CS 840 **Fall 2016**

Text Search and Succinct Data Structures: Unit 4

Given a text file, or several text files, how do we search for a query string?

Note the query/pattern is not of fixed length, unlike key searches.

If we are not able to preprocess the text file (length m), the "obvious method" is O(mn) where n is the length of the query.

- Knuth-Morris-Pratt: showed how to construct a simplified automaton in O(n) time so that the query is answered in O(m) time. Indeed the method essentially operates with two pointers into the text and on each comparison one is advanced.
  Knuth D.E., Morris (Jr) J.H., Pratt V.R., 1977, Fast pattern matching in strings, *SIAM Journal on Computing* 6(2):323-350.

- Boyer-Moore: give a method by which the pattern is aligned with the text, then the pattern is matched from right to left (i.e. back to front). If a mismatch is found early (near the end of the pattern string), the pattern may be advanced a substantial distance. It is thus possible to answer some queries in as little as O(m/n) time.
  Boyer R.S., **Moore J.S.**, 1977, A fast string searching algorithm. *Communications of the ACM*. 20:762-772.

- Rabin-Karp: this is a simple hashing scheme. The key idea is to have a hash function on strings of length n such that Hash($a_{i+1}\ldots a_{i+n}$) is easily computed from Hash($a_i\ldots a_{i+n-1}$). The usual approach is to use $\sum a_j x^j \bmod r$ for some values of x and the modulus r.
  Karp R.M., **Rabin M.O.**, 1987, Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 31(2):249-260.

When text is allowed to be preprocessed:

Peter Weiner (Peter Weiner: Linear Pattern Matching Algorithms. *Proc. 14th IEEE SWAT* (old name of *FOCS*) pp. 1-11; 1973) proposed the suffix tree (he called it a position-tree).

McCreight (Edward M. McCreight: A Space-Economical Suffix Tree Construction Algorithm. JACM 23(2): 262-272 (1976)) and Ukkonen (Esko Ukkonen: On-Line Construction of Suffix Trees. Algorithmica 14(3): 249-260 (1995)) gave different and more space efficient construction algorithms.

See (Robert Giegerich, Stefan Kurtz: From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. Algorithmica 19(3): 331-353 (1997)) for a unified view of these algorithms.

Gonnet et al (OED project) and Manber and Meyers (Udi Manber, Eugene W. Myers: Suffix Arrays: A New Method for On-Line String Searches. SIAM J. Comput. 22(5): 935-948 (1993); also SODA 1990) demonstrated the importance of this structure, and a cut-down form of it, for large scale text indexing.

We deal with the text S[1,.m] where wlg the last character is an eof symbol \$.

Suffix tree: Digital tree (or trie) in which root to leaf paths are in correspondence with suffixes of S (i.e., S[i,m] I = 1,m). All internal nodes are of degree at least two, each node is labeled with a non-null string of characters so that the root to leaf path spells out the appropriate suffix. [Sibling labels differ at the earliest possible point].

Representational issues
- Denote an edge label by a pair [b,f] such that S[b,f] corresponds to the string (note that the choice of b and f is not necessarily unique).
- As there are no internal nodes of degree 1, there are $< m$ internal nodes.
- The total space required (excluding S[1,m] is O(m) "words" of lg m bits each.
- If the tree were binary (as is often done) the edge information can be reduced to the number of bits skipped (i.e. bits on which all suffixes matching thus far continue to match).

Clearly we can build a suffix tree in $O(m^2)$ time.
Clearly we can reduce this by a "straightforward" sort of the suffixes.

We will return to this issue of creation of a suffix tree.

Applications:

Given a suffix tree of the text S and a query string P of length n, it is immediately obvious that we can determine whether the string P is a substring of S. Indeed in O(n) time we can

- determine whether P occurs in S, and if so (by keeping the size of a subtree at its root), the number of occurrences, in O(n) time (even if this number is $>> n$). In fact a reference to the appropriate subtree may be a convenient way to pass on all matches.

- among other useful information the suffix tree contains the longest repeated substring.

Applications abound:
- text processing
- bioinformatics
- compression

There are (at least) three linear time algorithms for creating suffix trees [Weiner 1973], [McCreight, *JACM* 1976] [Ukkonen, *Algorithmica* 1995].
All suffer from random access issues that make them costly for structures not fitting in main memory.

Weiner's approach (…the original so others are time/space reducing refinements):

Basic idea:

For $i = m$ downto 1 do insert suffix S[i,m] but we keep some back pointers (call this tree T_i).

Where does S[i,m] branch off?

- At the longest prefix of S[i,m] that is a substring of what you have already seen i.e. S[i+1,m]
- Call this Head(i)

So the method is linear if we have Head(i).
Unfortunately, at this stage the time per step is proportional to the length of this prefix.

Finding Head(i)

Key idea: keep two vectors (indicator) I and (link) L (of alphabet size)

I is binary
L is null or pointer
$I_v(x)$ is indicator of character x at node v
$L_v(x)$ is a reference

Key properties

- For any character x, node u, $I_u(x) = 1$ in T_{i+1} iff there is a path from the root labeled $x\alpha$ where α is the path to u. ($x\alpha$ need not end at a node, $I_u(x) = 1$, $L_u(x)$ = null handles that case).
- For any character x, $L_u(x)$ in T_{i+1} points to the internal node v in T_{i+1} iff v has path label $x\alpha$ and u has label α. Otherwise $L_u(x)$ is null.

Key notion: Start at leaf of $Suff_{i+1}$ in T_{i+1} walk up to find first node
v where $I_v(S(I)) = 1$ and
v' where $I_{v'}(S(i)) = 1$ and L_v is non-null.

