CSE 584A Class 11

Jeremy Buhler

February 24, 2016

1 Let's Talk Suffix Trees

Recall that suffix trees are like suffix arrays with common prefixes merged.

- Linear space, just like suffix arrays.
- Can be built directly from string in linear time, just like suffix arrays. (see, e.g., Ukkonen's algorithm (1995))
- For pattern P of length m, we've seen that we can find LCP of P and text in time $\Theta(m)$. (Suffices for matching.)
- (Homework asks you to investigate extension to substring matching.)
- With linear time and space preprocessing, suffix trees can be augmented to permit LCP queries on any two suffixes in constant time.
- In general, they're pretty awesome... except for space costs as we previously discussed, and in particular dependence on size of string alphabet Σ.
- Today, we'll talk about some applications of suffix trees that cannot (obviously) be done using suffix arrays alone.
- (Stay tuned for an explanation of the "obviously")

2 How To Build a Suffix Tree

First, let's see a quick and easy algorithm to build a suffix tree given what we already know about suffix arrays.

- Suppose you have sorted suffix array A and adjacent LCP array L.
- Here is a linear-time algo to build suffix tree.
- Start with empty tree (root node only).
- Create leaf node $\ell_{A[1]}$ off of root for suffix $\sigma_{A[1]}$.
- Let $d = \text{LCP}(A[1], A[2])$.
- "Go up" $|\sigma_{A[1]}| - d$ chars from leaf $\ell_{A[1]}$.

- If upward traversal ends at a node, let x be this node. Otherwise, if traversal ends in middle of edge e, create x by splitting e.
- Create leaf node $\ell_{A[2]}$ hanging directly off x.
- Repeat for all further suffixes in their order in A.

Does this result in a correct suffix tree?

- Consider insertion of suffix $\sigma_{A[i]}$, and let $d = \text{LCP}(A[i-1], A[i])$.
- Node x from which we do insertion is d chars below root, and its label matches the first d chars of $\sigma_{A[i]}$.
- Hence, leaf for $\sigma_{A[i]}$ belongs below x.
- All previous suffixes inserted into tree are $< \sigma_{A[i]}$.
- Hence, all share at most d chars in common with $\sigma_{A[i]}$.
- Conclude that $\sigma_{A[i]}$ cannot be added to any existing branch below x.
- Hence, our insertion is always correct.

Cost?

- We create $\Theta(n)$ nodes, and therefore $\Theta(n)$ edges.
- For each added suffix, we first move down to its newly created leaf in a single edge traversal (which takes constant time). This yields $\Theta(n)$ time spent moving down.
- Other than these initial downward moves on new leaf edges, we only traverse tree edges *upwards*, never downwards; hence, we never move up the same edge twice.
- Hence, we do $\Theta(n)$ upward edge traversals, each in constant time.
- Conclude that cost of construction is only $\Theta(n)$ overall.

3 Repeat Finding and Alignment

Repeat finding was a "killer app" for suffix trees in computational biology. It was widely implemented in the 1990s.

- Recall that repeats are common in genomes because of duplication events and self-replicating sequences.
- Repeats account for at least 50% of human genomic DNA!
- If repeated segments haven't diverged too much, they probably contain long (15-25 bases) exactly matching substrings
- **Goal**: find repeats by finding these exactly repeated segments

Another application of repeat finding is building **scaffolding for global alignment**.

- Suppose we want to globally align two or more related genomes
- (E.g. strains of bacteria, or mammalian genomes)
- **Step 1**: find local areas where the two genomes match perfectly (i.e. repeats shared by both)
- **Step 2**: Organize the matches into a scaffold (may entail recognizing genome rearrangements)
- **Step 3**: fill in gaps in scaffold using more aggressive alignment algorithms that handle mutations

4 Generic Repeat-Finding Problem

- **Defn**: Let S be a string of length n. Call α a **repeat** in S if it occurs as a substring of S in at least two different positions.
- **Defn**: an **instance** of α is a pair of starting points i, j such that there are copies of α beginning at both $S[i]$ and $S[j]$.
- **Problem**: given a sequence S, find all repeats in S, and enumerate all their instances.
- What is the "obvious" solution?
- Build a suffix tree for S, and find all internal nodes v of character depth at least d. The label of each such v represents a repeated sequence, and each pair of leaves ℓ_i, ℓ_j below v marks the starts of an instance.
- **Sloppiness**: if we just return sequences at nodes, we're not exactly solving the problem!
- **Example**:

- If $d = 3$, then technically we should report "aca," "cag," "agg," "acag," "cagg," and "acagg"
- But of these repeats, only "agg," "cagg," and "acagg" correspond to nodes of tree.
- We *could* modify the algorithm to find repeats not ending at nodes as well.
- But is it useful to report "cag" if it is always a substring of "cagg"? Probably not.
- **First Rule of Repeat Finding**: be careful what you wish for. You don't want to be buried alive in redundant output!

