COMS E6998-8: Advanced Data Structures (Spring'19) April 18, 2019

Lecture Lecture 12: String Matching and Suffix Trees

Instructor: *Omri Weinstein* Scribes: *Garrison Grogan*

1 Outline

We will focus on the following topics:

- Pattern matching in strings
- Tries, lexicographic search/library search
- suffix trees and applications
- $O(n)$ time contrustion of suffix trees

There are many applications to today's data structures.
The main hero of the lecture is the suffix trees.
In general what is our problem?

- Given text T (a huge data base of strings or collection of texts $T_1, ..., T_k$) we want to preprocess them such that we can answer all sorts of string/pattern matching queries on T.

  For example, if we were looking at DNA, we may want to search for the pattern $ACCTGA$, in this case $|\Sigma| = 4, \Sigma = \{A, C, T, G\}$. Many string data applications have come from genomics.

  We want a linear construction of our data structure, and $O(1)$ search.

2 Goal

Given a pattern P, we want to find/report/count the occurances of P in T in time $O(|P| + \underbrace{occ_T(P)}_{=k})$.

The dependence on $occ_T(P)$ is optimal.

Note there is no dependence on $|T|$, as the entire text/database is large.

How will we do this?

3 Σ-ary Trie

[1] We've seen tries before for binary search but they actually came first from string matching.

[1] Here and in future uses we want our alphabet size $|\Sigma|$ to be treated as a parameter. It is non-constant. We do this since lots of datastructures operate by blowing up $|\Sigma|$

A trie is a tree representation of a string, where every node can have a branching factor $|\Sigma|$. Each edge of the tree is lableled with a character of the alphabet. Each node then represents a string on characters.

Formally, a node n represents the string on letters on the edges followed to get to from the root node to n.

Our alphabet is small, so we can use an array of pointers at each node to point to the subtrees underneath it.

These tries are solutions to the following warmup problem: Predecessor/Lexicographic search in $T_1...T_k$. To search, we just follow the query string in the trie until it finishes or falls off. If we fall off the trie, the string we fell off at must be the predessor. Cleary this takes $O(|P|)$ time.

The use of Σ-ary tries then led to...

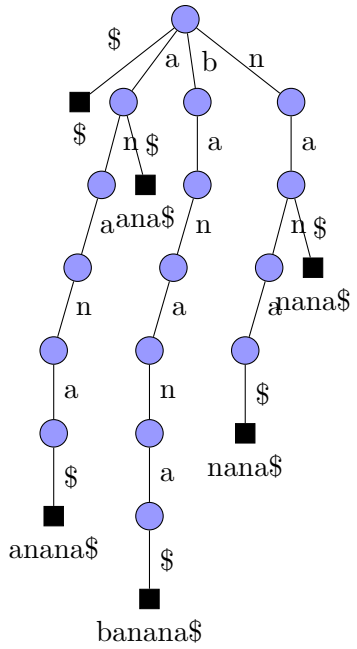
4 Suffix Trees

• Idea: A useful way to solve any any string problem is to construct a trie that represents all suffixes of the string. Ex: $T =$ "*banana*" We don't want one suffix to be a prefix of another, so we append \$, a small character $\notin \Sigma$. To construct the suffix tree, we build each suffix, and insert each one into a Σ-ary Trie

For the banana example, we construct the following suffixes:

1. **banana\$**
2. **anana\$**
3. **nana\$**
4. **ana\$**
5. **na\$**
6. **a\$**
7. **\$**

which then yields the following tree:



Now how does this help us?

Well if we are given a pattern P, Checking occurances is easy. We just walk down the tree. For example "na" appears in $na\$$ and $ana\$$. However there are few problems with our suffix tree-

1. Our suffix tree requires $O(n^2)$ letters of space, technically $O(|\Sigma|n^2)$ since we store arrays of size $|\Sigma|$. This
2. The preprocessing time for our tree is $O(|\Sigma|n^2)$
3. The query time is $O(\log |Sigma|)$

We can resolve these problems in the following way-

1. Many of our tree nodes have branching factor 1, so we can compress them. An easy fix is to contract all nodes of branching factor 1 into a single node. This process yields the **Compressed Suffix Tree**, hereby denoted as the **cST**.

