched in this suffix tree in O(*m*) time.

Definition of a suffix tree

- Let ***S*=*S*[1..*n*]** be a string of length *n* over a **fixed alphabet Σ**. **A suffix tree for *S* is a tree with *n* leaves (representing *n* suffixes) and the following properties:**
  1. Every internal node other than the root has at least 2 children
  2. Every edge is labeled with a nonempty substring of *S*.
  3. The edges leaving a given node have labels starting with different letters.
  4. The concatenation of the labels of the path from the root to leaf *i* spells out the *i*-th suffix *S*[*i*..*n*] of *S*. We denote *S*[i..n] by S_i.

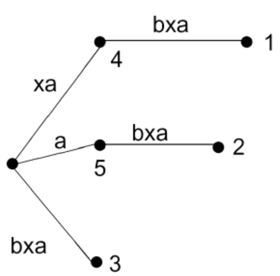
An example suffix tree

- The suffix tree for string: 1 2 3 4 5 6
  x a b x a c



What about the tree for xabxa?

- The suffix tree for string: 1 2 3 4 5
  x a b x a



xa an *a* are not leaf nodes.

Problem

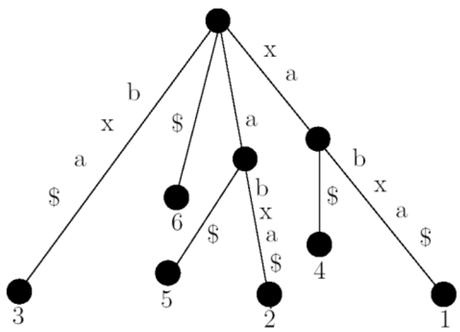
- Note that if a suffix is a prefix of another suffix we cannot have a tree with the properties of suffix tree
  - **e.g. xabxa**

The fourth suffix *xa* or the fifth suffix *a* won't be represented by a leaf node.

Solution: the terminal character $

Solution: insert a special **terminal character at the end such as $.**
Therefore xa$ will not be a prefix of the suffix xabxa.

The suffix tree for xabxa$



Suffix tree construction

- Start with a root and a leaf numbered 1, connected by an edge labeled *S*$.
- Enter suffixes *S*[2..*n*]$; *S*[3...*n*]$; ... ; *S*[*n*]$ into the tree as follows:
- To insert $K_i = S[i..n]$\$, follow the path from the root matching characters of K_i until the first mismatch at character $K_i[\,j\,]$ (which is bound to happen)
  - (a) If the matching cannot continue from a node, denote that node by *w*
  - (b) Otherwise the mismatch occurs at the middle of an edge, which has to be split

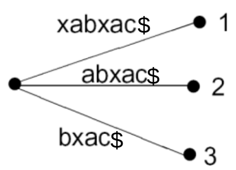
Suffix tree construction - 2

- If the mismatch occurs at the middle of an edge $e = S[u \ldots v]$
  - let the label of that edge be $a_1 \ldots a_l$
  - If the mismatch occurred at character a_k, then create a new node *w*, and replace ***e*** **by two edges S[u ... u+k-1] and S[u+k ... v] labeled by** $a_1 \ldots a_k$ **and** $a_{k+1} \ldots a_l$
- Finally, in both cases (a) and (b), create a new leaf numbered *i*, and connect *w* to it by an edge labeled with $K_i[j \ldots |K_i|]$

Example construction

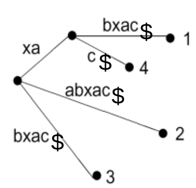
- Let's construct a suffix tree for xabxac$
- Start with:

- After inserting the second and third suffix:



Example cont'd

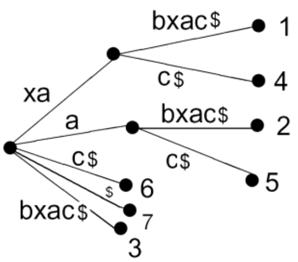
- Inserting the fourth suffix xac$ will cause the first edge to be split:



- Same thing happens for the second edge when ac$ is inserted.

