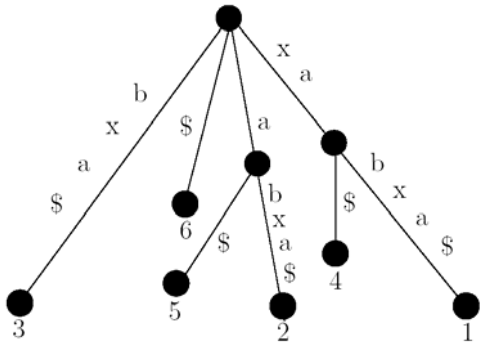
Suffix Trees and Arrays

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:
  - Every internal node other than the root has at least 2 children
  - Every edge is labeled with a nonempty substring of S.
  - The edges leaving a given node have labels starting with different letters.
  - 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.

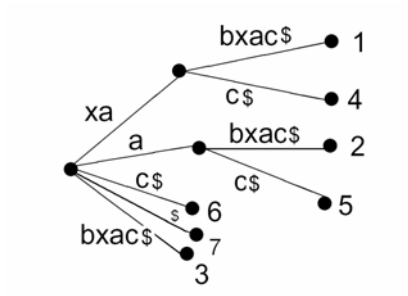
The suffix tree for xabxa$



Suffix tree construction

- Start with the complete string.
- Enter suffixes $S[2..n]$\$; $S[3...n]$\$; ... ; $S[n]$\$ into the tree as follows:
- To insert $S[i..m]$\$, follow the path from the root matching characters of the suffix until the first mismatch at character j
  - (a) If the matching cannot continue from a node, denote that node attach and the edge $S[j..m]$\$ to that node
  - (b) Otherwise the mismatch occurs at the middle of an edge, which has to be split into two edges at the position of the mismatch. The mismatched portion of the edge and S[j..m]$ are new edges coming out of the newly inserted node.

Example



Storing the edge labels efficiently

- Note that, we do not store the actual substrings $S[i ... j]$ of S in the edges, but only their start and end indices (i, j).
- Nevertheless we keep thinking of the edge labels as substrings of S.
- This will reduce the space complexity to $O(n)$

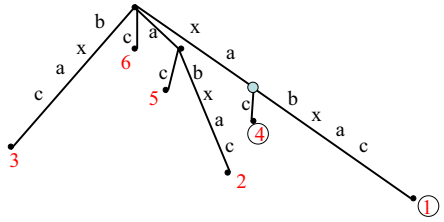
Using suffix trees for pattern matching

- 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. Of course, as a first step, we construct the suffix tree for *S*. Using the naive method this takes quadratic time, but *linear-time* algorithms (e.g., Ukkonen's algorithm) exist.
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
- Pattern P_2: xb



Generalized suffix trees

- Build a suffix tree for a set of strings S = {S_1, …, S_z}
- Nodes in tree may corresponds to substrings of potentially multiple strings S_i
  - compact edge labels: need 3 fields (start position, stop position, source strings)
  - leaf labels now a set of pairs indicating starting position and source strings

Longest common substring problem

- Build a generalized suffix tree for $S_1\$_1S_2\$_2$. Here $\$_1$ and $\$_2$ are different new symbols not occurring in S_1 and S_2.
- Mark every internal node of the tree with {1}, {2}, or {1,2} depending on whether its path label is a substring of S_1 and/or S_2.
- Find the *internal* node which is labeled by {1,2} and has the largest "*string depth*".
- Example: (with the applet)
  - pessimist%mississippi$

Selecting probes for microarrays

- Wikipedia: **Oligonucleotides** are short sequences of nucleotides (RNA or DNA), typically with twenty or fewer base pairs.
- Given a set of genomic sequences, the problem is to identify at least one signature oligonucleotide (probe) for each sequence. These probes must hybridize to only the desired sequence. The algorithm produces a GST from the reverse compliment of all the genomic sequences (candidate probe sequences). Using the GST, the algorithm identifies all common substrings and rejects these regions because probes designed in them would not be specific to a single genomic sequence. Criteria such as probe length are used to further prune this tree.
- http://www.zaik.uni-koeln.de/bioinformatik/arraydesign.html.en

Suffix arrays

- Suffix arrays were introduced by Manber and Myers in 1993
- 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

- Naive in place 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
- Manber and Myers give a O(m log m) construction in their 1993 paper.

Suffix arrays

- O(m) space where m is the size of the database string
- Space efficient. However, there's an increase in query time
- Lookup query
  - Binary search
  - O(n log m) time; n is the size of the query
  - Can reduce time to O(n + log m) using a more efficient implementation

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]}
```

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	$

Assignment #2

- Programming assignment

(1) Construct suffix arrays for 5000 randomly generated proteins of random length between 100 and 1000 (random generation will use the amino acid frequencies in nature) using the naive suffix array generation algorithm (similar to insertion sort)

(2) Plot the construction time vs. length of proteins
Do you observe the O(n^2) running time complexity in your graph? Comment on your graph.

Assignment #2

(3) Generate 5000 random amino acid patterns of random length between 3 and 20 (random generation will use the amino acid frequencies in nature)

(4) Search your patterns in 10 representative proteins in your database (randomly chosen) with lengths: 100, 200, 300, 400, ..., 1000, using the binary search algorithm described in class (slide 16 in week4_2.pdf)

(5) Plot 10 “pattern search time” vs. “pattern length” graphs for these 10 representative proteins.

What do you observe in these graphs?

(6) Plot 10 “number of pattern occurrences” vs. “query length” graphs. Explain the graphs.

Useful links

- http://pauillac.inria.fr/~quercia/documents-info/Luminy-98/albert/JAVA+html/SuffixTreeGrow.html
- http://home.in.tum.de/~maass/suffix.html
- http://homepage.usask.ca/~ctl271/857/suffix_tree.shtml
- http://homepage.usask.ca/~ctl271/810/approximate_matching.shtml
- http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic7/
- http://dogma.net/markn/articles/suffixt/suffixt.htm
- http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Suffix/