Suffix Trees and Arrays

Some 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 $S_1[i .. i+k-1] = S_1[j .. j+k-1]$ and k is as large as possible.
- Any solutions? How do you 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
  - Michael Hohl et al. 2002
  - Longest common substring problem
  - Common substrings of more than two strings
- Selection of signature oligonucleotides for DNA arrays
  - Kaderali and Schliep, 2002
- Identification of sequence repeats
  - Kurtz and Schleiermacher, 1999

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.
  - Whereas, a regular sequential search would take O(*n*) time.

History of suffix trees

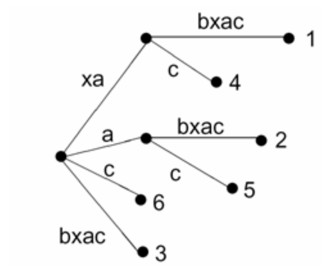
- Weiner, 1973: suffix trees introduced, linear-time construction algorithm
- McCreight, 1976: reduced space-complexity
- Ukkonen, 1995: new algorithm, easier to describe
- In this lecture, we will only cover a naive (quadratic-time) construction.

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.

An example suffix tree

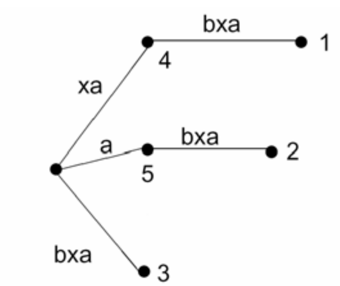
- The suffix tree for string: 1 2 3 4 5 6
  x a b x a c



Does a suffix tree always exist?

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.

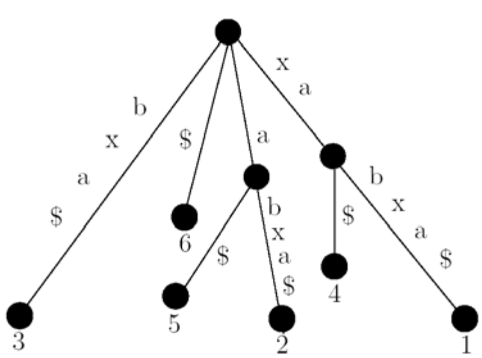
The terminal character $

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

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

- 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

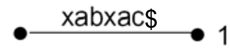
- Start with a root and a leaf numbered 1, connected by an edge labeled $S$$.
- Enter suffixes S[2..n]$; S[3...n]$; ... ; S[n]$ into the tree as follows:
- To insert K_i = S[i..m]$, follow the path from the root matching characters of K_i until the first mismatch at character K_i[j] (which is bound to 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

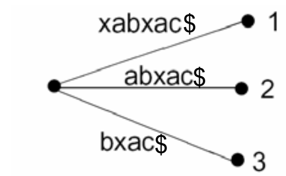
- If the mismatch occurs at the middle of an edge *e* = *S*[*u* ... *v*]
  - let the label of that edge be $a_1 \ldots a_l$
  - If the mismatch occurred at character a_k, then create a new node *w*, and replace *e* by two edges S[u ... u+k-1] and S[u+k ... v] labeled by $a_1 \ldots a_k$ and $a_{k+1} \ldots a_l$
- 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:

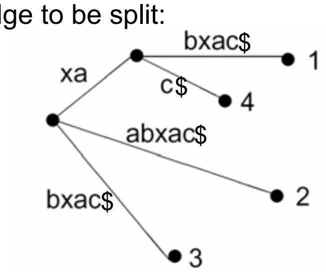


- After inserting the second and third suffix:



Example contd...

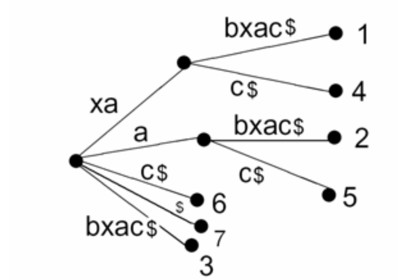
- Inserting the fourth suffix xac$ will cause the first edge to be split:



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

Example contd...