Numerous details (including degenerate case where v or v' don't exist). See for example Gusfield, Algorithms on Strings Trees and Sequences, for details.

Problem: Space for suffix tree (even after it has been created)

O(m) words (2 or 3 pointers, 2 or 3 indices)

in addition to the text of m characters.

In OED project, binary suffix tree was of size 5 times text even though index points only every 5th character or so.

Reference to text at leaves needed (in OED case this space is roughly equal to text).

But both Gonnet et al and Manber and Meyers tossed suffix tree in favour of simply keeping references to index points in the order of suffixes referred to. Hence search for string takes O(lg n) time (includes getting all matches).

This leads to the question of tree representations. How much space does it take to represent a tree (say a binary tree) on n nodes, so that we can efficiently "navigate the tree?"

A starting point: Indexing an m character text for searches starting at the beginning of each (English) word or special symbol.

- m bytes of text
- $m/5$ index points

Implement as (binary) suffix tree with references at leaves to text positions

Naïve implementation:	2m/5 nodes,	
Each requiring	2 pointers	= 8 bytes
	1 subtree size	= 4 bytes
	skip/flag	= 1 byte
		13 bytes

$13 * 2m/5 = 5\ 1/5\ m$ bytes for index on m characters.

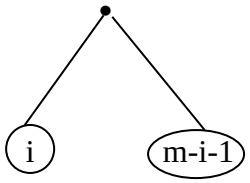
We can only reduce this by realizing the leaves are only pointers and get away with about $4\ 3/5\ m$ bytes.

Neither is acceptable.

What is the information theoretic minimum for representing a binary tree on m nodes?

There are $\binom{2m}{m}/m+1$ binary trees.

{Proof:



$$\#m = \sum_{i=0}^{m-1} \#i * \#(m-i-1)$$

$$\#0 = \#1 = 1$$

}

$$\binom{2m}{m}/(m+1) \approx c2^{2m}/m^{3/2} \quad (c = \text{small consistent})$$

bits required is lg of this value, or

$$2m - \frac{3}{2}\lg m + O(1)$$

↑ ↑

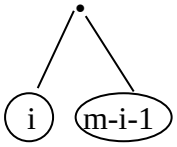
!!! and this is negative, hence recursive build up has a chance

Clearly trees can be represented in $2m$ bits, but can we still navigate?

A first approach?

The $-\frac{3}{2}\lg m$ term gives a hint.

To represent a subtree of size m we use Tree(m) bits. Then for



use rep(i) Tree(i) Tree(m-i-1) where rep(i) is the number bits to denote i, using a prefix code (we don't know anything about i other than it is less than m).

Clearly $2\lceil \lg i \rceil + 1$ bits suffice … $\lceil \lg i \rceil + 1$ 0's then write i in binary.
(But we can do better for large i).

But there is a problem even with $\lceil \lg i \rceil$ bits when i is close to m e.g.

T(m) = lg m + 0 + T(m-2) leads to an Θ(m lg m) solution.

Solution to this:

Let i = size of smaller tree, then use 1 bit to say whether small tree is left or right.

Reasonable tuning gives a 3m bit representation method by which we can navigate to children and find subtree size [can actually do a bit better, but the real problem is small subtrees and especially encoding i].

Now we have a mapping of nodes to 1…m. Root is 1, then do a preorder scan of left subtree, then right. (This way we can reduce suffix tree rep to this plus leaf pointers).

However, another approach leads to a $2m+o(n)$ bit representation and, ultimately, to more powerful navigation. Jacob [*FOCS* 1989], Munro and Raman [*FOCS* 1997].

Use a heaplike ordering on a tree, that is:

i) Append leaves to all positions
ii) Read nodes level by level
iii) Write 1 for an internal node, 0 for an external

If the original tree has m nodes, we append m+1 external nodes, so have a 2n+1 bit representation.

Key Lemma: The children of i^{th} internal node are 2i and 2i+1.

Proof: Straightforward induction. Assume for a tree and position i, now convert to position i+1 by converting external node to internal. There are a couple of details

Up to and including internal node i there are i internal nodes and j external nodes.

Between internal node i and its left child there are i-j more nodes. So left child is node 2i.

Note:

i^{th} 1 corresponds to i^{th} internal node in level scan

i^{th} 0 corresponds to i^{th} external node

Consider the following operations on a bit string

rank(i): find #1's up to position i
select(i): find the position of i^{th} 1

[Note: these work on any bit string so this could be used as a data structure over [1…n] for rank queries]

The key issue is to answer queries quickly with this base vector and o(m) more space.

But first
left-child(i) = 2 rank(i)
right-child(i) = 2 rank(i) + 1
parent(i) = select($\lfloor i/2 \rfloor$)

This is "just like" a heap

Doing rank:

For every k^{th} position write down rank(i k) i = 0,…,m/k. This takes (m lg m)/k bits {k, the big block size, will be w(lg n), but wait}

So an answer is between two spots.

For every l^{th} position, write down rank since the preceding k^{th} position. This takes $(m \lg k)/l$ bits {l, the little block size, is also to be defined}

To get the answer we simply look at the bits since the beginning of the last small block. Let $l = 1/2 \lg m$ and use table lookup for all possible bit strings. This table has size $\sqrt{m}$.

So with $k = (\lg m)^2$, the big block information takes $O(m/\lg m)$ bits; and the little block information $O(m \lg\lg m / \lg m) = o(m)$ bits.

Select is trickier. Jacobson's original idea required inspecting O(lg m) bits widely scattered. Munro and Raman [*FOCS* 1997] avoid this problem.

That paper actually uses another related approach to give subtree size. Follow up papers have dealt with ordered trees, trees of fixed degrees (quartary trees) and dynamic versions of the problem).