   Example:
   n
   |
   a
   |
   $
   becomes a single node $na\$$

   Claim 1. *in a cST, space$(cST) = O(|\Sigma|n)$, in fact there is a 1 to 1 correpsondence between nodes and suffixes.*

   Proof. We will give a brief proof via induction.

Say we insert *ana*, either we add a leaf to a current branch in the tree, or we fall off the tree in the middle of a branch and create a new brancing node. Clearly at most 1 branching node can be added.

From this we having the following invariant: $\forall$ new suffixes added (in order), we create ≤ 1 new branching nodes. Thus we have removed the n^2 dependence and our CST. $\square$

Now we can still do better, and get rid of the $|\Sigma|$ factor with the following observation: the cST still naively stores an array at each node. We want to spend $O(|P|)$ time per node, the minimal time per node for predecessor search, and have a minimal space trade off. [2]

A binary search tree will give $O(n)$ space and $O(|P| \log |\Sigma|)$ search time. We just store a small BST, lexicographically ordered on the children in the cST.

We can still do better though and apply the more advanced predessor structures we've seen in class in a black box. Using VeB trees and space reduction via hashing we obtain $O(n)$ space and $O(|P| \log \log |\Sigma|)$ search time.

This is still suboptimal however. Using a Tray [3], we can achieve $O(n)$ space and $O(|P| + \log |\Sigma|)$ query time.[4]

It is, however, an open problem on doing predecessor search in $O(|P| + \log \log |\Sigma|)$ time.

One application of Trays is "Library Search", string sorting on a collection of texts $T_1 ... T_k$. The naive algorithm takes $O(k|T|)$ time, but we can do better.

The high level idea is that if has $<< |\Sigma|$ children, we can compress it into a mega node, in a sort of heavy light decomposition. Doing this we can obtain an algorithm that takes $O(\sum_{i=1}^{k} |T_i| + \log |\Sigma|) = O(|T| + k \log ||\Sigma|)$ time, where $|T|$ is the size of all the texts in our library.

Now we know we can solve string sort/patter match and similar problems in linear space and nearly constant time, but there is still a preprocess time problem. This was the open problem in string/pattern searching for a long time.

Before getting to that, we must discuss some other useful applications of Suffix Trees:

(a) PM counting/reporting ie, the first k occurances of a pattern in a string.
we can solve this problem in $O(|P| + k)$ time

(b) Solving the repeated string problem - finding the longest repeated substring.
To solve, we augment the cST with a letter depth counter at each node, which tracks the number of letters traversed since on that path in the tree since the root node. Then in $O(|T|)$ time, sweep the trie, and find the string with branching factor ≥ 2 and largest letter depth.

(c) Longest Common Prefix (LCP): Here we find the longest common prefix in our library of strings T, starting from two indices in T, denoted as $LCP(T[i:], T[j:]$
Easily, we can do it in $O(j - i)$, but there is an $O(1)$ solution. To see it, note that LCP is equivalent to a Least Common Ancester query on $T[j:]$ and $T[i:]$. We just look at the ith and jth suffix and look at the LCA. The LCA can be found in constant time.

[2] If we were just checking $\exists$ of the pattern, we could use hashing, but this doesn't support predecessor search.
[3] not covered in this course
[4] Via Information Theory, it is often possible to separate a multiplicative log factor into an additive one

(d) Longest Common Substring[5] : given T_1, T_2, how can we do this in linear time? it's on the homework.

(e) Document retrieval problem: Given $T_1...T_k$ report all occurances of a pattern in each of the T_i's. We could do this in $O(k|P|)$ time, building k suffix trees. Ideally, we want the run time to be instance optimal, proportional to the output size, i.e $OPT = \propto |P| + \#matches$

If we build a single cST using all of the texts concatenated together, $cST(T_1\$T_2...\$T_k)$ and leveld the ancestor data structure, we can embed a document retrieval query as a LA query in $O(OPT \log\log|T|)$.

