Suffix trees

©kuko 26.11.2023

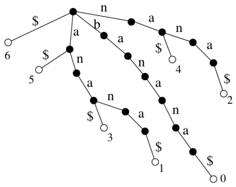
1 Motivation

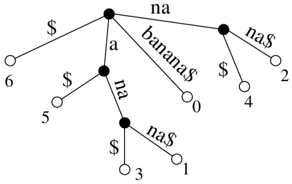
- given text T of length n, pattern P of length m; find occurrences of P in T
- trivial search: $O(m \times n)$; KMP: $O(m)$ preprocessing of the pattern and $O(n)$ search; practical algorithms (Boyer-Moore)
- *indexing*: could we preprocess the text (instead of the pattern) and then search in time $O(m)$?
- longest common substring – open problem for many years, can be solved in $O(n)$?

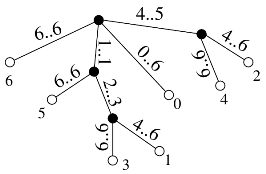
- 2 easier problems:
- preprocess given texts $T_1, \ldots, T_d$, so that for a given string P, we can find all texts which start with P
  - trivial: no preprocessing; search in $O(m \times d)$
  - sort, then binary search: $O(m \times \log d)$
  - build a trie (radix tree), then go down the path P; the leaves in the subtree are all documents starting with P
- given $T_1, \ldots, T_d$, find the longest common prefix of any two texts T_i, T_j
  - again, can be solved trivially, using sorting, or using a trie
- in general, a trie is good for problems concerning prefixes

2 Sufix trees

- sufix tree – crazy idea: store *all the suffixes of string* T in a single trie
- every substring is a prefix of some suffix (let that sink in) so this trie will be good for problems concerning substrings





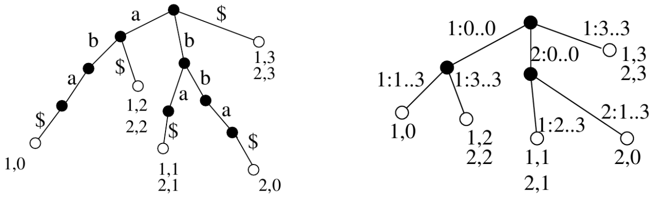


- every node v corresponds to a substring spelled on the path from root to v
- leaves in subtree v correspond occurrences of this substring

- space: all suffixes have total length $\Omega(n^2)$ – that's a problem (fig. left)
- solution:
- 1) each path that doesn't split is represented as a single edge (middle fig.)
- we get a tree where all internal nodes have degree ≥ 2, so the number of internal nodes $<$ number of leaves; i.e. $O(n)$ nodes in total
- 2) for every edge, only store the *indices* of substrings in T (not whole substrings; fig. right)
- this way, every edge takes $O(1)$ space and we have just 1 copy of T; that's $O(n)$ space in total
- sufix tree can also be constructed in $O(n)$ time; how to do that – later
- note that we can work with the succinct representation (right), but still imagine we conceptually have the tree on the left; e.g. when going from root along path "`ban`", we imagine there is a node as in the left fig., while we represent it as imaginary node (e_0,3), where e_0 is the edge from root to 0 and "3" means go 3 chars down along this edge

- generalization: given a set of “documents”(texts) $\mathcal{D} = \{T_1, T_2, \ldots, T_d\}$, a sufix tree contains all the suffixes of all the documents; e.g., think wikipedia $\approx 6M$ articles, tens of GBs of data
- it is sufficient to build a suffix tree for a single combined string $T_1\#T_2\#T_3\# \cdots T_d\#\$$, where $\#$ and $\$$ are two special symbols, which *do not occur* in texts $T_1, \ldots, T_d$
- the only difference is that now, the leaves and edges need to specify the document they are referring to

Example: $S_1 = \mathtt{aba\$}$, $S_2 = \mathtt{bba\$}$:



3 Many Applications

- *string search*: does P occur in the text T? find the first/all occurrences
  - just go down from the root along the path P; the leaves in the subtree are all the occurrences
  - if we want the first occurrence, we precompute for each node a pointer to the leaf with the smallest suffix number (or directly the position of the first occurrence) by traversing the tree bottom-up (postorder) in $O(n)$; value in node is min of its children
  - if we want all occurrences, just search the entire subtree – if there are k occurrences, the subtree has size $O(k)$
  - precomputation: $O(n)$, first occurrence: $O(m)$, all occurrences: $O(m + k)$, where $k = \#$occurrences
- *longest repeating substring* in T
  - a node with at least two leaves underneath represents a repeating substring (these are all internal nodes; #leaves = #occurrences)
  - for each node, we can precompute the ”string-depth(v)-the number of characters on the path from the root to v (note that this is not the classical depth of a node – we do not want the number of edges, but the length of the text on the edges)
  - the result is the internal node with the maximum string-depth – we can find it in $O(n)$
- *longest common substring* of T_1 and T_2
  - build the generalized suffix tree of T_1 and T_2; color all the leaves with 2 colors – depending on whether the suffix belongs to T_1 or T_2
  - find a node which has leaves of both colors underneath (we can precompute this info by a bottom-up traversal)
- the shortest unique substring / the most common substring of length $\geq k$ – can be solved similarly