So what does current algorithm actually return?

- If repeated string α ends at an internal tree node, then there exist chars $x \neq y$ (one of which could be terminator \$) such that both αx and αy appear somewhere in S.
- Repeat instances satisfying above property are called **right-maximal**.
- Algo finds every repeat for which *some* instance is right-maximal.
- Note that S can contain both right-maximal and non-right-maximal instances of α. Consider "aca" in "acagacatacag."

5 Enumerating Instances

- **Fact**: at most n distinct repeated strings α occur in S w/at least one right-maximal instance.
- (Follows immediately from tree having at most n internal nodes.)
- Total number of instances of all repeats might be as large as $O(n^3)$ (n internal nodes, $O(n^2)$ pairs of leaves below each)
- However, we only want to report right-maximal instances, of which there can be only $O(n^2)$.
- Why? Each right-maximal instance of α should be reported only for single node v labeled with α.
- Can we list all right-maximal instances in time proportional to their number (plus $O(n)$ for tree traversal)?

Let's propose an algorithm.

- For each node v of suffix tree, define $L(v)$ to be set of all leaves below v.
- Clearly, for leaf ℓ_i, $L(\ell_i) = \{\ell_i\}$.
- Moreover, $L(v) = \bigcup_{w \in \text{children of } v} L(w)$.
- For given node v, want to enumerate all pairs of leaves that came from *different* subtrees of v.

- That is, want all pairs ℓ_i, ℓ_j such that for two children $w_1 \neq w_2$ of v, $\ell_i \in L(w_1)$ and $\ell_j \in L(w_2)$.

Here's a postorder recursive algo to enumerate all these pairs.

```
EnumInstances(v)
  if v is a leaf
    return {v}
  else
    let w_1 ... w_k be children of v
    for i in 1..k do
      L(w_i) ← EnumInstances(w_i)
      for j in 1..i − 1 do
        report all pairs from L(w_j) × L(w_i)
    return ⋃_i L(w_i)
```

Example:

Does this algo take time proportional to number of enumerated pairs?

- Number of children of each v is at most $|\Sigma|$, so work to enumerate instances from all pairs of children is $O(|\Sigma|^2)$ (a constant) plus # instances.
- What about cost of computing union?
- If we create $L(v)$ by copying all the $L(w_i)$, we could spend $\Theta(n)$ time in copying at a single node.
- Entire enumeration would therefore be $\Theta(n^2)$, regardless of # of matches!
- Alternative: store sets $L(v)$ in a form that permits constant-time union, e.g. as circular linked lists.
- If two sets can be merged in constant time, then union of sets for all children takes time $O(|\Sigma|)$.
- Conclude that entire enumeration algo can run in time $O(n|\Sigma|^2+\#$ right-maximal instances).

6 Left-Maximality

Our list of repeats is still rather redundant. For example, given $S = acaggacaggt$ above, we return both "cagg" and "acagg," even though one occurs only as a suffix of the other.

- **Defn**: An instance of a repeat α is *left-maximal* if either (1) one of the copies occurs starting at $S[1]$ or (2) the characters preceding the two copies are distinct.
- **Defn**: An instance is **maximal** if it is both left- and right-maximal.
- Maximal repeat instances are not embedded in a longer instance of another repeat.

Can we tweak our enumeration algorithm to return only **maximal** repeats, again in time proportional to their number (plus $O(m)$)?

- Given a pair of leaves ℓ_i, ℓ_j, $i < j$, when does it represent a left-maximal repeat instance?
- When either $i = 1$, or $S[i-1] \neq S[j-1]$.
- **Idea 1**: minimally tweak previous algo to check each pair of leaves for left-maximality before reporting it.
- *Problem*: we spend time to check instances that we don't return. So, time not proportional to actual output size!
- **Idea 2**: For each leaf ℓ_i, define "previous character" of leaf to be $S[i-1]$, or "*" if $i = 1$.
- Keep leaves with different previous chars separate throughout algorithm!
- Define $L_c(v)$ to be the set of all leaves ℓ below v whose prev char is c.
- For leaf, $L_c(\ell) = \{\ell\}$ if leaf's prev char is c, or empty set if it is not.
- Clearly, we can compute all $L_c(v)$'s bottom-up for all nodes of tree just as above, in time $O(n|\Sigma|^2)$.
- Moreover, to enumerate pairs of leaves at v with different prev chars, replace enumeration of single set $L(w_j) \times L(w_i)$ by enumeration of $O(|\Sigma|^2)$ sets $L_a(w_j) \times L_b(w_i)$, for chars $a \neq b$.
- Resulting algo is $O(n|\Sigma|^4 + \#$ maximal instances).

One last note: there are other notions of "right" set of repeats to return besides the maximal ones.