2. Now we're still dealing with $O(n^2)$ steps in building the cST, essentailly we have an algorithm problem to generate the cST offline. It turns out that the following two objects suffice to construct a cST in $O(n)$ time:

   (a) Suffix Array

   (b) Longest Common Prefix (LCP) Array

   The Suffx Array stores indices of all suffixes of T in lex. sorted order, for example *banana* has the suffixes, in order,
   \$ *a*\$ *ana*\$ *anana*\$ *banana*\$ *na*\$ *nana*\$ and when given indices corresponding to their construction, the suffix array is $\boxed{7,6,4,2,1,5,3}$

   Suffix arrays can be constructed in linear time, as we will see later.

   Our 2nd object, the LCP array is easier to construct. We simply go through the SA, and for each pair of suffixes in order, store the length of the LCP. In the banana example, the LCPA is $\boxed{0,1,3,0,0,2}$.

   Claim 2. *the LCPA + SA give us a cST in linear time*

   Proof. Say we are inductively adding suffixes to the tree in order. The LCP array will tell us precisely how many letters to jump before inserting the entire string, so we don't spend time branching.

   Alternatively, there is a recursive algorithm using cartesian trees, not covered in the class. □

   We still need to construct the LCPA and SA in linear time. The LCPA is much easier, so we will focus on the SA.

5 Suffix Array Construction

Our goal is $\approx O(n)$ time construction of the SA.

Try #1: $O(n\log^2 n)$ expected time. Observation: If we can compare every pair of suffixes in time $f(n)$, then we can construct the SA in time $O(n\log(n)f(n))$.

Now we need to do $f(n)$ fast.

[5] continous substring, so no DP solution

Claim 3. *can compare any two suffixes of T in $O(\log n)$ randomized expected time using a "rolling hash" Karp-Rabin finger printing.*

Proof. Now we can check the equality of two suffixes with normal hashing but the worry is that constructing hashes of longer suffixes naively takes $O(n)$ time per hash since we have to read the entire suffix as the key, and we get $O(n^2)$ hashes.

But, if we have a hash that is linear with respect to character appending, i.e $h(x \circ x_i) = h(x_{<i}) + h(x_i)$, we can avoid reading the entire suffix on every new hash. This hashing also allows us to do equality checks.

We pay $\log n$ time to do smaller/greater comparisons by doing binary search and using a rolling hash at each comparision to find the first differing index of the two suffixes in constant time. □

Thus we have $n \log^2 n$ total expected time.

Now to do this in linear time? We use a three way divide and conquer algorithm. This algorithm runs in time $O(n + \underbrace{sort(\Sigma)}_{<<n}) \approx O(n)$

The algorithm does the following:

1. sort Σ
2. Partition T into 3 overlapping groups
   (a) $T_0 = \langle \underbrace{T[3i], T[3i+1], T[3i+2]}_{\forall i} \rangle$
   (b) T_1,T_0 with a shift of 1 letter to the right
   (c) T_2,T_1 with a shift of 1 letter to the right

   Each element of these partitions is dubbed a "mega letter", $\in \Sigma^3$. Each partition is of length $n/3$.

   We recursively apply this procedure on $\langle T_0 \circ T_1 \rangle$, and we know that $\langle T_0 \circ T_1 \rangle \in [\Sigma^3]^{2/3}$.

   Thus when the call returns we have via radix sort, all suffixes starting in positions 0 or 1 mod 3 sorted.

   Radix sort takes $O(n)$ time if the alphabet is constant size, and we have a recurrence relation $R(n) = R(2/3n) + O(n)$ to construct the SA, which is $O(n)$.

   Not forgetting the 2 mod 3 suffixes, we can sort and merge them into the the list from the recursive call given sorted 0 and 1 mod 3 suffixes, so we just merge all 3 sorted lists and get the SA in $O(n)$ time.