15-853: Algorithms in the Real World

Suffix Trees

Exact String Matching

- Given a text $T$ of length $m$ and pattern $P$ of length $n$
- "Quickly" find an occurrence (or all occurrences) of $P$ in $T$
- A Naïve solution: Compare $P$ with $T[i...i+n]$ for all $i$ --- $O(nm)$ time
- How about $O(n+m)$ time? (Knuth Morris Pratt)
- How about $O(m)$ preprocessing time and $O(n)$ search time?

Suffix Trees

- Preprocess the text in $O(m)$ time and search in $O(n)$ time
- Idea:
  - Construct a tree containing all suffixes of text along the paths from the root to the leaves
  - For search, just follow the appropriate path

Suffix Trees

A suffix tree for the string $x a b x a c$

Search for the string $a b x$
Constructing Suffix trees

- Naive $O(m^2)$ algo
- For every $i$, add the suffix $S[i..m]$ to the current tree

Ukkonen's linear-time algorithm

- We will start with an $O(m^3)$ algorithm and then give a series of improvements
- In stage $i$, we construct a suffix tree $T_i$ for $S[1..i]$
- Incrementing $T_i$ to $T_{i+1}$ naively takes $O(i^2)$ time because we insert each of the $i$ suffixes
- Thus a total of $O(m^3)$ time
**Going from $T_i$ to $T_{i+1}$**

- In the $j^{th}$ substage of stage $i+1$, we insert $S[j..i+1]$ into $T_i$. Let $S[j..i] = \beta$.

- Three cases
  - **Rule 1**: The path $\beta$ ends on a leaf $\Rightarrow$ add $S[i+1]$ to the label of the last edge
  - **Rule 2**: The path $\beta$ continues with characters other than $S[i+1]$ $\Rightarrow$ create a new leaf node and split the path labeled $\beta$
  - **Rule 3**: A path labeled $\beta S[i+1]$ already exists $\Rightarrow$ do nothing.

---

**Idea #1 : Suffix Links**

- In each substage, we first search for some string in the tree and then insert a new node/edge/label
- Can we speed up looking for strings in the tree?

- In any substage, we look for a suffix of the strings searched in previous substages
- Idea: Put a pointer from an internal node labeled $x\alpha$ to the node labeled $\alpha$
  - Such a link is called a “Suffix Link”

---

**Suffix Links - Bounding the time**

- Steps in each substage
  - Go up 1 link to the nearest internal node
  - Follow a suffix link to the suffix node
  - Follow the path for the remaining string

- First two steps together make up $O(m)$ in each stage
- The third step follows only as many links as the length of the string $S[1..i]$
- Thus the total time per stage is $O(m)$
Maintaining Suffix Links

- Whenever a node labeled $x\alpha$ is created, in the following substage a node labeled $\alpha$ is created. Why?
- When a new node is created, add a suffix link from it to the root, and if required, add a suffix link from its predecessor to it.

---

Going from $O(m^2)$ to $O(m)$

- Can we even hope to do better than $O(m^2)$?
- Size of the tree itself can be $O(m^2)$
- But notice that there are only $2m$ edges! - Why?
- Idea: represent labels of edges as intervals
- Can easily modify the entire process to work on intervals

---

Idea #2 : Getting rid of Rule 3

- Recall Rule 3: A path labeled $S[j..i+1]$ already exists $\Rightarrow$ do nothing.
- If $S[j..i+1]$ already exists, then $S[j+1..i+1]$ exists too and we will again apply Rule 3 in the next substage
- Whenever we encounter Rule 3, this stage is over - skip to the next stage.

---

Idea #3 : Fast-forwarding Rules 1 & 2

- Rule 1 applies whenever a path ends in a leaf
- Note that a leaf node always stays a leaf node - the only change is to append the new character to its edge using Rule 1
- An application of Rule 2 in substage $j$ creates a new leaf node. This node is then accessed using Rule 1 in substage $j$ in all the following stages
Idea #3: Fast-forwarding Rules 1 & 2

- Fast-forward Rule 1 and 2
  - Whenever Rule 2 creates a node, instead of labeling the last edge with only one character, implicitly label it with the entire remaining suffix
- Each leaf edge is labeled only once!

Another Way to Think About It

- Insert finger
- Search finger
- Increment when $S[j..i]$ is not in tree (Rule 2)
- Increment when $S[j..i]$ is in tree (Rule 3)

1. Insert $S[j..n]$ into tree by branching at $S[j..i-1]$ if there is one
2. Create suffix pointer to new node at $S[j..i-1]$ if there is one
3. Use parent suffix pointer to move finger to $j+1$

Leaf edge labels are updated by using a variable to denote the end of the interval
Complexity Analysis

- Rule 3 is used only once in every stage
- For every j, Rule 1 & 2 are applied only once in the jth substage of all the stages.
- Each application of a rule takes O(1) steps
- Other overheads are O(1) per stage
- Total time is O(m)

Extending to multiple lists

- Suppose we want to match a pattern with a dictionary of k strings
- Concatenate all the strings (interspersed with special characters) and construct a common suffix tree
- Time taken = O(km)
- Unnecessary complicated tree; needs special characters

Multiple lists - Better algorithm

- First construct a suffix tree on the first string, then insert suffixes of the second string and so on
- Each leaf node should store values corresponding to each string
- O(km) as before

Longest Common Substring

- Find the longest string that is a substring of both S1 and S2
- Construct a common suffix tree for both
- Any node that has leaf nodes labeled by S1 and S2 in the subtree rooted at it gives a common substring
- The “deepest” such node is the required substring
- Can be found in linear time by a tree traversal.
Common substrings of M strings

- Given M strings of total length n, find for every k, the length $l_k$ of the longest string that is a substring of at least k of the strings
- Construct a common suffix tree
- For every internal node, find the number of distinctly labeled leaves in the subtree rooted at the node
- Report $l_k$ by a single tree traversal
- $O(Mn)$ time – not linear!

Lempel-Ziv compression

- Recall that at each stage, we output a pair $(p_i, l_i)$ where $S[p_i .. p_i+l_i] = S[i .. i+l_i]$
- Find all pairs $(p_i, l_i)$ in linear time
- Construct a suffix tree for S
- Label each internal node with the minimum of labels of all leaves below it – this is the first place in S where it occurs. Call this label $c_v$.
- For every i, search for the string $S[i .. m]$ stopping just before $c_v \geq i$. This gives us $l_i$ and $p_i$. 