- *maximal repeats:* we want substrings $T[i \ldots i+k] = T[j \ldots j+k]$, such that $T[i-1] \neq T[j-1]$ and $T[i+k+1] \neq T[j+k+1]$, i.e. they are maximal in the sense that they cannot be extended to the left, nor to the right
  - just mark for each leaf corresponding to the i-th suffix the character before it, i.e., $T[i-1]$
  - find nodes which have at least two different characters in their two children subtrees
- given two positions i, j, find the *longest common prefix* (LCP) of $T[i \ldots]$ and $T[j \ldots]$
  - trivially in $O(k)$, if $T[i \ldots i+k-1] = T[j \ldots j+k-1]$ but $T[i+k] \neq T[j+k]$
  - in $O(1)$, with precomputed LCA (lowest common ancestor)

- *approximate search* with $\leq k$ mismatches
  - trivial in $O(n \times m)$ time (for every positions of a sliding window, count the number of mismatches)
  - better: in $O(n \times k)$ (build suffix tree of T and P and speed up the search by computing LCP (via LCA): similar as before, we search at every position, but instead of comparing all the characters, compute LCP and jump to the first mismatch in $O(1)$; then compute LCP of the rest and jump to the next mismatch until you reach the end of P or there are too many mismatches)
- *document counting problem*: find the number of documents containing P
  - imagine that we color the leaves with different colors, according to the document in which the given suffix is located; we have d colors and we want to know for each node, how many different colors are under it
  - trivial in $O(m + k)$ by searching the entire subtree (no precounting; $k = $#occurrences) – can we do better?
  - for each vertex, we precompute the set of colors below it – precomputation time and space $O(n \times d)$
  - better: use LCA; trick: let's fix some specific subtree; two nodes are in a subtree if and only if their LCA is also in this subtree
  - so if there are e.g. r red leaves in a subtree, then $r - 1$ consecutive pairs will have LCA *in* the given subtree
  - for each color, we compute the number of (occurrences minus 1); when we sum these up, we get the number of all leaves minus 1 for each color in the tree; i.e. we can compute the number of different colors in this roundabout way: we compute number of all leaves and subtract number of (occurrences minus 1) for each color
  - for each color, let's have the leaves of that color sorted from left to right
  - for each color, we successively go through the leaves of the given color; for every two consecutive leaves, compute their LCA and add 1 to this node
  - then sum up all the values for every subtree (traverse the tree bottom-up and for each node, add the sum of their children)
  - at the same time, count the number of leaves for each subtree and get the number of different colors as #leaves minus #LCAs
  - this way, the precomputation can be done in $O(n)$ time and space
- *document listing problem*: list all the documents containing P
  - trivially in $O(m + k)$ by traversing the whole subtree under P – can we do better? what if there are many documents?
  - define array A, s.t. $A[i]$ =number of the *preceding* node with the *same color*
  - all the occurrences of P correspond to leaves of some subtree, which correspond to an interval in array A (say $A[i \dots j]$)
  - we want to list all colors in a subtree; we achieve this by finding the leftmost node of each color in the interval; these are all the nodes such that their predecessor of the same color is *outside* of $[i \dots j]$, more precisely, $< i$
  - so the problem reduces to listing all positions k in interval $[i, j]$, such that $A[k] < i$
  - this can be done in time $O(|output|)$ (i.e., we list the documents in time $O($#documents$)$ instead of $O($#occurrences$)$ – note that each document can contain many many occurrences of P)
  - we precompute RMQ for A; then for given interval $[i, j]$, we find minimum and if the minimum is $< i$, we return it and also recursively search left and right sides

- even more complicated variants were studied: see e.g.

  `https://users.dcc.uchile.cl/~gnavarro/ps/soda12.pdf`,

  which solves the *top-k document retrieval* problem:

  - there is a predefined measure of how relevant is document D for pattern P (this can be e.g. some static rank of D, or it can depend on the number of occurrences of P in D, etc.)
  - problem: given a pattern P and number k, find the top-k most relevant documents which contain P

4 Summary

- suffix tree is a trie (radix-tree) containing all the suffixes of a given string (generalized suffix tree contains all suffixes of multiple strings)
- it takes $O(n)$ space and can be constructed in $O(n)$ time
- it is very useful in stringology because it reveals a lot of structure in substrings of a given string and many string problems can be reduced to problems on trees
  - $\longleftrightarrow$ position i $\longleftrightarrow$ leaf i
  - i-th suffix $\longleftrightarrow$ path from root to leaf i
  - substring $[i..j]$ $\longleftrightarrow$ path from root down (towards leaf i)
  - occurrences of P $\longleftrightarrow$ all the leaves under path P
  - all strings of length k in T $\longleftrightarrow$ cut the suffix tree at string-depth k
  - document $\longleftrightarrow$ color of the leaf
  - documents containing P $\longleftrightarrow$ different leaf colors under path P
  - common prefix $\longleftrightarrow$ common subpath from root toward two nodes
  - LCP of two substrings $\longleftrightarrow$ LCA of two nodes