Example cont'd

- After inserting the remaining suffixes the tree will be completed:



Complexity of the naive construction

- We need O(n-i+1) time for the i^{th} suffix. Therefore the total running time is:

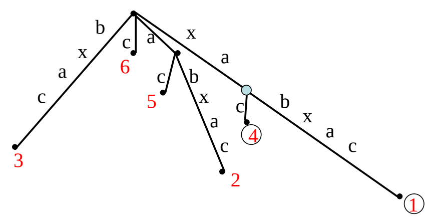
$$\sum_{1}^{n} O(i) = O(n^2)$$

Using suffix trees for pattern matching

- **Given S and P**. How do we find all occurrences of P in S?
- **Observation.** Each occurrence has to be a prefix of some suffix. Each such prefix corresponds to a path starting at the root.
  1. We construct the suffix tree for S.
  2. Try to match P on a path, starting from the root. Three cases:
     (a) The pattern does not match ➔ P does not occur in T
     (b) The match ends in a node u of the tree. Set $x = u$.
     (c) The match ends inside an edge (v,w) of the tree. Set $x = w$.
  3. **All leaves below x represent occurrences of P.**

Illustration

- T = xabxac
  - suffixes ={xabxac, abxac, bxac, xac, ac, c}
- Pattern P_1: xa



Running Time Analysis

- Search time:
  - O(m+k) where k is the number of occurrences of P in T and m is the length of P
  - O(m) to find match point if it exists
  - O(k) to find all leaves below match point

Scalability

- For very large problems a linear time and space bound is not good enough. This lead to the development of structures such as Suffix Arrays to conserve memory .

Two implementation issues

- Alphabet size
- Generalizing to multiple strings

Suffix arrays

- More space efficient than suffix trees
- A suffix array for a string x of length *m* is an array of size *m* that specifies the **lexicographic** ordering of the **suffixes of x**.

Suffix Arrays

Example of a suffix array for acaaacatat$

0	aaacatat$	3
1	aacatat$	4
2	acaaacatat$	1
3	acatat$	5
4	atat$	7
5	at$	9
6	caaacatat$	2
7	catat$	6
8	tat$	8
9	t$	10
10	$	11

Suffix array construction

- Similar to insertion sort
- Insert all the suffixes into the array one by one making sure that the new inserted suffix is in its correct place
- Running time complexity: $O(m^2)$ where m is the length of the string

Suffix arrays

- $O(n)$ space where n is the size of the database string
- Space efficient. However, there's an increase in query time
- Lookup query
  - Binary search
  - O(m log n) time; m is the size of the query

Search example

Search ***is*** in **mississippi$**

Examine the pattern letter by letter, reducing the range of occurrence each time.

First letter *i*:
•occurs in indices from 0 to 3
•So, pattern should be between these indices.

Second letter *s*:
•occurs in indices from 2 to 3
•Done.
Output: issippi$ and ississippi$

0	11	i$
1	8	ippi$
2	5	issippi$
3	2	ississippi$
4	1	mississippi$
5	10	pi$
6	9	ppi$
7	7	sippi$
8	4	sissippi$
9	6	ssippi$
10	3	ssissippi$
11	12	$

Searching for a pattern in Suffix Arrays

```
find(Pattern P in SuffixArray A):
  i = 0
  lo = 0, hi = length(A)
   for 0<=i<length(P):
     Binary search for x,y
     where P[i]=S[A[j]+i] for lo<=x<=j<y<=hi
     lo = x, hi = y
   return {A[lo],A[lo+1],...,A[hi-1]}
```

Suffix Arrays

- It can be built very fast.
- It can answer queries very fast:
- Disadvantages:
  - Can’t do approximate matching
  - Hard to insert new stuff (need to rebuild the array) dynamically.