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



Complexity of the naive construction

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

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

- What about space complexity?
  - Can also take $O(n^2)$ because we may need to store every suffix in the tree separately,
  - e.g., abcdefghijklmn

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*)

Suffix tree applet

- http://pauillac.inria.fr/~quercia/documents-info/Luminy-98/albert/JAVA+html/SuffixTreeGrow.html

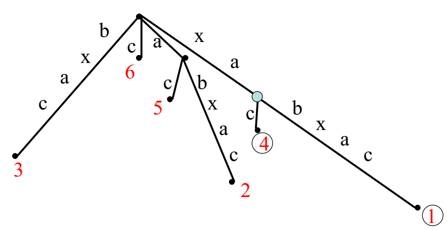
Using suffix trees for pattern matching

- 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



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

One way to compute

- Use a different end character $\$_i$ for each string S_i
- Concatenate all the strings together
- Make suffix tree of concatenated string
- Make artificial suffixes actual suffixes
  - For any internal node v, L(v) must be a substring of an original string
    - Only the leaf edge labels can span two original strings because of the uniqueness of each $\$_i$
  - Postprocess and shorten leaf edge labels appropriately

Effects of alphabet size on suffix trees

- We have generally been assuming that the trees are built in such a way that
  - from any node, we can find an edge in constant time for any specific character in Σ
    - an array of size $|\Sigma|$ at each node
- This takes $\Theta(m|\Sigma|)$ space.

More compact representation

- We can try to be more compact taking only O(m) space.
  - At each node, have pointers to only the edges that are needed
- This slows down the search time
- How much?
  - typically the minimum of O(log m) or O(log $|\Sigma|$) with a binary tree representation.
- This effects both suffix tree construction time and later searching time against the suffix tree.

Other methods are truly alphabet independent

- Z-computation, KMP, BM all have running times and space requirements that are truly independent of the alphabet size.
- This can make them superior to suffix tree approaches when $|\Sigma|$ is large.

Generalized suffix trees

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

One way to compute

- Use a different end character $\$_i$ for each string S_i
- Concatenate all the strings together
- Make suffix tree of concatenated string
- Make artificial suffixes actual suffixes
  - For any internal node v, L(v) must be a substring of an original string
    - Only the leaf edge labels can span two original strings because of the uniqueness of each $\$_i$
  - Postprocess and shorten leaf edge labels appropriately

Another way to compute

- Build tree for S_1
- Given tree for strings S_1 through S_i, add suffixes for S_{i+1} as follows:
  - Search for S_{i+1} in tree till mismatch in position j+1 of S_{i+1}
  - Existing tree implicitly has every suffix of $S_{i+1}[1..j]$
  - Resume Ukkonen's algorithm for S_{i+1} in phase j+1 from point of last match

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.
  - Example of a suffix array (mississippi)
- O(*n*) space
- Lookup query
  - Binary search
  - O(m log n) time; n is the size of the query
  - Can reduce time to O(m + log n) using a more efficient implementation

Suffix arrays

- Suffix Array. Give sequence:
  she_sells_seashell_on_the_sea_shore
- Suffix array is an array of pointers pointing to suffices of the sequence … sorted.
- E.g., all suffices of above are: e, re, ore, hore, shore, _shore, a_shore, ea_shore, sea_shore, and so on.
- Then these are sorted and their pointers (pointing to a suffix location in sentence) are stored in an array (the suffix array).

Suffix Arrays

- It can be built very fast.
- It can answer queries very fast:
  - How many times ATG appears (their pointers are all jammed together).
  - What is G-C contents.
- Disadvantages:
  - Can't do approximate matching
  - Hard to insert new stuff (need to rebuild the array) dynamically.
  - Pointers can cost too much space. 3G pointers?